LANGAGES DE PROGRAMMATION
IFT-3000
HIVER 2016

Travail pratique 1 - TRAVAIL INDIVIDUEL
Pondération de la note finale : 14.5%

À remettre, avant 17h, le vendredi 4 mars 2016

1 Énoncé

Ce travail pratique consiste à implanter différentes fonctions pour la gestion d'un réseau de transport permettant, à un utilisateur quelconque, de se renseigner facilement sur le réseau de transport en commun dans la ville de Québec. Vous devez utiliser les notions que nous avons vues dans le cours en faisant des manipulations de base sur un système qui est implanté en utilisant différents types de structures de données : listes, enregistrements, tuples, tables de hachage, etc.

1.1 Données ouvertes

Nous proposons d'utiliser les données ouvertes du Réseau de Transport de la Capitale (RTC), qui répertorient, entre autres, l'emplacement des stations de bus dans la ville, les informations sur les lignes de bus (comme les horaires et les arrêts), etc. Ces données sont disponibles dans un format compressé (zip) à l'adresse suivante [1] :

`http://www.rtcquebec.ca/Portals/0/Admin/DonneesOuvertes/googletransit.zip`

Une fois le fichier décompressé, vous verrez que celui-ci contient plusieurs fichiers au format CSV (même si l'extension des fichiers est txt). Le séparateur de colonnes dans tous les fichiers est la virgule (,) et le texte des données de certaines colonnes est entre guillemets ; la première ligne de chaque fichier précisant les titres des différentes colonnes (que nous utiliserons comme des noms de champs d'enregistrements). Toutes ces données sont sous format GTFS [2], un format standardisé pour communiquer des horaires de transports en commun et les informations géographiques associées ; ainsi, outre le RTC, la Société de Transport de Montréal (STM) utilise [3] ce même format pour rendre publics ses horaires d'autobus et de métro. L'intérêt étant qu'une application basée sur ce format pourra être utilisée pour traiter différents réseaux de transport en commun.

1.2 Données utilisées dans ce travail pratique

Le diagramme de classes présenté dans la figure 1 présente les structures de données utilisées dans ce TP. Il schématise le fait que le réseau de transport de le RTC a plusieurs stations qui sont représentées par le type *station*. Chaque station a un identifiant unique, et une position GPS qui indique son emplacement précis. Les lignes d'autobus qui relient les stations sont représentées par le type *ligne*. Chaque ligne est identifiée par un numéro (ex : "800", "11a" ou "350") et un identifiant «ligne_id» qui permettra d'identifier les voyages que fait cette ligne. Chaque voyage d'une ligne est effectuée dans une direction particulière vers une destination donnée. Cette notion de voyage est représentée par le type *voyage* qui est identifié de manière unique par le «voyage_id». Pendant un voyage, l'autobus fait une séquence d'arrêts à différentes stations ; le type *arret* permet de les représenter. Chaque arrêt est donc identifié de manière unique par la paire d'identifiants du voyage et de la station. De plus, chaque arrêt prévu par le RTC implique que l'autobus faisant le voyage arrive à la station correspondante à une heure donnée et reparte à une heure donnée. Pour terminer, noter que chaque voyage étant effectué à une date donnée, est aussi associé à la notion de date de service (type *service*). En effet, une date de service permet de faire le lien entre une date et une partie de l'horaire des voyages prévus par le RTC. Ainsi, pour une date donnée, nous pouvons avoir un ou plusieurs «service_id», lesquelles vont référer à différents groupes de voyages effectués à cette date.

1. Notez que nous vous remettons, avec le code du TP, les fichiers du RTC, en version réduite (données du 12-02-2016 au 04-03-2016), afin d'éviter trop de temps d'attente lors de leur chargement en mémoire.
2. https://fr.wikipedia.org/wiki/General_Transit_Feed_Specification.
3. http://www.stm.info/fr/a-propos/developpeurs/description-des-donnees-disponibles.

