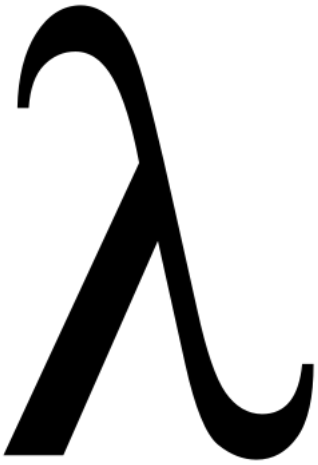
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Génie mathématique

Conception d'un Parser/Lexer sous ANTLR4



Ergi DIBRA ergi.dibra@insa-rouen.fr

Présenté à :
M. Habib ABDULRAB

Table des matières

1 Introduction

Ce projet est née lors d'un intérêt pour la découverte des nouveaux outils de génération des lexeurs et parseurs lors de la conception d'un grammaire. Sur ces pistes de recherche je suis tombé sur l'outil ANTLR4 et ensuite conçu 3 grammaires différentes en utilisant ce dernier.

La première grammaire correspond à un grammaire inventé qui mimique le langage qui s'utilise lors d'une conversation sur des plate-formes de chat sur internet. Cet exemple constitue aussi un petit témoignage de l'avantage de l'utilisation d'ANTLR4 et aussi un moyen de voire les différents moyens d'utilisation des parseurs et lexeurs générés par cet outil.

La deuxième grammaire conçue corresponds au langage de calcul des prédicats (qui est utilisé par des langages de programmation logiques comme Prolog etc) et la troisième grammaire corresponds au code morse.

2 ANTLR

ANTLR, sigle de **AN**other **T**ool for **L**anguage **R**ecognition, est un framework libre de construction de compilateurs utilisant une analyse LL(*), créé par Terence Parr à l'Université de San Francisco. ANTLR permet de générer des analyseurs lexicaux, syntaxiques ou des analyseurs lexicaux et syntaxiques combinés.

Un analyseur syntaxique peut créer automatiquement un arbre syntaxique abstrait qui peut alors à son tour être traité par un analyseur d'arbre. ANTLR utilise une notation identique pour définir les différents types d'analyseurs, qu'ils soient lexicaux, syntaxiques, ou d'arbre. Des actions peuvent être assignées aux branches de l'arbre syntaxique abstrait ainsi obtenu. Ces actions peuvent être directement insérées dans la spécification de la grammaire utilisée, ou utilisés de façon découplée à travers un système de traversée d'arbres fourni par ANTLR.

2.1 Pourquoi ANTLR4 ?

- **Generateur d'analyseur :** Ce programme permets la génération de parseur et lexeur dans la plupart des langages de programmation sans avoir besoin d'utilisation de Java en tant qu'intermediare (vu que ANTLR est programmé en Java).
- **GUI/TestRig :** L'interface graphique facilite le debugging de la grammaire (par exemple, vous pouvez voir les AST générés dans l'interface graphique avec aucun outil supplémentaire requis).
- **Lisibilité du code :** Le code généré est réellement lisible par l'homme (c'est l'un des objectifs d'ANTLR) et le fait qu'il génère des parseurs LL aide certainement à cet égard.

2.2 Guide d'utilisation

Ici je vais presenter une petite guide d'utilisation de cet outil :

2.2.1 Installation

L'installation est facile. Ca suffit à entrer les lignes de code suivantes dans le terminal et mettre les alias dans le *.bashrc* :

```
cd /usr/local/lib
wget https://www.antlr.org/download/antlr-4.7.1-complete.jar
export CLASSPATH=".:/usr/local/lib/antlr-4.7.1-complete.jar:$CLASSPATH"
alias antlr4='java -jar /usr/local/lib/antlr-4.7.1-complete.jar'
alias grun='java org.antlr.v4.gui.TestRig'
```

2.2.2 Utilisation

Ce qui reste c'est la génération des analyseurs pour le langage de programmation choisi :

```
antlr4 -Dlanguage=x Chat.g4
```

ou x sera remplacé par le nom de langage.

Après la génération des analyseurs on peut même utiliser ANTLR en soi pour générer nativement les analyseurs (en Java) et ensuite utiliser TestRig pour tester les règles de grammaire via :

```
antlr4 grammaire.g4
grun grammaire regle_de_grammaire fichier --options
```

Les grammaires (format .g4) écrites en ANTLR comprennent le lexeur et le parseur.

3 ANTLR-JS

Cet exemple va démontrer la génération d'un analyseur en JavaScript et l'utilisation de ce dernier pour la génération des éléments d'un page web. Premièrement on écrit le grammaire qu'on appelle *Chat.g4*.

3.1 Grammaire : Brève description

Voici le grammaire écrite sous ANTLR :

```
grammar Chat;

/*
 * Parser Rules
 */

chat                                : line+ EOF ;
```

```
line                                : name command message NEWLINE ;

message                               : (emoticon | link | color | mention | WORD |
↪  WHITESPACE)+ ;

name                                : WORD WHITESPACE;

command                               : (SAYS | SHOUTS) ':' WHITESPACE ;

emoticon                          : ':' '-'? ')'
                                  | ':' '-'? '('
                                  ;

link                                : TEXT TEXT ;

color                                : '/' WORD '/' message '/';

mention                               : '@' WORD ;


/*
 * Lexer Rules
 */

fragment A              : ('A'|'a') ;
fragment S            : ('S'|'s') ;
fragment Y          : ('Y'|'y') ;
fragment H          : ('H'|'h') ;
fragment O          : ('O'|'o') ;
fragment U          : ('U'|'u') ;
fragment T          : ('T'|'t') ;

fragment LOWERCASE  : [a-z] ;
fragment UPPERCASE  : [A-Z] ;

SAYS                                : S A Y S ;

SHOUTS                                : S H O U T S ;

WORD                                : (LOWERCASE | UPPERCASE | '_')+ ;

WHITESPACE                        : (' ' | '\t')+ ;

NEWLINE                               : ('\r'? '\n' | '\r')+ ;

TEXT                                : ('['|'(') ~[\])]+ (']'|')');
```

En ANLTR on écrit le parseur et le lexeur dans un même fichier et on écrit toujours le parseur avant le lexeur même si en réalité c'est le lexeur qui marche bien avant le parseur. Comme mentionnée ci-dessus ANTLR utilise une dérivation de type LL (leftmost derivation). On déclare les raccourcis/alias en utilisant le mot clé **fragment** et le nom qu'on veut lui donner.

