TDM 8 - Automates et Langages

novembre 2018

Analyse récursive descendante

L'**analyse syntaxique** est le processus qui consiste à vérifier si un mot peut être engendré par une grammaire, en trouvant l'arbre de dérivation qui permet de l'engendrer.
Un **analyseur syntaxique** est un logiciel dont la donnée est un « mot » (texte à analyser) et dont le résultat indique si ce texte est correct (c'est à dire engendré par la grammaire). En plus de ce travail de vérification syntaxique, l'analyseur déclenche le plus souvent des actions **sémantiques** guidées par le texte.
Par exemple un analyseur lit un texte devant correspondre à une expression arithmétique (vérification syntaxique) et calcule en même temps la valeur de l'expression (action sémantique).
L'**analyse récursive descendante** désigne une catégorie d'algorithmes d'analyse syntaxique.
Son principe est de créer un « sous-programme » pour chaque variable de la grammaire. Selon le langage de programmation utilisé, ce sous-programme sera une fonction, une méthode, une procédure... Par la suite nous parlerons de **méthode** puisque nous programmerons en Java.
L'implémentation de chaque méthode découle directement des règles existant pour la variable correspondante.
Supposons par exemple que pour une variable S on dispose des règles $S \to XY$ et $S \to aS$, l'implémentation de `S()` ressemblera à

```
S(){
  if (condition à déterminer)
    //appliquer S->XY
          X(); Y();
  else if (condition à déterminer)
   // appliquer S->aS
   Vérifier que la lettre à lire est 'a' et la consommer;
   S();
  else
   déclencher une erreur;
}
```

Comme l'on voit, il nous faut encore déterminer dans quel cas appliquer l'une ou l'autre règle, ou aucune des deux (erreur). Nous verrons plus loin que la réponse sera dans la **table d'analyse LL**.

1 Grammaires LL(1)

L'avantage de l'analyse récursive descendante est de fournir un algorithme simple et efficace ne nécessitant qu'une seule lecture du texte à analyser.
Son principal inconvénient est de ne pas pouvoir s'appliquer à toutes les grammaires : seules les grammaires vérifiant un propriété particulière (appelée LL(1)) pourront donner lieu à une analyse récursive descendante.
Nous verrons en cours comment déterminer si une grammaire convient et, si oui, comment construire sa table d'analyse LL(1). Retenez d'ores et déjà qu'une grammaire ambigüe ou récursive gauche n'est jamais LL(1).

1.1 Table d'analyse LL(1)

Cette table d'analyse est à deux dimensions. les clés sont les variables pour la première dimension et les symboles terminaux pour la deuxième.
Si V est une variable et x une tettre terminale, la case $T[V, x]$ de la table contient

- soit **une seule** règle, dont V est la partie gauche.
- soit rien (on note parfois dans ce cas le mot "erreur")

La table détermine quelle règle doit s'appliquer dans chaque circonstance.
Dans le cas de l'implémentation récursive descendante, elle définit le schéma d'implémentation des méthodes : pour la méthode `V()` , si la prochaine lettre à lire est x, on doit appliquer la règle figurant dans $T[V, x]$. Si $T[V, x]$ ne contient pas de règle, c'est que le mot ne peut être engendré et l'on déclenche une erreur.

2 Exemple

Considérons une grammaire et sa table d'analysee LL(1)

$S \to AB \mid Da$
$A \to aAb \mid \epsilon$
$B \to bB \mid \epsilon$
$D \to dD \mid e$

	S	A	B	D
a	$S \to AB$	$A \to aAb$	/	/
b	$S \to AB$	$A \to \epsilon$	$B \to bB$	/
d	$S \to Da$	/	/	$D \to dD$
e	$S \to Da$	/	/	$D \to e$
#	$S \to AB$	$A \to \epsilon$	$B \to \epsilon$	/

Le symbole # désigne le marqueur de fin du mot (symbole que l'on ne trouve qu'une seule fois, en fin de mot).
Voici maintenant l'implémentation complète de la méthode `S()` :

```
private void S() throws SyntaxException,ParserException {
        switch (current) {
        case 'a':
        case 'b':
        case END_MARKER:
                // S -> AB
                A();
                B();
                break;
        case 'd':
        case 'e':
                // S -> Da
                D();
                eat('a');
                break;
        default:
                // erreur
                throw new SyntaxException(ErrorType.NO_RULE,current);
        }
}
```

Les méthodes de l'analyseur ont accès au texte à analyser, qui est lu lettre par lettre, de gauche à droite :

- `current` : lettre courante
- `eat(caractère)` vérifie que le caractère passé en paramètre est bien le caractère courant. Si oui, la lecture progresse d'un caractère (`current` est donc modifié). Dans le cas contraire une erreur de syntaxe est signalée.

Chacune des méthodes est construite sur le même modèle. Chacune respecte les propriétés suivantes :

- une invocation de la méthode `V()` doit analyser une portion de texte pouvant être engendrée par la variable V.
- avant l'invocation de la méthode `V()`, `current` doit contenir la première lettre de la portion de texte à analyser par `V()`.
- à l'issue de l'exécution de `V()`, `current` contient la première lettre du texte qui SUIT la portion de texte qui vient d'être analysée.

Vous trouverez dans l'archive jointe le code complet d'un analyseur pour cette grammaire (classe `ArdExemple`).

