TD d'analyse syntaxique : extension/réduction du projet TL1

On souhaite programmer en PYTHON une calculette *en notation préfixe*[1]. Le principe d'une telle notation est que chaque opérateur est d'arité fixe (a un nombre d'arguments fixe) et qu'il est placé syntaxiquement en position *préfixe*, c'est-à-dire *avant* ses arguments. Ces contraintes suffisent à rendre les expressions du langage non-ambiguës sans recourir à des parenthèses. Concrètement, la syntaxe de cette calculette se décrit à l'aide d'une BNF[2] non-ambiguë très simple donnée ci-dessous.

La lexicographie de cette calculette est identique à celle du chapitre 2 du cours d'amphi, sauf sur les points suivants : les terminaux `OPAR` et `CPAR` sont superflus et donc absents ; il y a un terminal spécial `END` qui correspond à une sentinelle de fin de fichier.

On a donc les terminaux de la BNF (tokens) avec les langages de lexèmes correspondants :
`PLUS` $= \{$`'+'`$\}$ `MINUS` $= \{$`'-'`$\}$ `MULT` $= \{$`'*'`$\}$ `DIV` $= \{$`'/'`$\}$ `QUEST` $= \{$`'?'`$\}$
`END` $= \{$*end_of_input*$\}$ (fin de fichier, `^D`...)

On utilise aussi le token `NAT` $= \{$`'0'`$, \ldots,$ `'9'`$\}^+$ qui correspond aux entiers naturels en base 10 et le token `CALC` $= \{$`'#'`$\}$.`NAT` qui correspond aux valeurs des calculs déjà effectués. Ces deux tokens ont les profils d'attributs `NAT`$\uparrow\mathbb{N}$ et `CALC`$\uparrow\mathbb{N}$ où $\mathbb{N}$ est l'ensemble des entiers naturels. Ces attributs sont donc la valeur en base 10 de l'entier lu par l'analyseur lexical (après le `'#'` pour `CALC`). Voir le CM2 pour les explications.

Les profils d'attributs de la BNF sont :
- **input**$\downarrow\mathbb{L}\uparrow\mathbb{L}$, où $\mathbb{L}$ représente l'ensemble des listes d'entiers introduit au chapitre 2 ;
- **exp**$\downarrow\mathbb{L}\uparrow\mathbb{Z}$, où $\mathbb{Z}$ est l'ensemble des entiers relatifs ;

La calculette invoque l'axiome **input** avec comme liste héritée $[\,]$ qui représente la liste vide. Si les calculs ne contiennent pas d'erreur, elle récupère la liste synthétisée par **input** et l'affiche à l'écran. Comme vu au chapitre 2, la notation "$\ell[i]$" (pour $i \geq 1$) désigne le i-ième élément de la liste ℓ (ou correspond à une erreur si un tel élément n'existe pas).

input$\downarrow\ell\uparrow\ell'$	::=	`QUEST` **exp**$\downarrow\ell\uparrow n$ **input**$\downarrow(\ell \oplus n)\uparrow\ell'$	
	|	`END`	$\ell' := \ell$
exp$\downarrow\ell\uparrow n$	::=	`NAT`$\uparrow n$	
	|	`CALC`$\uparrow i$	$n := \ell[i]$
	|	`PLUS` **exp**$\downarrow\ell\uparrow n_1$ **exp**$\downarrow\ell\uparrow n_2$	$n := n_1 + n_2$
	|	`MINUS` **exp**$\downarrow\ell\uparrow n_1$	$n := -n_1$
	|	`MULT` **exp**$\downarrow\ell\uparrow n_1$ **exp**$\downarrow\ell\uparrow n_2$	$n := n_1 \times n_2$
	|	`DIV` **exp**$\downarrow\ell\uparrow n_1$ **exp**$\downarrow\ell\uparrow n_2$	$n := n_1 \,/\, n_2$

