Cours d'introduction à l'informatique
Examen de janvier 2020
Énoncés et solutions

Énoncés

1. (a) Écrire une fonction C prenant en argument une chaîne de caractères, et retournant le nombre de chiffres (c'est-à-dire, de caractères compris entre '0' et '9') que cette chaîne contient.

   (b) Par la méthode des invariants, démontrer que la valeur retournée par cette fonction est correcte.

2. En langage C :

   (a) Que cela signifie-t-il lorsqu'on dit qu'un opérateur s'évalue en circuit court ? Donner un exemple d'expression pour laquelle ce mécanisme est utilisé.

   (b) Qu'affiche l'exécution du fragment de code suivant ? (Justifier votre réponse.)

```
char s[] = "42", *t = "1", *u;

u = s;
*++u -= (*t) + 1;

printf("[%s]\n", s);
```

3. (a) Décrire, le plus simplement possible, l'opération effectuée par la fonction C suivante.

```
long f(int *t)
{
  if (!t[0])
    return 1;

  return t[0] * f(t + 1);
}
```

   (b) Quelle est la complexité en temps de cette fonction ?

(c) Écrire une fonction C réalisant exactement la même opération, mais sans effectuer d'appel récursif.

4. (a) Écrire un fragment de code C définissant une liste simplement liée d'entiers, c'est-à-dire une structure de données composée d'un ensemble ordonné d'éléments, tels que chacun mémorise une valeur entière et contient un pointeur vers son successeur.

(b) Écrire une fonction C allouant une liste liée vide.

(c) Écrire une fonction C capable d'insérer un entier donné en tête d'une liste liée donnée.

(d) Écrire une fonction C capable de calculer la somme des valeurs entières contenues dans une liste liée donnée.

Note : Pour résoudre ces problèmes, vous êtes libre de définir des structures de données et des fonctions auxiliaires.

Exemples de solutions

1. (a) Il suffit de parcourir la chaîne de caractères jusqu'à son symbole de terminaison, en comptant dans une variable `n` le nombre de chiffres lus. La fonction C suivante réalise cette opération :

```
unsigned nb_digits(char *s)
{
  unsigned n;

  for (n = 0; *s; s++)
    if (*s >= '0' && *s <= '9')
      n++;

  return n;
}
```

(b) On souhaite démontrer que le triplet

$$\{\texttt{s} = s_0\}$$
```
for (n = 0; *s; s++)
  if (*s >= '0' && *s <= '9')
    n++;
```
$$\{\texttt{n} = \text{nombre de chiffres dans la chaîne } s_0\}$$

est valide, où s_0 représente la valeur initiale de `s`. En développant la boucle `for`, cela revient à démontrer

$$\{\texttt{s} = s_0,\ \texttt{n} = 0\}$$

```
while (*s)
  {
    if (*s >= '0' && *s <= '9')
      n++;
    s++;
  }
```

$$\{\texttt{n} = \text{nombre de chiffres dans la chaîne } s_0\}.$$

Cherchons un invariant de boucle I. Celui-ci doit satisfaire les trois propriétés suivantes :

— I doit être impliqué par la précondition ($\texttt{s} = s_0$ et $\texttt{n} = 0$).

— Si I est vrai avant une itération de la boucle `while`, alors I doit également être vrai après cette itération.

— Après la dernière itération de la boucle, I doit impliquer la postcondition ($\texttt{n}$ = nombre de chiffres dans la chaîne s_0).

Ces trois propriétés sont vraies pour l'invariant

$$\begin{aligned} I : \exists i \geq 0\, (&\texttt{s} = s_0 + i \text{ et } \forall j \in [0, i-1]\, s_0[j] \neq 0 \text{ et} \\ &\texttt{n} = \text{nombre de chiffres dans la chaîne } s_0[0]s_0[1]\ldots s_0[i-1]). \end{aligned}$$

En effet :

— Initialement, on a

$$\begin{aligned} &(\texttt{s} = s_0 \text{ et } \texttt{n} = 0) \\ &\quad \Longrightarrow (\texttt{s} = s_0 + 0 \text{ et } \texttt{n} = \text{nombre de chiffres dans la chaîne vide}). \end{aligned}$$

— D'une itération à l'autre de la boucle, on a le triplet

$$\{I,\ \texttt{*s} \neq 0\}$$

```
if (*s >= '0' && *s <= '9')
  n++;
s++;
```

$$\{I\}.$$

Montrons que ce triplet est valide. Notons (respectivement) $\texttt{s}'$ et $\texttt{n}'$ les valeurs de `s` et `n` après l'exécution du fragment de code. On a

$$\texttt{s}' = \texttt{s} + 1,$$

donc si $\texttt{s} = s_0 + i$, alors $\texttt{s}' = s_0 + i'$ avec $i' = i + 1$.