Dans la partie lexeur, on définie les mots du type SAYS et SHOUTS qui correspondent à la syntaxe qui est utilise dans un logiciel du type chat pour dire quelquechose à quelqu'un ou a tout le monde. Ensuite un mot WORD qui peut être la combinaison des miniscules ou majuscules ou le tire bas ; le WHITESPACE (espace blanc) qui est soit une espace ou un tab, le NEWLINE (retour a la ligne), et le token TEXT qui sera utilisé afin de reconnaître les liens entrées.

En ce qui concerne le parseur on définit chat comme etant la combinaison d'une ligne et d'une end-of-file, une ligne comme étant compose d'un nom, un ordre (command) (SHOUT ou SAYS), un message et du retour a la ligne. Le message peut être un emoticon ou un lien ou un couleur, un mot ou une référence (mention).

Le nom est un nom suivi d'un WHITESPACE et un ordre est soit 'SAYS' ou 'SHOUTS' suivi de ' :'. Un emoticon est soit **:-)** out **:-(**, un couleur est du format **/nomcouleur/message/** et une référence est du format **@lareference**.

3.2 Les analyseurs et leur utilisation

Afin de générer les analyseurs ça suffit de faire exécuter la commande :

```
antlr4 -Dlanguage=JavaScript Chat.g4
```

On observe tout de suite la génération des analyseurs comme suit :

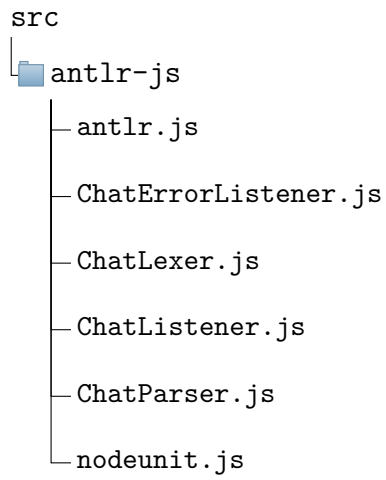


Figure 1 – Arborescence du code source

Le fichier que nous intéresse est ChatListener.js, on ne va rien y modifier, mais il contient des méthodes et des fonctions que nous allons surcharger avec notre propre Listener. Nous n'allons pas le modifier, car les changements seraient écrasés à chaque régénération de la grammaire. Voici le contenu de ce fichier :

```
// Generated from Chat.g4 by ANTLR 4.7.1
// jshint ignore: start
var antlr4 = require('antlr4/index');

// This class defines a complete listener for a parse tree produced by ChatParser.
function ChatListener() {
        antlr4.tree.ParseTreeListener.call(this);
        return this;
}

ChatListener.prototype = Object.create(antlr4.tree.ParseTreeListener.prototype);
ChatListener.prototype.constructor = ChatListener;

// Enter a parse tree produced by ChatParser#chat.
ChatListener.prototype.enterChat = function(ctx) {
};

// Exit a parse tree produced by ChatParser#chat.
ChatListener.prototype.exitChat = function(ctx) {
};


// Enter a parse tree produced by ChatParser#line.
ChatListener.prototype.enterLine = function(ctx) {
};

// Exit a parse tree produced by ChatParser#line.
ChatListener.prototype.exitLine = function(ctx) {
};


// Enter a parse tree produced by ChatParser#message.
ChatListener.prototype.enterMessage = function(ctx) {
};

// Exit a parse tree produced by ChatParser#message.
ChatListener.prototype.exitMessage = function(ctx) {
};


// Enter a parse tree produced by ChatParser#name.
ChatListener.prototype.enterName = function(ctx) {
};

// Exit a parse tree produced by ChatParser#name.
ChatListener.prototype.exitName = function(ctx) {
};


// Enter a parse tree produced by ChatParser#command.
ChatListener.prototype.enterCommand = function(ctx) {
};
```

