

Lex & Yacc

Sullivan PINEAU Sebastien VALLEE

April 2017

1 Introduction

On a tous entendu parler un jour ou l'autre de lex et yacc, ne serait-ce que pour compiler certains logiciels ... Mais qu'est ce que c'est ?

Lex et yacc sont un ensemble d'outils très utile si un programme C/C++ a recours à un fichier de données structurées et qu'il faut le parcourir, en vérifier sa validité, en extraire les données utiles etc... Lex et Yacc permettent de décrire la syntaxe du fichier (les mots clefs structurants) ainsi que sa grammaire (les enchaînements de mots clefs et l'exploitation des données parsées) dans un langage de haut niveau. Une fois ce travail accompli, ces deux outils génèrent une fonction C facilement intégrable dans un projet C/C++.

2 Lex

Lex est un analyseur lexical, c'est à dire que celui-ci permet de transformer une suite de symboles en terminaux (un terminal peut être une lettre, un chiffre, un signe '*' ...). Après cette transformation faite, le main est repassé à l'analyseur syntaxique. Par conséquent, le but de cette analyseur lexical est de "prendre" des symboles, de les transformer en respectant ces règles qui lui sont propres et les donner à l'analyseur syntaxique. Chaque règles sont représenté sous la forme d'une expression régulière et d'actions. Dans ce cas, si un "match" a lieu entre l'expression saisie et l'expression régulière définit alors les actions qui lui sont associée sont exécutées. La compilation d'un programme en Lex génère un programme C/C++ et ce programme définit la fonction `yylex(void)` qui permet de représenter l'analyseur lexical.

Il faut savoir qu'un programme Lex possède 4 parties :

```
%{ (Optionnel)
Partie 1 : déclarations pour le compilateur C (Optionnel)
}% (Optionnel)
Partie 2 : définitions régulières (Optionnel)
%%
Partie 3 : règles (Optionnel)
%% (Optionnel)
Partie 4 : fonctions C supplémentaires (Optionnel)
```

2.1 Partie 1

Cette partie permet de spécifier les fichiers à inclure (" #include ..."). Ces lignes seront tout simplement copié au début du fichier généré.

2.2 Partie 2

Dans cette partie se trouve les définitions régulières qui permet donc de définir des notions non terminales. La forme de ces définition régulières doivent être :

NON-TERMINALE EXPRESSION_REGULIERE

Par exemple :

```
character [a-zA-Z]
digit [0-9]
word ({charater}|{digit})+
```

2.3 Partie 3

C'est dans cette troisième partie que se trouve les différentes règles. Une action est un morceau de code C, qui sera recopié tel quel, au bon endroit, dans la fonction yylex. Ces différentes règles devront être sous la forme :

EXPRESSION_REGULIERE {ACTIONS}

Par exemple :

```
yacc    printf("(1)%s",yytext);
word    {printf("word : "); printf("%s", yytext);}
```

NB: Il est possible d'effectuer plusieurs actions pour une même expression régulière (elles doit se trouver entre "{...}"). Si une expression régulière ne possède pas d'action alors Lex recopiera les caractères tels quels sur la sortie standard.

2.4 Partie 4

Dans cette dernière partie se trouve du code C qui sera tout simplement copié à la fin du fichier généré. Si celle-ci est vide alors le compilateur prendra le main dit par défaut qui est :