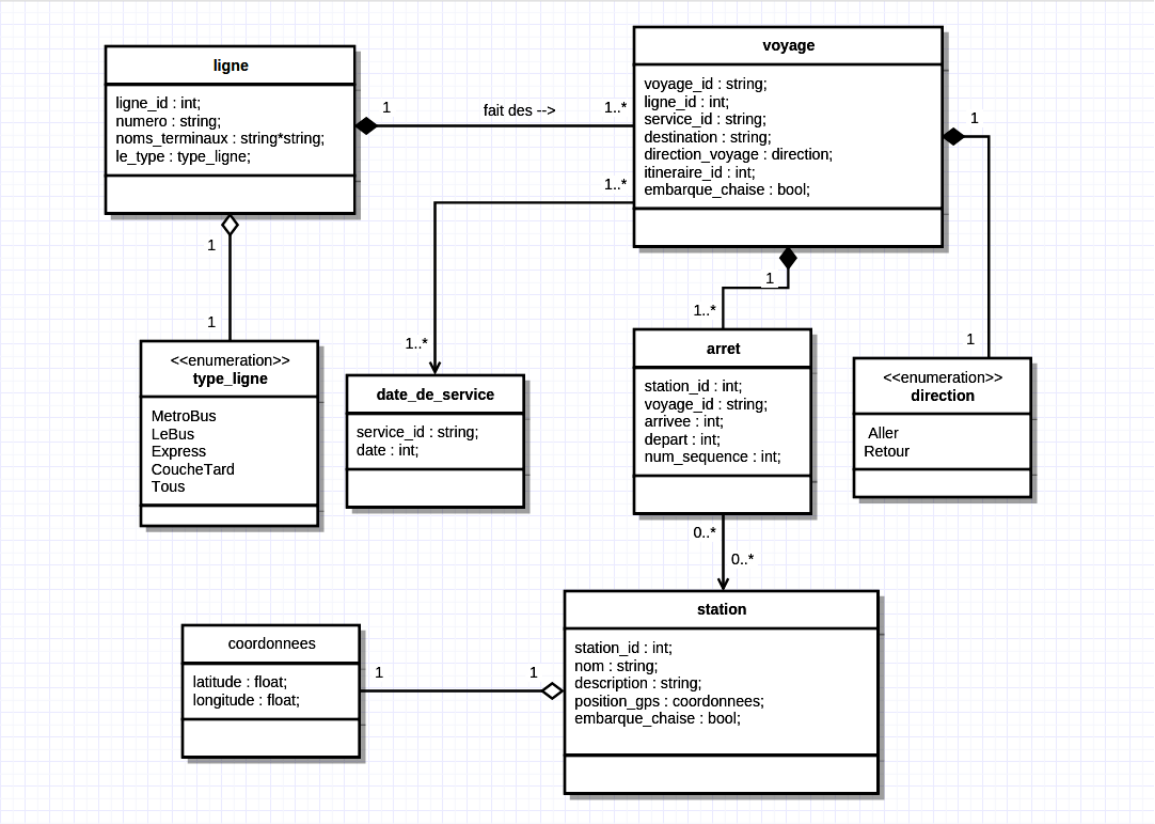

FIGURE 1 – Diagramme de classes du RTC.

1.2.1 Données du RTC utilisées dans ce travail pratique

Dans le cadre de ce travail pratique, nous nous limitons aux fichiers suivants (relativement à ceux fournis par le RTC) :

routes.txt Ce fichier comprend des données relatives aux lignes d'autobus du réseau. Par exemple, la "800", la "801" et la "13A" sont toutes des lignes différentes. Veuillez, noter qu'il y a deux entrées dans le fichier pour chaque ligne d'autobus, ce qui implique qu'un numéro de ligne est associé à deux identifiants différents. Pour votre information, le fichier est composé des champs suivants :

- route_id : identifiant unique de la ligne d'autobus ;
- agency_id : identifiant unique de l'agence (RTC) ;
- route_short_name : texte permettant d'identifier un parcours sur les horaires et panneaux de signalisation ; typiquement, il s'agit du numéro de la ligne ("7", "800", "801", "13A", "13B", etc.) ;
- route_long_name : texte permettant d'identifier, avec plus de précisions, une ligne sur les horaires et panneaux de signalisation (ce champ est vide dans le fichier fourni par le RTC) ;
- route_desc : texte décrivant la ligne ; le RTC a choisi de préciser les noms des terminaux ;
- route_type : identifie le type de véhicule de transport [4] ; la valeur est fixée à 3 pour indiquer qu'il s'agit d'autobus ;
- route_url : comprend une URL qui mène vers une page décrivant la ligne en question ;
- route_color : couleur permettant d'identifier visuellement la ligne ;
- route_text_color : couleur permettant d'identifier visuellement un texte décrivant la ligne.

