TD analyse syntaxique en notation préfixe : revisitons *plus rigoureusement* le TP de TL1...

L'objectif de ce TD est de programmer en PYTHON une variante de calculette *en notation préfixe* (comme en TL1). On rappelle que 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 non-ambiguë très simple donnée ci-dessous.

La lexicographie de cette calculette est identique à celle du chapitre 2 du cours d'amphi. On a donc les terminaux de la BNF (tokens) avec les langages de lexèmes correspondants :
`PLUS` $= \{$`'+'`$\}$ `MINUS` $= \{$`'-'`$\}$ `MULT` $= \{$`'*'`$\}$ `DIV` $= \{$`'/'`$\}$ `QUEST` $= \{$`'?'`$\}$

On utilise aussi le token `NAT` $= \mathit{DIGIT}^+$ qui correspond aux entiers naturels en base 10, avec $\mathit{DIGIT} = \{$`'0'`$, \ldots,$ `'9'`$\}$ (ce n'est pas un token, juste un ensemble de caractères). Enfin, 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`). Et de façon classique, on utilise un token spécial pour indiquer la fin de l'entrée : ici, le token `END`.

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. 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**$\downarrow(\ell \oplus n)\uparrow\ell'$	
	|	ϵ	$\ell' := \ell$
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) ? 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 Conception rigoureuse de l'analyseur lexical

Comme l'analyseur lexical ne traite qu'une union de langages réguliers, on va construire son implémentation progressivement, en passant par l'intermédiaire d'automates finis.

▷ **Question 2.** Donner un automate fini déterministe (mais éventuellement incomplet), sur le vocabulaire des caractères $\mathcal{C}$ qui reconnaît le langage des lexèmes défini par

$$\mathcal{L} = \mathit{SEP}^*.(\texttt{QUEST} \cup \texttt{PLUS} \cup \texttt{MINUS} \cup \texttt{MULT} \cup \texttt{DIV} \cup \texttt{NAT} \cup \texttt{CALC} \cup \texttt{END})$$

où $\mathit{SEP} = \{\texttt{' '}, \texttt{'\t'}, \texttt{'\n'}\}$ et $\texttt{END} = \{\texttt{''}\}$ (ici `''` représente un caractère spécial marquant la fin de fichier).

On pourra étiqueter les transitions directement avec des ensembles de caractères.

L'analyseur lexical est plus qu'un simple reconnaisseur : il doit notamment produire une *valeur* entière dans le cas où il reconnaît un lexème de `NAT` ou de `CALC`. On va donc raffiner l'automate précédent en une sorte de machine de Mealy qui produit une donnée de sortie en fonction de la séquence de caractères en entrée. Une *machine de Mealy* (aussi appelée *transducteur fini*) est une extension des automates finis dans laquelle les transitions possèdent également des sorties. Ici, on va convertir les caractères en entrées en des terminaux, possiblement avec un attribut. Pour se rapprocher du programme Python final, on définit cette donnée de sortie à l'aide de types Python. Les terminaux sont représentés par des entiers entre 0 et 7, via une énumération Python :

```
Token = enum.Enum('Token',['QUEST', 'PLUS', 'MINUS', 'MULT', 'DIV',
                           'NAT', 'CALC', 'END'], start=0)
```

On définit un *terminal attribué* comme un couple `(t, v)` accepté par `assert_attr_token(t, v)` ci-dessous :

```
def assert_attr_token(t, v):
    assert t in Token
    match t:
        case Token.NAT | Token.CALC:
            assert type(v) is int and v >= 0
        case _:
            assert v is None
```

▷ **Question 3.** Transformer l'automate de la question précédente en une machine de Mealy qui effectue lors des transitions des affectations sur des variables `t` et `v` pour que, dans les états finals de l'automate, `t` soit le terminal reconnu et que si $\texttt{t} \in \{\texttt{NAT}, \texttt{CALC}\}$ alors `v` soit la valeur décimale de l'entier lu. Typiquement, une transition de cette machine de Mealy aura une étiquette de la forme "D / `t`:=nom" où D est l'ensemble des caractères qui déclenchent la transition, et *nom* le token rangé dans `t`. On peut aussi avoir la forme "$c \in D$ / `v`:=exp" où c est le nom du caractère (dans D) effectivement lu, utilisable dans l'expression "exp" ; on peut évidemment combiner des affectations simultanées à `t` et `v`... Pour alléger la notation, on assimilera abusivement un *chiffre* ('4') au *nombre* qu'il représente en base 10 (4).

