

Année universitaire 2016-2017
Université de Caen Basse-Normandie

Rapport sur le deuxième semestre de TPA

Alexis Carreau
Thomas Lécluse
Emma Mauger
Théo Sarrazin
L2 Informatique

Réalisation d'un IDE en Python

Table des matières

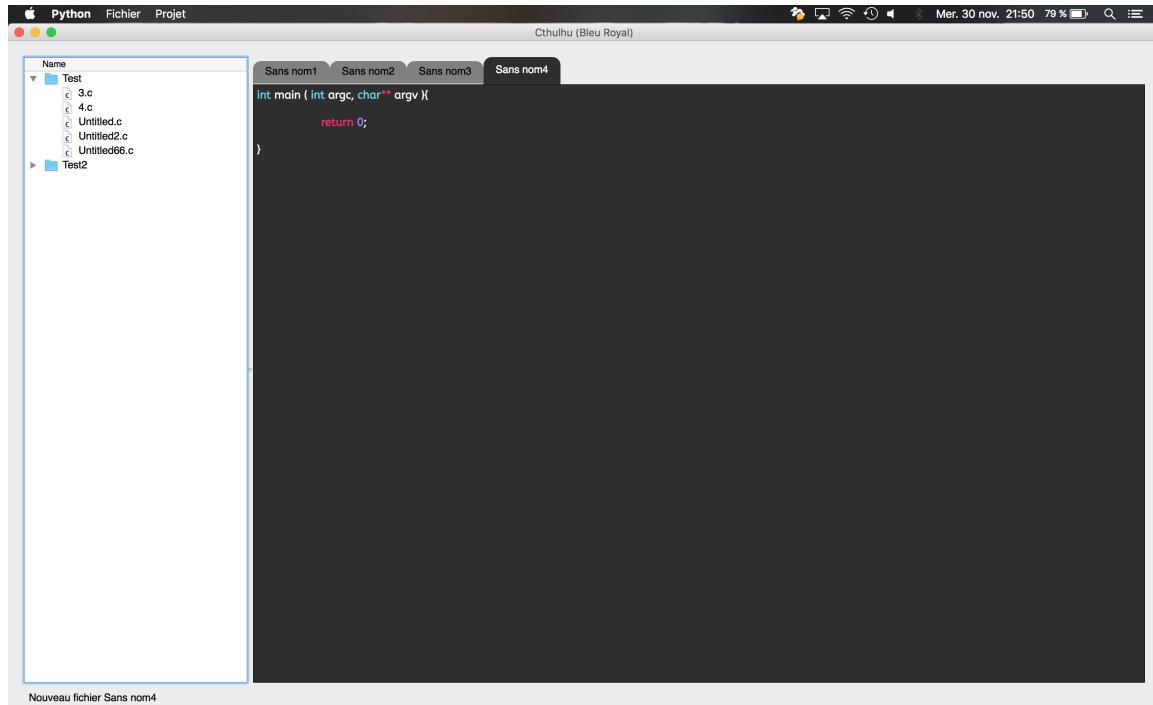
1	Introduction	2
1.1	Grammaire pour Lex et Yacc	2
1.2	Pour ce semestre	3
2	Les thèmes et les styles	4
2.1	Changer de thème	5
2.2	Gestion des thèmes	5
2.3	Création de thèmes	7
2.4	Style externe	7
3	Informations sur le code	8
3.1	Une autre barre de statut	8
3.2	Numérotation des lignes	9
3.2.1	Insertion dans l'interface graphique	9
3.2.2	Surcharger les méthodes originales	10
4	Recherche et édition du texte	11
4.1	Selection de la ligne courante	11
4.2	Selection du mot courant	12
4.3	Duplication	13
4.4	Recherche	14
4.5	Indentation du fichier	15
4.6	Commenter la sélection	16
4.7	Les snippets	16
4.7.1	Les snippets : explication	16
4.7.2	Intergration des bouts de code dans l'IDE	17
5	Fichier de configuration XML	17
5.1	Pourquoi le XML	17
5.2	Le module XML et le fichier de configuration	17
5.3	Les projets et le XML	18
6	Traitement des projets	19
6.1	La mise en place du concept de projet	19
6.2	La création d'un nouveau projet	19
6.3	Importation de projet	19
6.4	Suppression de projet	20
6.5	Informations de projet	20
6.6	Menu clic droit du navigateur de projets	21
7	Compilateur	23
7.1	Compilateurs utilisés	23
7.2	Integration de GCC à notre IDE	23
8	Inspecteur d'éléments	24
9	Personnalisation des raccourcis	25
9.1	Partie interface	25
9.2	Partie modification	26

10 Cache des documents	27
10.1 Utilité du cache	27
10.2 Intégration du cache dans l'IDE	27
11 Nouvelles grammaires	27
11.1 Grammaire Arithmétique	27
11.1.1 Lex	27
11.1.2 Yacc	28
11.2 Grammaire Python	29
11.2.1 Création de la grammaire	29
11.2.2 Utilisation de la librairie PlyPlus	29
12 Langue du système en Anglais	30
12.1 Gestion des langues	30
12.2 Rajouter une langue	31
13 Ajouts bonus	31

1 Introduction

Voici un résumé de ce que nous avions à la fin du premier semestre.

Notre IDE était capable d'ouvrir des documents avec l'extension .c ou .h à partir de projets qu'on avait créé. On pouvait ouvrir plusieurs documents, et avoir une liste d'onglets. Nous avions un navigateur de fichiers qui nous permettait de naviguer entre nos différents projets et leurs documents. Une barre de menu nous permettait d'accéder à nos différentes fonctionnalités, et on affichait à l'aide d'une barre de statut différents messages répondant aux requêtes de l'utilisateur.



Nous colorions le contenu des documents à l'aide du logiciel Lex, en particulier les différents tokens (éléments du code) selon leur fonction. Et nous analysions les documents pour détecter des erreurs de syntaxe à l'aide du logiciel Yacc. Nous utilisons pour cela une grammaire.

1.1 Grammaire pour Lex et Yacc

À l'aide du module PLY (Python Lex and Yacc), nous définissons une liste de tokens afin de déterminer tous les mots clefs du langage.

```
types = [
    "char",
    "bool",
    "double",
    "enum",
    "float",
    "int",
    "long",
    "short",
    "signed",
    "unsigned",
    "void"
]
```

FIGURE 1 – Exemple de liste de tokens pour les déclarations de type en C.

Nous avons aussi des fonctions à définir, pour Yacc afin de définir la syntaxe à suivre pour le langage (ici le langage C). Pour donner un exemple, sur les expressions de multiplication, nous indiquons qu'une multiplication peut prendre plusieurs formes.

```
def p_multiplicative_expression(p):
    '''multiplicative_expression : cast_expression
                                | multiplicative_expression TIMES cast_expression
                                | multiplicative_expression DIVIDE cast_expression
                                | multiplicative_expression MOD cast_expression'''
```

FIGURE 2 – Différentes formes d'expressions de multiplication.

Soit une "cast_expression", soit une expression multiplicative multipliée par une "cast_expression", soit une expression multiplicative divisée par une "cast_expression" ou soit expression multiplicative modulo une "cast_expression".

```
def p_cast_expression(p):
    '''cast_expression : unary_expression
                        | L_BRACKET type_name R_BRACKET cast_expression'''
```

FIGURE 3 – Une "cast_expression"

Une "cast_expression" étant définie par une expression unitaire ou bien un "type_name" entre parenthèses suivi d'une "cast_expression". Où un "type_name" est une autre fonction définissant ce que c'est.

```
def p_unary_expression(p):
    '''unary_expression : postfix_expression
                        | INC_OP unary_expression
                        | DEC_OP unary_expression
                        | unary_operator cast_expression
                        | SIZEOF unary_expression
                        | SIZEOF L_BRACKET type_name R_BRACKET'''
```

FIGURE 4 – Une expression unitaire

On décrit ci-dessus toutes les formes possibles d'expressions unitaires. C'est donc ainsi de suite et en remontant toutes les clauses que l'on définit les règles syntaxiques du langage.

1.2 Pour ce semestre

Nous avons beaucoup travaillé pour rajouter de nombreuses fonctionnalités à notre éditeur afin qu'il soit complet et qu'il ressemble à des éditeurs déjà existant au niveau du contenu.

Voici donc la liste de ce qui a été ajouté depuis les vacances de Noël :

- Différents thèmes et styles pour l'éditeur et ses éléments
- Traitement des projets
- Fichiers de configuration au format XML
- Intégration d'un compilateur
- Fonctionnalités diverses de recherche et d'édition de texte
- Inspecteur d'éléments
- Une autre barre de statut pour d'autres types d'informations

- Les numéros de lignes
- Fenêtre de paramétrage des raccourcis
- Ajout de cache afin d'optimiser l'ouverture de fichiers
- Grammaire arithmétique puis grammaire python
- Ajout de l'Anglais
- Quelques bonus

À noter que nous avions réalisé des rapports au fur et à mesure de l'avancement. Nous les reprennons ici donc en les ré-adaptant à l'aspect global évidemment. Mais il peut rester des images notamment sur la barre de menu où les fonctionnalités ne sont plus placées au même endroit. Par exemple, le menu apparence est maintenant placé dans Fichier/Paramètres.

