TDM 9 - Automates et Langages

novembre 2018

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 suivante, d'axiome E et d'ensemble de terminaux { `&&` , `||` , `!` , `(` `)` , `value` } (value désigne une valeur booléenne quelconque)

$$E \longrightarrow E \texttt{ \&\& } E \mid E \texttt{ || } E \mid \texttt{! } E \mid \texttt{(} E \texttt{)} \mid \texttt{value}$$

Cette grammaire, bien commode pour donner de manière compacte la syntaxe des expressions à traiter, est clairement ambiguë. En s'appuyant sur les priorités des opérateurs et après élimination de la récursivité gauche, on obtient la grammaire LL(1) équivalente suivante :

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

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

	E	E'	T	T'	F
`value`	$E \to TE'$		$T \to FT'$		$F \to$ `value`
`(`	$E \to TE'$		$T \to FT'$		$F \to$ `(` 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

On travaillera en réalité avec un analyseur lexical qui assurera le découpage du texte en *tokens*. Un analyseur, spécifié par un fichier **JFlex**, vous est fourni et vous n'avez donc pas à l'écrire.
Chaque type de tokens est implémenté par une classe spécifique. Chacune d'elle implémente :

```
public interface Yytoken {
 /**
  *   token source text
  */
 String image();
 /**
  * type of the token (enum)
  */
 TokenType getType();
}
```

`TokenType` est un type énuméré associant à chaque token un symbole unique. Son utilisation simplifiera la rédaction de l'analyseur syntaxique. Voici la correspondance entre les symboles du langage formel et les tokens de l'analyseur lexical :

Symbole formel	`TokenType`	`Classe`	Valeurs possibles
value	`CONSTANT`	`Constant`	`true` | `false` | `TRUE` | `FALSE`
value	`IDENT`	`Ident`	`[A-Za-z](_?[A-Za-z0-9])*`
!	`NOT`	`Not`	`!`
||	`OR`	`Or`	`||`
&&	`AND`	`And`	`&&`
(	`OPEN_BRACKET`	`OpenBracket`	`(`
)	`CLOSE_BRACKET`	`CloseBracket`	`)`
$	`EOD`	`EndOfData`	*fin des données*

On constate qu'une valeur peut être soit une constante (`true` ou `false`) soit un identificateur de variable. L'analyseur lexical vérifie également les particularités suivantes :

1. En fin de texte, il renvoie le token `EOD` (`EndOfData`).
2. En cas d'erreur lexicale (texte ne pouvant pas constituer un token), une exception `LexicalException` est déclenchée.

Un programme de test de l'analyseur lexical vous est fourni, bien qu'il ne soit pas, a priori, indispensable pour le travail à faire.

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`		$E' \rightarrow \varepsilon$		$T' \rightarrow \varepsilon$	

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). Plusieurs éléments vous sont fournis, notamment :

— la classe `AbstractParser` implémente quelques opérations de base communes à tous les analyseurs. Vous noterez une évolution par rapport à la version de la semaine précédente : la classe est générique pour permettre à la méthode `parse()` de renvoyer une valeur. Pour écrire un parser, il reste à étendre la classe `AbstractParser`, l'extension devant notamment implémenter la méthode `axiom()`

— la classe `BooleanExpParser` est la classe étendant `AbstractParser` dédiée à l'analyse fondée sur la grammaire ci-dessus. La version qui vous est fournie ... **est à compléter**.

— la classe `TestParser` est une classe de test du Parser.

4 Étape 1

Compléter la classe `BooleanExpParser` et réalisez **des** tests (portant à la fois sur des expressions correctes et des expressions incorrectes).

5 Étape 2

La classe `BooleanExpTranslator` est destinée à implémenter un traducteur d'expression. En plus de la vérification de l'expression il produit, dans le cas où elle est correcte, une version traduite dans laquelle les constantes sont toutes en minuscules, les opérateurs s'écrivent `and`, `or` et `not`. Cette fois encore, la classe fournie est à compléter et à tester.

6 Réduction d'expression

On veut maintenant simplifier ces expressions en fonction de l'occurrence des constantes booléennes `vrai` ou `faux`. On utilsera notamment les propriétés booléennes suivantes, valables pour tout e :

- `true` && e se simplifie en e, de même pour e && `true`
- `false` && e se simplifie en `false` , de même pour e && `false`
- `false` || e se simplifie en e, de même pour e || `false`
- `true` || e se simplifie en `true` , de même pour e || `true`
- (`true`) se simplifie en `true` , et (`false`) en `false`

Par exemple, `(Riche ou vrai) et Vieux ou non X ou Grand ou (G et non (vrai ou X))` se simplifie en `Vieux ou non X ou Grand` tandis que `(vrai ou Riche) ou (P et non Q)` sera transformée en `vrai`.

Attention, on ne fait pas toutes les simplifications possibles ! Par exemple, `(Truc ou non Truc)` sera inchangé (et non simplifié en `vrai`). De même, on peut admettre que `(vrai et X)` soit transformé en `(X)` et non en `X`.
Il s'agit donc de transformer l'analyseur syntaxique ou le traducteur précédent pour en faire une application qui prend en entrée une expression de départ non simplifiée et qui affiche sur la sortie standard l'expression simplifiée correspondant s'il n'y a pas d'erreur de syntaxe dans l'expression d'entrée et signale une erreur sinon.

6.1 Dernière étape

La classe `BooleanExpReductor` est le squelette d'un analyseur/réducteur d'expressions booléennes, **à compléter, puis à tester**. On remarquera que le résultat de la méthode `axiom()` est encore `String` (c'est le résultat final), mais il faudra définir une ou plusieurs nouvelles classes pour représenter les résultats des autres méthodes.

NB : Un fois réalisées les simplifications demandées, vous pouvez ajouter les simplifications suivantes, (id désigne un identificateur) :

- (id) se simplifie en id.
- id && id se simplifie en id, de même pour id || id

Ces simplifications sont également faciles à implémenter