L'automate considéré jusqu'ici décide si *une séquence de caractères donnée appartient à* $\mathcal{L}$. Notre analyseur lexical cherche à résoudre un problème un peu différent : dans la séquence de caractères restants, « *trouver le plus long préfixe qui appartient à* $\mathcal{L}$ ». On peut néanmoins réutiliser l'automate A de $\mathcal{L}$ pour résoudre ce dernier problème : il suffit de lire la séquence de caractères jusqu'à tomber dans **l'état d'erreur** de A (i.e. l'état puits). Alors, soit il n'y aucun préfixe appartenant à $\mathcal{L}$ si aucun état final de A n'a été rencontré (et c'est un cas d'erreur lexicale), soit les caractères lus jusqu'au **dernier** état final de A rencontré correspondent au plus long préfixe appartenant à $\mathcal{L}$.

▷ **Question 4.** Transformer la machine de Mealy précédente pour qu'elle reconnaisse *uniquement le plus long préfixe* de l'entrée appartenant à $\mathcal{L}$. Pour `NAT` et `CALC`, la machine est obligée de lire un caractère au-delà du lexème reconnu. Par souci de simplification, on décide que la machine lit *systématiquement* le caractère qui *suit* le lexème reconnu (sauf évidemment pour le token `END`).

Le caractère qui suit le lexème reconnu est appelé *caractère de pré-vision* (*look-ahead* en anglais), car c'est le premier caractère *déjà lu* pour la prochaine lecture.

▷ **Question 5.** Dans le cas général, pour un automate A (déterministe complet avec un seul état puits) quelconque de reconnaissance de lexèmes, on note P_A l'ensemble des *traces* – dites de *pré-vision*– allant d'un état final jusqu'à l'état puits sans boucler sur l'état puits, ni repasser par un état final. Quand ce langage P_A est fini, on note N_A la taille maximale d'un mot de P_A : N_A donne alors une borne sur la taille maximale de la suite de caractères de pré-vision à mémoriser pour la lecture du lexème suivant. En pratique, c'est très souvent 1. Mais pas en Python où `'1'` et `'1e2'` sont des lexèmes, mais pas `'1e'` alors qu'il y a d'autres lexèmes commençant par `'e'` : dans une entrée `'1eab'` en Python, on doit mémoriser les deux caractères de pré-vision `'ea'` après vu le `'1'`.[1]

1. Soit A l'automate de $\mathcal{L}$ à la question 3. P_A est-il fini ? Si oui, que vaut N_A ?
2. Même questions que précédemment, mais pour A l'automate (complété avec l'état puits) donné en TP de TL1 pour le langage `number`.
3. Montrer qu'utiliser un unique caractère de prévision dans l'analyse lexicographique du TP de TL1 (e.g. comme à la question 12) rend la calculatrice en notation préfixe incorrecte. On donnera un contre-exemple explicite (à inférer à partir de P_A).

Conclusion : la conception de l'analyse lexicale (ou syntaxique) est bien automatisable, alors que si on la fait à un niveau trop *intuitif*, on risque de produire un analyseur faux !

2 Programmer l'analyseur lexical [A FAIRE POUR LE TP]

On suppose maintenant que le caractère de pré-vision est rangé dans une variable globale `current_char` qui peut être positionnée sur le prochain caractère de l'entrée grâce à la fonction `update_current()`. Le code de l'analyseur lexical est écrit dans la fonction `next_token()` ci-dessous, qui doit retourner un couple `(t, v)` correspondant au prochain lexème de l'entrée ; elle suppose que le premier caractère du lexème à lire se trouve déjà dans `current_char`.

1. Cet exemple semble un peu artificiel en Python, car la grammaire interdit de toute façon la suite de lexèmes `'1' 'eab'`.

```
def next_token():
    while current_char in SEP: # skip separators
        update_current()
    try:
        return parse_token_after_separators()
    except KeyError:
        raise Error('Unknown start of token')
```

Après avoir consommé les séparateurs, cette fonction reconnaît un token à l'aide de la fonction `parse_token_after_separators`. Cette dernière décide quel terminal reconnaître en fonction du premier caractère du flot d'entrée et retourne un couple `(t, v)` où `t` est le terminal à retourner et `v` son attribut le cas échéant. On exploite ici le fait que le premier caractère qui suit un séparateur détermine le `t` à retourner. [2] Pour cela, on introduit un dictionnaire `TOKEN_MAP` associant un caractère à son terminal (sauf pour `NAT` traité à part). Pour les terminaux qui possèdent un attribut (`NAT` et `CALC`), il faut également calculer cet attribut et consommer pour cela les caractères correspondants en entrée ; on préfère donc les laisser dans des fonctions à part : `parse_NAT` et `parse_CALC`. Il faut également faire attention au terminal `END` pour lequel il ne faut pas consommer de caractère (il n'y en a plus !).

