Analyse Syntaxique - Rapport de projet

Florian KAUDER - Julie LOPEZ - Nathan TUTARD - Lilian CHAMPROY

6 Mai 2015

Table des matières

Chapitre 1

Organisation

1.1 Outil de versionnage

Afin d'avoir des sauvegardes régulières de nos travaux et de pouvoir partager notre projet, nous utilisons un répertoire Git disponible sur GitHub à l'adresse suivante : https://github.com/AamuLumi/mustached-wookie-parser

1.2 Développement modulaire

Le projet est le développement de trois modules :
— le module de traitement syntaxique du code C
— le module de création de la documentation
— le module d'affichage de la documentation

L'avantage d'un développement modulaire est qu'il est possible d'avancer sur plusieurs tâches à la fois : un module n'a pas besoin d'attendre l'autre pour pouvoir fonctionner.
Cependant, il faut veiller à avoir une bonne communication (sur le formatage des fichiers, la syntaxe à utiliser, etc.) afin d'avoir une intégration finale quasi-instantanée.

1.3 Nomenclature des fichiers générés

Les fichiers générés sont de la forme suivante :

monFichierCode.c.code.html Ce fichier contient le code mis en page du fichier monFichierCode.c

monFichierCode.c.doc.html Ce fichier contient la documentation extraite du fichier monFichierCode.c

1.4 Répartition des tâches

Générateur de page de code Julie LOPEZ - Lilian CHAMPROY

Générateur de documentation Nathan TUTARD

Documentation et Web Florian KAUDER

Première partie

Parser le code C

Chapitre 2

Objectif Consécutif au parcours du code

Cette partie a pour but de parser le code. C'est à dire le parcourir afin de récupérer les informations utiles. Le résultat de ce parseur sera une page html qui sera modifié ensuite par le code web de la dernière partie. Cette phase est très importante puisque c'est a elle d'extraire toutes les informations utiles afin que la documentation générée soit complète.

Parser le code C et avoir des résultats utilisable peut se résumer en 2 parties
— Le parsage des données pour récupérer le code brut
— Le parsage des données pour récupérer la partie documentation

2.1 Objectifs

Le principe comme énnoncé dans l'introduction est de parser du code C afin de créer le fichier HTML de base qui sera modifié par le javascript de la seconde partie.

2.2 Technologies utilisées

Afin d'obtenir un parsage complet et efficace 2 technologies ont été utilisées :

Lex Un analysateur syntaxique permettant de parser le code

Bison Un analysateur grammatical.

Chapitre 3

Parsage principal

Nous utilisons le fichier**c-grammar.y** comme "parseur principal" (fichier d'analyse grammaticale) qui va appeler **c-grammar.l** (fichier d'analyse lexicale).

3.1 c-grammar.l

Le fichier **c-grammar.l** analyse chaque mot d'un fichier .c et imprime toutes les balises de base pour définir ce qu'on lit comme mot.

Par exemple, si on lit le mot **if**, on écrira dans le fichier html : <keyword>if<\keyword>. Si on lit un type comme **int**, on écrira : <type>int<\type>. Si on lit un caractère, on écrira ce caractère dans le html. Sauf, si c'est **<**, on écrira **&gt**, pour **>** on écria **&lt**. Pour **}**, on fermera la balise avec <\div>. Pour le caractère **\n**, on écrira
\n car ce caractère s'écrit
 en html.

Pour les commentaires sur une ligne, on écrira <comment>le commentaire trouvé<\comment>. Ceux qui sont sur plusieurs lignes, on utilise une fonction qui tant qu'on n'a pas trouvé la fin du commentaire (le caractère */) on écrit ce que l'on lit.

Pour les # include en début de fichier, on utilise aussi une fonction qui écrit <directive># puis tout ce que l'on lit, et qui s'arrête une fois que l'on tombe sur un saut de ligne.

3.2 c-grammar.y

La fonction **main()** est placée dans ce fichier. Elle prend en paramètre un fichier .c qui sera lu en entrée et le fichier .code.html associé à ce fichier qui sera en sortie. Nous utilisons plusieurs parseurs. Nous les expliquerons plus en détail dans le chapitre suivant.

Dans ce fichier, deux fonctions ont été ajoutées pour écrire l'en-tête HTML et le fin du fichier HTML.

Lors de l'analyse grammaticale, on rajoute au fichier de sortie des mot-clés de la forme **=<=mon-mot-cle=<=**. Il sont utiles pour les deux parsages qui vont suivre.

3.2.1 Mots-clés pour définir les blocs de codes

A partir de ce moment là, on a donc notre premier fichier, disponible sous le nom de **tmp.txt**. On peut passer au second parseur.

Chapitre 4

Parseurs multiple

4.1 Création du HTML

Dans cette partie nous allons nous baser sur la création

4.1.1 Déclarations des fonctions - 1ère partie

Si on trouve que la ligne contient une déclaration de fonction (la ligne contient **=<=fun-declaration=<=**), il suffit juste d'ajouter en début de ligne une balise **<declaration>**. On passe aussi le booléen **funDeclarationAdded** pour signaler qu'une déclaration de fonction a déjà été ajoutée sur cette ligne, et qu' on ne rajoute pas d'autres déclarations tant que nous ne sommes pas passés à une nouvelle ligne.

