TP : Sérialisation/Désérialisation

Langages OCaml et C

Exercice 1.

Sérialisation/désérialisation d'arbres binaires en C.

Dans ce sujet, les arbres binaires ont des étiquettes entières positives. Les fils d'une feuille sont deux arbres vides. L'arbre vide est modélisé en C par le pointeur `NULL`.

On sérialise un arbre binaire en écrivant les étiquettes des nœuds selon l'ordre d'un parcours en profondeur préfixe. On utilise un marqueur pour l'arbre vide : par exemple -1.

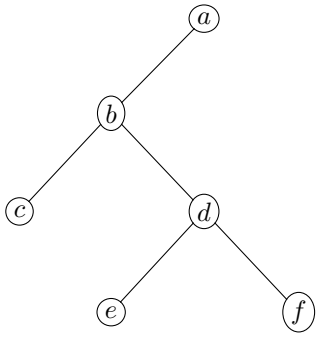

FIGURE 1 – Codage **a b c -1 -1 d e -1 -1 f -1 -1 -1**

- Télécharger l'archive pour ce sujet en entrant dans un terminal :

```
$ wget https://nussbaumcpge.be/public_html/Sup/MP2I/serialiser_btree.zip
```

L'archive contient le squelette d'un fichier C à compléter.

1. Écrire la fonction `void serialise(const char * filename, Node * root)` qui sérialise un arbre binaire dans un fichier texte dont le nom est donné. Le contenu attendu pour le fichier de sérialisation est donné en commentaire dans le `main`.
2. Écrire la fonction `Node * deserialise(const char * filename)` qui retourne l'arbre binaire résultat de la désérialisation du fichier dont le nom est donné en paramètre.

Exercice 2. Dans cet exercice on étudie une sérialisation de graphe en OCaml. On rappelle l'existence des fonctions suivantes :

```
# String.split_on_char ':' "\"0\":[0,1,4],";;
- : string list = ["\"0\""; "[0,1,4],"]
# String.sub "abcdefgh" 1 3;;
```

```
- : string = "bcd"
# int_of_string "32";;
- : int = 32
```

Les graphes sont représentés par le type

```
type graph = int list array;;
```

On veut sérialiser un graphe sous forme d'un fichier au format JSON. Par exemple le graphe `[|[0;1;4];[];[0;3];[2];[1]|]` est sérialisé en un fichier texte dont le contenu est le suivant :

Listing 1 – sérialisation de `[|[0;1;4];[];[0;3];[2];[1]|]`

```
{
"0":[0,1,4],
"1":[],
"2":[0,3],
"3":[2],
"4":[1]
}
```

Ainsi :

- Le contenu est entre deux accolades ;
- chaque numéro de sommet est présenté entre guillemets ;
- puis viennent deux points et une liste de voisins au formalisme `Python`. L'ordre est sans importance.
- on impose que chaque paire (clé,valeur) est séparée de la suivante par un passage à la ligne ;
- on considère qu'il n'y a pas d'espace ni de caractère de tabulation et que chaque sommet a sa liste de voisin.

1. Écrire la fonction `serialise (g:graph) (filename:string) : unit` qui sérialise le graphe g dans le fichier dont le nom est donné.
2. Écrire la fonction `deserialise (filename:string) : graph` qui retourne le graphe dont une représentation sérialisée est contenue dans le fichier appelé **filename**.

```
# let g = [|[0;1;4];[];[0;3]; [2];[1]|] in
serialise g "graphe"; deserialise "graphe";;
- : int list array = [|[0; 1; 4]; []; [0; 3]; [2]; [1]|]
```

Appendice

Lecture/écriture en C

Ouverture

```
FILE * fopen(const char * restrict filename,
             const char * restrict accessMode);
```

On indique un nom de fichier et un type d'accès (lecture/écriture et texte/binaire). On récupère un flot.

Vérification du succès de l'ouverture : le flot doit être différent de `NULL`.

Arguments :

filename : définit le nom du fichier à ouvrir. On peut spécifier un chemin absolu ou relatif. Pour plus de portabilité, préférer les chemins relatifs.

accessMode : mode d'ouverture du fichier.

Il faut faire figurer ces modes entre guillements (par exemple **"r"**).

En mode texte :

r : ouverture d'un fichier texte en lecture,

w : ouverture d'un fichier texte en écriture avec écrasement,

a : ouverture d'un fichier texte en écriture à la fin

Les modes binaires **rb,wb,ab** sont les pendants des précédents.

Fermeture

La fonction **fclose** ferme un flot et coupe la liaison avec le fichier qu'il pointe :

```
1  int fclose ( FILE * stream );
2
```

Si la fermeture se déroule

- sans erreur : la valeur **0** est retournée.
- avec erreur : la constante **EOF** de **stdlib.h** est retournée.
  Du fait de l'utilisation de **EOF**, il est utile d'importer **stdlib.h**.

C'est souvent au moment de la fermeture que le buffer est copié de la RAM vers le disque dur.

Écriture

```
1  int fprintf (FILE *stream, const char *format-string, argument-list);
```

Écrit une chaîne de caractères formatée sur un flot. La chaîne passée en argument n'est pas modifiée. Les arguments de **printf** sont d'abord le flot, puis la chaîne à formater et enfin ses arguments. Les descripteurs de format sont communs à **printf** ou **fprintf**. Par exemple, `%d` pour un entier, `%f` pour un flottant, `%s` pour une chaîne de caractères. Ainsi `fprintf(stream,"*");` écrit une astérisque dans le fichier pointé par le flot.