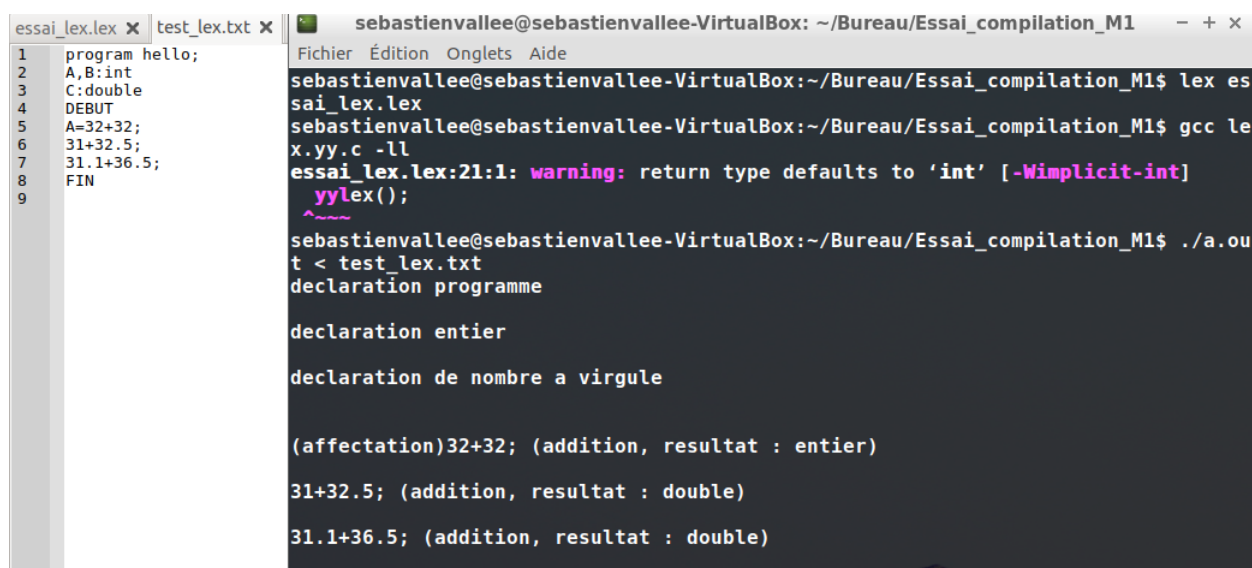
```
main() {
    yylex();
}
```

2.5 Exemple pour compiler un fichier Lex

Vous trouverez dans la racine du dossier de ce rapport un fichiers Lex qui se nomme "essai_lex.lex". Voici les différentes commandes à effectuer :

```
$ lex essai_lex.lex
$ mv lex.yy.c essai_lex.lex.c
$ gcc essai_lex.lex.c -ll
$ ./a.out < textlex
```

Le résultat de ce code est le suivant :



```
essai_lex.lex x test_lex.txt x sebastienvallee@sebastienvallee-VirtualBox: ~/Bureau/Essai_compilation_M1 - + x
1 program hello;
2 A,B:int
3 C:double
4 DEBUT
5 A=32+32;
6 31+32.5;
7 31.1+36.5;
8 FIN
9

sebastienvallee@sebastienvallee-VirtualBox:~/Bureau/Essai_compilation_M1$ lex es
sai_lex.lex
sebastienvallee@sebastienvallee-VirtualBox:~/Bureau/Essai_compilation_M1$ gcc le
x.yy.c -ll
essai_lex.lex:21:1: warning: return type defaults to 'int' [-Wimplicit-int]
    yylex();
    ^~~~~
sebastienvallee@sebastienvallee-VirtualBox:~/Bureau/Essai_compilation_M1$ ./a.ou
t < test_lex.txt
declaration programme

declaration entier

declaration de nombre a virgule

(affectation)32+32; (addition, resultat : entier)

31+32.5; (addition, resultat : double)

31.1+36.5; (addition, resultat : double)
```

Figure 1: Résultat du code lex

Sur la figure 1, on peut voir que le but du programme est de remplacer les lignes du programme du fichier en entrée, par une description des expression qu'il trouve. Par exemple, on peut lire sur la sortie standard que la deuxième ligne du programme est une déclaration d'entier ; ou que l'addition 32+32 est précédé d'une affectation à une variable, tandis que les deux additions suivante non. Lorsque qu'un mot n'est pas reconnu par Lex, il est par défaut envoyé sur la sortie standard, ce qui n'est pas le cas pour ce mot (le fichier en entrée).

Alors bien que Lex peut offrir des outils puissant, l'analyse de fichiers peut s'avérer assez difficile. On ne peut par exemple, pas savoir si les additions sont bien entre les balises "DEBUT" et "FIN" (sauf en écrivant un code c inutilement compliqué). Nous verrons plus tard qu'un autre outil est plus puissant dans ces cas là.

3 Yacc

Lex et yacc sont des outils similaire : ce sont tous les deux g rateur d'analyseur syntaxique.

Leur fonctionnement est tr s similaire dans le sens o  il permettent d'associer des actions lorsque qu'un mot du texte en entr e est reconnu.

Cependant, le fonctionnement de yacc est l g rement plus puissant. En effet, il permet de fonctionner avec une grammaire.

Le code yacc est divis  en 3 partie que nous allons voir rapidement.

3.1 premi re partie

cette premi re partie commence par `%{` et finie par `%}`. Elle permet de d finir tous les besoins du code `c/c++`. C'est   dire la d claration des variables, des constantes, des fonctions, etc...

3.2 deuxi me partie

Cette partie contient l'ensemble de la grammaire et des actions qui lui sont associ . C'est dans cette partie que yacc se distingue de Lex. En effet, l  o  Lex utilise des d finitions r guli res et un ensemble de r gles, Yacc se contente d'associer   chaque r gle d'une grammaire une ou des actions `c++`.

  l'ex cution, cette diff rence s'exprime par le fait que les r gles de Lex sont fixes. Lex va essayer de trouver le mot le plus long qui permet de matcher   une r gle. C'est   dire qu'il y   un match pour une seule r gle. Une fois le mot trouv , Lex "oublie" totalement la r gle utilis e et passe   la recherche du mot suivant. Yacc lui permet de remonter les r gles et donc d'effectuer plusieurs actions   diff rents moments sur une m me cha ne de caract re. C'est   dire qu'il va se souvenir des r gles qu'il traverse. Ce fonctionnement ressemble fortement   ce que nous avons fais dans le code : parcourir l'arbre de notre grammaire et effectuer des actions en fonction du noeud o  nous sommes. Yacc fonctionne exactement pareil.

cette grammaire se d finie de la mani re suivante :

NON_TERMINAL: expression

En premier, le non-terminal suivie de deux points. Puis une expression pouvant contenir des non-terminaux, des terminaux ou des symboles sp ciaux tel que `—` pour la r gle "ou" des grammaire, etc...

