Colle 7

Interpréteur arithmétique

MPII : Ocaml

Dans ce TP nous allons interpréter les programmes dans un langage très simple : les expressions arithmétiques. Le but est de lire une expression arithmétique sous forme d'une chaîne de caractères et d'en calculer la valeur.
Nous nous restreindrons au cas de valeurs entières positives en raison du double sens du symbole "-" ; pour introduire un entier négatif, on l'écrira `(0 - 14)` pour -14.
On n'utilise que les opérations "+", "-", "*" et "/", la division entière.
Rappels

- Une chaîne de caractères est encadrée par des guillemets doubles : `"Bonjour"`.
- Sa longueur est fournie par `String.length ch`.
- Le caractère d'indice i d'une chaîne est obtenu par `ch.[i]`.
- Un caractère est encadré par des guillemets simples.
- `Char.code` permet de renvoyer le code ascii d'un caractère.
- pour calculer le chiffre représenté par un caractère entre `'0'` et `'9'`, on peut écrire `Char.code c - Char.code '0'`. Pour `c = '5'` on obtient 5.
- `List.rev` permet de retourner une liste.

I Lexer très simple

La première étape consiste à lire la chaîne de caractères et à la séparer en ses objets de bases appelés lexèmes : ce seront les entiers, les opérateurs et les parenthèses.
On utilisera le type suivant :

```
type lexeme =  Nb of int                    (* les entiers *)
             |Plus |Moins |Fois |Divise     (* les opérateurs *)
             |ParO |ParF;;                  (* les parenthèses *)
```

On se contentera ici de nombres qui sont des chiffres (1, 2, 3, 4, 5, 6, 7, 8 et 9), on fournira ainsi des exemples utilisables dans les parties suivantes. On reviendra à un cas plus général à la fin. Bien entendu les calculs pourront donner des valeurs au delà de ces chiffres.

Question 1
Écrire une fonction `lecture : string -> lexeme list` qui reçoit une chaîne de caractères et qui renvoie la liste des lexèmes associée. Les caractères qui ne correspondent pas à un lexème seront ignorés.

`lecture "5*(4 + 3)a"` doit renvoyer

```
[Nb 5; Fois; ParO; Nb 4; Plus; Nb 3; ParF]
```

II Notation polonaise inversée

En 1920 le mathématicien polonais Jan Lukasiewicz propose la notation pré-fixé (ou polonaise) qui consiste à considérer les opérations comme des opérateurs à 2 variables, $7 + 11$ s'écrit $+\ 7\ 11$.
L'avantage est que les parenthèses deviennent inutiles :
$(7 - 2).\sin(2x + \pi/3)$ devient $*\ -\ 7\ 2\ \sin\ +\ *\ 2\ x\ /\ \pi\ 3$.
Hewlett-Packard a utilisé cette notation, en l'inversant, comme interface utilisateur avec les calculatrices de bureau de (HP-9100), puis avec la calculatrice scientifique HP-35 en 1972.
Cette notation polonaise inversée consiste simplement à placer l'opérateur **après** les deux opérandes, `"7 + 11"` s'écrit `"7 11 +"`.
L'expression `"8 - 2*((7 / 2)*3 - 6)"` se traduit donc par
`"8 2 7 2 / 3 * 6 - * -"`
L'avantage est que, combinée à une pile, cette notation permet d'effectuer les calculs sans faire référence à une quelconque adresse mémoire.

- On lit l'expression terme-à-terme :
- si on lit une valeur numérique, elle est empilée,
- si on lit une opération ♣,
  on dépile les deux derniers opérandes a et b
  et on empile le résultat du calcul a ♣ b.

Un exemple : on part de l'expression `"8 2 7 2 / 3 * 8 - * -"` :

Lecture	Action	Pile
	On crée une pile vide	
8	On empile	8
2	On empile	8 2
7	On empile	8 2 7
2	On empile	8 2 7 2
/	On dépile 2 éléments	3 2
	On empile 7/2	8 2 3
3	On empile	8 2 3 3
*	On dépile 2 éléments	8 2
	On empile $3 * 3$	8 2 9
6	On empile	8 2 9 6
-	On dépile 2 éléments	8 2
	On empile $9 - 6$	8 2 3
*	On dépile 2 éléments	8
	On empile $2 * 3$	8 6
−	On dépile 2 éléments	
	On empile $8 - 6$	2
	On dépile le résultat : 2	

On définit un type `pile` **impératif** : empiler et dépiler modifient la pile sans en créer une nouvelle.
On ne l'utilisera qu'au travers des fonctions

- `pileVide : unit -> pile` qui crée une pile vide,
- `sommet : pile -> élément` qui renvoie le sommet de la pile sans l'enlever,
- `empiler : pile -> élément -> unit` qui ajoute un élément au sommet de la pile,
- `depiler : pile -> unit` qui enlève l'élément au sommet de la pile,
- `est_pileVide : pile -> bool` qui teste si une pile est vide.

