Université de Lille — FACULTÉ DES SCIENCES ET TECHNOLOGIES — Département Informatique

Informatique

L3S5 Miage

Analyse Syntaxique – TP3

Arbres de dérivation

Ce TP a pour objectif de construire l'arbre de dérivation d'un mot quelconque pour un langage particulier, JSON. Ce langage est utilisé pour échanger des données structurées à travers le réseau indépendamment d'un langage de programmation et d'une architecture de processeurs (32/64 bits et Big/Little Endian).

JSON : JavaScript Object Notation

JSON est entièrement documenté dans la RFC (Request For Comments) rfc4627.txt que je vous invite à lire soigneusement. Vous y trouverez la grammaire de ce langage sous forme ABNF. La grammaire est reprise ici sous une forme qui vous est plus habituelle [1], $G_{JSON} = (\Sigma, V, S, P)$ avec :

$$
\begin{array}{rcl}
\Sigma & = & \left\{\begin{array}{l}\texttt{begin-array},\ \texttt{begin-object},\ \texttt{end-array},\ \texttt{end-object},\ \texttt{name-separator},\\ \texttt{value-separator},\ \texttt{false},\ \texttt{null},\ \texttt{true},\ \texttt{number},\ \texttt{string}\end{array}\right\} \\
V & = & \{JSONtext, Object, Array, Member, Value\} \\
S & = & JSONtext \\
P & = & \left\{\begin{array}{lcl} JSONtext & \rightarrow & Object \mid Array \\ Object & \rightarrow & \texttt{begin-object}\,(Member\,(\texttt{value-separator}\,Member)*)?\,\texttt{end-object} \\ Array & \rightarrow & \texttt{begin-array}\,(Value\,(\texttt{value-separator}\,Value)*)?\,\texttt{end-array} \\ Member & \rightarrow & \texttt{string}\,\texttt{name-separator}\,Value \\ Value & \rightarrow & \texttt{false} \mid \texttt{null} \mid \texttt{true} \mid Object \mid Array \mid \texttt{number} \mid \texttt{string} \end{array}\right.
\end{array}
$$

Question 1. *Créez un analyseur lexical avec JFlex pour l'ensemble Σ des terminaux du langage JSON définis ci-dessus. Vous pouvez utilisez le programme JsonScan.java qui affiche les terminaux jusque* `sym.EOF`*, ainsi que le fichier sym.java pour disposer des mêmes valeurs de constantes. Reprenez (ou consultez) également le fichier json.lex pour disposer des fonctions d'accès aux numéros de ligne/colonne dans la suite.*

Les terminaux sont définis dans la RFC, en voici un résumé :

```
ws = (
                %x20 /              ; Space
                %x09 /              ; Horizontal tab
                %x0A /              ; Line feed or New line
                %x0D                ; Carriage return
    ) *

begin-array = ws %x5B ws  ; [ left square bracket
```

1. Les terminaux sont en fonte `typewriter`, les non-terminaux ont une initiale en majuscule

```
begin-object    = ws %x7B ws  ; { left curly bracket

end-array       = ws %x5D ws  ; ] right square bracket

end-object      = ws %x7D ws  ; } right curly bracket

name-separator  = ws %x3A ws  ; : colon

value-separator = ws %x2C ws  ; , comma

false = %x66.61.6c.73.65   ; false

null  = %x6e.75.6c.6c      ; null

true  = %x74.72.75.65      ; true

number = (minus)? int (frac)? (exp)?

decimal-point = %x2E       ; .

digit1-9 = %x31-39         ; 1-9

e = %x65 | %x45            ; e E

exp = e ( minus | plus )? (DIGIT)+

frac = decimal-point DIGIT+

int = zero | ( digit1-9 DIGIT* )

minus = %x2D               ; -

plus = %x2B                ; +

zero = %x30                ; 0

string = quotation-mark (char)* quotation-mark

char = unescaped |
       escape (
               %x22 |          ; "    quotation mark  U+0022
               %x5C |          ; \    reverse solidus U+005C
               %x2F |          ; /    solidus         U+002F
               %x62 |          ; b    backspace       U+0008
               %x66 |          ; f    form feed       U+000C
               %x6E |          ; n    line feed       U+000A
               %x72 |          ; r    carriage return U+000D
               %x74 |          ; t    tab            U+0009
               %x75 (HEXDIG){4} )  ; uXXXX                U+XXXX

escape = %x5C              ; \

quotation-mark = %x22      ; "
```

```
unescaped = %x20-21 | %x23-5B | %x5D-10FFFF

DIGIT =  %x30-39 ; utilisez la classe [:digit:]
HEXDIG =  DIGIT | "A" | "B" | "C" | "D" | "E" | "F"
```