2 Les thèmes et les styles

Afin de pouvoir rendre plus personnalisable l'application, nous avons choisi de permettre la personnalisation du thème global. Voici quelques exemples de thèmes :

```

#include <stdio.h>
#include <stdlib.h>

int triplaire(int nombre)
{
    return 3 * nombre;
}

int main(int argc, char *argv[])
{
    int nombreEntree = 0;
    printf("Entrez un nombre : ");
    scanf("%d", &nombreEntree);
    // Le résultat de la fonction est directement envoyé au printf et n'est pas stocké dans une variable
    printf("Le triple de ce nombre est %d\n", triplaire(nombreEntree));
    return 0;
}

int kafdg()
{
    return 0;
}

```

FIGURE 5 – Thème Forêt

```

#include <stdio.h>
#include <stdlib.h>

int triplaire(int nombre)
{
    return 3 * nombre;
}

int main(int argc, char *argv[])
{
    int nombreEntree = 0;
    printf("Entrez un nombre : ");
    scanf("%d", &nombreEntree);
    // Le résultat de la fonction est directement envoyé au printf et n'est pas stocké dans une variable
    printf("Le triple de ce nombre est %d\n", triplaire(nombreEntree));
    return 0;
}

int kafdg()
{
    return 0;
}

```

FIGURE 6 – Thème de base

FIGURE 7 – Thème Pastel

2.1 Changer de thème

Le changement de thème est très simple. Il suffit de se rendre dans le menu "Fichier/Paramètres" puis de choisir son thème parmi ceux proposés dans les catégories clairs et sombres.

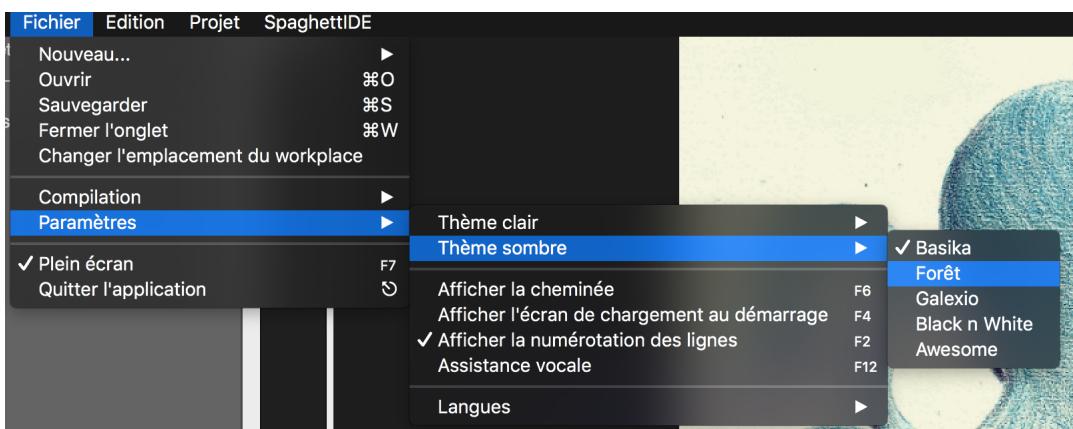


FIGURE 8 – Choix de son thème

Lorsqu'on sélectionne son thème il est immédiatement changé, il n'y a pas besoin de relancer l'application. De plus, une petite icône apparaît à côté du thème que vous avez choisi dans la barre de menu.

Le thème sélectionné est alors écrit de le fichier conf.xml qui est situé à la racine du projet. Ainsi, lorsque vous relancerez l'IDE, le dernier thème que vous avez utilisé sera rechargeé.

2.2 Gestion des thèmes

Les thèmes sont regroupés dans des répertoires distincts, le tout dans le répertoire "theme" situé à la racine du projet.

Chaque répertoire de thème regroupe les fichiers en format .json qui contiennent les couleurs en RGB de chaque élément de l'interface graphique.

Le module theme.py nous permet de récupérer le thème sauvegardé (dans le fichier conf.xml) lors du chargement de l'application notamment. Ici sont également contenues les méthodes per-

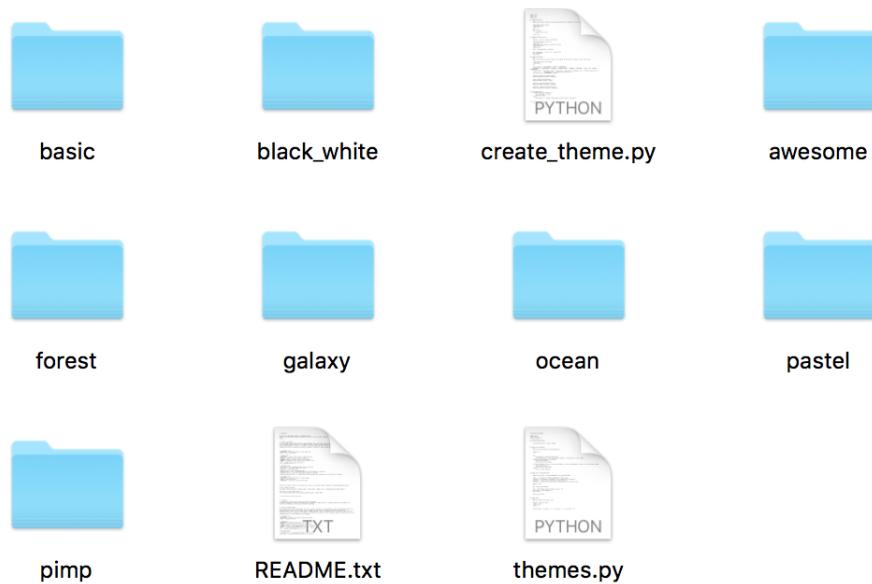


FIGURE 9 – Contenu du répertoire "theme"

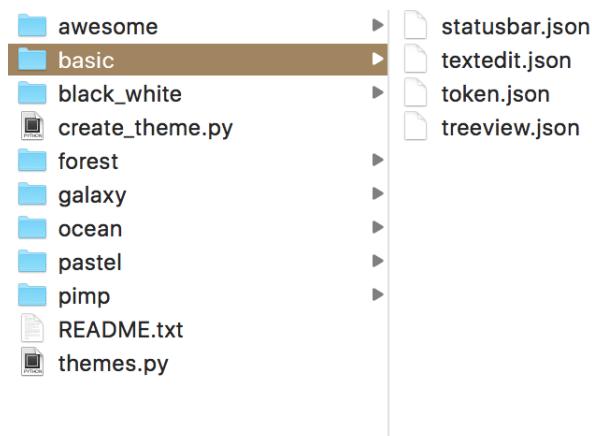


FIGURE 10 – Contenu du répertoire "basic" (le contenu est semblable pour tous les thèmes)

```

1  {
2      "text-back-color": [31, 31, 31],
3      "text-color": [255, 255, 255],
4      "tab-color": [0, 0, 0],
5      "tab-back-color": [100, 100, 100],
6      "tab-hover-back-color": [31, 31, 31],
7      "tab-hover-color": [255, 255, 255],
8      "tab-hover-bord-bot-color": [0, 0, 0]
9  }

```

FIGURE 11 – Exemple de fichier .json pour les thèmes.

mettant à l'interface d'aller chercher les couleurs qu'elle doit appliquer aux différents éléments.

Au niveau technique, nous utilisons la méthode `.setStyleSheet()` de QT qui peut s'appliquer à la majorité des widgets (tous dans notre cas) et qui nous permet donc de spécifier et de modifier les couleurs de fond ainsi que de police des widgets en fonction du thème choisi.

2.3 Crédit de thèmes

Vous pouvez utiliser les thèmes pré-définis, qui ont été pour la plupart validés et certifiés par la totalité du groupe comme étant jolis, mais vous pouvez aussi créer vos propres thèmes.

Le script "createtheme.py" vous permet cela, et la démarche à suivre est expliquée dans le fichier README.txt.

Pour résumer, on lance le fichier createtheme.py via un terminal en spécifiant le nom du thème. Tapez par exemple : "python3 createtheme.py monNouveauTheme" et cela créera un répertoire "monNouveauTheme" qui contiendra les fichiers nécessaires à la gestion de votre thème. Ouvrez ensuite les fichiers .json et définissez vos propres couleurs (par défaut tout est noir).

Une fois cela fait, vous devrez ajouter deux lignes dans le module menu.py pour que votre thème apparaisse dans la sélection.

```
nomTheme = MyAction(parent, "&monNouveauTheme", "monNouveauTheme", lambda: self.__change_theme_to("monNouveauTheme"))
```

FIGURE 12 – Ici nomTheme est le nom de la variable pour le thème, et monNouveauTheme le nom que vous avez donné à votre thème

```
self.set_group(nomTheme, groupe_theme, apparence_menu, "monNouveauTheme")
```

FIGURE 13 – Ici nomTheme est le nom de la variable pour le thème, et groupetheme le groupe (clair ou sombre) auquel appartient votre thème

Votre thème apparaît maintenant dans la barre de menu et il est sélectionnable !

2.4 Style externe

Dans le répertoire "gui" qui contient tout ce qui est relatif à l'interface graphique, il y a un répertoire "style" qui regroupe des éléments auxquels on applique également des feuilles de styles (via la méthode `setStyleSheet()` des widgets dans QT ; de même que pour les thèmes).

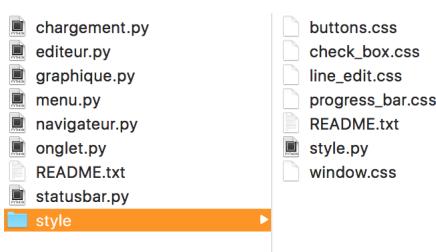


FIGURE 14 – Contenu du répertoire style

Chaque document .css contient le style relatif à des éléments. Nous retrouvons ici le style appliqué :

- aux boîtes de dialogue

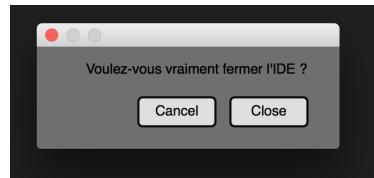


FIGURE 15 – Exemple de la fermeture de l’IDE : une popup qui apparaît demandant la confirmation.

