DS 3

Informatique MP2I

Julien Reichert

Toutes les questions de programmation sont à résoudre dans le langage précisé dans la question ou le titre de la section.

Questions de cours

Question de cours A1 : Définir la notion d'invariant de structure, exemple à l'appui (on n'écrira pas de programme).

Question de cours A2 : Expliquer la structure de données abstraite de tableau redimensionnable et en présenter l'implémentation usuelle (on ne réalisera pas l'implémentation).

Question de cours A3 : Présenter brièvement la notion de hachage.

Questions de cours en OCaml

Question de cours B1 : Donner la définition d'un type option, exemple à l'appui (ne pas écrire de long programme).

Question de cours B2 : Expliquer le problème de l'expression `Array.make 3 (Array.make 3 0)` et proposer une façon de procéder pour ne pas rencontrer ce problème.

Question de cours B3 : Rappeler dans quel contexte on utilise le mot-clé `mutable`.

Questions de cours en C

Question de cours C1 : Rappeler les 6 opérateurs de comparaison (qui produisent des booléens) et les 3 opérateurs booléens.

Question de cours C2 : On considère un pointeur de pointeurs, nommé `p`, qui a été alloué dynamiquement, de même que tous les pointeurs en question et dont le nombre est noté `n`. Écrire un morceau de code qui libère toute cette mémoire.

Question de cours C3 : Que se passe-t-il si un fichier contient `#include "foo.h";` et `#include "bar.h";` alors que `foo.c` contient `#include "bar.h";`, et comment éviter le problème sans effacer la moindre de ces directives ?

Exercices surtout théoriques dans un langage au choix

Exercice $\alpha 1$: **(Attention, cet exercice est long et il faut lire toute la consigne avant de commencer !)**
Écrire un fichier source bien encadré et dont le nom est précisé avant le cadre. Ce fichier doit contenir une fonction déterminant s'il existe un entier k tel que la somme des k premiers chiffres d'un nombre en premier argument soit égale à l'entier en second argument. Écrire à la suite du fichier une ligne de commande qui compile le fichier source puis une commande qui exécute le fichier exécutable obtenu avec deux arguments de ligne de commande : `1234` et `6`. Ces arguments de ligne de commande devront être traités par le fichier source écrit pour que le résultat de la fonction s'affiche dans la console en tant que `VRAI` puisque `6` est la somme des trois premiers chiffres de `1234`.

Exercice $\alpha 2$: On donne l'interface du type « maillon de liste chaînée de tableaux de soixante-quatre entiers » (en pratique utilisées comme des piles) ci-après. À l'aide de ceci, proposer une implémentation alternative des tableaux redimensionnables. L'ajout d'un élément nécessitera éventuellement de créer un maillon, et le retrait d'un élément fera libérer le dernier maillon s'il ne contient plus d'élément considéré comme étant dans le tableau. Le langage est au choix. Les fonctions attendues sont la création, l'ajout et le retrait d'un élément à la fin, l'accès et la modification à un indice précisé. La création du type n'est pas triviale !

Interface à utiliser sans programmer ce qui suit :
- Type des maillons de listes chaînées de tableaux d'entiers nommé `lcti`.
- Fonctions `creer_lcti`, `ajout_fin`, `retrait_fin`, `est_vide_lcti`, `contenu_maillon` et `maillon_suivant` (arguments et valeurs de retour au choix, et à préciser, sachant qu'ils dépendent du langage...).

Exercices en OCaml

Exercice $\beta 1$: Écrire une fonction prenant en argument un entier strictement positif et retournant le plus grand entier strictement positif utilisant les mêmes chiffres (on admet qu'il n'y a pas de débordement).

Exercice $\beta 2$: Écrire une fonction prenant en argument une liste de couples formés par une chaîne de caractères et un nombre et retournant une liste de couples ayant pour premiers éléments chaque chaîne (n'apparaissant plus qu'une fois) et pour seconds éléments la différence entre le maximum et le minimum des nombres ayant été associés à la chaîne en question dans la liste de départ. L'ordre des couples dans la liste retournée n'importe pas.

Par exemple, pour la liste `[("a", 4); ("c", 2); ("a", 0); ("b", 1); ("c", 9); ("a", 3)]`, la fonction retournera `[("a", 4); ("c", 7); ("b", 0)]` (à l'ordre près).

Exercice $\beta 3$: Écrire une fonction prenant en argument deux listes qui sont des permutations l'une de l'autre, à ceci près que la première liste a un élément en plus (éventuellement une occurrence en plus d'un de ses éléments), et qui retourne l'élément qui figure en plus.

