

Listes et fonctionnelles

1. **map**. Rappelons les définitions suivantes vue en cours :

```
def doublist(l: list[int]) -> list[int]:
  match l:
    case [h, *t]: return [2 * h, *doublist(t)]
    case _: return []

def inclist(l: list[int]) -> list[int]:
  match l:
    case [h, *t]: return [h + 1, *inclist(t)]
    case _: return []
```

La fonction `doublist` double chaque élément de la liste, alors que `inclist` incrémente les éléments. Par exemple :

```
doublist([1, 2, 3, 4]) == [2, 4, 6, 8]
inclist([1, 2, 3, 4]) == [2, 3, 4, 5]
```

Ecrire une fonction `map[A, B](f: Callable[[A], B], l: list[A]) -> list[B]` qui prend en paramètre la fonction `f` à appliquer à chaque élément de la liste; et qui généralise ainsi `doublist` et `inclist` :

```
def doublist(l: list[int]) -> list[int]:
  return map(lambda x: 2 * x, l)

def inclist(l: list[int]) -> list[int]:
  return map(lambda x: x + 1, l)
```

Autrement dit, `map` doit vérifier l'équation suivante :

```
map(f, [x1, x2, ..., xn]) == [f(x1), f(x2), ..., f(xn)]
```

2. **filter**. La fonction `filter[A](f: Callable[[A], bool], l: list[A]) -> list[A]` prend en paramètre un prédicat (i.e. une fonction booléenne) et extrait d'une liste tous les éléments qui vérifient ce prédicat. Par exemple, si l'on définit :

```
def even(n: int) -> bool:
  return (n % 2 == 0)
```

En évaluant l'expression `filter(even, [1, 2, 3, 4, 5, 6])` on doit obtenir la liste `[2, 4, 6]`. Ecrire la fonction `filter`. Peut-on la définir facilement à partir de `map` ? Que vaut l'expression `map(even, [1, 2, 3, 4, 5, 6])` ?

3. **fold**. Rappelons les définitions suivantes vues en cours :

```
def product(l: list[int]) -> int:
  match l:
    case [h, *t]: return h * product(t)
    case _: return 1

def concat[A](l: list[list[A]]) -> list[A]:
  match l:
```

```

    case [h, *t]: return h + concat(t)
    case _: return []

```

La fonction `sum` calcule la somme des éléments d'une liste, alors que `concat` concatène une liste de listes. Par exemple :

```

product([1, 2, 3, 4]) == 24
concat([[1, 2], [3, 4], [5, 6, 7]]) == [1, 2, 3, 4, 5, 6, 7]

```

Ecrire une fonction :

```

foldr[A, B](f: Callable[[B, A], A], l: list[B], init: A) -> A

```

qui prend en paramètre la fonction `f` à appliquer à chaque élément de la liste; et qui généralise ainsi `product` et `concat`.

```

def product(l: list[int]) -> int:
    return foldr((lambda x, y: x * y), l, 1)

def concat[A](l: list[list[A]]) -> list[A]:
    return foldr((lambda x, y: x + y), l, [])

```

Autrement dit, `foldr` doit vérifier l'équation suivante :

```

foldr(f, [x1, x2, ..., xn], init) == f(x1, f(x2, ..., f(xn, init)...))

```

De la même manière, définir la fonction `foldl` avec le même prototype :

```

foldl[A, B](g: Callable[[A, B], A], l: list[B], init: A) -> A

```

qui doit vérifier l'équation suivante :

```

foldl(g, [x1, x2, ..., xn], init) == g(...g(g(init, x1), x2), ..., xn)

```

On définit maintenant :

```

def product2(l: list[int]) -> int:
    return foldl((lambda x, y: x * y), l, 1)

def concat2[A](l: list[list[A]]) -> list[A]:
    return foldl((lambda x, y: x + y), l, [])

```

Que valent les expressions suivantes ?

```

product2([1, 2, 3, 4])
concat2([[1, 2], [3, 4], [5, 6, 7]])

```

4. On définit maintenant :

```

def subtract(l: list[int]) -> int:
    return foldr((lambda x, y: x - y), l, 0)

def subtract2(l: list[int]) -> int:
    return foldl((lambda x, y: x - y), l, 0)

```

Que valent les expressions suivantes ?

```

subtract([1, 2, 3])
subtract2([1, 2, 3])

```

5. Vérifier sur des exemples la propriété suivante :

```

foldr(f, l, init) == foldl(g, rev(l), init)

```

où la fonction `g` vérifie la propriété suivante :

```

g(x, y) == f(y, x)

```

6. Donner des versions impératives de `map`, `filter` et `foldl` (aussi appelé `reduce`).