- aux boutons, à qui on inverse les couleurs lorsqu’on les survole

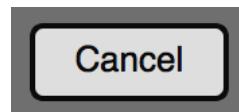


FIGURE 16 – Style appliquée à un bouton normal.



FIGURE 17 – Le même bouton lorsqu'il est survolé par la souris.

- à la barre de progression



FIGURE 18 – La barre de statut (voir ci-dessous).

3 Informations sur le code

3.1 Une autre barre de statut

Nous avons également travaillé à étoffer l’interface. Nous avons rajouté une seconde barre de statut (en bas à droite), servant à afficher des informations sur le code lui-même :



FIGURE 19 – Le nombre de lignes du fichier courant

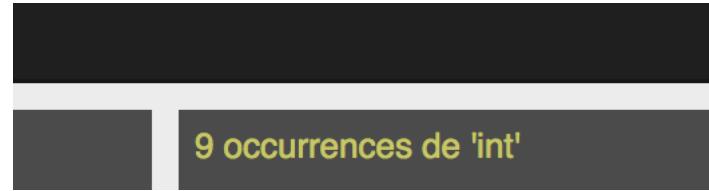


FIGURE 20 – Le nombre d’occurrences d’une recherche effectuée

Nous utilisons de plus cet emplacement (en bas à droite) pour afficher une barre de statut qui sert à indiquer la progression lors du chargement de projet (Yacc lisant tous les fichiers afin de récupérer les différentes fonctions à travers les modules, cela peut prendre plusieurs secondes).



FIGURE 21 – Barre de progression lors du chargement d’un projet de l’utilisateur

L’ouverture de projets est maintenant beaucoup plus rapide qu’au début du semestre, notamment grâce à l’utilisation du cache. La barre de progression apparaît toujours, mais c’est souvent très rapide.

3.2 Numérotation des lignes

3.2.1 Insertion dans l’interface graphique

L’objet que nous utilisons pour afficher et éditer le code est un QTextEdit de QT. Il ne possède pas de méthode pour afficher les numéros de lignes, et il n’y a pas d’autres alternatives à celui-ci pour faire cela non plus.

Il nous fallait donc un autre élément, afin d’afficher les numéros de lignes qui serait placé sur le côté, nous avons choisi le côté droit.

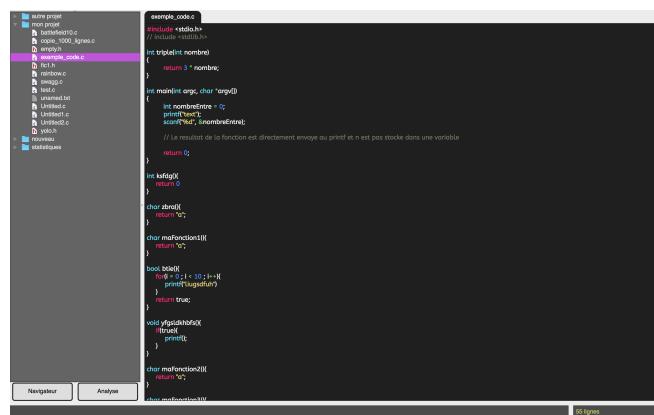


FIGURE 22 – Sans la barre de numérotation des lignes

The screenshot shows the SpaghettiIDE interface. On the left is a project tree titled "mon projet" containing files like "butterfield10.c", "1000_lignes.c", "example_code.c", "test.h", "test.c", "main.c", "fibonacci.c", "Unfiled.c", "fibonacci2.c", and "Vasch.c". The "example_code.c" file is selected and shown in the main code editor area. The code editor has a dark background with white text and includes line numbers from 1 to 47. Below the code editor are two buttons: "Navigateur" and "Analyses". In the bottom right corner of the code editor, it says "90 lignes".

FIGURE 23 – Avec la barre de numérotation des lignes

L'utilisateur peut choisir d'afficher ou non cette barre.

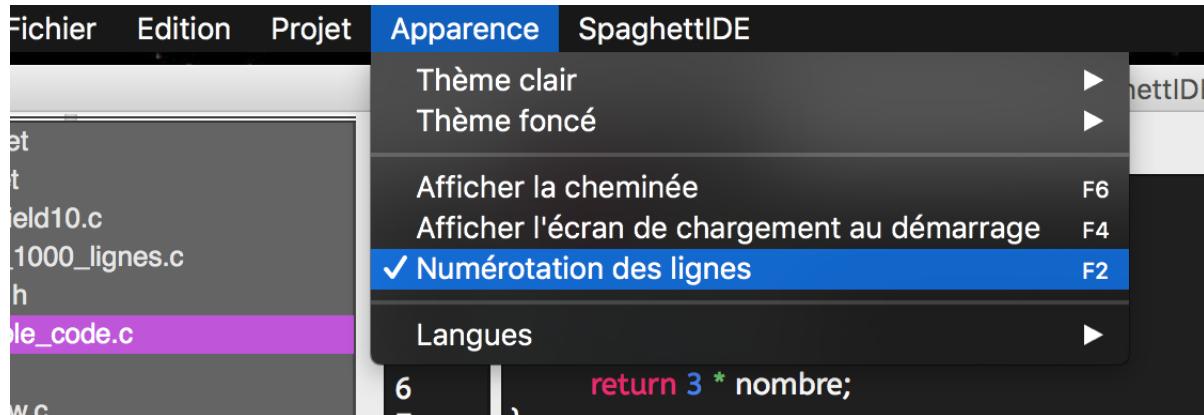


FIGURE 24 – Dans le menu apparence

3.2.2 Surcharger les méthodes originales

La difficulté n'était pas d'afficher une barre avec des numéros dedans, ni de récupérer le nombre de lignes, car nous avons une méthode pour cela. La difficulté était de synchroniser le défilement des deux éléments (widgets).

Lorsque l'on fait un défilement avec la molette de souris ou le pad **uniquement**, la méthode appelée sur le QTextEdit est le "wheelEvent(e)" où "e" est l'événement utilisé par QT pour effectuer le défilement.

Le principe consiste à appeler la fonction d'origine ainsi que celle de l'autre objet avec le même argument "e" lorsque la méthode "wheelEvent()" est appelée.

```

def wheelEvent(self, e, syncr=False):
    """
    Évenement appelé lors du scroll via la souris

    :param e: evenemnt
    :type e: object
    """
    QTextEdit.wheelEvent(self, e)
    if not syncr:
        self.master.codes[self.master.get_idx()].wheelEvent(e, True)

```

FIGURE 25 – Méthode de l'objet contenant la numérotation des lignes

Ici, "self.master.codes" désigne la liste des onglets de codes ouverts, et "self.master.get_idx()" retourne l'indice de l'onglet courant. On appelle donc la méthode "wheelEvent()" de l'onglet courant lorsque l'on fait défiler la liste de numérotation des lignes. L'argument booléen sert à dire qu'il ne faut pas rappeler la méthode "wheelEvent()" car sinon on rentrerait dans une boucle infinie.

```

def wheelEvent(self, e, syncr=False):
    """
    Évenement appelé lors du scroll via la souris

    On rappelle ici la même fonction sur l'objet pour afficher les lignes afin de les synchroniser.

    :param e: evenemnt
    :type e: object
    """
    QTextEdit.wheelEvent(self, e)
    if not syncr:
        self.parent.nb_lignes.wheelEvent(e, True)

```

FIGURE 26 – Méthode de l'objet contenant le code

"self.parent.nb_lignes" désigne l'objet contenant la numérotation des lignes, on y appelle donc la méthode "wheelEvent()" avec les mêmes arguments. Et toujours l'argument empêchant la boucle infinie.

Nous arrivons ainsi à synchroniser les deux éléments lors du défilement de l'un comme de l'autre avec la souris ou le pad.

4 Recherche et édition du texte

4.1 Sélection de la ligne courante

Dans un premier temps, nous avons ajouté une fonction permettant de sélectionner la ligne où se trouve le curseur. Pour cela, nous récupérons l'objet QTextCursor de notre classe Editeur (héritant de QTextEdit) puis nous utilisons la méthode **select** de cet objet qui nous permet de sélectionner du texte dans notre Editeur, cette méthode prend en paramètre une méthode de selection, QTextCursor.LineUnderCursor dans notre cas. La ligne où se trouve notre curseur

va donc être sélectionnée. Pour appliquer ces modifications, nous devons appliquer notre objet QTextCursor à notre Editeur, pour cela on utilise la méthode **setTextCursor** de l'objet Editeur et on lui passe en paramètre notre QTextCursor.

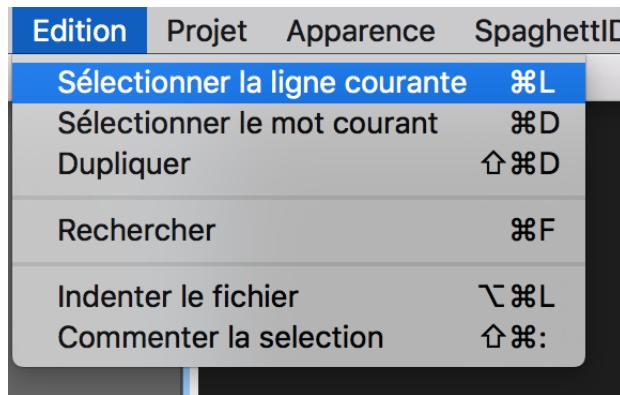


FIGURE 27 – Action du menu permettant la sélection de la ligne courante

```
int main (int argc, char** argv){  
    return 0;  
}
```

```
int main (int argc, char** argv){  
    return 0;  
}
```

FIGURE 28 – Résultat de l'utilisation de la fonction sélection de la ligne courante

4.2 Selection du mot courant

Pour l'ajout de la sélection du mot courant, la démarche est exactement la même que pour la sélection de la ligne courante, nous devons simplement changer la méthode de sélection, passant de QTextCursor.LineUnderCursor à QTextCursor.WordUnderCursor, afin de ne plus sélectionner la ligne mais le mot présent au niveau du curseur.