Exercices en C

Exercice $\gamma 1$: Écrire une fonction qui prend en argument un tableau de chaînes de caractères et sa taille (supposée d'au moins un, inutile de le vérifier) et qui retourne l'indice au sein du tableau de la plus longue chaîne de caractères parmi celles dont les caractères sont toujours croissants selon l'ordre usuel, la première rencontrée en cas d'égalité de longueur. [1] Si aucune ne respecte la condition demandée, on renverra `-1`.

Exercice $\gamma 2$: Écrire une fonction réalisant un tri dénombrement d'un tableau d'entiers en échangeant uniquement des éléments du tableau. On se servira d'un tableau de pointeurs annexes, dont la taille sera égale au nombre de valeurs possibles dans le tableau, ce nombre étant un argument supplémentaire de la fonction. Les valeurs possibles seront forcément les premiers entiers naturels.

Problème 1 : Sérialisation

Nous allons ici réaliser des exemples sommaires de sérialisations, qui peuvent être assimilés au « pretty-print » de Python.

On considèrera en OCaml des listes d'entiers et un type enregistrement `type foo = { a : int ; b : float }`, et en C des tableaux d'entiers et une structure `struct foo { int a ; double b ; };`.

Question P1.1 : Écrire en OCaml une fonction de signature `serialise_liste_entiers : int list -> string` qui construit une chaîne de caractères obtenue à partir d'une liste d'entiers, de sorte que `serialise_liste_entiers [1; 2; 3]` donne `"[1; 2; 3]"` (les espaces sont fixées par la fonction et ne dépendent évidemment pas de la présentation de la liste en argument.

Question P1.2 : Question P1.1 : Écrire en C une fonction de même effet (en particulier la chaîne retournée sera la même) mais agissant sur des tableaux d'entiers, dont le prototype sera `char* serialise_tableau_entiers(int* tab, int taille)`.

Question P1.3 : Écrire dans chaque langage la fonction de désérialisation associée. On admettra qu'en C le tableau obtenu ne sera pas vide et que dans les deux langages la chaîne sera bien formatée, gestion des espaces incluse. On signale aussi qu'en C la taille de la réponse n'est pas forcément connue au moment où la fonction est appelée, et il serait agréable de ne pas forcer à la recalculer...

Question P1.4 : Écrire en OCaml une fonction de signature `serialise_foo : foo -> string`, de sorte que `serialise_foo { a = 3; b = 3.3 }` donne `"{a = 3 ; b = 3.3}"` (les espaces sont également fixées et l'ordre est imposé). Écrire également la fonction de désérialisation associée, mais en considérant que l'ordre peut être permuté dans l'énonciation des attributs au niveau de la chaîne à désérialiser et que le nombre d'espaces est totalement arbitraire. Ceci étant, la chaîne restera correcte à part ceci dans le sens où si on évaluait son contenu en OCaml cela décrirait bien un `foo`.

Question P1.5 : Même exercice en C, avec comme prototype `char* serialise_foo(struct foo bar)` sous les mêmes conditions à ceci près que le contenu de la chaîne de caractères respectera la syntaxe de C (toujours avec des espaces et un ordre arbitraire, mais avec la garantie que les deux champs sont initialisés).

1. Voilà comment j'aurais formulé l'exercice $\gamma 3$ du DS 2 si cela avait été la consigne telle que beaucoup l'ont comprise...

Problème 2 : Flux

On considère la notion de flux (attention aux homonymies), qui peut être assimilée à une suite logique de données (ici des entiers) infinie. Pour implémenter cette structure de données, une possibilité en OCaml est un type enregistrement d'attributs un peu particulier : `type flux = { tete : int ; reste : flux -> flux}`.

L'attribut `reste` est donc une fonction qui au flux dans son entièreté associe un autre flux, dont on pourra consulter le premier élément ou le reste (associé a priori à la même fonction).

On peut voir les flux comme la version récursive des générateurs de Python (hors-programme de toute filière, mais si utiles).

Pour une adaptation en C, on utilisera des listes chaînées finies correspondant aux premiers éléments d'un flux tel qu'il serait géré en OCaml. En pratique, même finies, on pourra imaginer que ces listes chaînées comprendront de nombreux éléments.

Deux exemples « simples » pour fixer les idées en OCaml : un flux dont tous les éléments valent zéro s'écrit par exemple `let zero = { tete = 0 ; reste = fun fl -> fl}`, et un flux contenant tous les entiers naturels s'écrit `let nn = { tete = 0 ; reste = fun fl -> { tete = fl.tete + 1 ; reste = fl.reste }}`.

Question P2.1 : Donner la syntaxe pour récupérer la valeur `1` dans le flux `nn`.

Question P2.2 : Créer un flux contenant une répétition infinie des entiers `1`, `2` puis `3`.