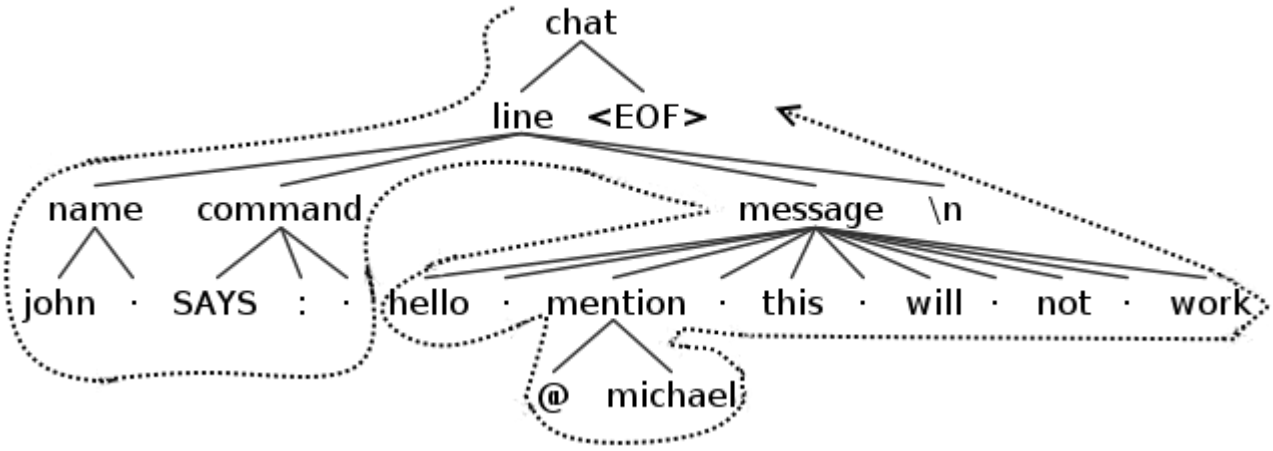
```
// Exit a parse tree produced by ChatParser#command.
ChatListener.prototype.exitCommand = function(ctx) {
};


// Enter a parse tree produced by ChatParser#emoticon.
ChatListener.prototype.enterEmoticon = function(ctx) {
};

// Exit a parse tree produced by ChatParser#emoticon.
ChatListener.prototype.exitEmoticon = function(ctx) {
};


// Enter a parse tree produced by ChatParser#link.
ChatListener.prototype.enterLink = function(ctx) {
};

// Exit a parse tree produced by ChatParser#link.
ChatListener.prototype.exitLink = function(ctx) {
};


// Enter a parse tree produced by ChatParser#color.
ChatListener.prototype.enterColor = function(ctx) {
};

// Exit a parse tree produced by ChatParser#color.
ChatListener.prototype.exitColor = function(ctx) {
};


// Enter a parse tree produced by ChatParser#mention.
ChatListener.prototype.enterMention = function(ctx) {
};

// Exit a parse tree produced by ChatParser#mention.
ChatListener.prototype.exitMention = function(ctx) {
};



exports.ChatListener = ChatListener;
```

En y regardant, vous pouvez voir plusieurs fonctions d'entrée/sortie, une paire pour chacune de nos règles d'analyseur. Ces fonctions seront appelées lorsqu'un morceau de code correspondant à la règle sera rencontré. C'est l'implémentation par défaut du Listener qui nous permet de simplement remplacer les fonctions dont vous avez besoin, sur notre Listener dérivé, et de laisser le reste tel quel.

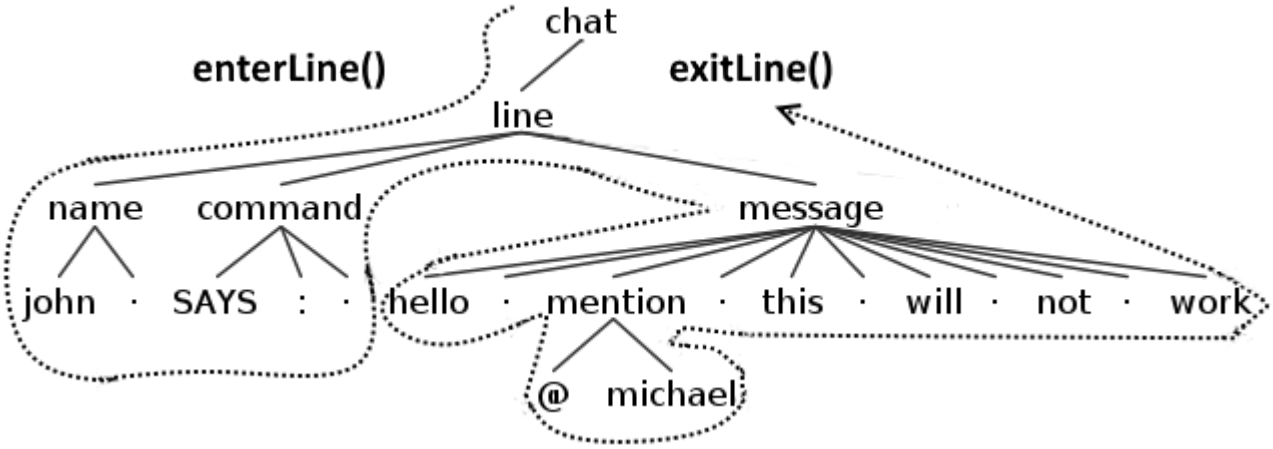
L'alternative à la création d'un Listener est la création d'un Visitor. Les principales dif-

férences sont qu'on ne peut ni contrôler le flux d'un Listener ni renvoyer quoi que ce soit de ses fonctions, alors qu'on peut faire les deux avec un Visitor. Donc, si vous aurait besoin de contrôler comment les nœuds de l'AST sont entrés, ou de recueillir des informations de plusieurs d'entre eux, on utiliserait probablement un visiteur. Le Listener et le Visitor ensemble, utilisent la recherche en profondeur (depth-first search).

Une recherche en profondeur signifie que lorsqu'un nœud sera accédé, ses enfants seront accédés, et si l'un des nœuds enfants a ses propres enfants, il sera accédé avant de continuer avec les autres enfants du premier nœud. L'image suivante illustre bien ce concept :

