Correction de l'examen final de programmation impérative

ÉNSIIE, semestre 1

mercredi 18 janvier 2023

Remarques préliminaires

Dans les `@requires`, il est inutile de rappeler les informations de type, qui sont déjà présente dans le prototype de la fonction. Par exemple, il ne sert à rien d'écrire `@requires un entier a et un entier b`. Par contre, il faut indiquer les propriétés qui doivent être vérifiées par les paramètres, en particulier les liens entre eux s'ils en ont. Par exemple `@requires t is an array of size at least s, and s > 0`.

Les accolades ne sont pas obligatoires après les **`if`**, les **`while`** et les **`for`** quand il n'y a qu'une seule instruction. Elles doivent être présentes pour grouper plusieurs instructions. Si les mettre quand même quand il n'y a qu'une instruction n'est pas faux en soi, cela nuit à l'esthétique du code et donc sa lisibilité, comme par exemple si vous écriviez `1 + (2 * 3)` où les parenthèses ne sont pas fausses mais sont inutiles.

Quand on demande un test, il faut écrire une fonction qui retourne `1` si le test est vrai, et `0` sinon. En effet, il faut utiliser ces valeurs si jamais le test est utilisé comme condition dans un **`if`** ou un **`while`**.

Exercice 1 : Fonctions simples (6 points)

1.
```
/*@requires pa and pb valid addresses
  @assigns *pa and *pb
  @ensures swap the values of *pa and *pb */
void swap(double *pa, double *pb) {
  double tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```

2.
```
/*@requires t has size s and s > 0
  @assigns nothing
  @ensures returns the minimim value in t */
int min_array(int t[], int s) {
  int min = t[0]; // exists since s > 0
  for (int i = 1; i < s; i += 1)
    // start from 1 since t[0] is already min
```

```
    if (t[i] < min)
      min = t[i];
  return min;
}
```

3.
```
/*@requires nothing
  @assigns nothing
  @ensures returns an array of size 26
           containing the ASCII codes of 'A' to 'Z' */
char *ascii() {
  char *r = malloc(26);
  for (int i = 'A'; i <= 'Z'; i += 1)
    r[i - 'A'] = i;
  return r;
}
```

4.
```
/*@requires s is a null-terminated string
  @assigns nothing
  @ensures test if s contains the same character
           two times successively */
int same(char *s) {
  int i = 0;
  while (s[i] != '\0') {
    if (s[i] == s[i+1]) // at worst, t[i+1] is '\0'
      return 1;
    i += 1;
  }
  return 0;
}
```

5.
```
/*@requires s1 and s2 are null-terminated strings
  @assigns nothing
  @ensures test if s1 and s2 contains a common character */
int inter(char *s1, char *s2) {
  int r1 = 0;
  while (s1[r1] != '\0') {
    int r2 = 0;
    while (s2[r2] != '\0') {
      if (s1[r1] == s2[r2])
        return 1;
      r2 += 1;
    }
    r1 += 1;
  }
```

```
  return 0;
}
```

6.
```
/*@requires l is a well-formed, acyclic,
             and non-empty list
  @assigns nothing
  @ensures returns the mean of the values in l */
double mean(list l) {
  double sum = 0;
  int c = 0;
  while (l != NULL) {
    sum += l->val;
    c += 1;
    l = l->next;
  }
  return sum / c; // c != 0 since l is non-empty
}
```

Exercice 2 : Cordes linéaires (9 points)

Comme indiqué dans l'énoncé, une corde linéaire sera bien formée si elle est non-vide et que la taille des tableaux est strictement positive dans chaque maillon.

1.
```
typedef struct cell *lcord;

struct cell {
  int *array;
  int size;
  lcord next;
};
```

2.
```
/*@requires l is a well-formed linear cord
  @assigns nothing
  @ensures print the content of l */
void print_lcord(lcord l) {
  while (l != NULL) {
    for (int i = 0; i < l->size; i += 1)
      printf("%d␣", l->array[i]);
    printf(";␣");
    l = l->next;
  }
}
```

3.
```
/*@requires t has size s
  @assigns nothing
  @ensures returns a newly allocated linear cord
           with a single cell containing a copy of t */
lcord create_from_array(int t[], int s) {
  lcord l = malloc(sizeof (struct cell));
  l->next = NULL;
  l->array = malloc(s * sizeof (int));
  l->size = s;
  for (int i = 0; i < s; i += 1)
    l->array[i] = t[i];
  return l;
}
```

4.
```
/*@requires l is a well-formed linear cord
           and 0 < i < l->size - 1
  @assigns *l
  @ensures splits the first cell of l at position i */
void split_cell(lcord l, int i) {
  lcord n = create_from_array(&l->array[i], l->size - i);
  // it is also possible to build n without using create_from_array
  l->size = i;
  n->next = l->next;
  l->next = n;
}
```

5.
```
/*@requires p>0, out and in are well-formed linear cords
           there are at least p + 1 elements in out
  @assigns potentially all cells of out and in
  @ensures insert in in out at position p */
void insert(lcord out, lcord in, int p) {
  while (out->size < p) {
    if (out->next == NULL) {
      errno = EDOM;
      return;
    }
    p -= out->size;
    out = out->next;
  }
  if (out->size > p)
    split_cell(out, p);
  lcord next = out->next;
  out->next = in;
  while (in->next != NULL)
```