Le fichier `lexer.py`, fourni pour le TP, contient déjà un squelette de la fabrication de `parse_token_after_separators`. En l'état, celle-ci traite les terminaux sans attribut et `NAT`. Dans le cas de `NAT`, c'est la fonction `parse_NAT` qui doit s'occuper de la reconnaissance de l'entier à fabriquer et actuellement elle ne traite que des entiers d'un seul chiffre. Elle utilise une fonction fournie `parse_digit` qui lève une erreur si `current` n'est pas un chiffre décimal et sinon, appelle `update_current` en retournant la valeur décimale du chiffre lu.

```
def parse_NAT():
    print("@ATTENTION: lexer.parse_NAT à finir !") # LIGNE A SUPPRIMER
    v = parse_digit()
    return (Token.NAT, v)

def parse_token_after_separators():
    print("@ATTENTION: lexer à finir !") # LIGNE A SUPPRIMER
    if current_char in DIGITS:
        return parse_NAT()
    tok = TOKEN_MAP[current_char]
    match tok:
        case Token.END:
            return (Token.END, None)
        case _: # default case without attribute
            update_current()
            return (tok, None)
```

FIGURE 1 – Squelette de code pour l'analyseur lexical

▷ **Question 6. [A FAIRE POUR LE TP]**
Corriger le code fourni de `parse_NAT` et de `parse_token_after_separators` fourni à la figure 1.

2. Ce n'est pas toujours le cas, cf. `<` et `<=` dans la plupart des langages de programmation...

3 Programmation de l'analyseur syntaxique

La calculette en notation préfixe peut s'implémenter en suivant les principes de l'analyse LL(1) donnés au chapitre 3 du cours. Pour vous donner un exemple, voilà un squelette de code de la calculette, qui traite en fait la BNF suivante :

$$\begin{aligned} \textbf{input}{\downarrow}\ell{\uparrow}\ell' ::= \ & \textbf{exp}{\downarrow}\ell{\uparrow}n && \ell' := \ell \oplus n \\ \textbf{exp}{\downarrow}\ell{\uparrow}n ::= \ & \texttt{NAT}{\uparrow}n \\ | \ & \texttt{PLUS}\ \textbf{exp}{\downarrow}\ell{\uparrow}n_1\ \textbf{exp}{\downarrow}\ell{\uparrow}n_2 && n := n_1 + n_2 \end{aligned}$$

```
def parse_input(l):
    n = parse_exp(l)
    return l+[n]

def parse_exp(l):
    match get_current():
        case Token.NAT:
            return parse_token(Token.NAT) # retourne l'attribut associé au Token.NAT
        case Token.PLUS:
            parse_token(Token.PLUS) # on ignore l'attribut associé (qui vaut None)
            n1 = parse_exp(l)
            n2 = parse_exp(l)
            return n1+n2
```

Ci-dessus, `get_current()` retourne le terminal courant *sans le consommer*. De plus, `parse_token(tok)` vérifie que le terminal courant est `tok` et retourne la valeur de son attribut en *consommant* ce terminal.

▷ **Question 7.** Programmer une première version de la calculette où l'opération sur les listes "$\ell \oplus n$" est implémentée par le code Python "`l+[n]`" comme dans le squelette fourni. Note : l'accès "$\ell[i]$" doit être implémenté en Python par "`l[i-1]`" (les listes Python commençant à l'indice 0).

▷ **Question 8. [OPTIONNEL]**
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 Θ(`len(l)`). En revanche, "`l.append(n)`" modifie la liste "`l`" en place avec un coût (amorti) constant. Pour éviter que les modifications en place ne soient visibles de l'extérieur, on peut commencer par dupliquer `l` (e.g. avec `l = list(l)`) avant le premier calcul d'expression : cette copie n'aura lieu qu'une seule fois, à l'initialisation, au lieu d'avoir lieu à chaque calcul d'expression.