

Année universitaire 2016-2017  
Université de Caen Basse-Normandie

# Rapport sur la progression du projet de création d'une IDE

Alexis Carreau  
Thomas Lécluse  
Emma Mauger  
Théo Sarrazin  
*L2 Informatique*

# Table des matières

<b>1</b>	<b>Séance 1, 09/09/16</b>	<b>2</b>
1.1	Un IDE, qu'est-ce que c'est ? . . . . .	2
1.1.1	Définition . . . . .	2
1.1.2	Que contient-il ? . . . . .	2
1.2	Quel(s) langage(s) supporte-il ? . . . . .	2
1.2.1	Que doit-il être capable de faire selon un utilisateur lambda ? . . . . .	3
1.2.2	Options de correction du code . . . . .	3
<b>2</b>	<b>Séance 2, 23/09/2016</b>	<b>3</b>
2.1	Lex . . . . .	3
2.1.1	Théorie . . . . .	3
2.1.2	Pratique . . . . .	3
2.2	Yacc . . . . .	3
2.2.1	Théorie . . . . .	3
2.2.2	Pratique . . . . .	4
2.2.3	Et dans notre projet ? . . . . .	4
<b>3</b>	<b>Séance 3, 07/10/16</b>	<b>4</b>
3.1	Sortie de Lex . . . . .	4
3.2	Sortie de Yacc . . . . .	5
3.3	Analyse syntaxique produisant l'arbre syntaxique abstrait pour l'auto-complétion	5
3.4	Coloration syntaxique avec Lex . . . . .	5

# **1 Séance 1, 09/09/16**

## **1.1 Un IDE, qu'est-ce que c'est ?**

### **1.1.1 Définition**

Un IDE ou Environnement de Développement Intégré (Integrated Development Environment) est un logiciel qui fournit des facilités au programmeur pour le développement logiciel. Il a pour but de maximiser la productivité du programmeur.

### **1.1.2 Que contient-il ?**

Un IDE contient généralement un éditeur de texte, un interpréteur, un debugger et un compilateur.

#### **L'éditeur de texte**

L'éditeur de texte présente une zone de saisie de texte. Il ne permet pas la mise en forme de ce dernier.

#### **L'interpréteur**

L'interpréteur analyse, traduit et exécute les instructions écrites dans un langage informatique. Ces opérations d'analyse et de traduction sont effectuées à chaque fois que l'on décide d'exécuter le programme.

#### **Le debugger**

Un debugger est un logiciel qui permet d'analyser les bugs d'un programme (tels que des erreurs de syntaxe). Il permet d'exécuter le programme pas-à-pas, d'arrêter le programme, de l'observer et de le contrôler.

#### **Le compilateur**

Un compilateur est un programme informatique qui transforme un code source écrit dans un langage de programmation (langage source) en un autre langage (langage cible), afin qu'il puisse être interprété par la machine (qui ne comprend un langage dit de bas niveau et traductible en binaire).

Les langages utilisés pour programmer sont dits "de haut niveau" (car facilement compréhensible par l'homme), tandis que les langages plus proches du langage machine (le binaire) sont dits de bas niveau.

#### **Un outil de gestion de projet**

Un projet se matérialise comme un dossier virtuel contenant des fichiers (fichiers de code source, fichiers de ressources, documentation...). L'outil de gestion de projet permet d'indexer les fichiers de celui-ci, d'ajouter ou d'enlever un fichier et associer des méta-données aux fichiers (telles que l'auteur, la description, les dates de création et de modification, les options de compilation).

## **1.2 Quel(s) langage(s) supporte-il ?**

Pour commencer, le langage que supportera notre IDE sera le langage C.

### 1.2.1 Que doit-il être capable de faire selon un utilisateur lambda ?

Lorsqu'un utilisateur quelconque ouvre un IDE, il s'attend à trouver plusieurs fonctionnalités telles que :

- Créer, Éditer et Supprimer un "projet". Un projet se matérialisant comme un dossier virtuel contenant des fichiers (fichiers de code source, fichiers de ressources, documentation...)
- Créer, Éditer et Supprimer un dossier.
- Créer, Éditer, Enregistrer et Supprimer des documents de l'extension de leur choix.
- Naviguer dans les dossiers et documents du projet.
- Des raccourcis pour des outils.
- Une interface graphique pour que cela lui soit plus intuitif.

### 1.2.2 Options de correction du code

Le langage avec lequel notre utilisateur codera bénéficiera d'une coloration syntaxique ainsi que d'une vérification syntaxique. Pour cela, nous allons utiliser Lex et Yacc.