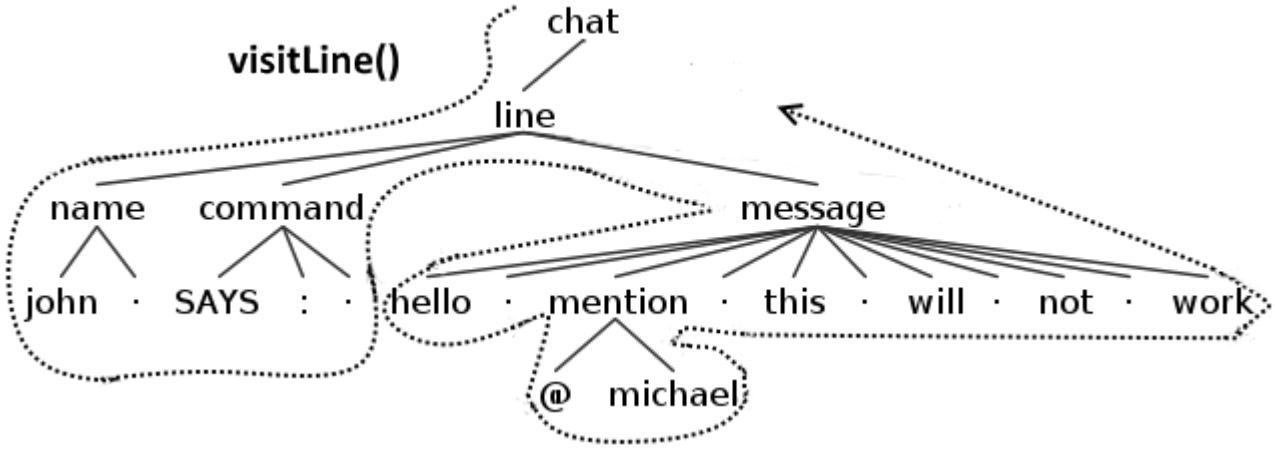


Ainsi, dans le cas d'un Listener, un événement d'entrée sera déclenché à la première rencontre avec le noeud et un autre sera déclenché après avoir quitté tous ses enfants. Dans l'image suivante, vous pouvez voir l'exemple des fonctions qui seront déclenchées lorsqu'un auditeur rencontrera un nœud de ligne (pour simplifier, seules les fonctions liées à la ligne sont affichées).



Avec un visiteur standard, le comportement sera analogue sauf, bien sûr, qu'un seul événement de visite sera déclenché pour chaque nœud. Dans l'image suivante, vous pouvez

voir l'exemple de la fonction qui sera déclenchée lorsqu'un visiteur rencontrera un nœud de ligne (pour simplifier, seule la fonction liée à la ligne est affichée). liées à la ligne sont affichées).



Regardons maintenant à quoi ressemble un programme ANTLR typique.

```
const http = require('http');
const antlr4 = require('antlr4/index');
const ChatLexer = require('./ChatLexer');
const ChatParser = require('./ChatParser');
const HtmlChatListener = require('./HtmlChatListener').HtmlChatListener;
const ChatErrorListener = require('./ChatErrorListener').ChatErrorListener;

http.createServer((req, res) => {

   res.writeHead(200, {
       'Content-Type': 'text/html',
   });

   res.write('<html><head><meta charset="UTF-8"/></head><body>');

   var input = "john SHOUTS: hello @michael /pink/this will work/ :-) \n";
   var chars = new antlr4.InputStream(input);
   var lexer = new ChatLexer.ChatLexer(chars);
   var tokens  = new antlr4.CommonTokenStream(lexer);
   var parser = new ChatParser.ChatParser(tokens);

   parser.buildParseTrees = true;
   var tree = parser.chat();
   var htmlChat = new HtmlChatListener(res);
   antlr4.tree.ParseTreeWalker.DEFAULT.walk(htmlChat, tree);

   res.write('</body></html>');
   res.end();

}).listen(1337);
```

Au début du fichier principal, nous importons (en utilisant require) les bibliothèques et fichiers nécessaires, antlr4 (le runtime) et notre analyseur généré, plus le Listener que nous verrons plus tard. Pour simplifier, nous obtenons en entrée une chaîne de caractères , alors que dans un scénario réel, elle proviendrait d'un éditeur. Cette chaine sera tout simplement : *john SHOUTS : hello @michael /pink/this will work/ :-) n.*

Les lignes 16-19 montrent l'essentiel de chaque programme ANTLR : vous créez le flux de caractères à partir de l'entrée, vous le donnez au lexeur et il les transforme en jetons, qui sont ensuite interprétés par l'analyseur.

Il est utile de réfléchir à cela : le lexer travaille sur les caractères de l'entrée, une copie de l'entrée pour être précis, tandis que l'analyseur travaille sur les jetons/tokens générés par l'analyseur. Le lexer ne fonctionne pas directement sur l'entrée et l'analyseur ne voit même pas les caractères.

C'est important de se souvenir au cas où on doit faire quelque chose d'avancé comme manipuler l'entrée. Dans ce cas, l'entrée est une chaîne, mais, bien sûr, il peut s'agir de n'importe quel flux de contenu.

Une fois que nous obtenons l'AST de l'analyseur, nous voulons généralement le traiter en utilisant un Listener ou un Visitor. Dans ce cas, nous spécifions un Listener. Notre Listener particulier prend un paramètre : l'objet de réponse. Nous voulons l'utiliser pour mettre du texte dans la réponse à envoyer à l'utilisateur. Après avoir etablit le Listener, nous parcourons finalement dans l'arbre avec notre Listener.

```
const antlr4 = require('antlr4/index');
const ChatLexer = require('./ChatLexer');
const ChatParser = require('./ChatParser');
var ChatListener = require('./ChatListener').ChatListener;

HtmlChatListener = function(res) {
    this.Res = res;
    ChatListener.call(this); //inherit default listener
        return this;
};
//inherit default listener
HtmlChatListener.prototype = Object.create(ChatListener.prototype);
HtmlChatListener.prototype.constructor = HtmlChatListener;
//overrides the default listener's behaviour
HtmlChatListener.prototype.enterName = function(ctx) {
    this.Res.write("<strong>");
};
HtmlChatListener.prototype.exitName = function(ctx) {
    this.Res.write(ctx.WORD().getText());
    this.Res.write("</strong> ");
};

HtmlChatListener.prototype.enterColor = function(ctx) {
    var color = ctx.WORD().getText();
```

```
    ctx.text = '<span style="color: ' + color + '">';
};

HtmlChatListener.prototype.exitColor = function(ctx) {
    ctx.text += ctx.message().text;
    ctx.text += '</span>';
};

HtmlChatListener.prototype.exitEmoticon = function(ctx) {
    var emoticon = ctx.getText();

    if(emoticon == ':-)' || emoticon == ':)')
    {
        ctx.text = "";
    }

    if(emoticon == ':-(' || emoticon == ':(')
    {
        ctx.text = "";
    }
};

HtmlChatListener.prototype.exitMessage = function(ctx) {
    var text = '';

    for (var index = 0; index <  ctx.children.length; index++ ) {
        if(ctx.children[index].text != null)
            text += ctx.children[index].text;
        else
            text += ctx.children[index].getText();
    }

    if(ctx.parentCtx instanceof ChatParser.ChatParser.LineContext == false)
    {
        ctx.text = text;
    }
    else
    {
        this.Res.write(text);
        this.Res.write("</p>");
    }
};

HtmlChatListener.prototype.enterCommand = function(ctx) {
    if(ctx.SAYS() != null)
        this.Res.write(ctx.SAYS().getText() + ':' + '<p>');

    if(ctx.SHOUTS() != null)
        this.Res.write(ctx.SHOUTS().getText() + ':' + '<p style="text-transform:
        ↪  uppercase">');
};
```

```
exports.HtmlChatListener = HtmlChatListener;
```

