Examen final de programmation impérative

ÉNSIIE, semestre 1

mercredi 8 janvier 2020

Durée : 1h45.
Tout document papier autorisé. Aucun appareil électronique autorisé.
Ce sujet comporte 3 exercices indépendants, qui peuvent être traités dans l'ordre voulu. Il contient 4 pages.
Le barème est donné à titre indicatif, il est susceptible d'être modifié. Le total est sur 20 points.
Il va de soi que toute réponse devra être justifiée, et que toute fonction devra être commentée (au minimum `require`, `assigns` et `ensures`).

Exercice 1 : Fonctions simples (6 points)

1. Proposer une procédure dont l'effet est d'incrémenter de 1 la valeur d'une variable de type `int` définies auparavant. (La fonction est définie *avant* `main`. À l'intérieur de `main`, la variable est définie et la fonction appelée. On n'écrira pas la fonction `main`.)
2. Proposer une procédure qui prend en paramètre un tableau `t` de `double` et deux entiers `i` et `j`, et qui échange le contenu des cases de `t` d'indices `i` et `j`.
   On fera bien attention en précisant les préconditions.
3. Proposer une fonction qui prend en paramètre une chaîne de caractère `s`, et qui retourne le nombre d'occurrence de la sous-chaîne `"ab"` dans `s`. Par exemple, pour `s = "cabaabb"` on retournera `2`.
4. Proposer une fonction qui prend en paramètre un tableau de `int` et sa taille, et qui retourne une copie inversée de ce tableau. Par exemple si le tableau contient 1, 2, 3, on retournera un tableau qui contient 3, 2, 1.

Exercice 2 : Arbres (10 points)

Les arbres sont une structure de données omniprésente en informatique. Un arbre est un nœud qui contient une valeur ainsi que des enfants qui sont eux-mêmes des arbres. On considère ici des arbres qui peuvent avoir un nombre quelconque d'enfants, y compris 0 auquel cas on parle de feuille.

On va donc stocker les enfants dans des tableaux alloués dynamiquement, et pour cela on va conserver la taille du tableau dans un enregistrement :

```
struct node_array {
  int nb_nodes;
  struct node *nodes;
};
```

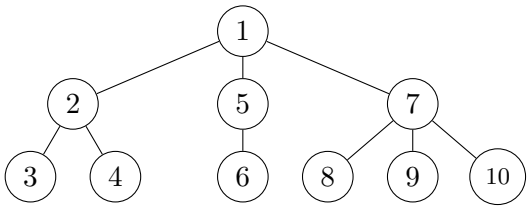
Un nœud est alors un enregistrement contenant la valeur et un tableau d'enfants :

```
struct node {
  int val;
  struct node_array children;
};
```

On peut définir les alias de type suivants pour les arbres et les forêts (tableaux d'arbres) :

```
typedef struct node tree;
typedef struct node_array forest;
```

1. Proposer une procédure `print_tree` qui prend en paramètre un arbre et qui l'affiche sur la sortie standard de la façon suivante : on affiche la valeur, puis si le nombre d'enfants est strictement positif, on affiche entre parenthèses les enfants séparés par des virgules.
   Par exemple, l'arbre

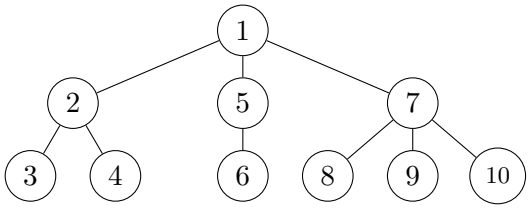
   sera affiché comme `1 (2 (3, 4), 5 (6), 7 (8, 9, 10))`.
2. Proposer un fonction `leaf` qui prend en paramètre un entier `e` et qui retourne un arbre dont la valeur est `e` et qui n'a pas d'enfant.
3. Proposer un fonction `binary_node` qui prend en paramètre un entier `e` et deux arbres `left` et `right` et qui retourne un arbre dont la valeur est `e` et dont les enfants sont `left` et `right`.
4. On considère la fonction suivante :

```
1 void f(int n) {
2   tree a;
3   tree b;
4   a = leaf(n);
5   b = binary_node(n-1, a, a);
6   b.children.nodes[1] = b;
7 }
```

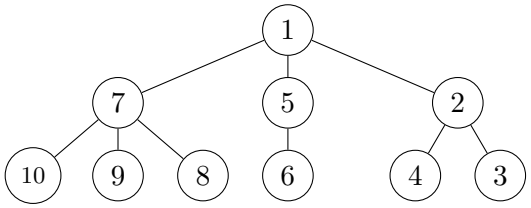
   Représenter l'état de la mémoire juste avant et juste après l'exécution de la ligne 6, lors d'un appel à `f(42)`.

Que peut-on remarquer ?

5. Proposer une fonction `inversed_copy` qui prend en paramètre une forêt et qui retourne une copie de cette forêt dans laquelle l'ordre des éléments a été inversé.
6. Proposer une procédure `mirror` qui prend en paramètre un arbre par référence et qui le modifie par effet pour en faire son image dans un miroir. Par exemple l'arbre



deviendra



On pourra utiliser un algorithme récursif. En cas d'utilisation de `inversed_copy`, on fera attention à bien libérer la mémoire inutilisée.

Exercice 3 : Étude de fonctions (4 points)

On prendra le soin de bien réfléchir avant de donner sa réponse.

1. On considère la fonction `test_all_equals` suivante :

```
int test_all_equals (int t[], int size) {
  int i = 0;
  while ((i < size - 1) && (t[i] = t[i+1]))
    i = i + 1;
  return (i >= size - 1);
}
```

Que fait l'appel `test_all_equals(t,5)` où `t` est un tableau de taille 5 contenant 0, 1, 1, 0, 1 ?

On considère maintenant la définition habituelle des listes chaînées, avec la fonction habituelle de construction par valeur `cons` :

```
list cons(elem x, list l) {
  list r;
  r = (list) malloc(sizeof(struct cell));
  r->val = x;
  r->next = l;
  return r;
}
```

2. Soit la fonction `length` suivante :

```
int length(list l) {
  int i = 0;
  while (l != NULL)
    i = i + 1;
    l = l->next;
  return i;
}
```

   Que fait l'appel `length(cons(1, cons(2, NULL)))` ?

3. Toujours sur les listes, on considère la fonction `remove_head` suivante :

```
void remove_head(list l) {
  list tmp;
  if (l != NULL) {
    tmp = l;
    l = l->next;
    free(tmp);
  }
}
```

   Que va faire le code suivant ?

```
list l = cons(1, cons(2, NULL));
remove_head(l);
printf("%d", l->val);
```