Corrigé de l'examen final de programmation impérative

ÉNSIIE, semestre 1

mercredi 19 janvier 2022

Remarques préliminaires

— Les éléments de type `char` sont **des entiers !** (sur 8 bits, donc compris en entre -128 et 127 ou entre 0 et 255 suivant qu'ils soient signés ou non.) On peut donc tout à fait appliquer les opérations arithmétiques dessus, sauf qu'on travaille modulo 256.

Ces entiers peuvent être interprétés grâce au code ASCII pour les voir comme des caractères, par exemple s'ils sont dans une chaîne de caractères (qui est, je le rappelle, un tableau de `char` qui se termine par 0).

On peut utiliser la notation entre apostrophes pour entrer des constantes de type `char`. Par exemple, la constante `'*'` est la même constante que `42` (ou que `0x2A`...).

Au passage, la fonction `atoi` de la bibliothèque standard prend une **chaîne de caractères** en paramètre et convertit l'entier écrit en base 10 dans la portion initiale de la chaîne de caractère en un entier `int`. Exemple : `atoi("42blabla") == 42`. Elle ne sert donc pas à convertir un `char` en `int` (conversion de toute façon inutile, puisque que `char` est un sous-ensemble de `int`).

— Dans les `requires`, il est inutile de rappeler les types.

Par ailleurs, écrire « l'entier `x` est valide » ne veut rien dire.

— Pour qu'ils puissent être utilisés dans les `if` et autres `while`, les tests en C doivent renvoyer 1 (ou en tout cas une valeur non nulle) quand ils réussissent, 0 sinon.

— Si on a uniquement besoin de parcourir une chaîne de caractère, il est inutile de commencer par calculer sa taille pour ensuite refaire une boucle `for` dessus. Il est plus efficace d'effectuer directement le traitement en faisant une boucle tant qu'on n'a pas atteint le caractère nul.

Exercice 1 : Fonctions simples (6 points)

1.
```
/*@ requires px is a valid address
    assigns *px
    ensures increment *px by c */
void incr(int *px, int c) {
  *px += c;
}
```

Exemple d'utilisation (pour votre culture, non demandé dans le sujet) :

```
int main() {
  int x = 0;
  incr(&x, 42);
  return 0;
}
```

2. On rappelle que **double** est le type des nombres à virgule flottante de double précision. (Ce ne sont donc pas des entiers.)

```
/*@ requires t is an array of size s > 0
    assigns nothing
    ensures return the mean value of t */
double avg(double *t, int s) {
  double r = t[0];
  int i;
  for (i = 1; i < s; i += 1)
    r += t[i];
  return r / s;
}
```

3. Il faut réutiliser les valeurs des cases précédentes pour ne pas avoir à recalculer les sommes et les produits à chaque tour de boucle.

```
/*@ requires n > 0
    assigns nothing
    ensures returns an array t of size n such that
    t[i].left is the sum of the integers from 1 to i
    t[i].right is the product of the integers from 1 to i */
struct couple *sumprod(int n) {
  struct couple *res = malloc(n * sizeof (struct couple));
  int i;
  res[0].left = 0; // pas de -> ici, res[0] n'est pas un pointeur !
  res[0].right = 1;
  for (i = 1; i < n; i += 1) {
    res[i].left = res[i-1].left + i;
    res[i].right = res[i-1].right * i;
  }
  return res;
}
```

Pour mieux faire, il faudrait tester que le `malloc` a bien fonctionné (c'est-à-dire qu'il restait de la mémoire disponible) et qu'il n'a donc pas renvoyé `NULL`.

4.
```
/*@ requires s is a nul-terminated string that
             contains only lower-case letters
```

```
    assigns all characters of s
    ensures encode s using Caesar cipher */
void Caesar(char *s) {
  int i;
  while (s[i]) {
    if (s[i] >= 'a' + 13)
      s[i] -= 13; // x + 13 - 26 == x - 13
    else
      s[i] += 13;
    i += 1;
  }
}
```

Remarque : il n'est pas utile de connaître les codes ASCII des caractères puisqu'on peut utiliser par exemple `'a'` au lieu de `97`.

5.
```
/*@ requires sub and str are nul-terminated strings
    assigns nothing
    ensures returns 1 if sub is a substring of str
                    0 otherwise */
int is_substring(char *sub, char *str) {
  int start_ss, offset;
  start_ss = 0; /* The index from which the presence
                   of the substring is tested */
  /* The loop ends because start_ss + offset will eventually reach
     the end position of str if the substring is not found earlier */
  while (1) {
    offset = 0;
    /* We advance in sub and str as long as they are not finished
       and their characters match. */
    while (sub[offset] &&
           str[start_ss + offset] &&
           sub[offset] == str[start_ss + offset])
      offset += 1;
    if (sub[offset] == '\0')
      /* all characters of sub are present in str starting at
         position start_ss */
      return 1;
    if (str[start_ss + offset] == '\0')
      /* sub is longer than the substring of str starting at
         start_ss, so it can no longer be found */
      return 0;
    start_ss += 1; // try the next starting position
  }
  return 0; // This statement is never reached
```

```
}
```

On ne peut pas intervertir l'ordre des deux **if** car si les deux chaînes se terminent en même temps, la première est bien incluse dans l'autre. (Le sous-mot est dans ce cas à la fin de l'autre.)

