Evaluation d'expressions arithmétiques

Dans tout ce TP, on considère uniquement le type `Z` pour représenter les nombres. On se restreint aux opérations binaires suivantes : addition, soustraction et multiplication.

1 Expressions arithmétiques infixes

1- Définir un type `expr` permettant de représenter les expressions arithmétiques infixes complètement parenthésées pour les opérateurs d'addition, soustraction et multiplication.

```
Inductive expr : Set := ...
```

2- Construire deux termes représentant les expressions suivantes `(2+5)*3` et `((2-1)+(7*6))-2`.

3- Programmer une fonction d'évaluation `eval_expr`. La tester avec les deux termes construits à la question précédente.

2 Expressions arithmétiques postfixes

On considère maintenant les expressions arithmétiques postfixes. Celles-ci seront représentées par des listes contenant des opérateurs et des nombres.

4- Construire le type de base nécessaire pour construire de telles listes.

5- Construire les listes correspondant aux expressions `2 5 + 3 *` et aussi `2 1 - 7 6 * + 2 -`.

6- Redéfinir en Coq les opérations de base sur les piles d'entiers relatifs : `pile_nouv`, `empiler`, `depiler`, `sommet`.

7- Programmer une fonction d'évaluation `eval_postfixe2` qui prend en arguments une expression à évaluer (sous forme de liste) et une pile. En déduire une fonction `eval_postfixe` qui démarre le calcul avec la pile vide.

8- Vérifier le comportement de cette fonction sur les deux exemples de la question précédente.

3 Compilation des expressions infixes vers des expressions postfixes

9- Définissez l'opération `append` (que l'on pourra noter `++`) de concaténation de deux listes. Programmer une opération `translate` qui transforme un terme représentant une expression infixe en un terme representant l'expression postfixe correspondante.

On veut prouver la propriété suivante :

```
Lemma interp_ok : forall e:expr, eval_expr e = (eval_postfixe (translate e)).
```

Pour cela, il va falloir passer pour un certain nombre de définitions et de démonstrations de lemmes intermédiaires. On commence par généraliser le lemme précédent et on considère maintenant le lemme suivant :

```
Lemma interp_ok' :
  forall e:expr, forall p, eval_expr e = sommet (eval_postfixe2 (translate e) p).
```

La première étape consiste à définir un *prédicat inductif* `well_formed` qui exprime que la liste fournie en arguments est bien formée, i.e. les nombres d'arguments et d'opérateurs sont cohérents.

10- Programmer ce prédicat en Coq.

```
Inductive well_formed : liste -> Prop :=
  | w_seul  : forall z:Z, well_formed(...)
  | w_plus  : ...
  | w_moins : ...
  | w_fois  : ...
```

11- Vérifier maintenant que la traduction d'une expression avec `translate` produit bien une liste bien formée.

```
Lemma wf_ok : forall e, well_formed (translate e).
```

12- Prouver ensuite les deux lemmes techniques suivants qui seront fondamentaux pour la preuve du théorème :

```
Lemma append_eval : forall e f, forall p,
  eval_postfixe2 (e ++ f) p = eval_postfixe2 f (eval_postfixe2 e p).

Lemma depiler_eval : forall f, well_formed f ->
 forall p, p=depiler(eval_postfixe2 f p).
```

13- A partir de ces deux lemmes, établir le lemme suivant qui simule l'évaluation d'expressions postfixes *à la manière de* l'évaluation infixe.

```
Lemma lemma_plus : forall e f, forall p, well_formed e -> well_formed f ->
  sommet (eval_postfixe2 e p) + sommet (eval_postfixe2 f (eval_postfixe2 e p))=
  sommet (eval_postfixe2 (e  ++ (f ++ ((s plus):: lvide))) p).
```

14- Prouver deux lemmes similaires avec la soustraction et la multiplication.

15- Utiliser ces lemmes pour démontrer le théorème `interp_ok'`, puis `interp_ok` par induction sur la structure de l'expression e traduite.