Listes : fonctions avancées

Le but de cette séance de Travaux Pratiques est d'implémenter des fonctions avancées sur les listes.

Vous trouverez sur Moodle une archive contenant un certain nombre de fichiers pour vous aider à démarrer. Après l'avoir téléchargée :

```
$ tar -xf sda1-tp6.tar.gz
$ cd sda1-tp6
```

Les fichiers sont répartis sur plusieurs répertoires correspondant aux différents exercices décrits dans la suite de ce document. Le fichier `Makefile` à la racine permet de compiler l'ensemble des exercices. Des fichiers `Makefile` sont également disponibles dans chacun des répertoires.

```
$ make
$ ./test_liste
```

Une implémentation de base des listes est fournie afin que vous puissiez vous concentrer sur les fonctions qui vous sont demandées dans les exercices ci-dessous. Néanmoins, il sera peut-être nécessaire d'observer attentivement le comportement de l'implémentation fournie.

Rappel : si nécessaire (par exemple pour vérifier l'absence de fuites mémoire) il est possible de customiser les options de compilation :

```
$ make clean
$ make CFLAGS='-Og -ggdb -fsanitize=address' LDFLAGS='-fsanitize=address'
```

1 `liste_range()`

Commencez par implémenter la fonction `liste_range()` :

```
liste* liste_range(S debut, S fin, S pas);
```

Cette fonction construit une liste d'éléments de l'intervalle $[debut, fin[$ espacés de `pas`. Par exemple :

— `liste_range(0, 10, 1)` = `[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]`
— `liste_range(10, 0, -1)` = `[ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]`
— `liste_range(0, 20, 2)` = `[ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 ]`

2 Implémentation des opérations avancées

Implémentez maintenant les fonctions suivantes :

```
liste* liste_reverse(const liste* l);
liste* liste_map(const liste* l, S (*f)(S));
liste* liste_filter(const liste* l, bool (*f)(S));
S liste_reduce(const liste* l, S (*f)(S, S));
void liste_fold_left(const liste* l, void (*f)(void*, S), void* resultat);
void liste_fold_right(const liste* l, void (*f)(S, void*), void* resultat);
```

Voici quelques indications sur ces fonctions :

— `liste_reverse()` inverse l'ordre des éléments de la liste.
— `liste_map()` consiste à appliquer la fonction `f` fournie sur chaque élément. En d'autres termes :

$$map([x_1, x_2, \ldots, x_n], f) = [f(x_1), f(x_2), \ldots, f(x_n)]$$

— `liste_filter()` ne conserve que les éléments pour lesquels `f` est vraie. C'est-à-dire :

$$filter([x_1, x_2, \ldots, x_n]) = [x_i \mid 1 <= i <= n \; et \; f(x_i) = true]$$

— `liste_reduce()` correspond à la réduction d'une liste :

$$reduce([x_1, x_2, \ldots, x_n], f) = f(\ldots f(x_1, x_2) \ldots, x_n)$$

— `liste_fold_left()` correspond à :

$$fold\_left([x_1, x_2, \ldots, x_n], f, r) = f(\ldots f(f(r, x_1), x_2) \ldots, x_n)$$

— `liste_fold_right()` correspond à :

$$fold\_right([x_1, x_2, \ldots, x_n], f, r) = f(x1, f(x_2, \ldots f(x_{n-1}, f(x_n, r)) \ldots))$$

Les fonctions qui retournent une liste doivent retourner une liste sans modifier la liste d'entrée.

3 Application des opérations avancées sur les listes

Implémentez les fonctions suivantes en utilisant les fonctions de l'exercice précédent :

```
S liste_somme(const liste* l);
S liste_produit(const liste* l);
S liste_minimum(const liste* l);
S liste_maximum(const liste* l);
liste* liste_carre(const liste* l);
bool liste_any(const liste* l, bool (*f)(S));
bool liste_all(const liste* l, bool (*f)(S));
```

telles que :

— `liste_somme()` retourne la somme des éléments d'une liste.
— `liste_produit()` retourne le produit des éléments d'une liste.
— `liste_minimum()` retourne le minimum des éléments d'une liste.
— `liste_maximum()` retourne le maximum des éléments d'une liste.
— `liste_carre()` retourne une liste dont les éléments sont les carrés des éléments de la liste d'entrée.
— `liste_any()` retourne `true` s'il existe un élément dans la liste pour lequel `f` retourne `true`.
— `liste_all()` retourne `true` si pour tous les éléments de la liste `f` retourne `true`.