TDM 9 - Automates et Langages

février 2022

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` }

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

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

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

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	$E \rightarrow TE'$		$T \rightarrow FT'$		$F \rightarrow$ CONSTANT
IDENT	$E \rightarrow TE'$		$T \rightarrow FT'$		$F \rightarrow$ IDENT
OPEN_BRACKET	$E \rightarrow TE'$		$T \rightarrow FT'$		$F \rightarrow$ OPEN_B E CLOSE_B
NOT	$E \rightarrow TE'$		$T \rightarrow FT'$		$F \rightarrow$ NOT F
AND				$T' \rightarrow$ AND FT'	
OR		$E' \rightarrow$ 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 Action sémantique 2 : traduction d'expression

Écrivez une classe `Exobool_postfix` implémentant un traducteur en une expression postifixée.
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`