

TP PLC n°3 Analyse lexicale et syntaxique

Ce TP est à faire sur 1 séance de 2h. Un compte-rendu est à rendre au plus tard pour le vendredi 27 octobre 2006 à 8h. Le compte-rendu, contenant des explications et le code Sicstus du TP, est à envoyer par mail à belleann@irisa.fr et ridoux@irisa.fr sous forme d'un fichier PDF.

1 Analyser le langage du calcul des propositions

On désire écrire un programme PROLOG pour lire des formules logiques du calcul des propositions et pour les représenter en PROLOG sous forme de termes. Les formules logiques lues en entrée respectent la syntaxe concrète suivante :

<i>expr</i>	→	<u>non</u> <i>expr</i>
		<i>expr</i> <i>opérateur-logique</i> <i>expr</i>
		(<i>expr</i>)
		<i>variable-propositionnelle</i>
<i>opérateur-logique</i>	→	<u>et</u> <u>ou</u>
<i>variable-propositionnelle</i>	→	<i>identificateur</i>

Un *identificateur* est une suite de lettres minuscules, à l'exclusion des identificateurs réservés non, et et ou qui représentent les opérateurs logiques. La priorité des opérateurs est habituelle :

$$\text{priorité}(\underline{\text{non}}) > \text{priorité}(\underline{\text{et}}) > \text{priorité}(\underline{\text{ou}})$$

de sorte que la chaîne d'entrée "a ou d et (b ou c)" représente la proposition :

$$a \vee (d \wedge (b \vee c))$$

Rappel: une chaîne de caractères est assimilée par PROLOG à la liste des codes ASCII de ces caractères. Ainsi, la chaîne "a ou xb" est équivalente à la liste [O'a, O', O'o, O'u, O', O'x, O'b]. Le parcours de la chaîne d'entrée se fera sur la liste associée.

Les questions qui suivent vont vous permettre de construire progressivement un analyseur, lexical puis syntaxique, pour le langage du calcul des propositions.

1.1 Analyse lexicale

Il s'agit, avant d'entreprendre l'analyse syntaxique de la phrase, de transformer la chaîne d'entrée en une liste d'unités lexicales.

Question 1 *En suivant les consignes des sections 1.1.1 et 1.1.2, écrire le prédicat `analyse_lexicale(+S, -L)` tel que L est la liste de lexèmes qui correspond à la chaîne de caractères S . Les caractères «espace» n'ont pas d'autre signification qu'un rôle de délimiteur dans la chaîne d'entrée. Ils n'ont pas à être produits dans la liste de lexèmes. Voici deux exemples :*

Chaîne d'entrée	liste de lexèmes
"a et non(b ou c)"	$[id(a), op(et), op(non), sep(ouvp), id(b), op(ou), id(c), sep(fermp)]$
"non non a et (gauche ou droite)"	$[op(non), op(non), id(a), op(et), sep(ouvp), id(gauche), op(ou), id(droite), sep(fermp)]$

note Vous pourrez utiliser le prédicat prédéfini suivant :

- `name(?Const, +String)` tel que la chaîne de caractères *String* est le nom de l'atome *Const*.

Ainsi, l'exécution du but `name(C, "toto")` produit `{C = toto}`.

1.1.1 Spécifications de l'analyseur lexical

L'analyse lexicale de la chaîne d'entrée consiste à encoder cette chaîne en une liste de lexèmes. On vous demande d'utiliser trois types de lexèmes :

1. Les séparateurs : `(` et `)`.
2. Les opérateurs : `non`, `et` et `ou`.
3. Les identificateurs.

Avec les codages suivants :

<code>(</code>	<code>sep(ouvp)</code>
<code>)</code>	<code>sep(fermp)</code>
<code>non</code>	<code>op(non)</code>
<code>et</code>	<code>op(et)</code>
<code>ou</code>	<code>op(ou)</code>
<code>toto</code>	<code>id(toto)</code>

En cas d'ambiguïté pour le découpage de la chaîne d'entrée, on choisira la plus longue chaîne. Ainsi, la chaîne "nona et a" sera lue comme la suite des lexèmes 'nona', 'et', 'a' (et non comme 'non', 'a', 'et', 'a')