Après les appels de fonction requis, nous faisons en sorte que notre HtmlChatListener étende ChatListener.L'argument *ctx* est une instance d'un contexte de classe spécifique pour le nœud que nous entrons/quittons.

Donc, pour *enterName* on a *NameContext*, pour *exitEmoticon* on a *EmoticonContext*, etc. Ce contexte spécifique aura les éléments appropriés pour la règle, ce qui rendrait possible d'accéder facilement aux tokens et sous-règles respectifs. Par exemple, *NameContext* contiendra des champs comme WORD() et WHITESPACE() ; *CommandContext* contiendra des champs comme WHITESPACE(), SAYS() et SHOUTS().

Ces fonctions, enter et exit, sont appelées par le parcoureur de foret chaque fois que les nœuds correspondants sont entrés ou sortis pendant qu'il traverse l'AST qui représente le retour à la ligne du programme. Un Listener nous permet d'exécuter du code, mais il est important de se rappeler que nous ne pouvons pas arrêter l'exécution du parcoureur et l'exécution des fonctions.

Nous commençons par imprimer une étiquette forte ** parce que nous voulons que le nom soit en gras, alors sur *exitName* nous prenons le texte du mot WORD et fermons le tag. Notez que nous ignorons le token WHITESPACE, rien ne dit que nous devons tout montrer. Dans ce cas, nous aurions pu tout faire soit sur la fonction d'entrée ou de sortie.

Sur la fonction *exitEmoticon*, nous transformons simplement le texte de l'émoticône en un caractère emoji. Nous obtenons le texte de la règle entière car il n'y a pas de tokens définis pour cette règle d'analyseur. Sur *enterCommand*, à la place, il pourrait y avoir deux tokens SAYS ou SHOUTS, nous vérifions donc lequel est défini. Et puis nous modifions le texte suivant, en transformant en majuscule, s'il s'agit d'un SHOUT. Notez que nous fermons la balise *p* à la sortie de la règle de ligne, car la commande, sémantiquement parlant, modifie tout le texte du message.

Avec Javascript, nous pouvons modifier les objets dynamiquement, nous pouvons donc profiter de ce fait pour changer l'objet *Context eux-mêmes. On utilise cela pour modifier le couleur etc.

Nous ajoutons un champ de texte à chaque noeud qui transforme son texte, puis à la sortie de chaque message, nous imprimons le texte s'il s'agit du message principal, celui qui est directement enfant de la règle de ligne. Si c'est un message, c'est aussi un enfant de couleur, on ajoute le champ de texte au noeud que l'on quitte et on laisse la couleur s'imprimer. Nous vérifions cela (ligne 30), et nous regardons le nœud parent pour voir s'il s'agit d'une instance de l'objet LineContext. Ceci est également une preuve supplémentaire de la façon dont chaque argument ctx correspond au type correct.

Entre les lignes 23 et 27 (le for loop), nous pouvons voir un autre champ de chaque nœud de l'arbre généré : children, qui contient évidemment les nœud enfants. Vous pouvez observer

que si un texte de champ existe, nous l'ajoutons à la variable appropriée, sinon nous utilisons la fonction habituelle pour obtenir le texte du nœud.

3.3 Jeu de test

3.3.1 ANTLR-JS

Tout ce qu'il reste à faire maintenant est de lancer le node, avec :

```
nodejs antlr.js
```

et de pointer notre navigateur web à son adresse, habituellement à

```
http://localhost:1337/
```

et nous serons accueillis avec l'image suivante.