## 2 Séance 2, 23/09/2016

### 2.1 Lex

#### 2.1.1 Théorie

Lex peut :

- convertir les chaînes de caractères en Tokens
- reconnaître les rôles des éléments du code grâce à des expressions régulières qui correspondent à des patterns
- renvoyer où il a trouvé tel élément et à quoi il correspond.

#### 2.1.2 Pratique

Pour faire fonctionner Lex, nous devons lui spécifier une liste de token (qui doit s'appeler "tokens"), qui sont une représentation numérique du code. Ainsi que des expressions régulières qui correspondent à chacun des token(s), permettant à Lex de les identifier.

On lui passe donc une liste de token qu'il doit retrouver, et il va retourner tout ce qu'il va trouver avec les expressions régulières.

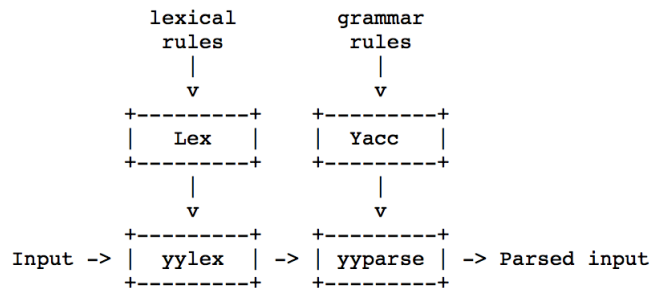
### 2.2 Yacc

#### 2.2.1 Théorie

- Yacc récupère la liste de token renvoyée par Lex
- On lui passe une grammaire : façon d'assembler les token(s)
- Analyse les token(s) et crée un arbre syntaxique qui impose une structure hiérarchique des token(s).
- Dès qu'il trouve un élément qui correspond à la grammaire qu'on lui a passé, il (dans notre cas) exécute une action.

## 2.2.2 Pratique

On commence donc par définir une grammaire, contenue dans une docstring d'une fonction (ici python), qui correspond à la syntaxe du langage que nous souhaitons traiter. Yacc va parcourir la liste de token renvoyée par Lex et dès qu'il rencontrera une syntaxe correspondante, il exécutera la fonction dans laquelle est contenue la docstring.



## 2.2.3 Et dans notre projet ?

Lex et Yacc nous permettront de vérifier la syntaxe du code ainsi que de colorer les mots clefs.

Lorsque Yacc reconnaîtra des tokens envoyés par Lex, il exécutera la fonction dans laquelle la docstring permettant la reconnaissance des tokens est située.

```
def p_primary_expression(p):
    '''primary_expression : IDENTIFIER
                           | CONSTANT
                           | STRING_LITERAL
                           | L_BRACKET expression R_BRACKET'''
    # Cette fonction est appelée lorsque Yacc trouve une primary expression, c'est-à-dire soit le token
    # IDENTIFIER, soit le token CONSTANT, ou encore STRING_LITERAL, ou bien une expression entre parenthèses.
    print("primary_expression")
```

FIGURE 1 – Exemple de fonction de reconnaissance YACC

## 3 Séance 3, 07/10/16

Nous devons maintenant nous pencher sur l'analyse des sorties générées par Lex et Yacc. On doit les interpréter et les comprendre pour pouvoir les exploiter pour notre IDE. Nous devons également obtenir un arbre syntaxique abstrait grâce à l'analyse syntaxique afin de gérer l'auto-complétion.

### 3.1 Sortie de Lex

Lex génère donc une liste de token qui correspondent au type de chaque élément qu'il a reconnu grâce aux règles lexicales. Cette liste de token va nous permettre de colorer les mots en fonction de leur étiquetage. Pour notre IDE, nous allons donc utiliser Lex pour la coloration syntaxique.

### 3.2 Sortie de Yacc

Yacc, en revanche, grâce à la liste de token récupérée par Lex, exécute la fonction dans laquelle est contenue la règle grammaticale si cette dernière est reconnue. Yacc va donc exécuter une action donnée lorsque la syntaxe du bloc d'instructions est correcte. Avec Yacc, nous allons par exemple pouvoir empêcher l'enregistrement du document en cours, si sa syntaxe n'est pas correcte. Pour l'instant, nous savons reconnaître les éléments seulement quand Yacc leur détecte une bonne syntaxe, mais c'est plus compliquée pour identifier les éléments ayant une mauvaise syntaxe. C'est pour cela, que nous travaillons actuellement sur la sortie générée par Yacc.

### 3.3 Analyse syntaxique produisant l'arbre syntaxique abstrait pour l'auto-complétion

Pour avoir l'arbre syntaxique abstrait permettant l'auto-complétion, nous devons utiliser YACC de PLY. Pour ce faire, nous avons mené des investigations et nous avons notamment trouvé des informations intéressantes sur ce site internet :

[http://www.matthieuamiguet.ch/media/documents/MA-compil-03-Analyse\\_Syntaxique.pdf](http://www.matthieuamiguet.ch/media/documents/MA-compil-03-Analyse_Syntaxique.pdf).