1.1.2 Réalisation de l'analyseur lexical

Pour vous guider dans la réalisation de l'analyseur lexical, voici les entêtes de prédicats que vous êtes invités à programmer.

```
% lettre( +Char )  
réussit si Char est le code ASCII d'une lettre minuscule. Ainsi l'exécution du but  
lettre(0'a) réussit. lettre(97) aussi.
```

```
% identificateur( +Chaine, -Identificateur, -ResteChaine )  
réussit si la chaîne Chaine commence par l'identificateur Identificateur (une liste  
de lettres minuscules) et se poursuit par la chaîne ResteChaine. Ainsi l'exécution  
du but identificateur("toto et non b",I,R) réussit avec {I = "toto", R =  
" et non b"}.
```

```
% filtre( +Identificateur, -UniteLexicale )  
réussit si le lexème UniteLexicale correspond à la chaîne Identificateur. Lec-  
ture opérationnelle du prédicat: filtre transforme une chaîne, représentant un  
opérateur ou un identificateur, en un lexème associé.  
Ainsi, si Identificateur désigne un opérateur, UniteLexicale sera de la forme  
op(_), sinon il sera de la forme id(_). Ainsi l'exécution du but filtre("toto",U)  
réussit avec {U = id(toto)}.
```

```
% unite_lexicale( +Chaine, -UniteLexicale, -ResteChaine )  
réussit si la chaîne Chaine commence par une unité lexicale, éventuellement précédée  
d'espaces, correspondant au lexème UniteLexicale, et se poursuit par ResteChaine.  
Ainsi l'exécution du but unite_lexicale(" toto et non b",U,R) réussit avec {U  
= id(toto), R = " et non b"}.
```

```
% analyse_lexicale( +Chaine, -ListeUnitesLexicales )  
réussit si la chaîne Chaine (une liste de codes ASCII) correspond à la liste de lexèmes  
ListeUnitesLexicales.  
Ainsi l'exécution du but analyse_lexicale("toto et non(a)", L) réussit avec  
{L = [id(toto),op(et),op(non),sep(ouvvp),id(a),sep(fermp)]}.
```

1.1.3 Test du programme

Il est utile d'écrire un programme de test qui permet de tester de nombreuses valeurs d'entrée, sans avoir à saisir chaque nouvelle valeur en ligne de commande SICSTUS-PROLOG. Le prédicat autotest ci-dessous est un exemple d'un tel programme.

```
% autotest : y mettre les exemples donnés en complément de spécification  
% plus tous les problèmes rencontrés lors de la mise au point  
autotest :-  
    member( Chaine,  
% cas de test. ATTENTION!!! member doit être écrit pour le mode(-,+)  
    ["a et non(b ou c)",  
    "a et b et c",  
    "non non a 22 (gauche ou droite)",  
    "non non a et (gauche ou droite)",  
    "non non a et (gauche ou droite)      "  
    ] ) ,  
% pilote de test
```

```

format( "AUTOTEST:analex", [] ) , nl ,
format( " + ~s + ", [ Chaine ] ) , nl ,
( analyse_lexicale( Chaine, ListesUnitesLexicales )
-> format( " - ~p - ", [ ListesUnitesLexicales ] ) , nl
; format( " NO ", [] ) , nl
) ,
fail .
% arrêt quand tout les tests ont été effectués
autotest .

```

1.2 Analyse syntaxique

Partant d'une liste de lexèmes telle que définie ci-dessus, nous allons maintenant construire un analyseur syntaxique du langage du calcul des propositions.

Question 2 *Donnez une grammaire, d'axiome `expr`, pour le langage du calcul des propositions qui soit équivalente à la grammaire ci-dessus, qui prenne en compte les priorités d'opérateurs et qui puisse servir de base pour le codage en PROLOG d'un analyseur du langage.*

... et voici une réponse!

$$\begin{array}{ll}
expr & \longrightarrow expr_ou \underline{op(ou)} expr \mid expr_ou \\
expr_ou & \longrightarrow expr_et \underline{op(et)} expr_ou \mid expr_et \\
expr_et & \longrightarrow \underline{op(non)} expr_et \mid expr_primaire \\
expr_primaire & \longrightarrow \underline{sep(ouvp)} expr \underline{sep(fermp)} \mid \underline{id(ident)}
\end{array}$$

En voici même une autre, qui a l'avantage d'être LL(1)

$$\begin{array}{ll}
expr & \longrightarrow expr_ou \ suite_ou \\
suite_ou & \longrightarrow \underline{op(ou)} expr_ou \ suite_ou \mid \epsilon \\
expr_ou & \longrightarrow expr_et \ suite_et \\
suite_et & \longrightarrow \underline{op(et)} expr_et \ suite_et \mid \epsilon \\
expr_et & \longrightarrow \underline{op(non)} expr_et \mid expr_primaire \\
expr_primaire & \longrightarrow \underline{sep(ouvp)} expr \underline{sep(fermp)} \mid \underline{id(ident)}
\end{array}$$

Question 3 *Exprimer cette deuxième grammaire sous forme de dcg afin d'obtenir un analyseur des formules du calcul des propositions.*

Ainsi l'appelphrase(expr, [id(a), op(ou), op(non), id(c)]) doit réussir.