Dans le cadre de ce travail, nous n'utiliserons qu'une partie de ces données ; plus précisément, un type enregistrement «ligne» est défini comme suit :

4. Dans la page https://developers.google.com/transit/gtfs/reference#routes_fields, on retrouve les différents types de véhicules pouvant être considérés, comme le métro, le train, etc.

```
type ligne =
    {
        ligne_id: int;                    (* correspond à route_d *)
        numero: string;                   (* correspond à route_short_name *)
        noms_terminaux: string * string;  (* construit à partir de route_desc *)
        le_type: type_ligne;              (* construit à partir de route_color *)
    }
```

Aussi, deux tables de hachage sont associées à ces données :

```
let lignes = Hashtbl.create 5000
(* Associe aux numéros de la ligne (clé) l'enregistrement au complet *)

let lignes_par_id = Hashtbl.create 5000
(* Associe aux identifiants de la ligne (clé) les numéros d'autobus  *)
```

stop.txt Ce fichier comprend des données relatives aux stations ; pour votre information, il est composé des champs suivants :

- stop_id : identifiant unique d'une station ;
- stop_name : contient le nom de la station ;
- stop_desc : contient la description de la station ;
- stop_lat : contient la latitude de la station au format wgs84 [5] ;
- stop_lon : contient la longitude de la station au format wgs84 ;
- stop_url : contient l'URL d'une page décrivant une station en particulier ;
- location_type : différencie les stations entre-elles ; si ce champ est vide, ce qui est le cas dans les données publiées par le RTC, c'est une station par défaut ;
- wheelchair_boarding : identifie si l'embarquement de fauteuil roulant est possible à l'arrêt ; une valeur
  - 0 (ou vide) indique que cette information n'est pas disponible pour cette station ;
  - 1 indique qu'un véhicule passant par cette station peut accueillir au moins un fauteuil roulant ;
  - 2 indique qu'aucun fauteuil roulant ne peut être pris en charge à cette station.

Dans le cadre de ce travail, nous n'utiliserons qu'une partie de ces données ; plus précisément, un type enregistrement «station» est défini comme suit (notez que les champs «embarque_chaise» et «description», bien que présents dans la structure, ne sont pas utilisés dans ce TP) :

```
type station =
    {
        station_id: int;           (* correspond à stop_id *)
        nom: string;               (* correspond à stop_name *)
        description: string;       (* correspond à stop_desc *)
        position_gps: coordonnees; (* construit à partir de stop_lat et stop_lon *)
        embarque_chaise: bool      (* construit à partir de wheelchair_boarding *)
    }
```

Une table de hachage est associée à ces données :

```
let stations = Hashtbl.create 100000
(* Associe aux identifiants de stations (clé) l'enregistrement au complet *)
```

trips.txt Ce fichier comprend des données relatives aux voyages effectués par les autobus ; pour votre information, il est composé des champs :

- route_id : identifiant unique d'une ligne ;
- service_id : identifiant unique d'un ensemble de dates durant laquelle la RTC a un horaire de service défini ;
- trip_id : identifiant unique d'un voyage ;
- trip_headsign : indique aux passagers la destination du voyage ;
- trip_short_name : texte permettant d'identifier un trajet sur les horaires et panneaux de signalisation ;
- direction_id : valeur binaire indiquant le sens du voyage ;
- block_id : champ identifiant le bloc auquel appartient le voyage ; un bloc est constitué de deux ou plusieurs voyages successifs effectués avec le même véhicule ;
- shape_id : contient l'identifiant de la forme de l'itinéraire ;

5. https://fr.wikipedia.org/wiki/WGS_84.

– wheelchair_accessible : identifie si l'embarquement de fauteuil roulant est possible à une certaine station.