Question P2.3 : Écrire une fonction de signature `flux_to_list : flux -> int -> int list` récupérant les premiers éléments d'un flux (on admet que le nombre d'éléments à récupérer est positif).

On impose l'interface suivante pour les listes chaînées.

```
struct m_l_c_i { int valeur; struct m_l_c_i* suivant; };
typedef struct m_l_c_i milc;

struct l_c_i { milc* debut; };
typedef struct l_c_i flux;

flux creer_flux(); // Création d'un flux vide
bool est_vide_flux(flux fl); // Test de vacuité
milc* tete_flux(flux fl); // Premier maillon
bool est_dernier(milc* maillon); // Test qu'il n'y a pas de suivant
milc* suivant_flux(milc* maillon); // Maillon suivant
int acceder_maillon(milc* maillon); // Récupération de la valeur
int modifier_maillon(milc* maillon, int x); // Modification de la valeur
void retirer_flux(flux* fl, milc* m); // Retrait du maillon et mise à jour éventuelle du champ début
```

Question P2.4 : En se servant de fonctions de l'interface, écrire une fonction analogue à la fonction attendue dans la question P2.3 de prototype `int* flux_to_array(flux fl, int taille)`. On admet cette fois-ci que le nombre d'éléments à récupérer est strictement positif et que la liste chaînée en contient suffisamment.

Question P2.5 : Écrire une fonction de prototype `void retire(flux* fl)`, tel qu'un élément sur deux soit retiré, en commençant par retirer le premier.

Question P2.6 : Écrire une fonction de prototype `void fusionne(flux fl)`, tel que chaque élément du flux soit fusionné avec son successeur (les entiers sont additionnés) qui disparaît (on peut admettre que la taille du flux est paire). Ainsi, à partir des valeurs `1`, `2`, `3`, `4` etc. on obtient les valeurs `3`, `7`, `11` etc.

Question P2.7 : Écrire une fonction de prototype `void rassemble(flux fl)`, tel que chaque entier du flux soit remplacé par la somme de tous les entiers depuis le début du flux et jusqu'à lui-même inclus.

Question P2.8 : Que contient le flux correspondant à l'adaptation en C de `nn` après avoir subi `retire` puis `rassemble` ?

Annexe

On rappelle à toute fin utile des fonctions du module `Hashtbl` avec des informations sur leur spécification :

- `Hashtbl.create n` crée une table de hachage avec `n` places pour commencer, mais en adaptant si besoin (donc on devine `n` sans qu'il n'y ait de risque si l'estimation est mauvaise) ;
- `Hashtbl.add th cle valeur` ajoute une association à la table de hachage, en masquant une éventuelle clé déjà existante (l'autre valeur sera de nouveau accessible en cas de retrait de ce qui l'a masqué) ;
- `Hashtbl.find th cle` détermine la valeur associée à la clé dans la table de hachage, en déclenchant l'erreur `Not_found` si la clé est absente ;
- `Hashtbl.mem th cle` détermine si la clé est présente dans la table de hachage ;
- `Hashtbl.remove th cle` retire une occurrence de la clé dans la table de hachage s'il y en a une (sinon la fonction n'a pas d'effet) ;
- `Hashtbl.replace th cle valeur` remplace la valeur associée à la clé dans la table de hachage par une nouvelle valeur (une éventuelle valeur masquée n'est pas impactée) en ajoutant la clé si elle n'y était pas encore.
- `Hashtbl.find_opt th cle` agit comme la fonction `find`, mais retourne une option pour éviter de lever une exception si la clé est absente ;
- `Hashtbl.iter f th` appelle la fonction fournie, prenant des clés et des valeurs (dans cet ordre) en argument, à tous les éléments de la table de hachage, sans contrôle sur l'ordre, sachant que si des clés sont masquées par d'autres clés identiques, elles subiront aussi la fonction (et on sait que ce sera dans l'ordre inverse de leur apparition dans la table de hachage).

Pour la manipulation des chaînes en C, on propose les fonctions suivantes :

- `atoi(chaine)` renvoie l'entier représenté dans la chaîne en argument (ne pas l'utiliser si la chaîne ne correspond pas exactement à un entier), la fonction est `atof` pour renvoyer un flottant de type `double` ;
- `sprintf(chaine, "%d", n)` écrit dans la chaîne en premier argument l'entier en troisième argument, ce qui écrase la chaîne et suppose que sa taille soit suffisante, le format devient `"%f"` pour un flottant ;
- `strcat(chaine1, chaine2)` écrit à la suite de la chaîne en premier argument la chaîne en deuxième argument (même remarque sur la taille) ;
- `strcpy(chaine1, chaine2)` écrit au début de la chaîne en premier argument la chaîne en deuxième argument (idem).