TD d'analyse syntaxique : préparation au TP1

L'objectif du TP1 est de programmer en C 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. Elle affiche au fur et à mesure chaque nouveau résultat inséré via l'opération $\oplus$ dans la liste synthétisée (comme dans l'implémentation en YACC du chapitre 2). La notation "$\ell[i]$" 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**$\downarrow\ell\uparrow n_1$ **exp**$\downarrow\ell\uparrow n_2$	$n := n_1 - n_2$
|	`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 ASCII qui reconnaît le langage des lexèmes défini par

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

où $\texttt{SPACE} = \{\text{`}\ \text{'}, \text{`}\backslash\texttt{n}\text{'}, \text{`}\backslash\texttt{t}\text{'}\}$ et $\texttt{END} = \{\text{EOF}\}$ (ici EOF est le caractère ASCII spécial pour marquer 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 un entier 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 sortie en fonction de la suite de caractères en entrée. Pour se rapprocher du programme C final, on code ces sorties dans des variables. Les terminaux sont définis comme les valeurs du type énuméré `token` suivant. On définit aussi 2 variables globales :

```
typedef enum { INT, VAR, QUEST, PLUS, MINUS, MULT, DIV, END } token;
token t; int v;
```

▷ **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 "$c \in D$ / $\texttt{v} := f(c); \texttt{t} := e$" où c est le nom du caractère de D étant effectivement lu, utilisable dans l'expression "$f(c)$" calculée dans la variable v. Pour simplifier, on assimilera abusivement le caractère correspondant à un chiffre (comme `'2'`) avec le nombre correspondant à ce chiffre (comme 2).

◁

▷ **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 standard grâce à la procédure `update_current()`.

```
char current;
void update_current();
```

▷ **Question 5** (A PREPARER EN TEMPS LIBRE AVANT LA SEANCE DE TP)**.** Transformer la machine de Mealy précédente en procédure `next` qui positionne ses paramètres de sortie `*t` et `*v` conformément au premier terminal de l'entrée : elle suppose que le premier caractère du terminal à lire se trouve déjà `current`. En cas d'erreur lexicale, la procédure interrompt l'exécution du programme en invoquant `exit(1)` (en ayant au préalable affiché un message d'erreur à l'utilisateur).

```
void next(token *t, int *v);
```

◁

2 Programmation de l'analyseur syntaxique

Pour écrire le code de l'analyseur syntaxique, on exploite la propriété ci-dessous, caractéristique des BNF de notation préfixe (cf. Chapitre 3 du CM).

> Soit $\mathcal{V}_T = \{\texttt{INT}, \texttt{VAR}, \texttt{QUEST}, \texttt{PLUS}, \texttt{MINUS}, \texttt{MULT}, \texttt{DIV}, \texttt{END}\}$. Pour tout mot w de $\mathcal{V}_T^*$, il existe *au plus* un préfixe u de w tel que $u \in$ **exp** ou $u \in$ **input**.

▷ **Question 6** (A PREPARER EN TEMPS LIBRE AVANT LA SEANCE DE TP)**.** En utilisant l'analyseur lexical précédent (supposé initialisé), écrire une procédure *récursive* qui reconnaît l'unique préfixe u dans le flot de terminaux qui dérive de **exp** et calcule la sémantique associée $\textbf{exp}{\downarrow}l{\uparrow}v$. Dans le cas où un tel préfixe n'existe pas, la procédure interrompt l'exécution du programme avec un message d'erreur. Sinon, elle retourne le v obtenu.

```
int parse_exp(list l);
```

◁

▷ **Question 7** (A PREPARER EN TEMPS LIBRE AVANT LA SEANCE DE TP)**.** Même question que précédemment mais pour un mot de $\textbf{input}{\downarrow}l{\uparrow}l'$. Ici, comme on utilise l'opérateur `append` avec effets de bord pour implémenter $\oplus$, la valeur de l' se trouve, après l'exécution de la procédure, dans le paramètre `l` (la valeur initiale l de `l` est donc perdue).

```
void parse_input(list l);
```

Dans un premier temps, on peut utiliser une procédure récursive. Dans un deuxième temps, en remarquant que cette procédure est *récursive terminale*, on peut remplacer la récursion par une simple boucle.

◁