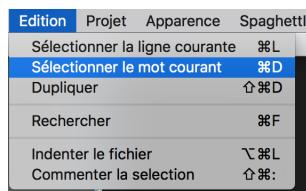


FIGURE 29 – Action du menu permettant la sélection du mot courant

```
int main (int argc, char** argv){  
    return 0;  
}
```

```
int main (int argc, char** argv){  
    return 0;  
}
```

FIGURE 30 – Résultat de l'utilisation de la fonction sélection du mot courant

4.3 Duplication

Pour l'ajout de la duplication du texte, nous avons choisi de différencier deux cas, le premier où rien n'est sélectionné et le second où du texte est déjà sélectionné. Dans le premier cas toutes la ligne est dupliquée et dans le second seulement la partie sélectionnée est dupliquée.

Pour cela, nous récupérons une nouvelle fois le QTextCursor de notre Editeur, puis pour savoir dans quel cas nous sommes on utilise la méthode selectedText de l'objet QTextCursor. Ainsi si aucun text n'est sélectionné nous sélectionnons la ligne courante de la même façon que précédemment de plus on assigne la valeur

n à la variable **return_** en effet si on duplique une ligne entière, on ajoute on retourne à la ligne entre la sélection d'origine et la partie dupliquée. Puis on ajoute le texte dans notre objet Editeur grâce à la méthode **insertText** avec en paramètre la sélection du QTextCursor (recupérée grâce à la méthode **selectedText**) suivie de la variable **return_** elle même suivie de la sélection du QTextCursor.

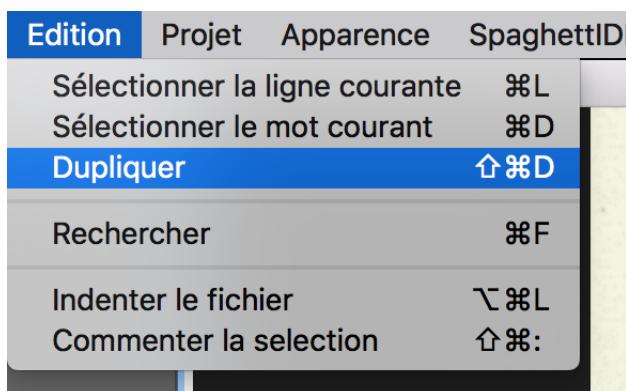


FIGURE 31 – Action du menu permettant de dupliquer

```
int main (int argc, char** argv){  
    return 0;  
    return 0;  
}
```

A screenshot of a code editor window showing two identical blocks of C code. Each block consists of the same three lines: 'int main (int argc, char** argv){', ' return 0;', and '}'. The code is color-coded with syntax highlighting, where 'int', 'main', 'argc', 'argv', 'return', and '}' are in blue, and 'char' is in cyan.

FIGURE 32 – Résultat de l'utilisation de la fonction permettant de dupliquer

4.4 Recherche

Pour la recherche dans le document, nous avons décidé d'ajouter une boîte de dialogue permettant d'entrer le texte à rechercher. Pour cela nous avons créé une classe SearchDialog (héritant de QDialog), lors de l'affichage de cette boîte de dialogue nous utilisons la méthode `exec`, qui rend impossible l'interaction avec la fenêtre en arrière plan tant que la boîte de dialogue est ouverte.

Cette boîte de dialogue nous permet de taper le texte à rechercher, de choisir si on recherche en avant ou en arrière, mais aussi si on veut être sensible à la case.

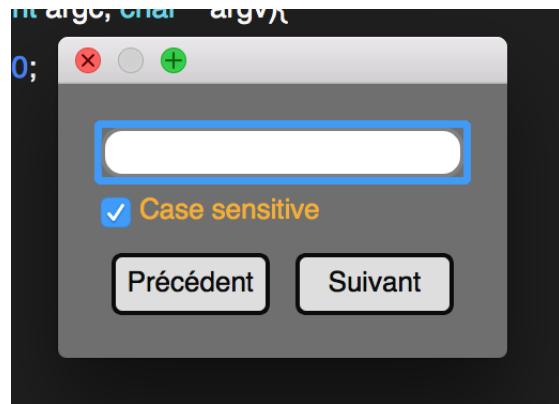


FIGURE 33 – Boîte de dialogue relative à la recherche

Pour la recherche on utilise la méthode `find` de notre objet Editeur. Cette méthode prend en paramètre le texte à rechercher, suivi de différents drapeaux. Dans notre cas nous utilisons le drapeau permettant d'exécuter la recherche en arrière et le drapeau permettant de faire la recherche en étant sensible à la case (respectivement les drapeaux `QTextDocument.FindBackward` et `QTextDocument.FindCaseSensitively`)

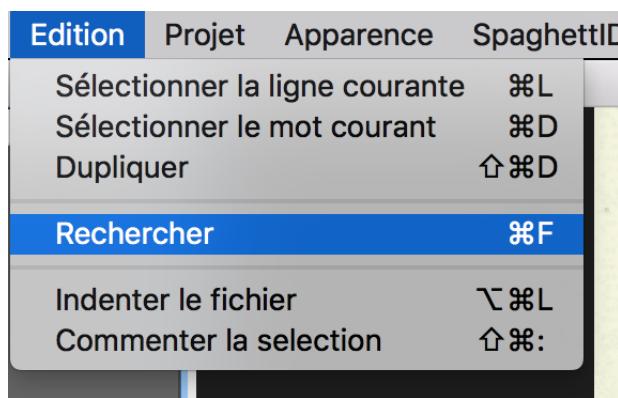


FIGURE 34 – Action du menu permettant de rechercher

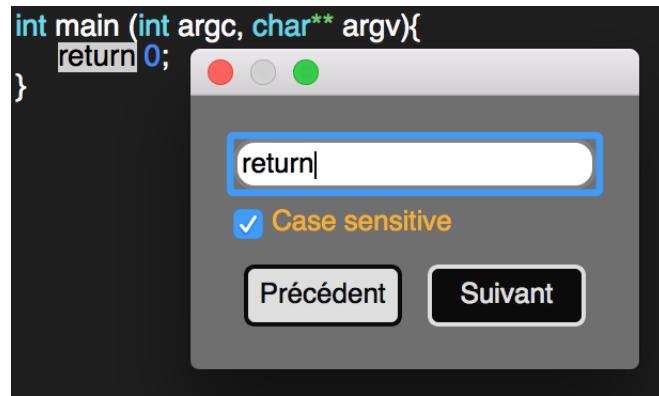


FIGURE 35 – Résultat de l'utilisation de la fonction permettant de rechercher

4.5 Indentation du fichier

Pour l'indentation du document, nous allons changer son contenu (ajout/retrait de tabulation) nous devons donc stocker la position courante du curseur (grâce à la méthode **blockNumber** de l'objet QTextCursor). Par la suite on récupère le contenu du document grâce à la méthode **toPlainText** de l'objet Editeur. On crée une variable **indent_level**, qui contient le niveau courant d'indentation, puis on parcourt toutes les lignes de notre document, si la ligne contient le caractère "}", on retire 1 au niveau d'indentation puis on change la ligne pour ajouter au début de cette dernière **indent_level** fois une tabulation, puis on ajoute 1 au niveau d'indentation si la ligne contient "{". Pour finir on définit le nouveau texte ainsi obtenu comme texte de notre document avec la méthode **setPlainText** et on replace le curseur au bon endroit.

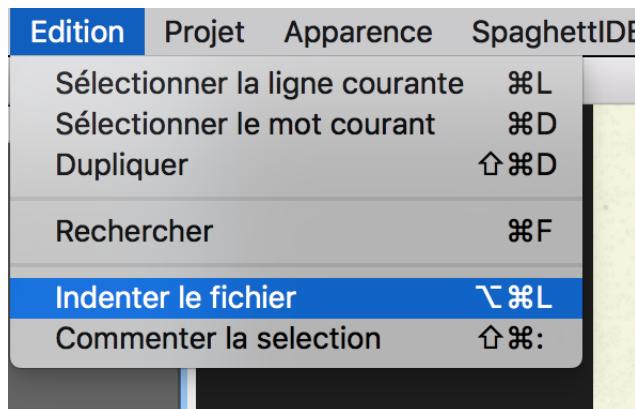


FIGURE 36 – Action du menu permettant d'indenter le fichier



FIGURE 37 – Résultat de l'utilisation de la fonction permettant d'indenter le fichier

4.6 Commenter la sélection

De la même façon que pour la duplication du texte, nous avons séparé cette action en deux cas, soit du texte est sélectionné soit rien n'est sélectionné. Dans le cas où du texte est sélectionné nous commenterons seulement à partir du début de la sélection. Si plusieurs lignes sont sélectionnées elles seront évidemment toutes commentées. Si il n'y a pas de texte sélectionné, on commente la ligne courante.