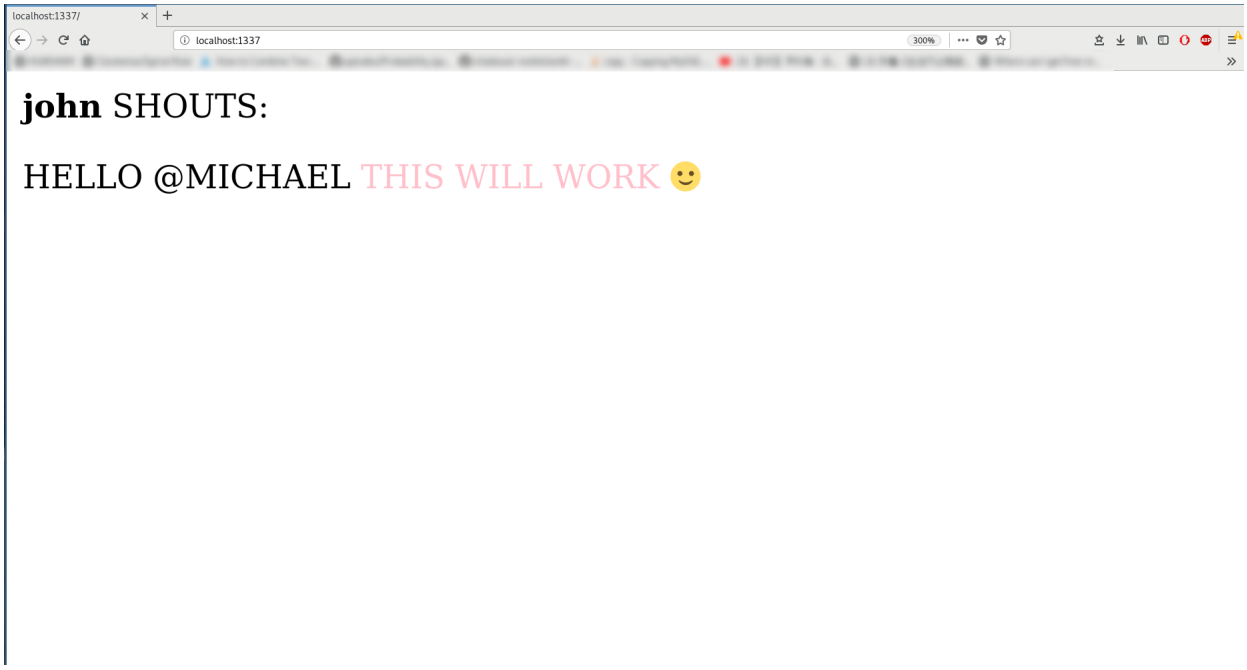


FIGURE 2 – Jeu de test - JS

On peut bien sur étendre ce type d'utilisation pour créer des logiciels qui créent eux-mêmes automatiquement des pages de web à partir du code écrit sans avoir besoin d'écrire de JavaScript derrière chaque fois qu'on veut éditer/ajouter une fonction. Une utilisation envisagée peut être la conception d'une plate-forme de chat enligne qui utilise un langage un peu plus avance (un langage formel issu d'un sous-ensemble de l'anglais par exemple etc).

3.3.2 ANTLR-Native

Dans cet exemple on teste notre grammaire en utilisant l'outil TestRig de ANLTR4. Donc on saisie la commande :

```
echo "hello world" | grun Chat chat -tokens
```

qui prends le string hello world et le mets en entrée du parseur afin de tester le grammaire. Cette commande nous donne :

```
[@0,0:4='hello',<WORD>,1:0]
[@1,5:5=' ',<WHITESPACE>,1:5]
[@2,6:10='world',<WORD>,1:6]
[@3,11:11='\n',<NEWLINE>,1:11]
[@4,12:11='<EOF>',<EOF>,2:0]
line 1:6 mismatched input 'world' expecting {SAYS, SHOUTS}
```

qui montre clairement le fait d'avoir rencontré un token du type WORD "hello", une espace " " et un autre mot "world". Ensuite cela detecte aussi le token de retour à la ligne et le fameux EOF. Au final l'outil TestRig nous démontre ou l'erreur a eu lieu en affichant qu'il attends un mot du type SAYS ou SHOUTS au lieu du "world" qui correspond bien aux règles qu'on avait établies.

Essayons maintenant :

```
echo "john SHOUTS: hello @michael /pink/this will work/ :-) \n" | grun Chat
↪  chat -tokens
```

ce qui ne donne pas d'erreur comme attendu.

```
[@0,0:3='john',<WORD>,1:0]
[@1,4:4=' ',<WHITESPACE>,1:4]
[@2,5:10='SHOUTS',<SHOUTS>,1:5]
[@3,11:11=':',<':'>,1:11]
[@4,12:12=' ',<WHITESPACE>,1:12]
[@5,13:17='hello',<WORD>,1:13]
[@6,18:18=' ',<WHITESPACE>,1:18]
[@7,19:19='@',<'@'>,1:19]
[@8,20:26='michael',<WORD>,1:20]
[@9,27:27=' ',<WHITESPACE>,1:27]
[@10,28:28='/',<'/'>,1:28]
[@11,29:32='pink',<WORD>,1:29]
[@12,33:33='/',<'/'>,1:33]
[@13,34:37='this',<WORD>,1:34]
[@14,38:38=' ',<WHITESPACE>,1:38]
[@15,39:42='will',<WORD>,1:39]
```

```
[@16,43:43=' ',<WHITESPACE>,1:43]
[@17,44:47='work',<WORD>,1:44]
[@18,48:48='/',<'/'>,1:48]
[@19,49:49=' ',<WHITESPACE>,1:49]
[@20,50:50=':',<':'>,1:50]
[@21,51:51='-',<'-'>,1:51]
[@22,52:52=')',<')'>,1:52]
[@23,53:53=' ',<WHITESPACE>,1:53]
[@24,55:55='n',<WORD>,1:55]
[@25,56:56='\n',<NEWLINE>,1:56]
[@26,57:56='<EOF>',<EOF>,2:0]
```

On peut même aller plus loin et écrire :

```
echo "john SHOUTS: hello @michael /pink/this will work/ :-) \n" | grun Chat
↪  chat -gui
```

qui génère l'AST (arbre syntaxique abstraite) suivant :

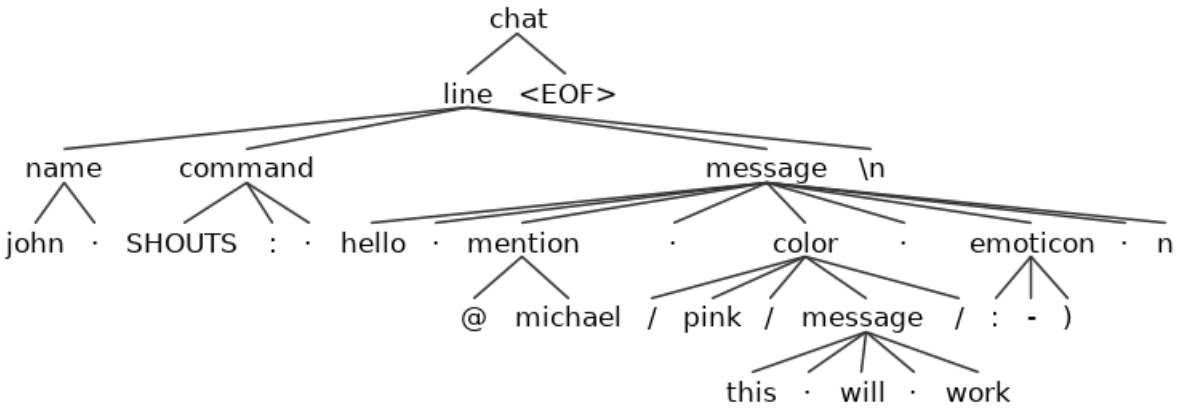


FIGURE 3 – AST

qui correspond bien a ce qu'on attendait.