Question 2
Définir un type `pile` et écrire les fonctions.

Question 3
Écrire une fonction `calculNPI : string -> int` qui calcule la valeur d'une expression **écrite au format NPI** sous forme d'une liste de lexèmes.

III Cas général

Le procédé ci-dessus demande d'écrire la formule mathématique en notation N.P.I. ce qui n'est pas pratique. Nous allons maintenant traduire une formule "normale" en une formule N.P.I.
Pour cela on aura à gérer la priorité des opérations :
`"3 + 4 * 5"` est traduit en `"3 4 5 * +"` alors que `"3 * 4 + 5"` est traduit en `"3 4 * 5 +"`
On donne la priorité 1 aux opérations $+$ et $-$ et la priorité 2 aux opérations $*$ et $/$.

Pour traduire une formule on effectue les opérations suivantes.

- On crée une pile vide,
- on lit la formule terme-à-terme :
  - si on lit une valeur numérique, elle est ajoutée à la sortie,
  - si on lit une opération ♣, on dépile les opérations de priorité supérieure ou égale à celle de ♣ et on les ajoute à la sortie, on empile ensuite ♣ sous forme d'un couple avec sa priorité,
  - si on lit une parenthèse ouvrante on l'empile avec la priorité 0,
  - si on lit une parenthèse fermante, on dépile et on ajoute à la sortie toutes les opérations jusqu'à ce qu'on trouve une parenthèse ouvrante, on dépile cette parenthèse ouvrante.
- Quand on a lu tous les lexèmes, on dépile et on ajoute à la sortie toutes les opérations restantes.
- La sortie est une liste que l'on doit renverser pour obtenir la formule N.P.I.

Un exemple On part de l'expression `"2*(11-2+3*2)"` qui a été transcrite en

```
[Nb 2; Fois; ParO; Nb 11; Moins; Nb 2; Plus;
                                 Nb 3; Fois; Nb 2; ParF]
```

Pour des raisons de place on écrit les lexèmes sous forme "naturelle" dans la pile et la sortie.

Lecture	Pile	Sortie
Nb 2		[2]
Fois	(*, 2)	[2]
ParO	(*, 2) ((, 0)	[2]
Nb 11	(*, 2) ((, 0)	[11; 2]
Moins	(*, 2) ((, 0) (-, 1)	[11; 2]
Nb 2	(*, 2) ((, 0) (-, 1)	[2; 11; 2]
Plus	(*, 2) ((, 0) (+, 1)	[-; 2; 11; 2]
Nb 3	(*, 2) ((, 0) (+, 1)	[3; -; 2; 11; 2]
Fois	(*, 2) ((, 0) (+, 1) (*, 2)	[3; -; 2; 11; 2]
Nb 2	(*, 2) ((, 0) (+, 1) (*, 2)	[2; 3; -; 2; 11; 2]
ParF	(*, 2)	[+; *; 2; 3; -; 2; 11; 2]
		[*; +; *; 2; 3; -; 2; 11; 2]

La notation N.P.I. est donc

```
[Nb 2; Nb 11; Nb 2; Moins; Nb 3; Nb 2; Fois; Plus; Fois]
```

Question 4
Écrire une fonction `convertirNPI : lexeme list -> lexeme list` qui convertit une liste de lexèmes qui exprime une expression sous forme naturelle en une liste de lexèmes qui exprime une expression sous forme N.P.I.

Question 5
En déduire une fonction `calcul : string -> int` qui calcule la valeur d'une expression naturelle dans une chaîne de caractères et qui calcule sa valeur.

IV Lexer général

Pour générer les nombres généraux (positifs), on doit accumuler les lectures de chiffres et ne les ajouter à la liste de lexèmes que lorsque l'on voit un autre caractère ou à la fin.
Si on veut discerner le cas où il n'y a pas de nombre et celui où il y a un nombre qui vaut k, on peut utiliser un type optionnel :

```
type 'a opt = None | Some of 'a
```

`Some 412` signifie que la variable contient un nombre et qu'il vaut 412, `None` signifie que la variable ne contient pas de valeur.

Question 6
Écrire une fonction `lecture : string -> lexeme list` qui reçoit une chaîne de caractères et qui renvoie la liste des lexèmes associée. Les caractères qui ne correspondent pas à un lexème seront ignorés.

V Aller plus loin

Au cas où vous auriez fini ...

- Introduire la fonction modulo qui calcule le reste de la division euclidienne (`mod` en OCaml). On choisira avec soin sa priorité.
- Introduire l'exponentiation (dont on devra écrire la fonction en OCaml).
  Il y a ici un problème de priorité : en effet on lit généralement `a ** b ** c` sous la forme a^{b^c} et non $(a^b)^c$ qui se simplifie en a^{bc}. On pourra donner une priorité différente au symbole d'exponentiation dans la pile et hors de la pile.