

## Analyse récursive descendante - suite

### 1 La grammaire et sa transformation

Le but de ce TP est de réaliser un analyseur récursif LL(1) pour des expressions booléennes à la *Java*. Plus précisément, les expressions à analyser sont les expressions engendrées par la grammaire LL(1) suivante, d'axiome  $E$  et d'ensemble de terminaux  $\{ \&\& , || , ! , ( ) , c , i \}$

$$\begin{aligned} E &\longrightarrow T E' \\ E' &\longrightarrow || T E' \mid \varepsilon \\ T &\longrightarrow F T' \\ T' &\longrightarrow \&\& F T' \mid \varepsilon \\ F &\longrightarrow ! F \mid ( E ) \mid c \mid i \end{aligned}$$

Cette grammaire a pour table d'analyse LL(1) :

|        | $E$                 | $E'$                         | $T$                 | $T'$                         | $F$                   |
|--------|---------------------|------------------------------|---------------------|------------------------------|-----------------------|
| $c$    | $E \rightarrow TE'$ |                              | $T \rightarrow FT'$ |                              | $F \rightarrow c$     |
| $i$    | $E \rightarrow TE'$ |                              | $T \rightarrow FT'$ |                              | $F \rightarrow i$     |
| $($    | $E \rightarrow TE'$ |                              | $T \rightarrow FT'$ |                              | $F \rightarrow ( E )$ |
| $!$    | $E \rightarrow TE'$ |                              | $T \rightarrow FT'$ |                              | $F \rightarrow ! F$   |
| $\&\&$ |                     |                              |                     | $T' \rightarrow \&\& FT'$    |                       |
| $  $   |                     | $E' \rightarrow    TE'$      |                     | $T' \rightarrow \varepsilon$ |                       |
| $)$    |                     | $E' \rightarrow \varepsilon$ |                     | $T' \rightarrow \varepsilon$ |                       |
| $\#$   |                     | $E' \rightarrow \varepsilon$ |                     | $T' \rightarrow \varepsilon$ |                       |

### 2 L'analyseur lexical

Voici la définition des tokens de l'analyseur lexical :

| Token         | Token value | texte                           |
|---------------|-------------|---------------------------------|
| CONSTANT      | booléen     | true   false (case insensitive) |
| IDENT         | chaîne      | [A-Za-z](_?[A-Za-z0-9])*        |
| NOT           | chaîne      | !                               |
| OR            | chaîne      |                                 |
| AND           | chaîne      | &&                              |
| OPEN_BRACKET  | chaîne      | (                               |
| CLOSE_BRACKET | chaîne      | )                               |

Écrivez un analyseur lexical répondant à cette spécification. L'analyseur ignorera les espaces, tabulations et fins de ligne entre tokens.

Testez-le

### 3 L'analyseur syntaxique LL(1)

Comme lors de la séance précédente, l'analyseur utilisera la programmation récursive descendante, en se fondant sur la table LL(1).

La table d'analyse LL(1), une fois adaptée à la définition des tokens devient :

|               | $E$                 | $E'$                            | $T$                 | $T'$                             | $F$  |
|---------------|---------------------|---------------------------------|---------------------|----------------------------------|--|
| CONSTANT      | $E \rightarrow TE'$ |                                 | $T \rightarrow FT'$ |                                  | $F \rightarrow \text{CONSTANT}$                    |
| IDENT         | $E \rightarrow TE'$ |                                 | $T \rightarrow FT'$ |                                  | $F \rightarrow \text{IDENT}$                       |
| OPEN_BRACKET  | $E \rightarrow TE'$ |                                 | $T \rightarrow FT'$ |                                  | $F \rightarrow \text{OPEN\_B } E \text{ CLOSE\_B}$ |
| NOT           | $E \rightarrow TE'$ |                                 | $T \rightarrow FT'$ |                                  | $F \rightarrow \text{NOT } F$                      |
| AND           |                     |                                 |                     | $T' \rightarrow \text{AND } FT'$ |  |
| OR            |                     | $E' \rightarrow \text{OR } TE'$ |                     | $T' \rightarrow \varepsilon$     |  |
| CLOSE_BRACKET |                     | $E' \rightarrow \varepsilon$    |                     | $T' \rightarrow \varepsilon$     |  |
| EOD           |                     | $E' \rightarrow \varepsilon$    |                     | $T' \rightarrow \varepsilon$     |  |

L'analyseur descendant « simple » (sans actions sémantiques) vous est fourni .  
Testez-le avec votre analyseur lexical.

## 4 Action sémantique 1 : évaluation d'expression

Écrivez une classe `Exobool_eval` qui implémente un parser d'expressions booléennes (sur le modèle de celui fourni) mais qui, en plus, renverra la valeur de l'expression (et donc un résultat booléen). Comme on n'a pas implémenté de dictionnaire associant une valeur aux identificateurs , on supposera pour cette question que tout identificateur possède la valeur `False`.  
Testez bien évidemment ...

## 5 Actions sémantiques 2 : nombre de variables, profondeur

1. Écrivez une classe `Exobool_vars` dont le parser renvoie le nombre d'occurrences de variables figurant dans l'expression. Exemple : pour `a && b && (TRUE || a || c)` on obtient 4.
2. Écrivez une classe `Exobool_nested` dont le parser renvoie le niveau maximal d'imbrications de parenthèses Exemples :
  - pour `a && b && (TRUE || a || c)` on obtient 1.
  - pour `a && b && TRUE || a || c` on obtient 0
  - pour `( a && b ) || (a && (b || c))` on obtient 2

## 6 Action sémantique 3 : traduction d'expression

Écrivez une classe `Exobool_postfix` implémentant un traducteur en une expression postfixée. En plus de la vérification de l'expression il produit, dans le cas où elle est correcte, une chaîne contenant une expression équivalente, mais en notation postfixée.  
Les constantes seront toutes en minuscules et les opérateurs s'écriront `and`, `or` et `not`.  
Exemples :

`a&&b` se traduira en `a b and`  
`a&&b&&c` se traduira en `a b and c and`  
`a||b&&c` se traduira en `a b c and or`  
`(a||b)&&c` se traduira en `a b or c and`  
`(a||b)&&(c||d)` se traduira en `a b or c d or and`