TP n°2

Chaînes et fonctions

Quelques fonctions prede nies

Dans les exercices qui suivent, vous pourrez utiliser les fonctions prede nies suivantes pour realiser des a chages a l'ecran :

- { `print_string : string -> unit` pour a cher une cha^ne de caracteres ;
- { `print_int : int -> unit` pour a cher un entier ;
- { `print_newline : unit -> unit` pour passer a la ligne { cette fonction prend en argument la constante `()` de type `unit`, un appel de cette fonction s'ecrit donc `print_newline ()`.

Les fonctions suivantes seront egalement utiles pour manipuler des cha^nes de caracteres :

- { `String.length : string -> int` donne la longueur d'une cha^ne ;
- { `String.sub : string -> int -> int -> string` retourne, pour `s n l` donnes, la sous-cha^ne de `s` de longueur `l` commencant a la position `n` (la premiere position est `0`).

Rotations de cha^nes de caracteres

Etant donnee une cha^ne de caracteres `s` : la *rotation gauche* de `s` est la cha^ne obtenue en deplacant en n de cha^ne le premier caractere de `s` ; la *rotation droite* de `s` est la cha^ne obtenue en deplacant en debut de cha^ne le dernier caractere de `s`. Par exemple, les rotations gauche et droite de `"abcde"` sont respectivement `"bcdea"` et `"eabcd"`.

Exercice 1. Le but de cet exercice est d'ecrire une fonction a chant toutes les rotations gauches ou droites successives d'une cha^ne. Ecrire les fonctions suivantes, dont chacune peut utiliser les precedentes et/ou les fonctions prede nies :

1. `rotation_g : string -> string`
   renvoyant la rotation gauche d'une cha^ne.
2. `rotation_d : string -> string`
   renvoyant la rotation droite d'une cha^ne.
3. `iter_g : string -> int -> unit`
   telle que `iter_g s n` a che `n+1` lignes avec : sur la premiere ligne, la cha^ne `s` ; sur chaque ligne suivante, la rotation gauche de la cha^ne a chee sur la ligne precedente.

   ```
   # iter_g "abcde" 2 ;;
   abcde
   bcdea
   cdeab
   ```

4. `iter_d : string -> int -> unit`
   similaire a la precedente, mais avec une rotation droite.

5. `enum_g : string -> unit`
   telle que `enum_g s` a che sur des lignes successives toutes les chaînes obtenues en partant de `s` et en e ectuant des rotations gauches successives.

   ```
   # enum_g "abcde";;
   abcde
   bcdea
   cdeab
   deabc
   eabcd
   ```

6. `enum_d : string -> int -> unit`
   similaire a la precedente, mais avec une rotation droite.

Exercice 2 ($\star$). Les codes de `iter_g` et `iter_d` sont tres similaires. Pour eviter cette duplication de code, essayez d'ecrire une fonction `iter_rotation` prenant en argument `s`, `n`, puis en argument supplementaire l'une au l'autre des fonctions `rotation_g` ou `rotation_d`, et e ectuant le traitement de `iter_g` ou `iter_d` correspondant. De même, ecrire une fonction `enum_rotation` factorisant `enum_g` et `enum_d`.

Diviseurs

Exercice 3. Ecrire une fonction `diviseurs : int -> unit` qui a che successivement les di erents diviseurs possibles d'un entier donne en argument. Par exemple :

```
# diviseurs 12 ;;
1 2 3 4 6 12
```

Suggestion : Utiliser une fonction auxiliaire qui prend en argument deux entiers `n` et `k` et qui a che les diviseurs de `n` superieurs ou egaux a `k`.

Exercice 4. Ecrire une fonction `est_premier : int -> bool` qui determine si un nombre entier est premier (c'est-a-dire sans aucun diviseur autre que 1 et lui-même). Est-on oblige d'essayer tous les diviseurs possibles comme pour la fonction `diviseurs` ?

Exercice 5 ($\star$). Ecrire une fonction qui a che successivement les di erentes decompositions possibles d'un entier. Ici 1 ne sera pas considere comme un facteur possible. Par exemple :

```
12 = 2*6 = 2*2*3 = 2*3*2 = 3*4 = 3*2*2 = 4*3 = 6*2
```

Suggestion : Utiliser une fonction auxiliaire ayant un argument de type `string` contentant le debut de la factorisation (par exemple `"2*2"`).
Extension possible : Modi er votre fonction a n qu'elle evite les redondances comme `2*6` et `6*2`.

Fonctions mutuellement recursives

Exercice 6. Dans un bois, il y a des lapins et des renards. Au matin du premier jour, il y a 37 lapins et 5 renards. Chaque jour, a midi, chacun des renards mange 2 lapins { mais il reste toujours au moins 10 lapins qui se sauvent toujours. Chaque lapin qui n'est pas mange fait na^tre un autre lapin au matin du jour suivant. A chaque fois que 3 lapins sont manges, un nouveau renard na^t au matin du jour suivant. Finalement, chaque soir, 1/10 des renards (arrondi a l'entier inferieur le plus proche) meurt de vieillesse.

Ecrire trois fonctions mutuellement recursives `lapins`, `renards`, `manges` de type `int -> int`, telles que `(lapins n)`, `(renards n)`, `(manges n)` renvoient respectivement, les populations de lapins et de renards le matin apres les naissances eventuelles, et le nombre de lapins manges a midi, au bout de `n` jours. Evaluez ces fonctions pour les premieres deux semaines. Changez les nombres de lapins et/ou renards du premier jour et observez les changements.

Fibonacci, le retour

On considere de nouveau les nombres de Fibonacci, de nis par $f(0) = 1$, $f(1) = 1$ et $f(n) = f(n-1) + f(n-2)$ si $n \geq 2$.

Exercice 7. Si vous ne l'avez pas deja fait au TP precedent, ecrivez une fonction recursive `fib` implementant cette de nition. Pour des petites valeurs de `n`, determinez le nombre d'appels recursifs engendres par le calcul de `(fib n)`. Vous pouvez utiliser la directive `#trace fib` pour obtenir de l'information sur les sous-calculs e ectues par Caml, puis `#untrace fib` quand vous avez ni. Comment evolue le nombre d'appels recursifs lorsque `n` augmente ? Jusqu'ou peut-on calculer en temps raisonnable avec cette version de `fib` ?

Exercice 8 ($\star$). Ecrivez une fonction `fib_iter` qui attend des arguments `x y n` ou `x` et `y` sont deux nombres de Fibonacci consecutifs, et qui calcule le nombre de Fibonacci situe `n` etapes apres `y`. En deduire une nouvelle implementation optimisee de `fib`. Jusqu'ou peut-on calculer avec cette version ? Que se passe-t'il pour `(fib_opt 90)` ?