6. On sait que **sizeof (char)** vaut 1.

   La chaîne retournée a besoin de `j - i + 1` octets car il faut aussi stocker le `'\0'` final.

```
/*@ requires str is a nul-terminated string of length n such
      that n >= j >= i >= 0
    assigns nothing
    ensures returns a copy of the substring comprised between
      positions i (included) and j (excluded) */
char *copy_substring(char *str, int i, int j) {
  char *res = malloc (j - i + 1);
  int k;
  for (k = i; k < j; k += 1)
    res[k - i] = str[k];
  res[j - i] = '\0';
  return res;
}
```

Exercice 2 : Listes (7 points)

On rappelle la définition du type des listes :

```
typedef struct bucket *list;

struct bucket {
  int val;
  list next;
};
```

On rappelle qu'une liste est bien formée si c'est l'adresse `NULL` ou si c'est l'adresse valide d'un **struct bucket** dont le champ **next** est une liste bien formée.

1.
```
/*@ requires l is well formed
    assigns nothing
    ensures returns 1 if n is present in l,
                    0 otherwise */
int mem(int n, list l) {
  while (l != NULL) {
    if (l->val)
      return 1;
```

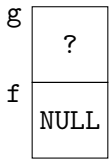
```
    l = l->next;
  }
  return 0;
}
```

La terminaison de cette fonction nécessite que la liste soit acyclique.

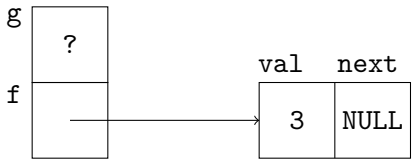
2.
```
/*@ requires pl is the address of a well-formed list
    assigns one of the buckets of *pl
    ensures add n at the end of *pl */
void add_last(list *pl, int n) {
  if (*pl == NULL)
    *pl = cons(n, NULL);
  else {
    list visit = *pl;
    while (visit->next != NULL)
      visit = visit->next;
    visit->next = cons(n, NULL);
  }
}
```

On pouvait aussi faire ses propres allocations plutôt que d'utiliser `cons` (qui fait elle-même une allocation).

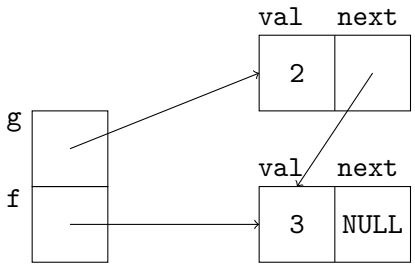
3. Après ligne 2 :



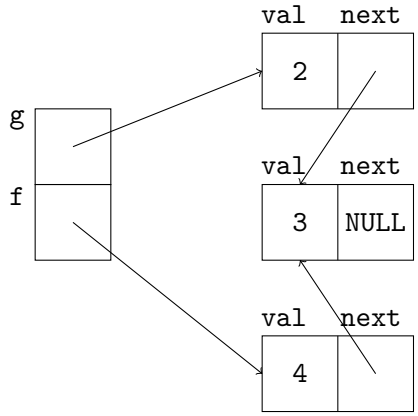
Après ligne 3 :



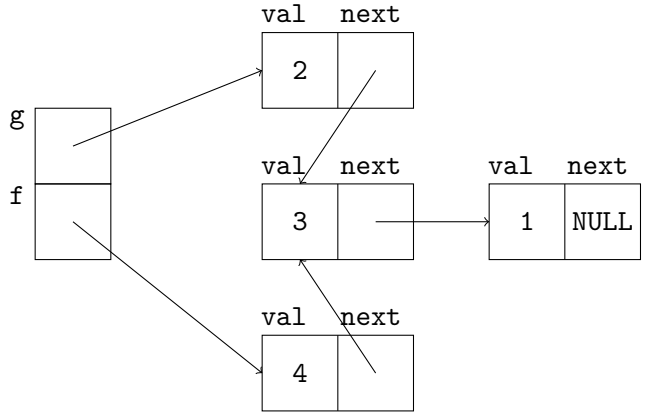
Après ligne 4 :



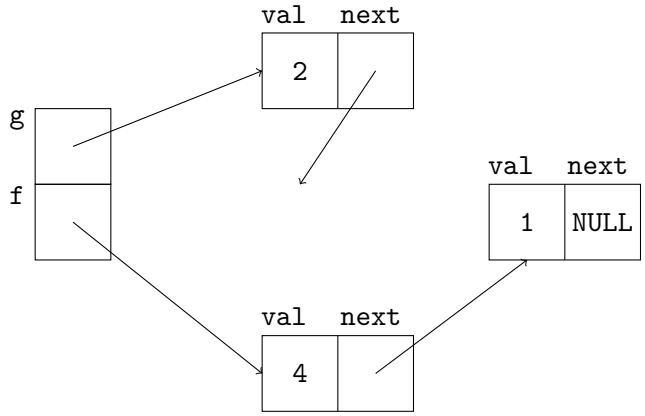
Après ligne 5 :



Après ligne 6 :



Après ligne 7 :



4. La valeur 2 est trouvée dans le premier maillon de `g`, on affiche donc `1` suivi d'un retour à la ligne sur la sortie standard.