Les caractères Unicode sont préfixés en JFlex par `\u` ou `\U` au lieu de `%x` : `\udddd` pour un caractère sur 16 bits, et `\Uddddddd` pour un caractère sur 24 bits [2].

La figure 1 présente un exemple d'exécution du programme sur le fichier json sample.json.

Note. Pour ceux connaissant ant, un fichier build.xml permet de construire les exercices dans le répertoire `/tmp` depuis un répertoire source de votre choix. Consulter le contenu du fichier pour respecter la hiérarchie utilisée pour les sources.

Arbres de dérivation en Java : la classe `ADNode`

Nous vous fournissons une classe `ADNode` dans un paquetage `ad`, qui permet de représenter un arbre dont les nœuds ont un type et une valeur. Cette classe est générique sur ces types et valeurs. Concernant les arbres de dérivation, nous utiliserons deux types de nœuds :

- les nœuds correspondant aux règles de la grammaire : le type du nœud est un entier correspondant à une règle (fichier `rule.java`), la valeur du nœud est un entier représentant la position dans la règle de la dérivation en cours ;
- les nœuds correspondant aux terminaux (et qui seront les feuilles de l'arbre) : le type du nœud est un entier correspondant au symbole défini dans le fichier `sym.java`, sa valeur est la valeur du texte associé au lexème (soit la valeur de `yytext()`).

Ces nœuds sont respectivement implémentés dans les classes `RuleNode` et `LeafNode`. Un visualisateur d'arbre est également fourni dans le paquetage `ad` : il affiche sous forme de `treenode`, dans un composant graphique JTree, un arbre en utilisant la représentation textuelle de chaque nœud qui est fournie par la méthode `nodetoString()` (les méthodes `toString` de la classe `ADNode` donnent une représentation textuelle d'un nœud *et* de ses fils, inadéquate dans ce cas). La figure 2 présente l'arbre associé au fichier d'exemple précédent. Les `LeafNode` y sont représentés avec un nom symbolique entre parenthèses suivi du numéro du token associé et de la valeur textuelle reconnue par JFlex. Les `RuleNode` y sont représentés par le nom symbolique de la règle suivi de la position dans la règle de la dérivation en cours. Cette position est ici maximale pour tous les nœuds car la reconnaissance est terminée. Les `RuleNode` sont les seuls nœuds du `treenode` pouvant etre étendus (ce sont bien les seuls à posséder des fils). Dans la figure, le premier membre du deuxième objet du tableau JSON est complètement étendu.

Construction de l'arbre de dérivation d'un mot

Pour construire l'arbre de dérivation associé à un texte JSON, vous allez implémenter directement la dérivation gauche du mot à partir de la grammaire donnée ci-dessus. La dérivation se fait en lisant le mot de gauche à droite (par un appel à `next_token()`), on dispose ainsi toujours d'un token *en avance* [3], désigné par une variable `symb`. A un pas donné de la dérivation, la règle en cours de dérivation correspond au nœud `RuleNode` courant, désigné par une variable `current` : la règle en cours de dérivation est ainsi désignée par `current.getType()`, et la position dans le membre droit de la règle est donnée par la valeur associée à ce nœud, soit `current.getValue()`. A chaque pas de la dérivation, trois actions sont possibles :