3.3 troisi me partie

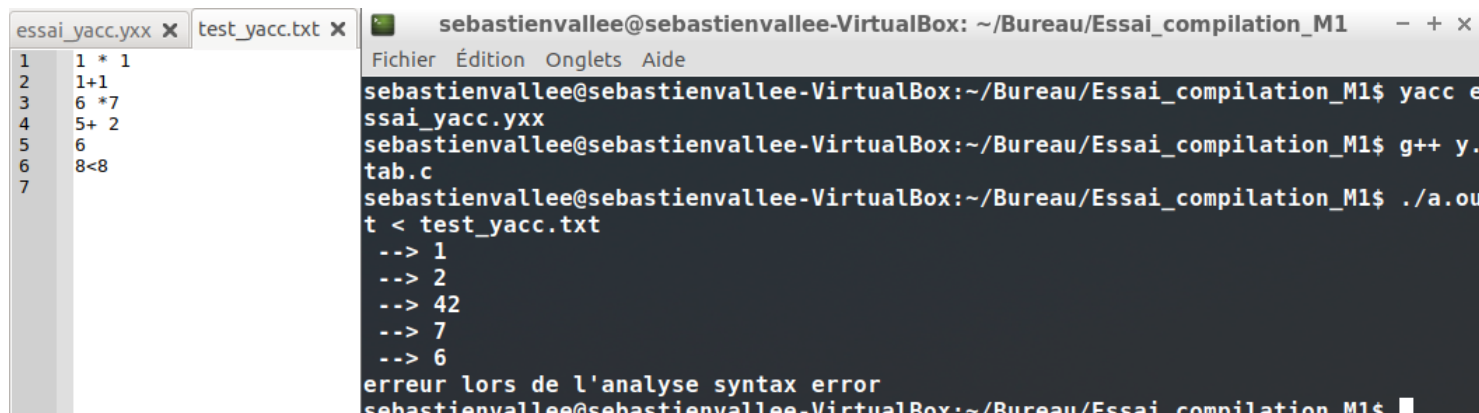
Cette partie est la d claration des fonctions `c++` de la premi re partie et de la d claration de la fonction principale du programme.

3.4 exemple de code Yacc

Vous pouvez trouver un exemple de code Yacc dans le dossier de ce rapport. Ce code yacc permet de récupérer des opérations d'additions ou de multiplication entre des entiers (deux, trois, quatre, ...) et d'afficher le résultat dans la sortie standard. Le code est contenu dans le fichier `essai_yacc.yxx`. Celui-ci peut être compilé avec les commandes :

<pre>\$ yacc essai_yacc.yxx</pre>
cela générera un fichier <code>y.tab.c</code> mais qui devra être compilé avec un compilateur <code>c++</code> (dû à l'utilisation d' <code>iostream</code> et des <code>cout, cin</code>)
<pre>\$ g++ y.tab.c</pre>
(générera l'exécutable qui peut être lancé avec la commande <code>\$./a.out</code>)

Le résultat du code est le suivant, de la forme " -- > resultat_operation ":



```
essai_yacc.yxx x test_yacc.txt x
1 1 * 1
2 1+1
3 6 *7
4 5+ 2
5 6
6 8<8
7

sebastienvallee@sebastienvallee-VirtualBox: ~/Bureau/Essai_compilation_M1
Fichier Édition Onglets Aide
sebastienvallee@sebastienvallee-VirtualBox:~/Bureau/Essai_compilation_M1$ yacc e
ssai_yacc.yxx
sebastienvallee@sebastienvallee-VirtualBox:~/Bureau/Essai_compilation_M1$ g++ y.
tab.c
sebastienvallee@sebastienvallee-VirtualBox:~/Bureau/Essai_compilation_M1$ ./a.ou
t < test_yacc.txt
--> 1
--> 2
--> 42
--> 7
--> 6
erreur lors de l'analyse syntax error
sebastienvallee@sebastienvallee-VirtualBox:~/Bureau/Essai_compilation_M1$
```

Figure 2: resultat du code yacc

Sur la figure 2, on peut voir que le programme s'arrête sur une erreur : la dernière ligne "`8 < 8`" ne respecte aucune règle donc le mot (tout le fichier) est faux et le programme s'arrête.

4 Conclusion

Pour conclure, nous pouvons constater qu'il est assez trivial de prendre rapidement en main Lex et Yacc.

À l'aide de Lex et de Yacc, il serait donc possible de créer un compilateur d'un langage relativement simple possédant des fonctionnalités basiques.