La valeur de retour est le nombre de caractères produit dans la chaîne. En cas d'erreur, la valeur `EOF` est renvoyée.

Lecture de chaînes de caractères formatées

```
int fscanf (FILE *stream, const char *format-string, argument-list);
```

stream : représente le flot sur lequel les extractions de valeurs devront être réalisées.

format : représente le format à utiliser pour décoder la chaîne de caractères.

Les descripteurs de format sont communs à **scanf** ou **fscanf**. Par exemple, `%d` pour un entier, `%f` pour un flottant, `%s` pour une chaîne de caractères. Ainsi, `fscanf(stream, "%d", &x);` affecte à l'entier `x` la valeur lue dans le flot.

Valeur de retour : le nombre de paramètres qui ont pu être extraits ou bien `EOF` (fin de fichier atteinte ou erreur). La fonction `fscanf` est utilisée en général quand on connaît la forme dans laquelle un fichier est écrit.

Lecture d'un caractère

La fonction `fgetc` lit le caractère à la position courante du flux considéré.

```
int fgetc( FILE * stream );
```

Une fois le caractère lu, la position de la tête de lecture associé au flux considéré est décalée sur le prochain caractère à lire.

Cette fonction retourne le caractère lu (casté en un `unsigned char`) ou bien `EOF` si la fin de fichier a été atteinte ou une erreur s'est produite.

Cette fonction est à préférer à `scanf` si on ne connaît pas exactement le format dans lequel les données du fichier exploré sont listées.

Lecture/écriture en OCAML

Les fonctions d'entrées sorties calculent une valeur (parfois de type `unit`) et modifient l'état des périphériques d'entrées-sorties :

- modification du buffer du clavier,
- affichage à l'écran,
- écriture dans un fichier
- ou modification du pointeur de lecture.

Deux types prédéfinis `in_channel` et `out_channel` décrivent les canaux de communication d'entrée et de sortie.

Ouverture en lecture Dans un fichier **essai.tex** du répertoire courant, écrivons trois lignes :

```
un
deux
trois et quatre
```

sans retour chariot après « quatre ».

La fonction `open_in` de type `string -> in_channel` permet d'ouvrir un fichier en lecture à partir de son chemin d'accès (ou son nom si le fichier à ouvrir est dans le répertoire courant). Elle ouvre un canal de communication (un flot) avec le fichier s'il existe et déclenche une exception `Sys_error` sinon (notamment si le fichier n'existe pas).

Notre fichier est ouvert avec l'instruction

```
let ic = open_in "essai.txt";; (*création d'un canal de com. vers essais.txt*)
```

On accède aux lignes du fichier grâce à la fonction

```
 # input_line;;
- : in_channel -> string = <fun>
```

Ci-dessous, nous lisons et affichons la première ligne du fichier :

```
let s = input_line ic in Printf.printf "%s\n" s;;
un
- : unit = ()
```

Lisons donc les deux autres lignes :

```
# let s = input_line ic in Printf.printf "%s\n" s;;
deux
- : unit = ()
# let s = input_line ic in Printf.printf "%s\n" s;;
trois et quatre
- : unit = ()
```

Si on essaye de lire une nouvelle ligne, on se doute bien qu'on va au devant d'un problème : une exception `End_of_file` sera soulevée. Ne tentons pas le diable et fermons le canal de communication `ic` :

```
# close_in ic;;
- : unit = ()
```

Le canal `ic` est fermé. Toute tentative de lire une ligne de **essai.txt** se solde par une exception `Sys_error "Bad file descriptor"`.

En résumé, pour lire et afficher toutes les lignes d'un fichier et le fermer proprement, il suffit de rentrer dans une boucle infinie qui affiche les lignes une à une et de récupérer l'exception **End_of_file** qui finira par arriver. On ferme alors le canal.

```
 let ic = open_in "test1.tex" in
let rec lire () =
  let s = input_line ic in Printf.printf "%s\n" s;
  lire();
in
try
  lire();
with End_of_file ->  close_in ic;;
```

Après compilation et exécution, le contenu du fichier **test1.tex** s'affiche (s'il existe).

Ouverture en écriture Quand on ouvre un canal de communication en écriture, le fichier correspondant est ouvert s'il existe ou créé s'il n'existe pas.

La fonction **open_out** permet d'ouvrir le fichier en mode écriture « avec écrasement ». La fonction **close_out** referme le canal

```
let oc = open_out "sortie.txt" in close_out oc;;
```

Nous avons juste ouvert puis refermé le fichier mais il a bien été créé :

```
$ ls sort*
sortie.txt
```

La fonction **output_string** permet d'écrire une chaîne de caractère dans le fichier :

```
 # let oc = open_out "sortie.txt" in
output_string oc "un";
output_string oc "deux";
output_string oc "trois";
close_out oc;;
        - : unit = ()
```

Attention, aucun saut de ligne n'a été inséré, il ne faut donc pas oublier les \n si on veut passer à la ligne :

```
$ cat sortie.txt
undeuxtrois
```

Plus proche de ce que nous connaissons est la fonction **Printf.fprintf** qui reconnaît les spécifieurs de format :

```
let oc = open_out "sortie.txt" in
Printf.fprintf oc "%f\n" 3.45;
Printf.fprintf oc "%d\n" 26;
Printf.fprintf oc "%s\n" "fini";
close_out oc;;
```

Vérification

```
$ cat sortie.txt
3.450000
26
fini
```