   Le fait que le champ `next` du premier maillon de `g` contienne maintenant une adresse invalide n'est pas un problème puisque comme on a trouvé la valeur re-

cherchée, on n'y accède pas.

Exercice 3 : Jeu des 7 erreurs (7 points)

1. La division `1 / 2` va se faire sur les entiers, puisque `1` et `2` sont des entiers. Elle vaudra donc zéro et on renverra zéro dans tous les cas. On peut forcer un des deux à être un nombre à virgule flottante (par exemple en écrivant `1.`), ou on peut utiliser directement `0.5`.

```
/*@ requires m >= 0
    assigns nothing
    ensures returns the kinetic energy given
    mass m and velocity v */
double kinetic_energy(double m, double v) {
  return .5 * m * v * v;
}
```

   NB : le parenthésage était bien correct ; on calcule $\frac{1}{2}mv^2$.

2. Il faut allouer suffisamment de mémoire pour les champs de **`struct node`**. Or `tree` est le type des pointeurs vers **`struct node`** et est donc plus petit.
   Il fallait donc écrire

```
  tree res = malloc(sizeof (struct node));
```

3. Comme les **`char`** sont des entiers sur 8 bits, il y a un fort risque de débordement lors de la somme. Par exemple, si on a un tableau qui contient trois fois la valeur 100, la somme vaudra finalement 44 et la valeur renvoyée sera 14 et non 100.
   Par pallier ce problème, on peut stocker la somme dans un **`int`**. On renverra par contre bien un **`char`**. (La valeur sera prise modulo 2^8, mais de toute façon elle sera plus petite que la borne.)

```
/*@ requires t has size s > 0
    assigns nothing
    ensures returns the mean value of the
    elements of t       */
char mean(char t[], int s) {
  int i;
  int r = 0;
  for (i = 0; i < s; i += 1)
    r += t[i];
  return r / s;
}
```

   Remarque : il faudrait aussi corriger la spécification pour dire que la taille doit être strictement positive.

4. On aura une erreur de segmentation lors du premier `visit->next` si la liste passée en paramètre est vide. Il faut donc rajouter un traitement pour ce cas.

```
/*@ requires l1 and l2 are well-formed lists
    assigns potentially one of the buckets of l1
    ensures returns the concatenation of l1 and l2 */
list concat(list l1, list l2) {
  if (l1 == NULL) return l2;
  list visit = l1;
  while (visit->next != NULL)
    visit = visit->next;
  visit->next = l2;
  return l1;
}
```

   Remarque : la liste `l1` sera modifiée même si elle n'est pas passée par référence. En particulier, si elle n'est pas vide, elle sera égale, après la fonction, à la concaténation des listes initiales.

5. Le code fourni prend `r` par valeur et non par référence. On travaille donc sur une copie et la valeur initiale n'est pas modifiée.

```
/*@ requires r->den != 0
    assigns r
    ensures inverse r by effect */
void inverse(struct rat *r) {
  int tmp = r->num;
  r->num = r->den;
  r->den = tmp;
}
```

   Rappel : `r->num` est du sucre syntaxique pour `(*r).num`.

6. Le mot-clef **`sizeof`**, quand il est utilisé avec une variable, renvoie la taille du type de la variable. Ici, `t` est un tableau passé en argument, c'est donc une adresse (celle de la première case du tableau). Sa taille est celle d'un pointeur vers un **`int`**, c'est donc une constante et elle n'a par conséquent a priori aucun lien avec le nombre d'éléments dans le tableau.

   Il n'est pas possible en C de récupérer la taille d'un tableau passé en argument d'une fonction, il faut donc passer un argument supplémentaire à la fonction.

```
/*@ requires t is a valid array of size s >= 0
    assigns nothing
    ensures print the content of t
    on the standard output   */
void print_tab(int t[], int s) {
  int i;
  for (i = 0; i < sizeof t / sizeof (int); i += 1)
```

```
    printf("%i␣", t[i]);
}
```

Remarque : **sizeof t / sizeof (int)** ne fonctionne pas non plus ici. Cela ne fonctionne quand dans le cas d'un tableau déclaré localement avec sa taille, puisque dans ce cas **sizeof** donne la taille prise en mémoire par le tableau.

7. Dans la condition du deuxième **if** on a un simple **=** (donc une affectation), il y aura donc une erreur de syntaxe car ce qui est à gauche n'est pas quelque chose dans lequel on peut affecter une valeur. Il faudrait écrire :

```
if (m % 2 == 0)
```