Ces outils sont donc très utiles lors de la création et le debugging d'une nouvelle grammaire.

4 ANLTR-Native

4.1 Langage de calcul des prédicats

Dans cet exemple on veut créer une grammaire pour le langage de calcul des prédicats (aussi appelé logique du premier ordre). Ce type de langage est utilisé dans la programmation logique (même si en fait cette utilisation fait appel au *calcul propositionnel*).

4.1.1 Grammaire : Brève description

Cette section donne une brève présentation de la syntaxe du langage formel du calcul des prédicats.

On se donne pour alphabet :
- un ensemble de symboles appelés variables, toujours infini : x, y, z etc.
- un ensemble de symboles de fonctions f, g, h etc.
- un ensemble de symboles de prédicats P, Q, R etc.

Chaque symbole de fonction et chaque symbole de prédicat a une arité : il s'agit du nombre d'arguments ou d'objets auxquels il est appliqué. Par exemple, le prédicat B pour " est bleu(e) " a une arité égale à 1 (on dit qu'il est unaire ou monadique), tandis que le prédicat *amis* pour " être amis " a une arité de deux (on dit qu'il est binaire ou dyadique).

- Les symboles $forall$ (quel que soit) et $\exists$ (il existe), appelés quantificateurs.
- Les symboles $\neg$ (non), $\wedge$ (et), $\vee$ (ou) et $\implies$ (implique), qui sont des connecteurs que possède aussi le calcul des propositions.
- Les symboles de ponctuation ") " et " (".

On pourrait se contenter d'un seul quantificateur $\forall$ et de deux connecteurs logiques $\neg$ et $\wedge$ en définissant les autres connecteurs et quantificateur à partir de ceux-ci. Par exemple $\exists x P(x)$ est défini comme $\neg(\forall x \neg P(x))$.

Les formules du calcul des prédicats du premier ordre sont définies par induction :

- $P(t_1, \ldots, t_n)$ si P est un symbole de prédicat d'arité n et $t_1, \ldots t_n$ sont des termes (une telle formule est appelée un atome ou une formule atomique)
- $\neg e$ si e est une formule
- $(e_1 \wedge e_2)$ si e_1 et e_2 sont des formules
- $(e_1 \vee e_2)$ si e_1 et e_2 sont des formules
- $(e_1 \implies e_2)$ si e_1 et e_2 sont des formules
- $\forall x e$ si e est une formule
- $\exists x e$ si e est une formule

Cette grammaire est été ensuite traduite en grammaire ANTLR comme suit :

```
/*
* FOL rewritten for antlr4
*
*/

grammar fol;

/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/
```

```
 condition
   :formula (ENDLINE formula)* ENDLINE* EOF
   ;
 formula
   : formula bin_connective formula
   | NOT formula bin_connective formula
   | NOT formula
   | FORALL LPAREN variable RPAREN formula
   | EXISTS LPAREN variable RPAREN formula
   | pred_constant LPAREN term (separator term)* RPAREN
   | term EQUAL term
   ;

term
   : ind_constant
   | variable
   | func_constant LPAREN term (separator term)* RPAREN
   ;

bin_connective
   : CONJ
   | DISJ
   | IMPL
   | BICOND
   ;
//used in FORALL|EXISTS and following predicates
variable
   : '?'CHARACTER*
   ;
//predicate constant - np. _isProfesor(?x)
pred_constant
   : '_'CHARACTER*
   ;
//individual constant - used in single predicates
ind_constant
   :'#'CHARACTER*
   ;
//used to create functions, np. .presidentOf(?America) = #Trump
func_constant
   :'.'CHARACTER*
   ;

LPAREN
   :'('
   ;
RPAREN
   :')'
   ;
separator
   :','
   ;
EQUAL
   :'='
   ;
```

```
NOT
   :'!'
   ;
FORALL
   :'Forall'
   ;
EXISTS
   :'Exists'
   ;
CHARACTER
   :('0' .. '9' | 'a' .. 'z' | 'A' .. 'Z')
   ;
CONJ
   :'\\/'
   ;
DISJ
   :'^'
   ;
IMPL
   :'->'
   ;
BICOND
   :'<->'
   ;
ENDLINE
   :('\r'|'\n')+
   ;
WHITESPACE
   :(' '|'\t')+->skip
   ;
```

4.1.2 Jeu de test

Premièrement on exécute :

```
antlr4 -Dlanguage=Java fol.g4
javac *.java
```

afin de générer les analyseurs et les compiler. Ensuite on choisi un des tests dans le fichier "examples" :

```
_isProf(#Lucy)
Forall(?x)_isProf(?x)->_isPerson(?x)
```

et ensuite on exécute la commande :

```
grun fol condition examples/example1.txt
```

qui nous n'envoie pas d'erreur (on teste la règle "condtition") donc cela signifie que l'exemple est bien en accord avec notre grammaire. Si on exécute :

```
grun fol condition examples/example1.txt -gui
```

on obtient l'arbre suivant :