Il y a deux cas à considérer :

— Si `*s` $\in$ [`'0'`, `'9'`] : Alors $\texttt{n}' = \texttt{n}+1$, et la chaîne $s_0[0]s_0[1]\ldots s_0[i = i'-1]$ contient effectivement un chiffre de plus (correspondant à `*s` $= s_0[i]$) que la chaîne $s_0[0]s_0[1]\ldots s_0[i-1]$.

— Si `*s` $\notin$ [`'0'`, `'9'`] : Alors $\texttt{n}' = \texttt{n}$, et les chaînes $s_0[0]s_0[1]\ldots s_0[i'-1]$ et $s_0[0]s_0[1]\ldots s_0[i-1]$ contiennent le même nombre de chiffres.

Enfin, on a `*s` $= s_0[i] \neq 0$, donc $\forall j \in [0, i'-1]\, s_0[j] \neq 0$, car par hypothèse $\forall j \in [0, i-1]\, s_0[j] \neq 0$.

— En sortie de boucle, on a

$$(I \text{ et } \texttt{*s} = 0),$$

qui implique bien que `n` est égal au nombre de chiffres contenus dans la chaîne s_0.

2. (a) Cela signifie que, lors de l'évaluation d'une expression où cet opérateur est appliqué à des opérandes, l'évaluation de ces opérandes cesse dès que la valeur finale de l'expression peut être déterminée.

Par exemple, l'évaluation de l'expression `f() && g()` n'effectuera pas l'appel `g()` si la valeur retournée par `f()` est nulle.

(b) — Après l'exécution de `u = s`, la variable `u` contient un pointeur vers le premier caractère de la chaîne `"42"` pointée par `s`.

— L'évaluation de `(*t) + 1` retourne le code du caractère pointé par `t` (c'est-à-dire `'1'`) augmenté de 1, c'est-à-dire le code du caractère `'2'`.

— `++u` incrémente `u` de 1, ce qui fait donc pointer `u` vers le deuxième caractère de la chaîne `"42"` pointée par `s`.

— Ensuite, la valeur pointée par `u`, c'est-à-dire le deuxième caractère de la chaîne pointée par `s`, est diminuée du résultat de l'évaluation de `(*t) + 1`, c'est-à-dire le code du caractère `'2'`. Cet opération a donc pour effet de remplacer le deuxième caractère de la chaîne pointée par `s` par un octet nul, terminant cette chaîne de caractères.

— En résumé, à l'issue de ces opérations, la chaîne de caractères pointée par `s` est désormais égale à `"4"`. La dernière instruction affichera donc "`[4]`" (suivi par un retour à la ligne).

3. (a) Cette fonction retourne le produit de tous les entiers contenus dans le tableau qui lui est passé en argument, jusqu'à la première valeur nulle (non incluse).

   (b) Appelons N le nombre d'entiers contenus dans un tableau `v`, jusqu'à la première valeur nulle (non incluse). En d'autres termes, soit N le plus petit entier positif ou nul tel que $\texttt{v}[N] = 0$.

   L'évaluation de `f(v)` provoquera $N + 1$ appels à la fonction `f`, chacun de ces appels effectuant un nombre borné d'opérations. La complexité en temps de la fonction `f` vaut donc $O(N + 1) = O(N)$.

   (c) Voici une solution possible :

```
long f(int *t)
{
  long r = 1;

  while (*t)
    r *= *t++;

  return r;
}
```

4. (a)
```
typedef struct _element
{
  int              valeur;
  struct _element *suivant;
} element;

typedef struct
{
  element *premier;
} liste_liee;
```

   (b)
```
liste_liee *creer_liste_vide()
{
  liste_liee *l;

  l = malloc(sizeof(liste_liee));
  if (!l)
    return NULL;
```

```
  l -> premier = NULL;

  return l;
}
```

(c)
```
void inserer_dans_liste(liste_liee *l, int v)
{
  element *e;

  e = malloc(sizeof(element));
  if (!e)
    return;

  e -> valeur = v;
  e -> suivant = l -> premier;

  l -> premier = e;
}
```

(d)
```
int somme_liste(liste_liee *l)
{
  element *e;
  int      s;

  for (s = 0, e = l -> premier; e; e = e -> suivant)
    s += e -> valeur;

  return s;
}
```