Dans un premier temps, comme pour la duplication, si rien n'est sélectionné on sélectionne la ligne courante, puis on sauvegarde le texte sélectionné que l'on récupère grâce à la méthode **selectedText**, puis on supprime le texte sélectionné avec la méthode **removeSelectedText**. Pour savoir si le texte est déjà commenté, nous parcourons toutes les lignes, si une des lignes ne commence pas par "://" la sélection est considérée comme non commentée. Par la suite nous parcourons chaque ligne du texte puis pour chaque ligne nous ajoutons/retirons les caractères "://" en fonction de si le texte est oui ou non déjà commenté.

Puis on ajoute le texte ainsi modifié à notre document en utilisant la méthode **insertText** (de l'objet **QTextCursor**).

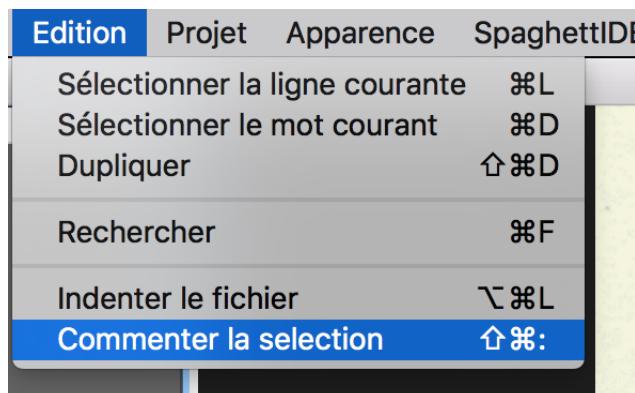


FIGURE 38 – Action du menu permettant de commenter le fichier

The image shows two side-by-side snippets of C code. On the left, the code is:

```
int main (int argc, char** argv){  
    return 0;  
}
```

On the right, the code is identical but includes a double slash comment at the end of the first line:

```
int main (int argc, char** argv){  
//    return 0;  
}
```

FIGURE 39 – Résultat de l'utilisation de la fonction permettant de commenter le fichier

4.7 Les snippets

4.7.1 Les snippets : explication

Les snippets sont des bouts de code qui sont réutilisables d'un projet à un autre (comme les conditions, les boucles et les définitions de fonction par exemple). Dans notre IDE nous utilisons des mots clés que l'on associe à ces bouts de code. Lorsque l'on tape un des mots clés puis que l'on presse la touche **tabulation**, le mot clé est remplacé par le bout de code qui correspond.

4.7.2 Intergration des bouts de code dans l'IDE

Utilisation du JSON Pour l'intégration des snippets dans l'IDE, nous avons utilisé des fichiers JSON, qui contiennent des dictionnaires avec comme clé le mot clé qui correspond au bout de code et en valeur le-dit bout de code, suivi de nombre de lignes que le cursor doit remonter, puis le nombre de caractère char que le cursor doit parcourir vers la droite pour être à l'emplacement voulu. De plus nous avons un fichier JSON par language (C, python, etc ...)

```
{  
    "int" : ["int maFonction(){\n\treturn 0;\n}", 2, 12],  
    "void" : ["void maFonction(){\n\t", 2, 12],  
    "bool" : ["bool maFonction(){\n\treturn true;\n}", 2, 12],  
    "char" : ["char maFonction(){\n\treturn \"a\";\n}", 2, 12],  
    "double" : ["double maFonction(){\n\treturn 0.0;\n}", 2, 12],  
    "if" : ["if(true){\n\tsomeStuffHere();\n}", 2, 4],  
    "elseif": ["else if(true){\n\tStuffHere();\n}", 2, 10],  
}
```

FIGURE 40 – Extrait du fichier JSON pour le language C

Connection entre le JSON et l'IDE Pour capturer l'appui sur la touche **tabulation**, nous utilisons la méthode **keyPressEvent** de notre classe **Editeur**. Cette méthode est appelée à l'appui sur une des touches du clavier. Par la suite, si la touche est la touche **tabulation**, nous parsons le JSON pour avoir tous les couples (mot clé, bout de code). Puis en fonction du mot placé au niveau du curseur, on le remplace ou non par le bout de code correspondant.



FIGURE 41 – Exemple d'utilisation des snippets avec une boucle for en Python

5 Fichier de configuration XML

5.1 Pourquoi le XML

Au fur et à mesure de l'ajout de fonctionnalités dans notre IDE, l'utilisation du XML s'est vite imposée du fait de son efficacité et également par convention. Nous devons ainsi lire, parcourir et écrire dans un fichier XML afin de subvenir à nos besoins dans l'élaboration des différentes fonctionnalités de notre IDE dans différentes conditions. Nous utilisons plusieurs fichiers xml, un pour chaque projet, un pour tous les projets et un fichier de configuration.

5.2 Le module XML et le fichier de configuration

Nous avons donc ainsi créé un module XML pour les fonctions de parcours et d'écriture de fichier XML et bien entendu, le fichier de configuration XML. Voici d'ailleurs leur disposition dans notre IDE, ils sont à la racine du projet.

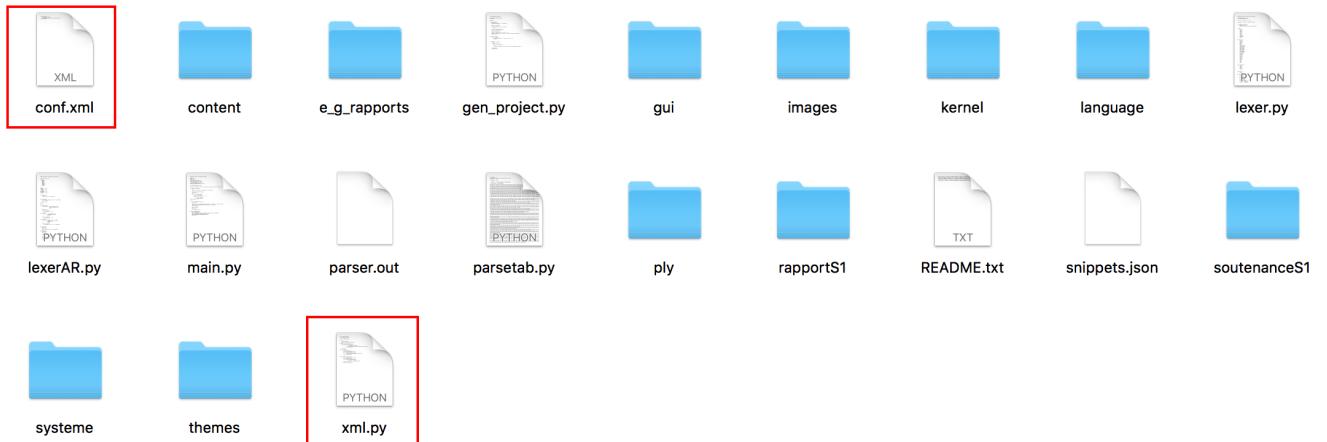


FIGURE 42 – Disposition du module et du fichier de configuration

Dans le module XML, nous avons une fonction `open_xml`, qui ouvre, parse et parcours le fichier de configuration XML pour retourner un dictionnaire avec comme clés le nom des balises XML, et comme valeurs la valeur des balises. Nous avons également une fonction `write_xml` qui permet d'écrire des modifications dans le fichier de configuration. On rentre le nom d'une balise et la valeur voulue en arguments, la fonction trouve la balise et lui associe cette valeur, puis nous écrivons dans le fichier. Enfin, pour éviter tout problème, si le fichier de configuration n'existe pas ainsi que le fichier comportant tous les projets, on les crée et leur ajoute les valeurs de configuration de base.

Dans le fichier de configuration ".xml", nous utilisons une balise pour stocker : le thème sélectionné actuellement, le booléen suivant l'activation ou non de l'assistance vocale dans l'interface graphique, le booléen suivant l'activation ou non de l'écran de chargement au démarrage, celui de la numérotation des lignes, le langage de l'IDE (Français ou Anglais) ainsi que la localisation du workplace (voir partie suivante).

```
<configuration>
    <theme>ocean</theme>
    <assistance_vocale>False</assistance_vocale>
    <loading>False</loading>
    <numerote_lines>True</numerote_lines>
    <language>en</language>
    <current_workplace>/Users/alexiscarreau/workplace/</current_workplace>
</configuration>
```

FIGURE 43 – Fichier de configuration

5.3 Les projets et le XML

Il y a un fichier ".xml" dans chaque projet. Ce fichier contient notamment le langage du projet, la date de création, son nom, son emplacement, le nombre de fichiers qu'il contient, l'emplacement de son compilateur et l'emplacement de son fichier de sortie.

On peut donc aisément rajouter et récupérer les informations qu'il contient afin de les utiliser pour diverses raisons, comme compiler le projet sans avoir à remettre toutes les options nécessaires.

```

<project>
  <name>Projet</name>
  <creation_date>2017-04-10 23:27:48.479969</creation_date>
  <language>C</language>
  <location>/Users/alexiscarreau/workplace/Projet</location>
  <number_files>5</number_files>
  <compil></compil>
  <compil_json></compil_json>
</project>

```

FIGURE 44 – Fichier de configuration

6 Traitement des projets

6.1 La mise en place du concept de projet

Le concept, le principe de projet dans IDE consiste à avoir un dossier Workplace qui comporte tous les dossiers de projet et ce dossier Workplace peut être placé où on veut. Après avoir réalisé ce concept, il ne restait plus qu'à faire les différentes fonctions associées aux projets, comme la création d'un nouveau projet, l'import, la suppression et les informations de chaque projet.

6.2 La création d'un nouveau projet

Pour créer un nouveau projet, il suffit de cliquer sur le bouton du menu "Nouveau projet", une fenêtre s'ouvre et propose de choisir un nom de projet et un langage et après validation le dossier de projet est créé avec l'xml contenant ses informations dedans. Le dossier est considéré comme un projet et est stocké dans le fichier xml contenant tous les projets.