2. Ne pas utiliser `Uddddddd` avec une version de JFlex inférieure à 1.4.3 incluse.
3. On dit *Lookahead(1)*, ce n'est pas comme vous qui "voyez" l'ensemble du mot.

```
[
        {
           "Latitude":  37.7668,
           "Longitude": -122.3959
        },
        {
           "Latitude":  37.371991,
           "Longitude": -122.026020
        }
]
```

```
Read symbol: 10 = [

Read symbol: 12 = {

Read symbol: 20 = "Latitude"
Read symbol: 14 = :
Read symbol: 19 = 37.7668
Read symbol: 15 = ,

Read symbol: 20 = "Longitude"
Read symbol: 14 = :
Read symbol: 19 = -122.3959
Read symbol: 13 =
        }
Read symbol: 15 = ,

Read symbol: 12 = {

Read symbol: 20 = "Latitude"
Read symbol: 14 = :
Read symbol: 19 = 37.371991
Read symbol: 15 = ,

Read symbol: 20 = "Longitude"
Read symbol: 14 = :
Read symbol: 19 = -122.026020
Read symbol: 13 =
        }

Read symbol: 11 = ]

End of scanning
```

FIGURE 1 – Scan d'un fichier json

FIGURE 2 – Visualisateur d'arbre

— si la position dans la règle en cours de dérivation correspond à un non terminal X et que le token lu correspond à un *début* de membre droit de X dans la grammaire, alors on crée un nœud `RuleNode` de type X dans l'arbre, fils du nœud courant et placé en dernière position (voir `ADNode.appendchild`). Le nœud courant devient le nœud ajouté, la position de la dérivation dans ce nœud étant initialisée à zéro. Le token sera lu au prochain pas en dérivant X, on n'avance donc pas dans le mot d'entrée. Notez que si X peut se dériver en le mot vide, il faut aussi tenir compte des tokens qui *suivent* x dans la règle courante.

— si la position dans la règle en cours de dérivation correspond à un terminal `t` et que le token lu est égal à `t`, on crée un nœud `LeafNode` de type t dans l'arbre, fils du nœud courant et placé en dernière position (voir `ADNode.appendchild`). On avance alors d'un pas dans la règle courante (`current.SetValue(current.GetValue()+1);` et on passe au token suivant (`symb=yy.next_token();`. Si le token lu n'est pas égal au terminal `t`, c'est une erreur de syntaxe.

— après avoir effectué une des deux actions précédentes (ou aucune), il est possible que l'on soit parvenu à la fin de la règle en cours de dérivation, i.e. que la position soit égale à la longueur de la règle courante. Dans ce cas il faut "remonter" dans l'arbre : le nœud courant devient égal au nœud parent (voir `ADNode.getParent`) et la position courante dans ce nœud parent est incrémentée de 1 ;

Les deux premières actions se nomment communément *décalage (shift)*, et la dernière action se nomme *réduction (reduce)*.