Dans le cadre de ce travail, nous n'utiliserons qu'une partie de ces données ; plus précisément, un type enregistrement «voyage» est défini comme suit (notez que les champs «direction_voyage», «itineraire_id» et «embarque_chaise», bien que présents dans la structure, ne sont pas utilisés dans ce TP) :

```
type voyage =
    {
        voyage_id: string;            (* correspond à trip_id *)
        ligne_id: int;                (* correspond à route_id *)
        service_id: string;           (* correspond à service_id *)
        destination: string;          (* correspond à trip_short_name *)
        direction_voyage: direction;  (* construit à partir de direction_id *)
        itineraire_id: int;           (* correspond à shape_id *)
        embarque_chaise: bool;        (* correspond à  wheelchair\_accessible *)
    }
```

Trois tables de hachage sont associées à ces données :

```
let voyages = Hashtbl.create 500000
(* Associe à l'identifiant du voyage (clé) l'enregistrement au complet *)

let voyages_par_service = Hashtbl.create 500000
(* Associe à l'identifiant du service (clé) l'identifiant du voyage    *)

let voyages_par_ligne = Hashtbl.create 500000
(* Associe à l'identifiant de la ligne (clé) l'identifiant du voyage   *)
```

stop_times.txt Ce fichier comprend des données relatives aux arrêts effectués par les autobus ; pour votre information, il est composé des champs :

– trip_id : identifiant du voyage ;
– arrival_time : spécifie l'heure d'arrivée de l'autobus pour cet arrêt (d'un voyage en particulier) ;
– departure_time : spécifie l'heure de départ de l'autobus pour cet arrêt (d'un voyage en particulier) ;
– stop_id : identifiant unique d'une station ;
– stop_sequence : identifie le numéro de séquence de cet arrêt dans l'ensemble des arrêts effectués pour un voyage en particulier (il permet ultimement de reconstituer l'ensemble des arrêts effectués pour un voyage dans le bon ordre) ;
– pickup_type : indique si les passagers peuvent embarquer à l'arrêt en question ou si seules les descentes sont permises ;
– drop_off_type : indique si les passagers sont déposés à l'arrêt selon l'horaire prévu ou que le débarquement n'est pas disponible.

Dans le cadre de ce travail, nous n'utiliserons qu'une partie de ces données ; plus précisément, un type enregistrement «arret» est défini comme suit (notez que les champs «depart», «embarque_client» et «debarque_client», bien que présents dans la structure, ne sont pas utilisés dans ce TP) :

```
type arret =
    {
        station_id: int;          (* correspond à stop_id *)
        voyage_id: string;        (* correspond à trip_id *)
        arrivee: int;             (* correspond à arrival_time *)
        depart: int;              (* correspond à departure_time *)
        num_sequence: int;        (* correspond à stop_sequence *)
        embarque_client: bool;    (* correspond à pickup_type *)
        debarque_client: bool;    (* correspond à drop_off_type *)
    }
```

Trois tables de hachage sont associées à ces données :

```
let arrets = Hashtbl.create 3000000
(* Associe à la paire d'identifiants de la station et du voyage (clé) l'enregistrement au complet*)

let arrets_par_voyage = Hashtbl.create 3000000
(* Associe à l'identifiant du voyage (clé) la paire d'identifiants de la station et du voyage   *)

let arrets_par_station = Hashtbl.create 3000000
(* Associe à l'identifiant de la station (clé) la paire d'identifiants de la station et du voyage*)
```

calendar_dates.txt Ce fichier comprend des données relatives aux dates où les services sont disponibles ; pour votre information, il est composé des champs :

- service_id : identifiant unique d'un ensemble de dates durant lesquelles la RTC a un horaire de service défini ;
- date : spécifie une date durant laquelle un service est offert ;
- exception_date : indique si le service est disponible à la date spécifiée.