3 Exercice 1

$S \rightarrow ERS \mid \epsilon$
$E \rightarrow L \mid (S)$
$R \rightarrow C \mid \epsilon$
$L \rightarrow a \mid b \mid c$
$C \rightarrow 0 \mid 1 \mid ... \mid 9$

	S	E	R	L	C
a,b,c	$S \rightarrow ERS$	$E \rightarrow L$	$R \rightarrow \epsilon$	$L \rightarrow a\|b\|c$	/
0,..,9	/	/	$R \rightarrow C$	/	$C \rightarrow 0\|1\|...\|9$
(	$S \rightarrow ERS$	$E \rightarrow (S)$	$R \rightarrow \epsilon$	/	/
)	$S \rightarrow \epsilon$	/	$R \rightarrow \epsilon$	/	/
#	$S \rightarrow \epsilon$	/	$R \rightarrow \epsilon$	/	/

3.1 Analyse simple

En prenant exemple sur la grammaire précédente, implémentez un analyseur récursif descendant en étendant la classe `Ard`.
Notez que pour alléger la lecture de la table d'analyse LL(1) on a rassemblé sur une même ligne des cas qui se traitent de façon identique (a, b, c ou $0, 1, ...9$). En théorie il aurait fallu écrire une ligne pour chaque caractère.

3.2 Actions sémantiques

Voici quelques exemples de mots du langage engendré : $aaab, a2b3a2, (a(bc)2)3(ba)2$
On peut voir chacun de ces mots comme une expression condensée décrivant une suite de lettres. Un chiffre indique combien de fois l'élément qui le précède immédiatement doit être répété. Par exemple

- `ba2` désigne le mot $ba^2 = baa$
- `(ba)2` désigne le mot $(ba)^2 = baba$
- `(a(bc)2)3(ba)2` désigne le mot $(a(bc)^2)^3(ba)^2 = (abcbc)^3(ba)^2 = abcbcabcbcabcbcbaba$

Les syntaxe des expressions se décline à partir des règles de la grammaire :

- une expression S est soit vide, soit constituée d'un élément E puis d'un facteur de répétition R puis d'une autre expression S.
- un élément E est soit une lettre L, soit une expression entre parenthèses.
- un facteur de répétition R est soit vide, soit un chiffre C.

Le mot développé correspondant à une expression est le mot équivalent qui ne contient ni parenthèses ni chiffre. Par exemple `baba` est le développé de `(ba)2`.
Vous allez transformer votre analyseur simple pour faire en sorte qu'il calcule le mot développé, au fur et à mesure de l'analyse.
Pour cela chaque méthode va renvoyer un résultat qui représente la valeur sémantique de la sous-expression qu'elle vient d'analyser.

1. `String L()` : renvoie une chaîne contenant le caractère.
2. `int C()` : renvoie la valeur du caractère, vu comme un chiffre décimal.
3. `int R()` : renvoie le nombre de répétitions représenté.
4. `String E()` : renvoie le mot développé correspondant à cette portion d'expression.
5. `String S()` : renvoie le mot développé correspondant à cette expression.

4 Exercice 2

En pratique les « lettres terminales » sont rarement de simples caractères, mais plutôt des tokens issus d'une analyse lexicale.
Vous implémenterez un nouveau parseur/évaluateur utilisant ces 2 niveaux d'analyse du texte :

1. écrivez un analyseur lexical jflex. Celui-ci distinguera les tokens suivants
   - `Entier` : entier décimal non signé. La valeur associée sera le nombre représenté.
   - `Lettre` : une lettre sera soit une lettre au sens usuel, soit un caractère quelconque précédé d'un \ (caractère d'échappement). La valeur associée sera une chaîne contenant la lettre (pour les lettres usuelles) ou le caractère échappé (dans le deuxième cas). Par exemple, la valeur de `a` est `"a"`, la valeur de `\3` est `"3"`
   - `Ouvrante` : parenthèse ouvrante
   - `Fermante` : parenthèse ouvrante
   - `EOD` : marqueur de fin.

Comme vous l'aviez fait lors des séances consacrées aux analyseurs lexicaux, vous créerez les classes nécessaires à la représentation des tokens.

2. dans la grammaire, on remplace les caractères `(` et `)` par les tokens `Ouvrante` et `Fermante` ; on supprime les variables L et C en les remplaçant par des symboles terminaux, respectivement les tokens `Lettre` et `Nombre`.

$S \rightarrow ERS \mid \epsilon$
$E \rightarrow Lettre \mid Ouvr\ S\ Ferm$
$R \rightarrow Nombre \mid \epsilon$

	S	E	R
Lettre	$S \rightarrow ERS$	$E \rightarrow Lettre$	$R \rightarrow \epsilon$
Nombre	/	/	$R \rightarrow Nombre$
Ouvr	$S \rightarrow ERS$	$E \rightarrow OuvrSFerm$	$R \rightarrow \epsilon$
Ferm	$S \rightarrow \epsilon$	/	$R \rightarrow \epsilon$
Eod	$S \rightarrow \epsilon$	/	$R \rightarrow \epsilon$

Créez un nouvel analyseur syntaxique / évaluateur, adapté du précédent mais qui cette fois lit des tokens et non plus directement des caractères.