TD d'analyse syntaxique: préparation au TP1

L'objectif du TP1 est de 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 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 :
$\texttt{PLUS} = \{\texttt{'+'}\}$ $\texttt{MINUS} = \{\texttt{'-'}\}$ $\texttt{MULT} = \{\texttt{'*'}\}$ $\texttt{DIV} = \{\texttt{'/'}\}$ $\texttt{QUEST} = \{\texttt{'?'}\}$

On utilise aussi le token $\texttt{NAT} = \{\texttt{'0'}, \ldots, \texttt{'9'}\}^+$ qui correspond aux entiers naturels en base 10 et le token $\texttt{CALC} = \{\texttt{'\#'}\}.\texttt{NAT}$ qui correspond aux valeurs des calculs déjà effectués. Ces deux tokens ont les profils d'attributs $\texttt{NAT}{\uparrow}\mathbb{N}$ et $\texttt{CALC}{\uparrow}\mathbb{N}$ 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`).

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 élèment 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) ? 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.

1 Programmation 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

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

où $\texttt{SEP} = \{\texttt{' '}, \texttt{'\textbackslash t'}, \texttt{'\textbackslash n'}\}$ 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 :

```
TOKEN_PREFIX = ('?', '+', '-', '*', '/', '', '#', '')
TOKENS = tuple(range(len(TOKEN_PREFIX)))
QUEST, PLUS, MINUS, MULT, DIV, NAT, CALC, END = TOKENS
```

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 TOKENS
    if t in (NAT, CALC):
        assert type(v) is int and v >= 0
    else:
        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 utiliser l'automate A de $\mathcal{L}$ pour résoudre ce dernier problème : il suffit de lire la séquence de caractères jusqu'à tomber dans un état d'erreur de A (i.e. un état puits). Alors, soit il n'y aucun préfixe appartenant à $\mathcal{L}$ si aucun état final n'a été rencontré, soit les caractères lus jusqu'au dernier état final 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)[2]. On suppose ici qu'il 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`.

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

Après avoir consommé les séparateurs, cette fonction reconnaît un token à l'aide de la fonction `get_token`. 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 terminal à retourner.[3] Pour cela, `get_token` appelle pour les terminaux sans attribut la fonction `parse_others` qui utilise le dictionnaire `TOKEN_MAP` associant un caractère à son token (sans attribut). 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 `get_token`. 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.La fonction `parse_others` utilisée pour les terminaux sans attribut doit être corrigée.

2. pré-vision et non post-lecture, car en fin de compte c'est le premier caractère *déjà lu* pour la prochaine lecture

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

```python
def parse_END():
    return (END, None)

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

def parse_others():
    print("@ATTENTION: lexer.parse_others à corriger !") # LIGNE A SUPPRIMER
    raise KeyError

def get_token():
    print("@ATTENTION: lexer.get_token à finir !") # LIGNE A SUPPRIMER
    if is_digit(current_char):
        return parse_NAT()
    elif current_char == TOKEN_PREFIX[END]:
        return parse_END()
    else:
        return parse_others()
```

FIGURE 1 – Squelette de code pour l'analyseur lexical

▷ **Question 5.** Corriger le code fourni de `parse_NAT`, de `parse_others` et de `get_token`, écrire celui de `parse_CALC`. ◁

2 Programmation de l'analyseur syntaxique (À PRÉPARER EN TEMPS LIBRE AVANT LA SÉANCE DE TP)

L'analyseur de la calculette peut s'implémenter en suivant les principes de l'analyse LL(1) donnés au chapitre 3 du cours. Pour vous donner un exemple, le fichier `pcalc.py` contient un squelette de code de la calculette, qui traite en fait la BNF suivante :

$$\begin{aligned} \textbf{input}{\downarrow}\ell{\uparrow}\ell' &::= \quad \textbf{exp}{\downarrow}\ell{\uparrow}n \quad \ell' := \ell \oplus n \\ \textbf{exp}{\downarrow}\ell{\uparrow}n &::= \quad \texttt{NAT}{\uparrow}n \end{aligned}$$

▷ **Question 6.** 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). Exécutez `test_pcalc.py` pour tester votre calculette. ◁

▷ **Question 7.** 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. 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`. Pensez à relancer `test_pcalc.py` pour vérifier que votre calculette s'exécute comme attendu. ◁