Dans le cadre de ce travail, nous n'utiliserons qu'une partie de ces données ; plus précisément, un type enregistrement «service» est défini comme suit :

```
type service =
    {
        service_id : string;    (* correspond à service_id *)
        date : int;             (* correspond à date *)
    }
```

Une table de hachage est associée à ces données :

```
let services = Hashtbl.create 100000
(* Associe à la date (clé) l'enregistrement au complet *)
```

2 Éléments fournis

Le code qui vous est fourni comprend 1 répertoire et 6 fichiers :

1. «rtc_data» : répertoire qui comprend une version réduite des données du RTC.
2. «utiles.ml» : fichier qui comprend 2 modules utiles pour la réalisation du TP :
   (a) Utiles : comprend des fonctions utiles pour réaliser les fonctions à implanter dans votre TP.
   (b) Date : comprend des fonctions de manipulation d'heures et de dates.
3. «reseau.ml» : fichier qui comprend la définition du module «Reseau_de_transport» ; ce dernier comprend les différentes définitions des types enregistrements présentés dans la section précédente du présent document, ainsi que certains types «somme» ; on y trouve aussi la définition de fonctions utilisées lors du chargement, en mémoire, des données des différents fichiers fournis par le RTC ; notez que ce module comprend aussi quelques fonctions utiles liées aux structures de données définies.
4. «tp1.mli» : comprend la définition de la signature «GESTIONNAIRE_TRANSPORT» que doit respecter le module principal du TP. Cette signature précise donc les types des différentes fonctions à implanter dans ce travail ; notez que nous fournissions l'implantation de 4 d'entre elles.
5. «tp1.ml» : comprend la définition du module «Gestionnaire_transport», principal module du TP, où s'y retrouvent toutes les fonctions (13 en tout) que vous devez compléter ; un texte en commentaire, «À IMPLANTER/COMPLÉTER (XY PTS)», signale que vous devez compléter la définition d'une fonction, et précise le nombre de points accordés pour la fonction en question.
6. «tests.ml» : comprend un ensemble de tests des différentes fonctions à implanter dans ce TP.
7. «trace.ml» : indique les résultats attendus par les tests (point précédent), dans l'interpréteur OCaml.

2.1 Démarche à suivre

Voici la démarche que je vous suggère de suivre :

- ouvrer, à l'aide d'un éditeur (Emacs par exemple), les différents fichiers .ml du TP ; lancer l'interpréteur Ocaml ;
- au niveau de l'interpréteur, charger («**#use** "tp1.ml" ; ;») le fichier «tp1.ml» ; ceci entraine aussi le chargement des fichiers «tp1.mli» et «reseau.ml» (ce dernier entrainant le chargement de «utiles.ml»).
  Ainsi, à cette étape, tous les modules du TP sont disponibles au niveau de l'interpréteur ; il est alors possible :
  - d'ouvrir le module «Utiles» («**open** Utiles ; ;») et de tester les fonctions qui y sont définies ;
  - d'ouvrir le module «Reseau_de_transport» («**open** Reseau_de_transport ; ;»), de tester les fonctions qui y sont définies, de manipuler les différentes structures de données, et de charger en mémoire (construction des différentes tables de hachage) les données du RTC : ce chargement se faisant par la biais de la fonction «charger_tout» qui, par défaut, tente de chercher les données dans le répertoire «/home/etudiant/workspace/tp1/rtc_data/». Évidemment, il est possible de préciser à la fonction d'aller chercher les fichiers dans un autre répertoire :

```
# charger_tout ~rep:"rtc_data/" ();;
...
```

  - d'ouvrir le module «Gestionnaire_transport» («open Gestionnaire_transport ; ;») pour tester les fonctions qui y sont déjà définies.
- complétez progressivement les fonctions à implanter.

3 Interface (signature)

La signature *GESTIONNAIRE_TRANSPORT* (fichier tp1.mli) est définie comme suit :

```
module type GESTIONNAIRE_TRANSPORT = sig

  (* Fonctions fournies - déjà implantées *)
  val map_voyages_passants_itineraire : ?heure:int -> int -> int
                                      -> (Reseau_de_transport.arret
                                          * Reseau_de_transport.arret
                                          * string -> 'a)
                                      -> string list -> 'a list
  val lister_numero_lignes: unit -> string list
  val lister_id_stations: unit -> int list
  val trouver_voyages_sur_la_ligne: ?date:int option -> string -> string list

  (* Fonctions à compléter/à implanter    *)
  val lister_numero_lignes_par_type :
    ?types:Reseau_de_transport.type_ligne list ->
    unit -> (Reseau_de_transport.type_ligne * string list) list
  val trouver_service: ?date:int -> unit -> string list
  val trouver_voyages_par_date: ?date:int -> unit -> string list
  val trouver_stations_environnantes: Reseau_de_transport.coordonnees -> float
                                    -> (int * float) list
  val lister_lignes_passantes_station: int -> string list
  val lister_arrets_par_voyage: string -> int list
  val trouver_horaire_ligne_a_la_station: ?date:int -> ?heure:int -> string
                                       -> int -> string list
  val lister_stations_sur_itineraire_ligne: ?date:int option -> string
                                       -> (string * int list) list
  val ligne_passe_par_station: ?date:int option -> string -> int -> bool
  val duree_du_prochain_voyage_partant: ?date:int -> ?heure:int -> string
                                      -> int -> int -> int
  val duree_attente_prochain_arret_ligne_a_la_station: ?date:int -> ?heure:int
                                                     -> string -> int -> int
  val ligne_arrive_le_plus_tot: ?date:int -> ?heure:int -> int -> int
                              -> string * int
  val ligne_met_le_moins_temps: ?date:int -> ?heure:int -> int -> int
                              -> string * int

end
```