Le squelette de programme suivant implémente ces actions pour les règles $JSONText$ et $Array$ (et une version de $Value$ limitée aux nombres entiers).

```
RuleNode root=new RuleNode(rule.JSON_text, 0);
RuleNode current=root;
RuleNode newrule;
LeafNode newleaf;
Symbol symb = yy.next_token();
while (symb.sym != sym.EOF){
    switch (current.getType()) {
    case rule.JSON_text:
        if (current.getValue() == 0) {
            switch(symb.sym) {
            case sym.BEGINOBJECT:
                newrule=new RuleNode(rule.OBJECT, 0);
                current.appendChild(newrule);
                current=newrule;
                break;
            case sym.BEGINARRAY:
                newrule=new RuleNode(rule.ARRAY, 0);
                current.appendChild(newrule);
                current=newrule;
                break;
            default:
                throw new SyntaxException("waiting '{' or '[' at line "+yy.line()+ ", col " +yy.col() + ", found "+yy.yytext
            }
        } else {
            throw new SyntaxException("waiting EOF at line "+yy.line()+ ", col " +yy.col() + ", found "+yy.yytext());
        }
        break;
    case rule.ARRAY:
        switch (current.getValue()) {
        case 0:
// Array -> . begin-array (Value ( value-separator Value )*)?
end-array
            switch(symb.sym) {
            case sym.BEGINARRAY:
                newleaf=new LeafNode(sym.BEGINARRAY, yy.yytext());
                current.appendChild(newleaf);
                current.setValue(current.getValue() + 1);
                symb = yy.next_token();
// Axiome
// Début de règle
// symbole lu ==
// premier terminal de la règle Object
// on crée un noeud OBJECT
// que l'on ajoute à la fin du noeud courant
// et on descend dans l'arbre

// premier terminal de la règle Array
// on crée un noeud ARRAY
// que l'on ajoute à la fin du noeud courant
// et on descend dans l'arbre

// c'est assez clair je pense

// on est en fin de règle
// Fin de l'axiome et symb!=sym.EOF

// symbole lu ==
// begin-array
// on crée un noeud terminal que l'on ajoute au noeud courant
// on avance dans la règle
// et on lit le token suivant
```

```
        break;
    default:
        throw new SyntaxException("waiting ']' at line "+yy.line()+ ", col " +yy.col() + ", found "+yy.yytext());
    }
    break;
case 1:                                 // Array -> begin-array . (Value ( value-separator Value )*)?
    switch(symb.sym) {
    case sym.ENDARRAY:                  // ce qui suit le point peut se dériver en le mot vide, jusque end-array
        newleaf=new LeafNode(sym.ENDARRAY, yy.yytext());
        current.appendChild(newleaf);   // on crée un noeud terminal que l'on ajoute au noeud courant
        current.setValue(current.getValue() + 1); // on avance dans la règle
        symb = yy.next_token();                   // et on lit le token suivant
        current=current.getParent();              // on est en fin de règle, on remonte
        current.setValue(current.getValue() + 1); // et on avance dans la règle parente
        break;
    case sym.FALSE:                     // les terminaux possibles en début de Value
    case sym.NULL:
    case sym.TRUE:
    case sym.BEGINOBJECT:
    case sym.BEGINARRAY:
    case sym.NUMBER:
    case sym.STRING:
        newrule=new RuleNode(rule.VALUE, 0); // on crée un noeud Value
        current.appendChild(newrule);        // que l'on ajoute à la fin du noeud courant
        current=newrule;                     // et on descend dans l'arbre
        break;
    default:
        throw new SyntaxException("waiting Value or ']' at line "+yy.line()+ ", col " +yy.col() + ", found "+yy.yytext());
    }
    break;
case 2:                                 // Array -> begin-array (Value . ( value-separator Value )*)?
    switch(symb.sym) {
    case sym.ENDARRAY:                  // ce qui suit le point peut se dériver en le mot vide, jusque end-array
        newleaf=new LeafNode(sym.ENDARRAY, yy.yytext());
        current.appendChild(newleaf);
        current.setValue(current.getValue() + 1);
```

```
        current=current.getParent();
        current.setValue(current.getValue() + 1);
        symb = yy.next_token();
        break;
    case sym.VALUESEPARATOR:
        newleaf=new LeafNode(sym.VALUESEPARATOR, yy.yytext());
        current.appendChild(newleaf);
        current.setValue(current.getValue() + 1);
        symb = yy.next_token();
        break;
    default:
        throw new SyntaxException("waiting ', ' or ']' at line "+yy.line()+ ", col " +yy.col() + ", found "+yy.yytext
    }
    break;
case 3:                                 // Array -> begin-array (Value ( value-separator . Value )*)?
end-array
    switch (symb.sym) {
    case sym.FALSE:                     // les terminaux possibles en début de Value
    case sym.NULL:
    case sym.TRUE:
    case sym.BEGINOBJECT:
    case sym.BEGINARRAY:
    case sym.NUMBER:
    case sym.STRING:
        newrule=new RuleNode(rule.VALUE, 0); // on crée un noeud Value
        current.appendChild(newrule);       // que l'on ajoute à la fin du noeud courant
        current=newrule;                    // et on descend dans l'arbre
        break;
    default:
        throw new SyntaxException("waiting Value at line "+yy.line()+ ", col " +yy.col() + ", found "+yy.yytext());
    }
    break;
case 4:                                 // Array -> begin-array (Value ( value-separator Value ).*)?
end-array

                                        // pour la répétition on se replace au début
    current.setValue(2);                // Array -> begin-array (Value . ( value-separator Value )*)?
end-array
    break;
```

```
            default:                    // Normalement impossible ! et pas une SyntaxException d'ailleurs
                throw new SyntaxException("Internal parser error in ARRAY ");
            }
            break;
        case rule.VALUE:
            switch (current.getValue()) {                   // number en version simple
            case 0:                                         // Value -> . number
                switch(symb.sym) {                          // symbole lu ==
                case sym.NUMBER:                            // sym.NUMBER
                    newleaf=new LeafNode(sym.NUMBER, yy.yytext());
                    current.appendChild(newleaf);           // on crée un noeud terminal que l'on ajoute au noeud courant
                    current.setValue(current.getValue() + 1); // on avance dans la règle
                    symb = yy.next_token();                 // et on lit le token suivant
                    break;
                default:
                    throw new SyntaxException("waiting Value at line "+yy.line()+ ", col " +yy.col() + ", found "+yy.yytext());
                }
                break;
            case 1:
                current=current.getParent();                // on est en fin de règle, on remonte
                current.setValue(current.getValue() + 1); // et on avance dans la règle parente
                break;
            default:                                        // Normalement impossible ! et pas une SyntaxException d'ailleurs
                throw new SyntaxException("Internal parser error in VALUE ");
            }
            break;
        default:
            System.out.println("Règle "+current.nodetoString()+ " non implémentée!!! - réduction immédiate ");
            current=current.getParent();
            current.setValue(current.getValue() + 1);
            // c'est normalement une erreur interne
            // throw new SyntaxException("Internal parser error: Unknown rule");
        }
    }
    System.out.println("End of scanning");
```