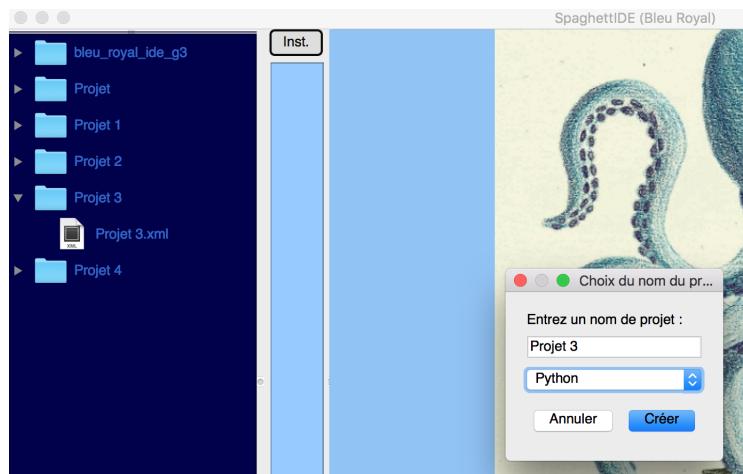


FIGURE 45 – Interface graphique de création de projet

Nous pouvons maintenant ajouter du code et donc des fichiers dans le nouveau projet créé.

6.3 Importation de projet

Pour importer un projet, un ancien projet en fait, dans le Workplace ou ailleurs, qui ne comprend donc pas de fichier xml, qui n'est donc pas considéré comme un projet encore par l'IDE, il suffit de cliquer sur le bouton du menu "Importer projet". Il l'ajoute au Workplace avec un fichier xml et fait toutes les formalités comme lors de la création d'un projet. Tous

les cas sont gérés, à savoir par exemple, l'ajout d'un projet déjà existant, le projet n'est pas écrasé mais rien ne se passe. Il est également possible d'importer un ancien projet qui est dans le Workplace avec le menu clic droit du navigateur de projets et en cliquant sur "Importer le projet".

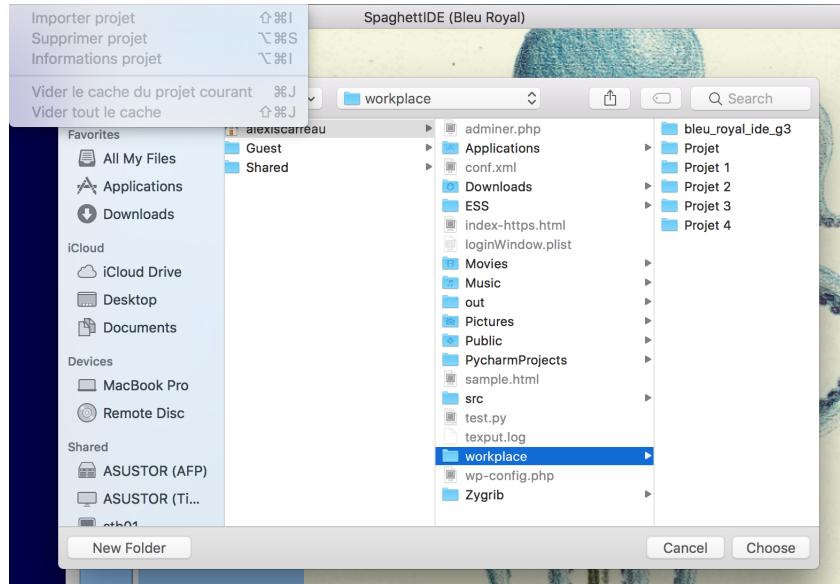


FIGURE 46 – Interface graphique d'import de projet

6.4 Suppression de projet

Pour supprimer un projet, il suffit de cliquer sur le bouton du menu "Supprimer projet". Ainsi, le dossier projet est supprimé du Workplace, nous avons utilisé la commande "shutil.rmtree". Avec la même méthode, nous pouvons supprimer un fichier et même un dossier dans un projet, via par contre le menu clic droit du navigateur de projets, dont nous allons parler dans une prochaine partie.

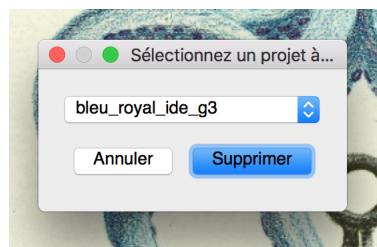


FIGURE 47 – Interface graphique de suppression de projet

6.5 Informations de projet

Pour accéder aux informations de projet, il suffit de cliquer sur le bouton du menu "Informations projet". Ainsi, une fenêtre s'ouvre et propose de choisir un projet. Une fois le projet choisi, ses informations sont affichées, soit le nom, le langage, la localisation, la date de création et le nombre de fichiers du projet. On peut également modifier les informations, enfin seulement le nom et le langage du projet, pour finir il faut simplement appliquer les modifications effectuées avec le bouton correspondant. Vous pouvez le voir avec les images ci-dessous :

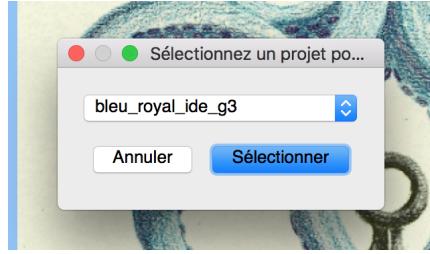


FIGURE 48 – Interface graphique de sélection d’informations de projet

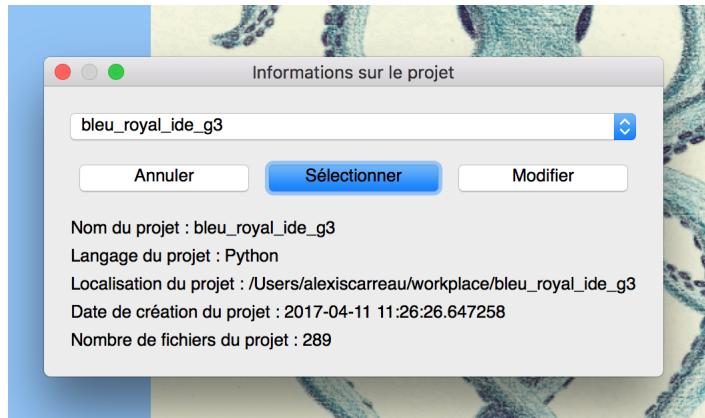


FIGURE 49 – Interface graphique des informations de projet

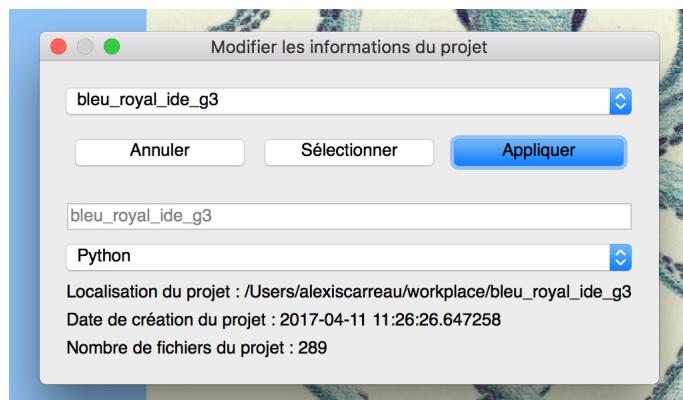


FIGURE 50 – Interface graphique de modification des informations de projet

6.6 Menu clic droit du navigateur de projets

Un menu s’affichant par un clic droit dans le navigateur de projets a été mis en place permettant d’importer un projet, de le supprimer et d’afficher ses informations. Il est également possible de renommer et de supprimer les dossiers et fichiers présents un projet. C’est donc les mêmes fonctionnalités que précédemment (avec en plus la suppression et la possibilité de renommer les dossiers et fichiers de projets) mais présentées de façon plus pratique, efficace, optimale pour travailler. Vous pouvez le voir avec les images ci-dessous :



FIGURE 51 – Importer projet via le menu clic droit

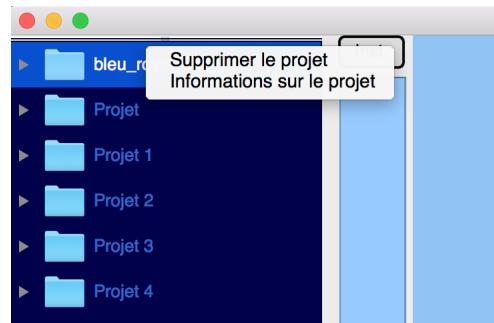


FIGURE 52 – Suppression et informations de projet via le menu clic droit

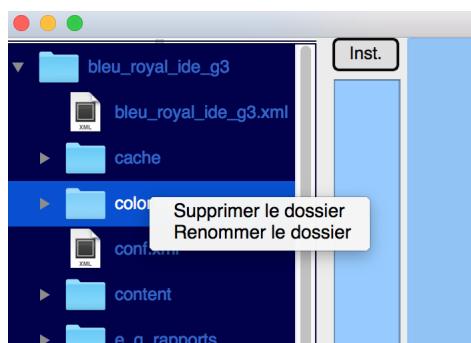


FIGURE 53 – Suppression et possibilité de renommer un dossier



FIGURE 54 – Suppression et possibilité de renommer un fichier

Lorsque que l'on clique sur le bouton pour renommer, nous obtenons l'affichage d'une fenêtre proposant d'écrire un nouveau nom :



