Année universitaire 2016-2017 Université de Caen Normandie

Rapport sur la création de la grammaire arithmétique

Théo Sarrazin

Table des matières

1	Lex
	1.1 Les tokens
	1.2 Les expressions régulières
	1.3 Fonctions et variables supplémentaires
	Yacc
	2.1 Création de la grammaire
	2.2 Création du parser

1 Lex

Comme nous avons pu le voir, les regles de lex sont définis par des tokens, chaques tokens étant associés à un expression régulière.

1.1 Les tokens

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

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

Pour être reconnnu 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 reconnaitre 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 : " $\mathbf{b}+$ ", le "

b" signifiant n'importe quel chiffre compris entre 0 et 9 de plus le + signifie répété une ou plusieur 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...)

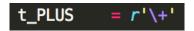


FIGURE 1 – Exemple de création d'une expression régulière

1.3 Fonctions et variables supplémentaires

De plus, pour l'utilisation de Lex, nous pouvons définir une variable **t_ignore** qui va preciser 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 \mathbf{t} _newline(\mathbf{t}), qui prend un token en paramètre et qui va être utilisée pour matcher tous les retours à la lignes grâce à l'expression régulière " \mathbf{n} +", de plus cette fonction va ajouter pour chaque ligne trouvé 1 à l'attribut lineno, qui contient le nombre de lignes.

Pour finir un autre fonction peut être utilisée, la fonction t_error(t), qui prend elle aussi on token en paramètre et qui va être appelée par le lexer à chaque token non reconnue 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 courrant grâce au quel on recupère le lexer.

2 Yacc

2.1 Création de la grammaire

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

Pour definir des règles nous devons créer des fonctions nommer de la façons suivante : **p_NomDeLaRegle**. Cette fonction doit contenir une docstring definissant les regles à respecter.

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 créée le lexer (en faisant appelle à Lex), puis créée le parser (en faisant appelle à Yacc), pour finir elle revoit les listes des propostions afin de pouvoir les ajouter à les listes des propostions de notre IDE.