

Année universitaire 2016-2017  
Université de Caen Normandie

# Rapport sur la création de la grammaire arithmétique

Théo Sarrazin  
*L2 Informatique*

# Table des matières

<b>1</b>	<b>Lex</b>	<b>2</b>
1.1	Les tokens . . . . .	2
1.2	Les expressions régulières . . . . .	2
1.3	Fonctions et variables supplémentaires . . . . .	2
<b>2</b>	<b>Yacc</b>	<b>3</b>
2.1	Création de la grammaire . . . . .	3
2.2	Création du parser . . . . .	3

# 1 Lex

Comme nous avons pu le voir, les regles de lex sont définis par des tokens, chaque token étant associé à une expression régulière.

## 1.1 Les tokens

Dans le cadre d'un grammaire arithmétique nous avons besoin de différents tokens :

- Number : le token qui va reconnaître les nombres
- Plus : le token qui va reconnaître le signe +
- Minus : le token qui va reconnaître le signe -
- Times : le token qui va reconnaître le signe \*
- Divide : le token qui va reconnaître le signe /
- LParen : le token qui va reconnaître la parenthèse gauche
- RParen : le token qui va reconnaître la parenthèse droite

Pour être reconnu par Lex ces tokens sont stockés dans un tuple **tokens**

## 1.2 Les expressions régulières

A chacun des tokens précédents, nous devons associer une expression régulière pour que Lex puisse reconnaître les tokens. Pour cela, nous devons simplement créer une variable de la forme `t_NomDuToken`, est qui prend comme valeur l'expression régulière à associer à ce token.

Pour le token Number, nous utilisons l'expression suivante : "

`b+`", le "

`b`" signifiant n'importe quel chiffre compris entre 0 et 9 de plus le `+` signifie répété une ou plusieurs fois. En effet un nombre est une suite de chiffre.

Pour les six autres tokens, nous utilisons simplement le signe qui doit correspondre au token (`+` pour le token Plus, `-` pour le token Minus, etc...)



FIGURE 1 – Exemple de création d'une expression régulière

## 1.3 Fonctions et variables supplémentaires

De plus, pour l'utilisation de Lex, nous pouvons définir une variable **t\_ignore** qui va préciser les éléments à ne pas prendre en compte, dans notre cas la variable vaut " `"`, nous ignorons donc les espaces et les tabulations.

Nous pouvons aussi définir une fonction **t\_newline(t)**, qui prend un token en paramètre et qui va être utilisée pour matcher tous les retours à la ligne grâce à l'expression régulière "`n+`", de plus cette fonction va ajouter pour chaque ligne trouvée 1 à l'attribut **lineno**, qui contient le nombre de lignes.

Pour finir une autre fonction peut être utilisée, la fonction **t\_error(t)**, qui prend elle aussi un token en paramètre et qui va être appelée par le lexer à chaque token non reconnu par notre lexer. Cette fonction va afficher un message d'erreur puis ignorer le token grâce à l'instruction **t.lexer.skip(1)**, `t` étant le token courant grâce auquel on récupère le lexer.

## 2 Yacc

### 2.1 Création de la grammaire

Pour fonctionner Yacc à besoin de différentes règles de grammaires, chaque règles définissant comment peuvent s'"assembler" les différents tokens reconnus pour Lex.

Pour définir des règles nous devons créer des fonctions nommer de la façons suivante : **p\_NomDeLaRegle**. Cette fonction doit contenir une docstring définissant les règles à respecter.

```
def p_terme(p):  
    '''terme :      terme TIMES facteur  
                  | terme DIVIDE facteur  
                  | facteur  
    ...  
    if not "/" in prop:  
        prop.extend(["/", "*"])
```

FIGURE 2 – Exemple d’une règle de grammaire de Yacc

Voici nos différentes règles :

- expression : correspond à une somme/soustraction de deux expressions. Une expression pouvant aussi être terme.
- terme : correspond à une multiplication/division de deux termes. Un terme pouvant aussi être un facteur.
- facteur : correspond à un nombre, une expression entre parenthèse ou - un facteur.

De plus, chacune de ces fonctions ajoute des propositions à une variable **prop** afin de les afficher par la suite dans notre IDE.

### 2.2 Création du parser

Par la suite, nous devons appliquer nos règles sur du contenu, pour cela nous avons créés une fonction **parse**, qui prend en paramètre le code à parser. Par la suite, cette fonction crée le lexer (en faisant appelle à Lex), puis crée le parser (en faisant appelle à Yacc), pour finir elle revoit les listes des propositions afin de pouvoir les ajouter à les listes des propositions de notre IDE.