FIGURE 55 – Renommer un dossier ou un fichier de projet

Lorsque l'on clique sur le bouton pour afficher les informations de projet, nous obtenons un affichage différents de cette même fonctionnalité présentée dans la partie précédente étant donné qu'il faut présenter les informations d'un seul projet. Il est impossible de modifier les informations de part ce menu clic droit, ni de renommer le nom de projet, il faut pour cela faire comme expliqué dans la partie précédente via le menu principal. Nous pouvons le voir avec l'image ci-dessous :

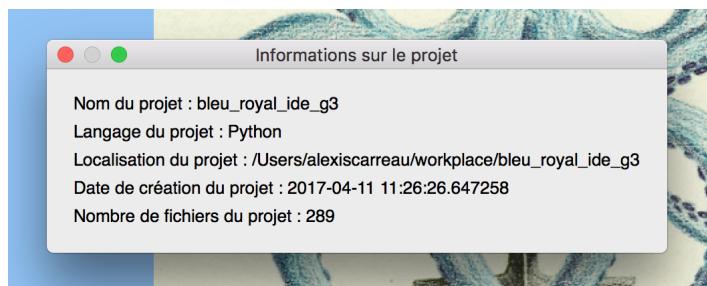


FIGURE 56 – Rendu des informations de projet via le menu clic droit

7 Compilateur

7.1 Compilateurs utilisés

Pour notre IDE, nous avons utilisé le compilateur **GCC**, pour compiler les projets de type C. Mais nous utilisons aussi les interpréteur de python afin d'interpréter les projets de type **Python**. Par la suite, pour chaque nouveaux langages il nous suffit d'utiliser le bon compilateur ou le bon interpréteur.

7.2 Integration de GCC à notre IDE

Pour utiliser **GCC** avec notre IDE nous avons créé une fenêtre de configuration afin de permettre à l'utilisateur de choisir la configuration adéquate à la compilation de son projet.

Chaque information de cette fenêtre est convertie en une chaîne de caractère du type :

```
gcc /Users/theosarrazin/workspaceC/ProjetDeTest/test.c -I /Users/theosarrazin/workspaceC/
ProjetDeTest/header -o MonExecutable
```

Cette chaîne de caractère est stockée dans le fichier XML de configuration du projet.

Pour finir, on exécute cette ligne de commande, puis on récupère sur la sortie d'erreurs les éventuelles erreurs afin de pouvoir les afficher à l'utilisateur après les avoir parsé pour avoir un bon formatage.

En ce qui concerne la partie interpréteur pour le langage Python, la démarche est semblable.

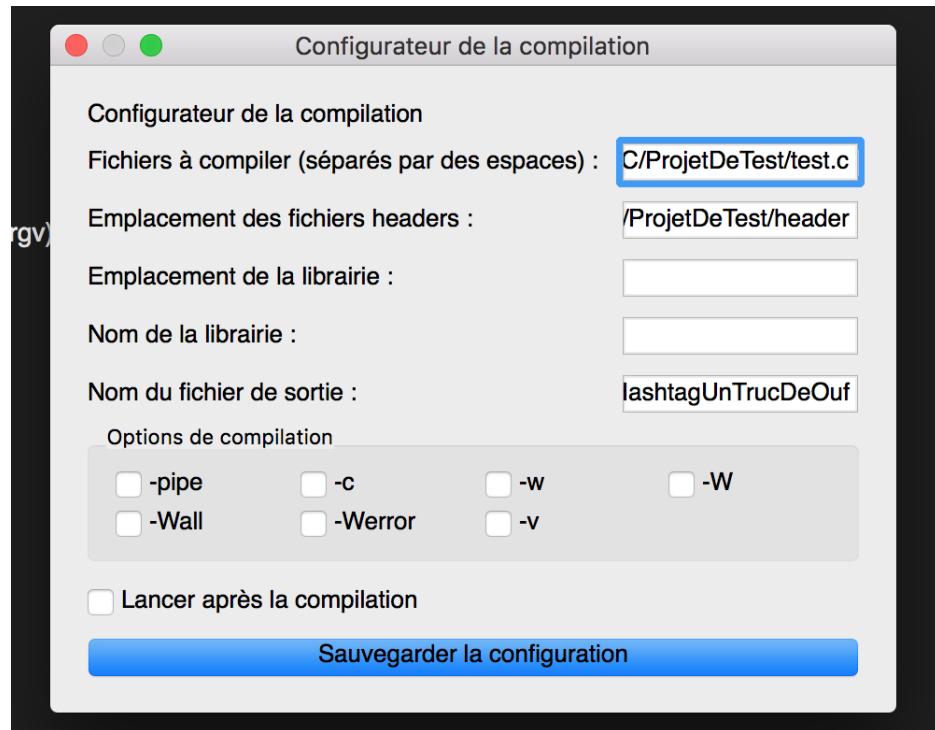


FIGURE 57 – Fenêtre de configuration de GCC

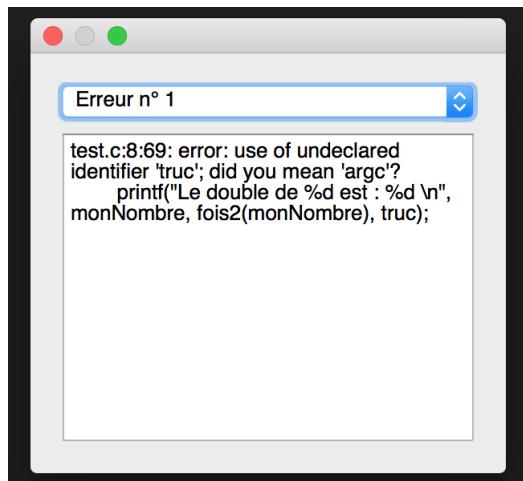


FIGURE 58 – Affichage des erreurs de GCC dans l'IDE

8 Inspecteur d'éléments

Lorsqu'un document est ouvert, nous avons la possibilité d'afficher l'inspecteur d'éléments. Pour cela, nous avons le bouton "Navigateur" sous le navigateur de fichiers, et lorsque l'on clique dessus, on affiche l'inspecteur à la place. De même si on re-clique dessus, on ré-affiche le navigateur.

Au niveau du code, le navigateur de fichiers est créé à l'aide d'un QTreeView (comme au premier semestre), et il est remplacé par un QTextEdit qui constitue l'inspecteur lorsqu'on clique sur le bouton.

On affiche dans l'inspecteur toute la structure du code grâce au logiciel Yacc, qui peut nous



FIGURE 59 – Respectivement l’inspecteur et le navigateur de fichiers sont affichés

renvoyer des listes contenant pour un fichier :

- Ses variables
- Ses fonctions
- Ses struct (en C) ou ses Class (en Python)

Lorsque l’on clique sur un élément par exemple un nom de variable, cette variable est sélectionné dans le document courant.

9 Personnalisation des raccourcis

Dans le menu de notre IDE, vous avez pu remarquer qu’à côté de certaines options, il y avait des raccourcis que nous avions choisis en fonction de ce qui nous semblait être le mieux. Mais ces choix étaient personnels, et tous les membres du groupe n’approuvant pas certains choix, nous avons décidé de les rendre personnalisables.

9.1 Partie interface

Nous avons trois menus dans lesquels il y a des raccourcis. Ainsi, nous avons décidé de créer un fichier "racc_defaut.json" qui contient un dictionnaire ayant pour clefs le noms de ces trois

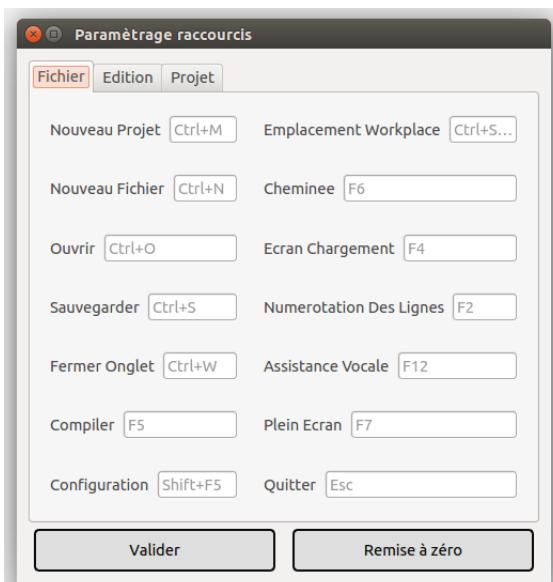
menus, et comme valeurs d'autres dictionnaires qui ont pour clefs les noms des fonctionnalités et comme valeurs une chaîne de caractères représentant le raccourci.

```
{
  "Fichier": {
    "Nouveau Fichier": "Ctrl+N",
    "Nouveau Projet": "Ctrl+M",
    "Ouvrir": "Ctrl+O",
    "Sauvegarder": "Ctrl+S",
  }
}
```

Ici, un aperçu du fichier "racc_defaut.json".

Nous pouvons donc conserver tous les raccourcis de base et en proposer dès le lancement de l'IDE, au lieu de forcer l'utilisateur à en définir. Il peut donc choisir, via le menu "Fichier", dans le sous-menu "Paramètres", l'option Raccourcis, qui modifiera le fichier "racc_utilisateur.json" créé lors du lancement de l'IDE si celui-ci n'existe pas déjà.

Cliquer sur cette option lui ouvrira une nouvelle fenêtre (une QDialog), dans laquelle trois onglets (des QTabWidget) seront disponibles. Ces trois onglets correspondent aux trois menus qui contiennent des raccourcis. Dans ces onglets, des lignes de saisies (des QLineEdit) seront apparentes permettront à l'utilisateur de rentrer ses propres raccourcis. Nous avons mis les raccourcis de base en fond pour que l'utilisateur ait un modèle. Cette interface est complétée par deux boutons, l'un qui valide les raccourcis et l'autre qui les remet à zéro, c'est-à-dire qu'il modifie les valeurs contenues dans le fichier "racc_utilisateur.json", et les remplace par les valeurs contenues dans le fichier "racc_defaut.json".



