avait une première composante qui était `return` ou `normal` selon que l'exécution de cette instruction avait rencontré une instruction `return` ou non. Il faut maintenant ajouter un troisième cas, `exception`, pour le cas où l'exécution de cette instruction lève une exception. De même, l'évaluation d'une expression peut maintenant se dérouler normalement ou lever une exception.

Le cas `exception` n'est pas très différent du cas `return`. En particulier, si dans une séquence $\{p_1\ p_2\}$ l'exécution de p_1 lève une exception, alors l'instruction p_2 n'est pas exécutée et le résultat de l'exécution de $\{p_1\ p_2\}$ est cette même exception.

Exercice 7.1

Étendre la définition de la fonction Σ de manière à prendre en compte les exceptions.

7.7 Caml

En Caml l'instruction `throw e` *s'écrit* `raise e`. *Il n'est pas nécessaire de déclarer qu'une fonction peut lever une exception. On déclare une nouvelle exception par l'instruction* `exception C`. *On récupère une exception avec la construction* `try p with _ -> q`.

Ainsi, on peut commencer par déclarer une exception

```
exception Pas_dans_la_liste
```

puis définir la fonction `assoc` *et l'utiliser dans le programme principal*

```
let rec assoc x l = if l = []
                    then raise Pas_dans_la_liste
                    else if fst(List.hd l) = x
                         then snd(List.hd l)
                         else assoc x (List.tl l)
in try print_int(assoc 5 [(3,4)])
   with _ -> print_string("Pas dans la liste")
```

Il n'y a pas d'exceptions en C, mais certaines constructions comme les sauts non locaux *s'en approchent un peu.*