Évaluation d'une formule arithmétique

On modélise les expressions arithmétiques par des listes de lexèmes selon la déclaration suivante :

```
type lexème =
  | Nombre of int
  | Op1 of (int -> int)
  | Op2 of (int * (int -> int -> int))
  | Parg
  | Pard
;;
```

- `Nombre`(n) représente une valeur numérique, on se limite aux valeurs entières ici ;
- `Op1`(f) représente une opération unaire, f est une fonction Caml implémentant cette opération ;
- `Op2`(p, f) représente une opération binaire, p est la priorité de l'opération (1 pour $+$ et $-$, 2 pour $\times$ et $\div$) et f est une fonction Caml implémentant cette opération ;
- `Parg` et `Pard` représentent les parenthèses gauche et droite.

La fonction suivante analyse une chaîne de caractères et la décompose en une liste de lexèmes :

```
let décompose s =
  let l = ref [] in
  for i=0 to string_length(s)-1 do match s.[i] with
    | ‘ ‘ -> ()
    | ‘~‘ -> l := Op1(minus_int)  :: !l
    | ‘+‘ -> l := Op2(1,add_int)  :: !l
    | ‘-‘ -> l := Op2(1,sub_int)  :: !l
    | ‘*‘ -> l := Op2(2,mult_int) :: !l
    | ‘/‘ -> l := Op2(2,div_int)  :: !l
    | ‘(‘ -> l := Parg  :: !l
    | ‘)‘ -> l := Pard  :: !l
    | ‘0‘..‘9‘ as c -> begin
        let x = char__int_of_char(c) - char__int_of_char(‘0‘) in
        match !l with
          | Nombre(y)::suite when (i > 0) & (s.[i-1] <> ‘ ‘)
              -> l := Nombre(x+10*y)::suite
          | _ -> l := Nombre(x) :: !l
      end
    | _ -> failwith "caractère imprévu"
  done;
  rev !l
;;
```

Les fonctions `minus_int`, `add_int`, `sub_int`, `mult_int`, `div_int` sont définies dans la bibliothèque standard de Caml et implémentent les opérations usuelles sur les entiers : opposé, addition, soustraction, multiplication et division entière. Noter que `décompose` distingue l'opposé et la soustraction par deux symboles différents : `~` et `-`. Exemple de décomposition pour une formule infixe et une formule postfixe :

```
#décompose "31 * (~4 + 5)";;
- : lexème list =
 [Nombre 31; Op2 (2, <fun>); Parg; Op1 <fun>; Nombre 4; Op2 (1, <fun>);
  Nombre 5; Pard]
#décompose "31 4 ~ 5 - *";;
- : lexème list =
 [Nombre 31; Nombre 4; Op1 <fun>; Nombre 5; Op2 (1, <fun>); Op2 (2, <fun>)]
```

L'objet du TP est de programmer l'algorithme d'évaluation d'une formule postfixe vue en cours et l'évaluation d'une formule infixe en respectant le parenthèsage et les règles de priorité.

1) Formule postfixe

Écrire une fonction récursive `eval_post : int list -> lexème list -> int` telle que `eval_post` *pile formule* calcule la valeur associée à *formule*, *formule* étant une formule postfixe supposée correcte et sans parenthèses et *pile* la pile des valeurs utilisée dans l'algorithme du cours. Pour évaluer une chaîne de caractères représentant une formule postfixe on utilisera le code : `let valeur_post(s) = eval_post [] (décompose s);;`

2) Formule infixe

L'évaluation d'une formule infixe peut se faire de la même manière en stockant dans une pile les lexèmes rencontrés et en réduisant à chaque étape le haut de la pile lorsque c'est possible selon les règles :

– remplacer la séquence $f\ x$ où f est un opérateur unaire et x un nombre par le nombre $f(x)$;
– remplacer la séquence $x\ f\ y\ g$ où f et g sont des opérateurs binaires et x, y des nombres par $z\ g$ si la priorité de f est supérieure ou égale à celle de g où z est le nombre $f\ x\ y$;
– remplacer la séquence $x\ f\ y\ )$ où f est un opérateur binaire et x, y des nombres par $z\ )$ avec $z = f\ x\ y$;
– remplacer la séquence $(\ x\ )$ où x est un nombre par le nombre x.

Selon ces règles, si deux opérations binaires ont même priorité alors celle de gauche doit être effectuée en premier, ce qui permet d'évaluer correctement une expression telle que $x - y - z$. Pour évaluer complètement une formule infixe f supposée correcte il suffit d'appliquer l'algorithme de réduction à la formule (f), le groupe de parenthèses forçant l'évaluation des opérations en instance lorsqu'on a terminé le parcours de f.

Programmer cet algorithme. On définira une fonction auxiliaire : `empile : lexème -> lexème list -> lexème list` qui reçoit en argument un lexème et la pile d'évaluation et retourne la nouvelle pile obtenue par insertion du lexème et application récursive des règles de réduction.

3) Prolongements possibles

– Selon l'algorithme précédent les opérations unaires sont interprétées de façon préfixe, ce qui correspond à l'usage pour la fonction « opposé », mais il existe aussi en mathématiques des opérations unaires placées en position postfixe (carré, factorielle). Modifier le type `lexème` et le programme d'évaluation infixe pour traiter cette situation.

– Dans les langages de programmation usuels l'opposé et la soustraction sont notées par le même symbole `-`. Modifier votre programme pour se conformer à cet usage.

– Lorsqu'une formule infixe est incorrecte peut-on afficher un message d'erreur explicite indiquant quel est l'erreur et à quel endroit elle figure dans la formule ?