Un aperçu de la fenêtre permettant la modification des raccourcis.

9.2 Partie modification

En ce qui concerne la modification des raccourcis, il est possible de d'écrire tout ce que l'on veut dans les lignes de saisies. C'est lorsque l'on clique sur le bouton valider que la fonction qui valide un raccourci va être appelée. Ainsi, certaines combinaisons de touches ne sont pas permises, telles que "M+Shift+Ctrl". Nous avons aussi décidé de faire en sorte que si l'utilisateur écrit quelque chose de ce style : "Ctrl+Shift+JHGFFUTRF", on écrive dans le fichier "racc_utilisateur.json" : "Ctrl+Shift+J".

10 Cache des documents

10.1 Utilité du cache

Afin de pouvoir faire de la coloration, nous avons utilisés **Lex** mais pour les fichiers de plusieurs centaines de lignes **Lex** prend plusieurs secondes pour nous donner la listes des tokens du fichier. Nous avons donc décidé d'utiliser un fichier de cache afin de ne pas utiliser **Lex** pour colorer des lignes que nous avons déjà coloré auparavant.

10.2 Intégration du cache dans l'IDE

Pour le cache, nous avons décidé d'utiliser un fichier JSON pour l'enregistrer. Le fonctionnement est assez simple la clé est le bout de code que l'on donne à **Lex** et la valeur est la réponse de ce dernier. Par la suite il nous reste plus qu'à utiliser les valeurs du fichier JSON à la place de **Lex**

```
{  
    "\tint monNombre;": [{"INT", "int"}, {"IDENTIFIER", "monNombre"}, {"SEMICOLON", ":"}],  
    "\treturn 2*a;": [{"RETURN", "return"}, {"CONSTANT", "2"}, {"TIMES", "*"}, {"IDENTIFIER", "a"}, {"SEMICOLON", ";"}],  
    "\tprintf(\"Le double de %d est : %d \\\n\", monNombre, fois2(monNombre), truc)": [{"IDENTIFIER", "printf"}, {"L_BRACKET": ["\t"]}],  
}
```

FIGURE 60 – Extrait d'un JSON contenant le cache

Grâce au cache nous avons un gain de temps considérable sur l'ouverture des documents.

Nous avons créé deux fonctions, l'une vide le cache du projet courant et l'autre vide tout le cache.

11 Nouvelles grammaires

11.1 Grammaire Arithmétique

11.1.1 Lex

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

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**

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...)

```
t_PLUS      = r'\+'
```

FIGURE 61 – Exemple de création d'une expression régulière

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 trouvé 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 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 auquel on récupère le lexer.

11.1.2 Yacc

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

Pour définir des règles nous devons créer des fonctions nommées de la façon suivante : **p_NomDeLaRègle**. 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 62 – 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.

Création du parser Par la suite, nous devons appliquer nos règles sur du contenu, pour cela nous avons créé 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.

11.2 Grammaire Python

11.2.1 Création de la grammaire

Dans un premier temps, nous avions commencé à rédiger une grammaire pour Python. Cependant nous avons rencontré plusieurs problèmes, notamment le fait qu'il n'est pas possible d'avoir des tokens imbriqués les uns dans les autres rendant ainsi les expressions régulières de **Lex** bien trop complexes. De plus la gestion de l'indentation n'était pas facile. En effet **PLY** ne peut pas analyser l'indentation par ces grammaires, il nous faut donc valider chacun des tokens identifiés par **Lex** dans un second temps. Chose qui n'est pas simple à faire avec la librairie **Ply**. Voilà donc pourquoi nous avons décidé d'utiliser la librairie **PlyPlus**

11.2.2 Utilisation de la librairie PlyPlus

Pour pallier aux problèmes précédents, nous avons donc choisi d'utiliser la librairie **PlyPlus**. Cette librairie se base sur la librairie **Ply** sur laquelle nous avons travaillé jusqu'à présent, il nous a donc été simple d'ajouter **PlyPlus** à notre projet. **PlyPlus** a plusieurs avantages, dont notamment le fait d'avoir un système de grammaire totalement différent de **Ply**, nous permettant ainsi de résoudre les problèmes cités auparavant. Ce ne sont pas les seuls avantages de **PlyPlus**. En effet il nous permet aussi de sélectionner très simplement un type d'élément (le nom des fonctions ou des classes présentes par exemple) rendant ainsi possible leur coloration. De plus **PlyPlus**, vient avec une grammaire python et un post-analyseur syntaxique permettant ainsi de traiter l'indentation du fichier.

Nous utilisons donc **PlyPlus** pour interpréter notre grammaire Python et nous gardons **Ply** pour la grammaire C, car elle suffit pour ce langage.

12 Langue du système en Anglais

12.1 Gestion des langues

Nous avons ajouté la possibilité de mettre la langue de l'IDE en Anglais en plus du Français. Il suffit d'aller dans : Fichier/Paramètres/Langue et de choisir sa langue.

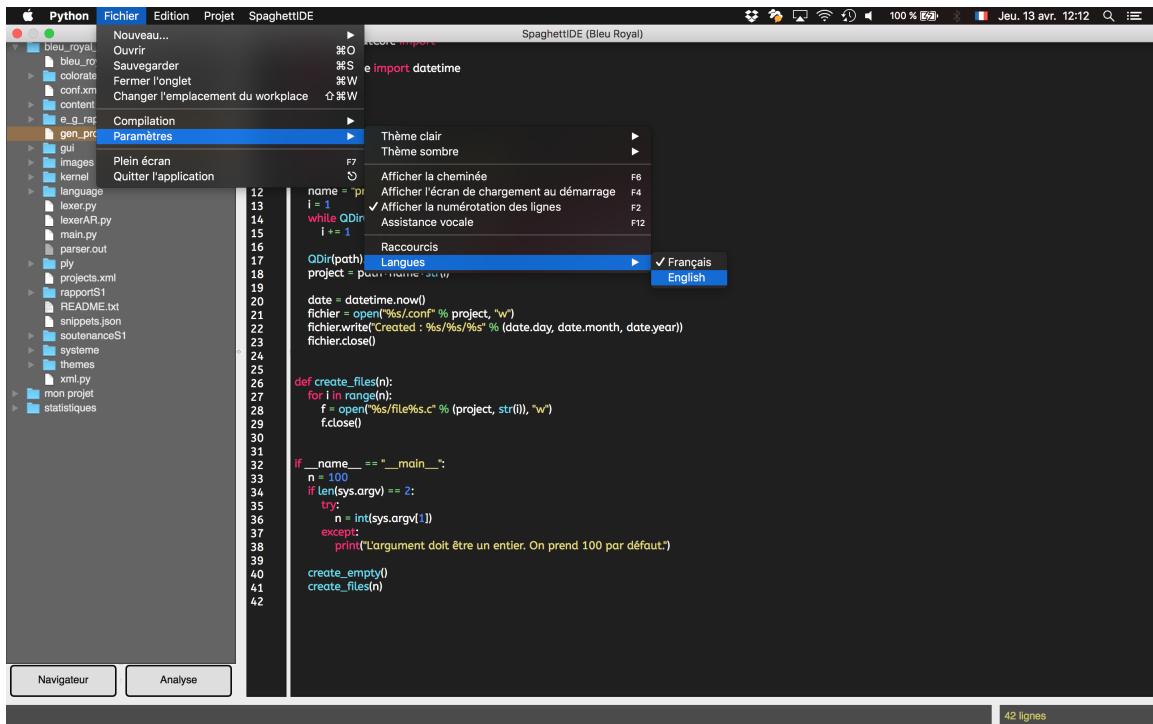


FIGURE 63 – Dictionnaire des textes

Il y a un répertoire langue situé à la racine du projet qui contient le module language.py qui permet de gérer le changement de langue ainsi que la récupération des textes selon la langue.

Pour l'aspect technique, nous utilisons un dictionnaire contenu dans un fichier ".json", qui associe à chaque élément sa traduction Française et Anglaise comme le montre l'exemple suivant.

```
"status_fic_analys": {
    "fr": "Le fichier courant a bien été analysé.",
    "en": "Current file has been successfully processed."
}, 

"status_fic_closed": {
    "fr": "Le fichier sélectionné a bien été fermé.",
    "en": "Current file has been successfully closed."
},
```

FIGURE 64 – Dictionnaire des textes

Pour chaque message, texte à afficher ou texte du menu, nous allons chercher la clef correspondante dans ce dictionnaire afin de l'afficher.

A noter que nous utilisons un second dictionnaire réservé aux textes de la barre de menu, cela nous permettait juste de ne pas avoir un très long fichier ".json" mais de répartir en deux pour des raisons de lisibilités. Mais le fonctionnement est strictement le même.

Nous n'avons pas trouvé de moyen d'actualiser la barre de menu lors du changement de langue. Il faut donc redémarrer l'IDE afin d'avoir les textes de la barre de menu dans la nouvelle langue choisie. En revanche tous les autres textes, c'est à dire les messages affichés sont directement dans la nouvelle langue.

12.2 Rajouter une langue

Si nous souhaitons intégrer une nouvelle langue, rien de plus simple !

Il nous suffit juste d'ajouter pour chaque textes des dictionnaires la langue et sa traduction. Par exemple si on ajoute l'espagnol, sur la figure 63, on aura une clef supplémentaire "es" à chaque sous-dictionnaire qui aura pour valeur le texte correspondant à la traduction.

13 Ajouts bonus