Nous sommes donc en train de voir actuellement comment implémenter cette analyse syntaxique à notre IDE afin d'obtenir l'arbre syntaxique abstrait pour gérer l'auto-complétion.

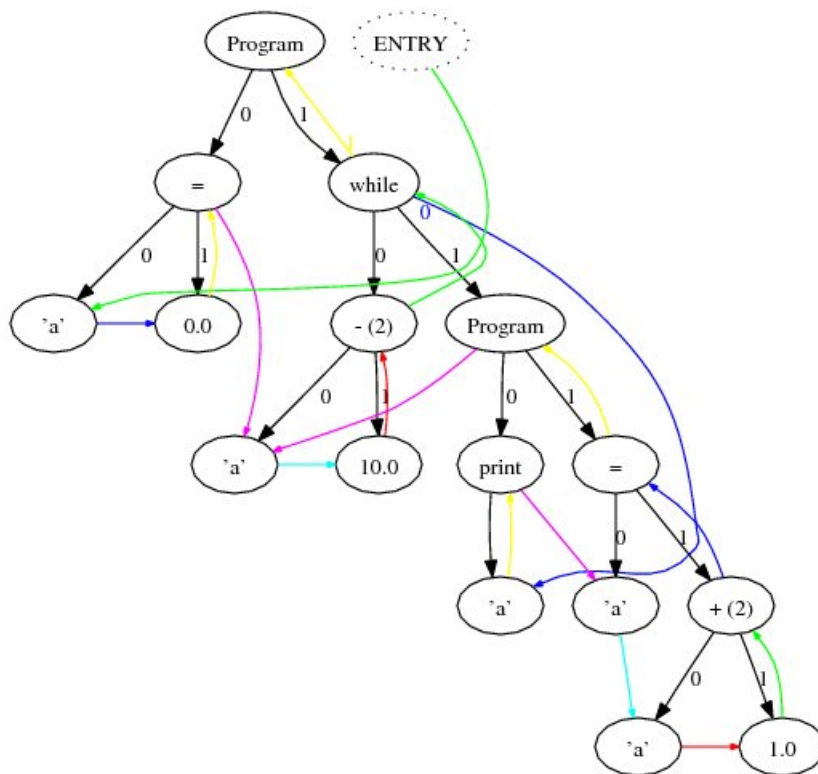
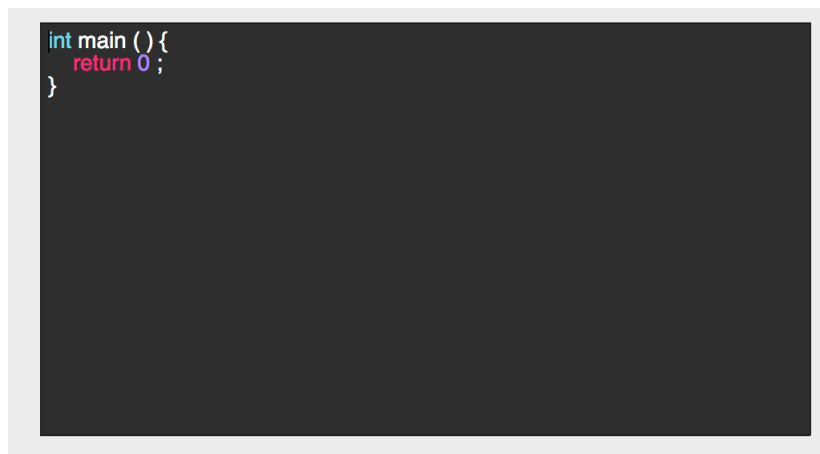


FIGURE 2 – Exemple d'arbre syntaxique

### 3.4 Coloration syntaxique avec Lex

Afin de colorer le texte présent dans notre zone de saisie, nous avons utilisés Lex. Pour cela, nous parcourons la liste des tokens trouvés par Lex. Puis pour chaque token nous ajoutons une balise HTML (span) avec un attribut style permettant de changer la couleur du texte. La couleur dépend du type du token (Identifieur, Keyword, Comment).

A screenshot of a code editor with a dark background. The code is written in a light color and is syntax-highlighted. The first line is 'int main () {' where 'int' is blue, 'main' is green, '()' is green, and '{' is green. The second line is 'return 0;' where 'return' is red, '0' is blue, and ';' is green. The third line is '}' which is green. The code is enclosed in a light gray border.

```
int main () {  
    return 0;  
}
```

FIGURE 3 – Exemple de coloration syntaxique