▷ **Question 1.** Quel est le comportement de la calculette sur l'entrée ci-dessous (implicitement terminée par une sentinelle de fin de fichier `END`) ? Dessiner l'arbre d'analyse avec la propagation d'attributs. On pourra noter "$([\,] \oplus a_1) \ldots \oplus a_n$" par "$[a_1, \ldots, a_n]$".

```
? + * 3 4 + 1 -3 ? * #1 / #1 2
```

1. Aussi appelée "*notation polonaise*", car inventée par le logicien polonais J. Łukasiewicz en 1924.
2. Backus-Naur Form, une façon courante de présenter des grammaires, utilisée ici.

L'analyseur de la calculette préfixée peut s'implémenter en suivant les principes présentés au chapitre 2 du cours.

```
def parse_exp(l):
    if current_token == NAT:
        n = val_current_token
        advance_token()
        return n
    elif current_token == PLUS:
        advance_token()
        n_1 = parse_exp()
        n_2 = parse_exp()
        return n_1 + n_2
    elif current_token == MINUS:
        advance_token()
        n_1 = parse_exp()
        return - n_1
    # ...
```

Sauf qu'ici, comme vu en cours, et comme on le constate sur la BNF attribuée, `parse_exp` doit avoir un paramètre : la liste des valeurs des calculs précédents...

Et évidemment, `parse_input` a une liste en paramètre et une liste en résultat.

▷ **Question 2.** Programmer une première version de la calculette où l'opération sur les listes "$\ell \oplus n$" est implémentée par le code PYTHON "`l+[n]`". Note : l'accès "$\ell[i]$" doit être implémenté en PYTHON par "`l[i-1]`" (les listes PYTHON commençant à l'indice 0).

▷ **Question 3.** Vérifiez vos fonctions "à la main" sur l'entrée
`QUEST MULT NAT`$\uparrow 2$ `NAT`$\uparrow 5$ `END`

Autrement dit : exécutez `parse_input([])`

Vérifiez que `advance_token` est bien appelée au bon moment.

▷ **Question 4.** Améliorer la calculette en implémentant l'opération sur les listes "$\ell \oplus n$" plus efficacement par "`l.append(n)`". En effet, "`l+[n]`" créé une nouvelle liste en recopiant intégralement "`l`", donc à coût $\Theta($`len(l)`$)$. En revanche, "`l.append(n)`" modifie la liste "`l`" en place avec un coût (amorti) constant. Il devient donc inutile que `parse_input` retourne ℓ' : à la fin la liste initiale a été modifiée en place au fur et à mesure des calculs par `parse_input`.

▷ **Question 5.** Qu'est-ce qui changerait si on avait défini

$$\begin{array}{rcll} \textbf{input}{\downarrow}\ell{\uparrow}\ell' & ::= & \textbf{exp}{\downarrow}\ell{\uparrow}n\ \texttt{QUEST}\ \textbf{input}{\downarrow}(\ell \oplus n){\uparrow}\ell' & \\ & | & \texttt{END} & \ell' := \ell \end{array}$$

▷ **Question 6.** Vérifiez vos fonctions "à la main" sur l'entrée
`MULT NAT`$\uparrow 2$ `NAT`$\uparrow 5$ `QUEST END`

Autrement dit : exécutez `parse_input([])`

Vérifiez que `advance_token` est bien appelée au bon moment.

Que faut-il faire sur `QUEST` ?

▷ **Question 7.** Qu'est-ce qui changerait si on avait défini

$$\begin{array}{rcll} \textbf{input}{\downarrow}\ell{\uparrow}\ell' & ::= & \textbf{input}{\downarrow}\ell{\uparrow}\ell''\ \textbf{exp}{\downarrow}\ell''{\uparrow}n\ \texttt{QUEST} & \ell' := \ell'' \oplus n \\ & | & \texttt{END} & \ell' := \ell \end{array}$$