```
    in = in->next;
  in->next = next;
}
```

6. ```
   /*@requires l is a well-formed linear cord
     @assigns *l and l->next
     @ensures merge the two first cells of l */
   void merge_two_first(lcord l) {
     if (l == NULL || l->next == NULL)
       return;
     int *new = malloc((l->size + l->next->size) * sizeof (int));
     for (int i = 0; i < l->size; i += 1)
       new[i] = l->array[i];
     for (int i = 0; i < l->next->size; i += 1)
       new[i + l->size] = l->next->array[i];
     free(l->next->array);
     free(l->array);
     l->array = new;
     l->size += l->next->size;
     lcord tmp = l->next;
     l->next = l->next->next;
     free(tmp);
   }
   ```

7. ```
   /*@requires l is a well-formed linear cord (hence non-empty)
     @assigns all cells of l
     @ensures merge all cells of l into one */
   void flatten(lcord l) {
     while (l->next != NULL)
       merge_two_first(l);
   }
   ```

8. ```
   /*@requires l is a well-formed linear cord
     @assigns all cells of l
     @ensures free all the memory allocated in l */
   void free_lcord(lcord l) {
     while (l != NULL) {
       lcord next = l->next;
       free(l->array);
       free(l);
       l = next;
     }
   }
   ```
   ```

Exercice 3 : Jeu des 5 erreurs (5 points)

1. La variable est passée par valeur, rien n'est donc modifié après l'exécution de la procédure. Correction possible :

```
/*@requires x valid address
  @assigns nothing
  @ensures increments x by 1 */
void incr(int *x) {
  *x += 1;
}
```

2. Lors de l'allocation de `r`, il faut allouer suffisamment d'espace pour contenir les adresses de chacune des lignes de la matrice. Il faut donc allouer `n` fois la taille d'un `int*` et non d'un `int`. Correction possible :

```
/*@requires n >= 0 and m >= 0
  @assigns nothing
  @ensures return a newly allocated matrix of dimension n * m */
int **create_matrix(int n, int m) {
  int **r = malloc(n * sizeof (int*));
  for (int i = 0; i < n; i += 1)
    r[i] = malloc(m * sizeof (int));
  return r;
}
```

3. Dans la deuxième boucle `for`, il faut incrémenter `j` et non `i`. Correction possible :

```
/*@requires t has dimension n * m
  @assigns nothing
  @ensures prints the content of t on the standard output */
void print_matrix(int **t, int n, int m) {
  for (int i = 0; i < n; i += 1) {
    for (int j = 0; j < m; j += 1)
      printf("%4d␣", t[i][j]);
    printf("\n");
  }
}
```

4. Comme `1` et `i` sont des entiers, la division `1 / i` est la division entière et vaut donc `0` pour `i > 1`. Il suffit que l'un des deux soit un flottant pour que ce soit la division flottante qui s'applique. Correction possible :

```
/*@requires n > 0
  @assigns nothing
  @ensures return the sum for i from 1 to n of the inverse of i */
double serie(int n) {
```

```
  double s = 0;
  for (int i = 1; i <= n; i += 1)
    s += 1. / i;
  return s;
}
```

NB : il n'y avait pas d'erreur de type ; l'expression `1 / i` est de type **`int`**, mais elle est convertie implicitement en **`double`** avant d'être affectée dans `s`.

5. Le `;` à la fin de la ligne du **`if`** correspond à l'instruction vide (qui ne fait rien). Par conséquent, le **`return;`** à la ligne suivante n'est pas attaché au **`if`** et il s'exécute dans tous les cas. On ne libère donc rien.

```
/*@requires nothing
  @assigns all cells of l
  @ensures free all cells of l */
void free_list(list l) {
  if (l == NULL)
    return;
  list next = l->next;
  free(l);
  free_list(next);
}
```

NB : il est tout à fait correct d'avoir des **`return;`** dans des procédures (type de retour **`void`**) ; ils ne prennent pas d'expression à retourner, mais ils servent à terminer la procédure pour revenir là où elle a été appelée.