Question 2. *Complétez le fichier* `JsonAD.java` *pour effectuer la construction d'un arbre de dérivation pour l'ensemble de la grammaire JSON. Vous trouverez les fichiers nécessaires dans l'archive tp3-json.tar.gz. Décompressez l'archive et copiez votre fichier JFlex de la question 1 dans le répertoire* `tp3-json/Q2-json-ad/` *sous le nom* `json.lex` *(l'archive contient une version limitée aux tableaux d'entiers). Vous pouvez alors utiliser le fichier* `build.xml` *pour construire le programme :*

```
# Pour compiler
cd ~/tp3-json
ant -v Q2-json-ad

# Pour exécuter
cd /tmp/build/Q2-json-ad
# test sur tableau d'entiers en mode debug (fonctionne avec le json.lex tel que fourni)
java -cp /usr/share/java/java_cup-runtime.jar:. JsonAD -d ~/tp3-json/array.json

# test sur l'exemple de Q1
java -cp /usr/share/java/java_cup-runtime.jar:. JsonAD  ~/tp3-json/sample.json
 # mode debug
java -cp /usr/share/java/java_cup-runtime.jar:. JsonAD -d ~/tp3-json/sample.json

# Un autre exemple
java -cp /usr/share/java/java_cup-runtime.jar:. JsonAD -d ~/tp3-json/sample1.json

# Pour utiliser le visualisateur sur un arbre sérialisé (sauvegardé dans le ficher result.ast)
java ad/ADViewer result.ast
```