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, avec un unique profil d'arguments, 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.

Autrement dit, on utilise les opérateurs suivants qui correspondent tous à un unique terminal
`PLUS` $:= \{$`'+'`$\}$ `MINUS` $:= \{$`'-'`$\}$ `MULT` $:= \{$`'*'`$\}$ `DIV` $:= \{$`'/'`$\}$ `QUEST` $:= \{$`'?'`$\}$
On utilise aussi le terminal `INT` $:= \{$`'0'`$, \ldots,$ `'9'`$\}^+$ qui reconnaît les entiers naturels en base 10 et le terminal `VAR` $:= \{$`'#'`$\}$.`INT` qui reconnaît les variables. Ces deux terminaux ont les profils d'attributs `INT`$\uparrow\mathbb{N}$ et `VAR`$\uparrow\mathbb{N}$ où $\mathbb{N}$ est l'ensemble des entiers naturels. Chacun de ces attributs correspond donc à l'entier en base 10 lu par l'analyseur lexical.

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$\downarrow\ell\uparrow n$ ::=	`INT`$\uparrow n$	
|	`VAR`$\uparrow i$	$n := \ell[i]$
|	`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 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.

1. Aussi appelée "*notation polonaise*", car inventée par le logicien polonais J. Łukasiewicz en 1924.

▷ **Question 2.** Donner un automate fini déterministe (mais éventuellement incomplet) sur le vocabulaire des caractères ASCII qui reconnaît le langage des lexèmes défini par

$$\mathcal{L} = \texttt{SPACES}^*.(\texttt{QUEST} \cup \texttt{PLUS} \cup \texttt{MINUS} \cup \texttt{MULT} \cup \texttt{DIV} \cup \texttt{INT} \cup \texttt{VAR} \cup \texttt{END})$$

où $\texttt{SPACES} = \{\texttt{' '}, \texttt{'\textbackslash n'}, \texttt{'\textbackslash t'}\}$ 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 ASCII.

◁

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 `INT` ou de `VAR`. On va donc raffiner l'automate précédent en une sorte de machine de Mealy qui produit une donnée de sortie en fonction de la suite de caractères en entrée. 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 :

```
TOKENS = tuple(range(8))
QUEST, PLUS, MINUS, MULT, DIV, INT, VAR, 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 (INT, VAR):
        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 des affectations sur des variables `t` et `v` de sorte que dans les états finaux de l'automate, `t` correspond au terminal reconnu et que si $\texttt{t} \in \{\texttt{INT}, \texttt{VAR}\}$ alors `v` correspond à la valeur décimale de l'entier lu. Typiquement, une transition de cette machine de Mealy aura une étiquette de la forme "D / $\texttt{t} := e$" où D est le sous-ensemble de caractères ASCII qui déclenchent la transition, et e est l'expression alors calculée dans la variable t. On peut raffiner cette forme en "$c \in D$ / $\texttt{v} := f; \texttt{t} := e$" où c est le nom du caractère de D effectivement lu, utilisable dans l'expression "f". Pour simplifier, on assimilera abusivement le caractère correspondant à un chiffre (comme `'2'`) avec le nombre correspondant à ce chiffre (comme 2).

◁

L'automate considéré jusqu'ici décide si *une suite de caractères ASCII donnée appartient à* $\mathcal{L}$. Mais notre analyseur lexical cherche à résoudre un problème un peu différent : dans la suite de caractères (non encore lus), « *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 suite de caractères jusqu'à tomber dans un état d'erreur de A (c-à-d. 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 de manière à celle qu'elle reconnaisse *uniquement le plus long préfixe* de l'entrée qui appartient à $\mathcal{L}$. Dans le cas des entiers et des variables, la machine est obligée de lire un caractère de trop. Du coup, pour simplifier, on prendra la convention que la machine lit *systématiquement* le caractère qui *suit* le lexème reconnu (sauf évidemment dans le cas où ce lexème est `END`).

◁

Le caractère qui suit le lexème reconnu est appelé *caractère de pré-vision* (*look-ahead* en anglais). On suppose ici qu'il correspond à une variable globale `current` qui peut être positionnée sur le prochain caractère non-lu 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 premier terminal de l'entrée : elle suppose que le premier caractère du terminal à lire se trouve déjà `current`.

```
def next_token():
    while current in SPACES:
        update_current()
    try:
        t, parser = SWITCH[current]
        return parser(t)
    except KeyError:
        raise Error('Unknown start of token')
```

Après avoir consommé les séparateurs, cette fonction utilise un dictionnaire `SWITCH` pour associer une transition de l'automate à chaque caractère qui démarre la reconnaissance du terminal proprement dit. On exploite ici le fait que le premier caractère qui suit un séparateur détermine le terminal à retourner. [2] L'accès au dictionnaire `SWITCH[current]` retourne donc un couple `(t, parser)` où `t` est le terminal à retourner et `parser(t)` va consommer tous les caractères associés à ce terminal et retourner le terminal attribué `(t, v)`.

Le fichier `lexer.py`, fourni pour le TP, contient déjà un squelette de la fabrication de `SWITCH` : ce dictionnaire est fabriqué une fois pour toutes pendant l'initialisation de l'analyseur lexical à l'aide de la fonction `mk_switch` (à compléter) donnée figure 1. En l'état, celle-ci ne traite que deux terminaux `END` and `INT`. Dans le cas de `INT`, c'est la fonction `parse_INT` qui doit s'occuper de la reconnaissance de l'entier à fabriquer. En l'état celle-ci 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. [3]

▷ **Question 5.** Corriger le code fourni de `parse_INT` et de `mk_switch`.

◁

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

3. La fonction `parse_digit` est donc très similaire dans son principe à la fonction `parse_token` introduite au chapitre 3 pour programmer les analyseurs LL(1), et en particulier l'analyseur syntaxique de la section 2. Ce n'est pas un hasard : l'analyse syntaxique LL(1) généralise la variante de machines de Mealy utilisée ici. De même, `mk_switch` est assez similaire à `mk_rules`.

```
def parse_END(t):
    return (t, None)

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

def mk_switch():
    print("@ATTENTION: lexer.mk_switch à finir !") # LIGNE A SUPPRIMER
    d = { '': (END, parse_END) }
    for digit in range(10):
        d[str(digit)] = (INT, parse_INT)
    return d

SWITCH = mk_switch()
```

FIGURE 1 – Squelette de code pour l'analyseur lexical

2 Programmation de l'analyseur syntaxique

(A PREPARER EN TEMPS LIBRE AVANT LA SEANCE DE TP)

L'analyseur de la calculette peut s'implémenter en suivant les principes de l'analyse LL(1) donné 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 :

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

▷ **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. Attention, l'accès "$\ell[i]$" doit alors être implémenté en PYTHON par "`l[i-1]`" (les listes PYTHON commençant à l'indice 0). En TP, exécutez `test_pcalc.py` pour vérifier que votre calculette s'exécute comme attendu.

◁

▷ **Question 7.** Améliorer la calculette en implémentant l'opération sur les listes "$\ell \oplus n$" plus efficacement par le code PYTHON "`l.append(n)`". En effet, "`l+[n]`" créé une nouvelle liste en recopiant intégralement "`l`", donc à coût Θ(`len(l)`). Au contraire, "`l.append(n)`" modifie la liste "`l`" en place avec un coût (amorti) constant. Il devient donc inutile que `parse_input` retourne ℓ', puisqu'en sortie la liste initiale est 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.

◁