Les 4 premières fonctions sont déjà implantées ; l'objectif du TP est de compléter la définition des 13 autres fonctions, en respectant les signatures de fonctions spécifiées.

4 Implantation (module)

Le module *Gestionnaire_transport* (fichier tp1.ml) est définie comme suit :

```
module Gestionnaire_transport : GESTIONNAIRE_TRANSPORT = struct
  ...
end
```

Ainsi, toutes les fonctions implantées dans ce module, et déclarées dans la signature *GESTIONNAIRE_TRANSPORT*, doivent respecter les types de fonctions précisées dans cette signature. Vous pouvez évidemment mettre en commentaire cette contrainte :

```
module Gestionnaire_transport (* : GESTIONNAIRE_TRANSPORT *) = struct
  ...
end
```

Notez cependant que, lors de la correction, votre module *Gestionnaire_transport* sera de nouveau contraint par rapport à la signature *GESTIONNAIRE_TRANSPORT* :

```
1  (* Fichier corrige.ml *)
2
3  # #use "tp1.ml";;
4    ...
5
6  # module M = (Gestionnaire_transport : GESTIONNAIRE_TRANSPORT);;
7  module M : GESTIONNAIRE_TRANSPORT
8
9  # open M;;
10
11 # (* Test de vos fonctions à partir du module M *)
```

Ainsi, si vous auriez mis en commentaire la contrainte, et que l'une de vos fonctions s'avère mal typée, la ligne 6 (voir ci-dessus) ne passera pas ; OCaml le signalera par une erreur de typage.

Pour terminer, notez que la description de toutes les fonctions à implanter se trouve, sous forme de commentaires, dans le fichier «tp1.ml» ; aussi, les fichiers «test.ml» et «trace.ml» comprennent des exemples d'utilisations de ces fonctions, ainsi que les résultats attendus.

5 À remettre

Il faut remettre un fichier .zip contenant uniquement le fichier "tp1.ml" complété. Le code doit être clair et bien indenté [6].

6 Remarques importantes

Plagiat : Tel que décrit dans le plan de cours, le plagiat est interdit. Une politique stricte de tolérance zéro est appliquée en tout temps et sous toutes circonstances. Tous les cas seront référés à la direction de la Faculté.

Travail remis en retard : Tout travail remis en retard se verra pénalisé de 25% par jour de retard. Chaque journée de retard débute dès la limite de remise dépassée (dès la première minute). Un retard excédant 2 jours (48h) provoquera le rejet du travail pour la correction et la note de 0 pour ce travail. La remise doit se faire par la boite de dépôt du TP1 dans la section «Évaluation et résultats».

Barème : Au niveau des fonctions à implanter, le barème est précisé dans le fichier "tp1.ml". Notez cependant que :
- (-10pts), si votre code ne compile pas (provoque une erreur/exception lors du chargement du fichier "tp1.ml" dans l'interpréteur, i.e. «**#use** "tp1.ml" ;;») ;
- (-5pts), si votre code n'est pas clair et bien indenté ;
- (-25%pts, par fonction), si une fonction est implantée en utilisant des références (paradigme impératif).

Bon travail.

6. Suggestion : http://www.cs.princeton.edu/~dpw/courses/cos326-12/style.php.