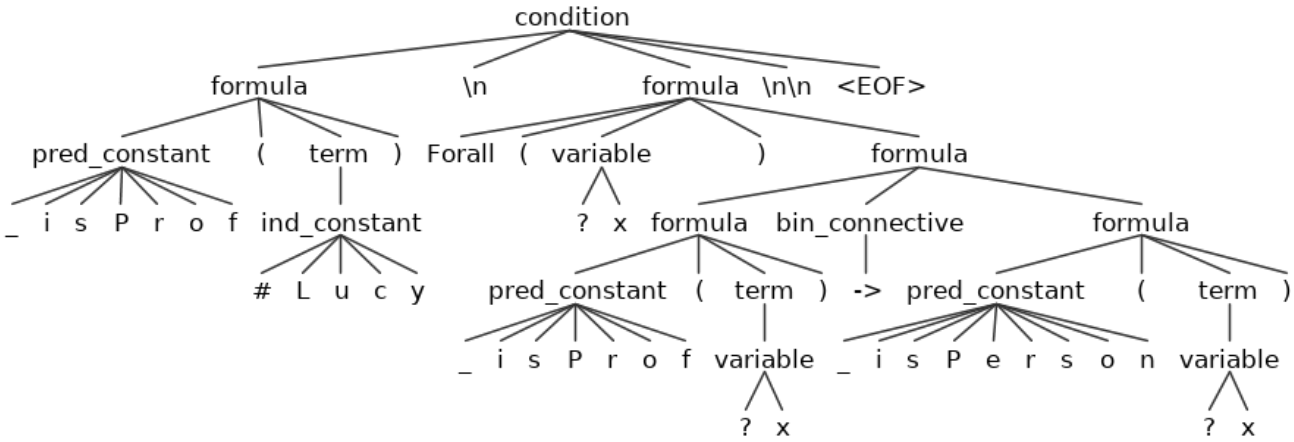


FIGURE 4 – AST : Calcul des prédicats

4.2 Code morse

4.2.1 Grammaire

De même on a :

```
grammar morsecode;

morsecode
    : letter (SPACE letter)+
    ;

letter
    : a
    | b
    | c
    | d
    | e
    | f
    | g
    | h
    | i
    | j
    | k
    | l
    | m
    | n
    | o
    | p
    | q
```

```
    | r
    | s
    | t
    | u
    | v
    | w
    | x
    | y
    | z
    | one
    | two
    | three
    | four
    | five
    | six
    | seven
    | eight
    | nine
    | zero
    ;

a
    : DOT DASH
    ;

b
    : DASH DOT DOT DOT
    ;

c
    : DASH DOT DASH DOT
    ;

d
    : DASH DOT DOT
    ;

e
    : DOT
    ;

f
    : DOT DOT DASH DOT
    ;

g
    : DASH DASH DOT
    ;

h
    : DOT DOT DOT DOT
    ;

i
```

```
    : DOT DOT
    ;

j
    : DOT DASH DASH DASH
    ;

k
    : DASH DOT DASH
    ;

l
    : DOT DASH DOT DOT
    ;

m
    : DASH DASH
    ;

n
    : DASH DOT
    ;

o
    : DASH DASH DASH
    ;

p
    : DOT DASH DASH DOT
    ;

q
    : DASH DASH DOT DASH
    ;

r
    : DOT DASH DOT
    ;

s
    : DOT DOT DOT
    ;

t
    : DASH
    ;

u
    : DOT DOT DASH
    ;

v
    : DOT DOT DOT DASH
    ;
```

```
w
    : DOT DASH DASH
    ;

x
    : DASH DOT DOT DASH
    ;

y
    : DASH DOT DASH DASH
    ;

z
    : DASH DASH DOT DOT
    ;

one
    : DOT DASH DASH DASH DASH
    ;

two
    : DOT DOT DASH DASH DASH
    ;

three
    : DOT DOT DOT DASH DASH
    ;

four
    : DOT DOT DOT DOT DASH
    ;

five
    : DOT DOT DOT DOT DOT
    ;

six
    : DASH DOT DOT DOT DOT
    ;

seven
    : DASH DASH DOT DOT DOT
    ;

eight
    : DASH DASH DASH DOT DOT
    ;

nine
    : DASH DASH DASH DASH DOT
    ;

zero
```

```
    : DASH DASH DASH DASH DASH
    ;


DOT
    : '.'
    ;


DASH
    : '-'
    ;

SPACE
     : ' '
     ;

WS
    : [\t\r\n] -> skip
    ;
```

4.2.2 Jeu de test

Premièrement on exécute :

```
antlr4 -Dlanguage=Java morsecode.g4
javac *.java
```

afin de générer les analyseurs et les compiler. Ensuite on choisi un des tests dans le fichier "examples" (SOS en morse) :

```
... --- ...
```

et ensuite on exécute la commande :

```
grun morsecode morsecode examples/SOS.txt
```

qui nous n'envoie pas d'erreur (on teste la règle "morsecode") donc cela signifie que l'exemple est bien en accord avec notre grammaire.

5 Conclusions et perspectives

5.1 Perspectives

En ce qui concerne la deuxième application, dans le futur je voudrais créer et utiliser des Listener's d'une telle manière pour qu'on puisse intégrer le paradigme de programmation logique dans une langage come C++ (plus ouverte au concept multi-paradigmale). Il en existent déjà des librairies qui font cela (CASTOR et LC++) mais elles fournissent que des petites fonctionnalités qui ne satisfont pas les besoins de conception d'un vrai langage de programmation logique.

5.2 Remerciements

Je tiens a remercier M. ABDULRAB qui m'a donne la possibilité de travailler sur ce projet en nous donnant ainsi la possibilité d'étudier pour la première fois la domaine de NLP et la façon de faire correcte en ce qui concerne la réalisation d'un projet de conception de grammaire/parseur/lexeur suivant toutes les étapes nécessaires. L'utilisation des outils comme *git* ont aussi facilite la réalisation de ce projet.