Programmation 1 – TP n° 1
Le zipper

Emacs et Tuareg

`emacs` admet un mode nommé `tuareg` facilitant le travail sous *OCaml*. Voici les raccourcis essentiels :

`C-x C-f`	Ouvrir un nouveau fichier.
`C-x C-s`	Sauvegarder le fichier courant.
`C-x b`	Changer de fichier courant.
`C-h C-h`	Aide sur l'aide d'emacs.
`C-c C-e`	Évaluer la phrase courante dans ocaml.
`C-c C-r`	Évaluer la région courante dans ocaml.
`C-c C-k`	Interrompre ocaml (quand ça boucle).
`C-c h`	Afficher l'aide *Ocaml* sur un symbole de la librairie standard.
`C-c C-h`	Afficher les raccourcis de `tuareg`.

1 Le *zipper* sur les listes

Dans la syntaxe d'OCaml, le type des listes est défini récursivement :

```
type 'a list = Nil | Cons of 'a * 'a list
```

(sauf que l'opérateur `Nil` s'appelle `[]` et l'opérateur `Cons` est infixe et s'appelle `::`)

Ce type de données fonctionnel est mal adapté à des algorithmes impératifs faisant plusieurs opérations en place dans la structure (insertions, remplacements, etc.). Au contraire, les fonction comme `mem`, qui recherche la présence d'un élement dans une liste, ou `length`, qui

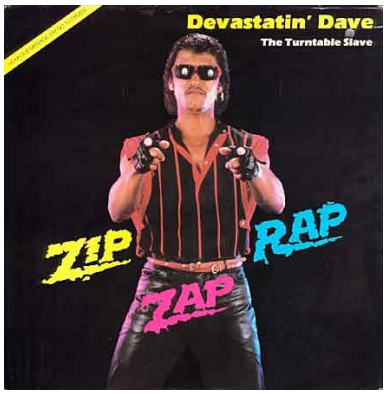


FIG. 1 – Exemples de zipper et de `attrape`

calcule la longueur d'une liste, sont facile car elles n'ont besoin que d'un seul parcours de la structure de données.

Q 1.1 Ecrire les fonctions `mem : 'a -> 'a list -> bool` et `length : 'a list -> int`.

Q 1.2 Programmer la fonction `append : 'a list -> 'a list -> 'a list` qui concatène deux listes.

Q 1.3 Ecrire la fonction `rev` de type `'a list -> 'a list` qui renverse une liste.

Nous allons nous inspirer de la fonction `rev` pour choisir une représentation qui contient toutes les informations de la liste et qui peut exprimer une position arbitraire dans celle-ci, facilitant ainsi les algorithmes impératifs. Une solution naturelle consiste à garder la trace des éléments déjà parcourus (le "chemin" du départ jusqu'à la position) :

```
type 'a path = Start | Succ of 'a path * 'a
```

Q 1.4 Que vous rappelle ce type de données ? Dans quel ordre sont rangés les éléments parcourus ?

Une position consiste alors en la donnée d'un chemin et du reste de la liste :

```
type 'a position = 'a path * 'a list
```

Q 1.5 Programmer la fonction `go_right : 'a position -> 'a position` qui avance dans la liste et la fonction `go_left` qui recule. Si l'opération est impossible, elles devront lever l'exception `Out_of_bounds`.

La fonction `lazy_beaver` parcourt une liste d'entier jusqu'à trouver un entier n plus grand que son argument m. Ensuite elle revient en arrière dans la liste de $n/4$ éléments et renvoie la valeur courante (ou `Out_of_bounds` le cas échéant).

Q 1.6 Ecrire la fonction `lazy_beaver` avec des listes puis des positions. Quels sont les avantages de cette structure ?

Q 1.7 Ecrire une fonction `append_pos : 'a position -> 'a position -> 'a position`. Commenter.

2 Pareil sur des arbres binaires

On définit un type d'arbre binaire dont les feuilles sont étiquetées par des valeurs de type `'a` et les noeuds par des valeurs de type `'b` :

```
type ('a,'b) tree = Leaf of 'a
                  | Node of ('a,'b) tree * 'b * ('a,'b) tree
```

On cherche, comme pour les listes, à définir un type de données permettant de représenter une position dans ces arbres afin de se déplacer dans la structure de façon impérative. On définit ainsi le type `path` qui contient les informations oubliées durant le parcours : les branches non-explorées et les étiquettes des noeuds parcourus.

```
type ('a,'b) path = Top
                  | Left of ('a,'b) path * 'b * ('a,'b) tree
                  | Right of ('a,'b) tree * 'b * ('a,'b) path

type ('a,'b) position = ('a,'b) path * ('a,'b) tree
```

Q 2.1 Comprendre ces nouveaux types de données et les similarités avec ceux utilisés pour les listes.

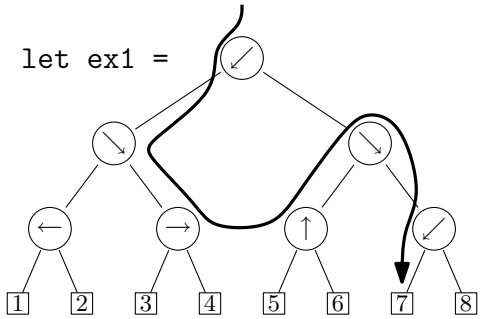
Q 2.2 Ecrire les fonctions `go_up : ('a,'b) position -> ('a,'b) position`, `go_down_left` et `go_down_right`.

On étiquette maintenant les noeuds par des directions (dont `Droite` et `Gauche` non-triviales, voir l'exemple) :

```
type dir = Bas_droite | Bas_gauche | Haut | Droite | Gauche

type 'a jeu = ('a,dir) tree
```

Un jeu sur un tel arbre sera un parcours dans l'arbre depuis la racine qui suit les directions des noeuds parcourus. Ainsi, une exécution du terme `ex1` devra renvoyer 7.



Q 2.3 Compléter le fichier `jeu_arbre.ml`.

Q 2.4 Proposer une implémentation pour repérer et éviter les boucles infinies dans ces jeux.

3 La fonction `attrape`

On considère enfin des arbres non vides variadiques (dont les noeuds ont un nombre arbitraire fini de fils) :

```
type 'a tree = Node of 'a * 'a tree list
```

Q 3.1 Ecrire une fonction `attrape : 'a -> 'a tree -> 'a tree` telle que `attrape a t` cherche un noeud de `t` étiquetté par `a` et créer l'arbre ayant la même structure mais enraciné en `a`.