Ca permet d'éviter le problème suivant :

```
int main(int a){=<=fun-declaration=<=
```

passe dans la règle, mais aussi

```
nt main(int a){=<=fun-declaration=<=
```

et

```
t main(int a){=<=fun-declaration=<=
```

et

```
 main(int a){=<=fun-declaration=<=
```

et ainsi de suite. Grâce au booléen, on ne passe qu'une fois dans la règle par ligne.

4.2 3e parseur - Recherche des autres mots-clés

4.2.1 Déclaration de variables

Le but est de faire seulement les déclarations de variables **dans les déclarations de fonctions**. On regarde juste ce cas :

```
int main(int a,=<=var-declaration=<= int b,=<=var-declaration=<= int c)=<=var-declaration{
```

On commence par la 3e règle (celle avec "(" au début). Elle permet de savoir que nous sommes dans le cas **(int a ... =<=var-declaration=<=**. On va donc ajouter la balise **<declaration>**, puis faire un REJECT ; pour continuer d'analyser le contenu.

Pour les deux premières règles :

- Si on trouve **,=<=var-declaration=<=**, on remplace par **</declaration>,<declaration>**. Ca veut dire qu'on a encore des déclarations de variables qui vont venir.
- Si on trouve **)=<=var-declaration=<=**, on remplace par **</declaration>)**. Ca veut dire qu'on a atteint la fin des déclarations de variables pour cette fonction. On peut donc réinitialiser le booléen.

4.2.2 Identifiants et déclarations

```
int <identifier>main</identifier>=<=identifier-declaration=<=(int a){
```

De la même manière : on récupère un identifiant qui est signalé comme identifiant de déclaration, et on le remplace avec les balises correspondantes puis on fait disparaître le mot-clé. On obtient donc :

```
int <identifier class="declaration">main</identifier>(int a){
```

4.2.3 Déclaration de fonctions - 2e partie

Les balises signalant le début d'une déclaration de fonction ont été placées par le 2e parseur, il n'y a plus qu'à signaler la fin de ces déclarations. Pour cela, on remplace **=<=fun-declaration=<=** par **</declaration>**.

4.2.4 La règle 5 (avec <declaration><img ...>)

On utilise cette règle pour corriger un bug qui ne mettait pas le code-expander en tout début de ligne, mais qui mettait la balise de déclaration de fonction. On va donc recopier en inversant les balises.

Dans le chapitre suivant, nous allons expliquer ce qui ne marche pas ou qui nous manque et expliquer ce que l'on pourrait améliorer pour le parasage du code.

Chapitre 5

Bugs et évolutions

5.1 Déclaration de variables

Nos parseurs ne prennent pas en compte toutes les déclarations de variables, il faudrait donc considérer toutes celles qui manquent.

Avec une analyse grammaticale plus apporfondie, on pourrait arriver à les gérer aussi.

5.2 Commentaires dédiés

On pourrait utiliser les champs **brief** et **return** pour ajouter des informations aux fonctions.

Ces informations seraient utilisées par l'affichage Web via les info-bulles sur les fonctions.

5.3 Meilleure précision sur les déclarations de variables

Pour le moment, les info-bulles des déclarations de variables possèdent des artefacts (des '(' pour certains paramètres). Le problème pourrait être corriger avec une meilleur analyse lexicale ou grammaticale.

5.4 Types non primitifs

Le programme ne gère actuellement que des types primitifs et des types dont la définition est faite dans le fichier analysé.

Ce problème est dû au fait que ces parseurs utilisent la grammaire de base du C. Cette grammaire, utilisée notamment par **gcc**, est utilisé après la phase d'édition de liens, c'est-à-dire que toutes les déclarations sont présentes dans notre fichier. Etant donné que nous faisons de l'analyse de fichier unique, les déclarations externes ne sont pas présentes. L'analyse grammaticale/lexicale ne comprend donc pas certains types, et provoque une erreur de syntaxe.

5.5 Optimisation des parseurs

On utilise actuellement 3 parseurs, c'est-à-dire 3 passages sur l'intégralité d'un fichier. Dans le cas de fichiers denses, l'analyse serait très lente. Une optimisation serait donc de rassembler le 2e et le 3e parseur.

Chapitre 6

Parsage De la Documentation

On utilise le fichier**c-grammar.y** comme "parseur principal" (fichier d'analyse grammaticale) qui va appeler **c-grammar.l** (fichier d'analyse lexicale).

6.1 c-grammar.l

Le fichier **c-grammar.l** analyse chaque mot d'un fichier .c et imprime tout les commentaires utiles, c'est à dire ceux contenus entre les balises /* */.

6.1.1 Mots-clés pour définir les blocs de commentaires

fn Correspond aux bloc de definition d'une nouvelle fonction

brief Correspond au definition en texte contenue dans les sources .c

Avec tout ça, on a donc notre premier fichier, disponible sous le nom de **tmpdoc.txt**. On peut passer au chapitre suivant.

Chapitre 7

Parseurs secondaire

7.1 Mise en place de deux parseurs

On a deux parseurs lexicaux (c-grammar.l second.l). Ce sont tous les deux des parseurs classiques.

Pour les faire fonctionner ensemble, il faut d'abord que les fonctions yyparse() et yylex() et yytext et ... aient des noms différents. C'est pour cela que, quand on compile les parseurs (dans le fichier **compile.sh**, on compile ces parseurs de cette façon là :

```
bison -d ./c-grammar.y
flex -p yy ./c-grammar.l
gcc ./c-grammar.tab.c ./lex.yy.c -o ./test-c
rm lex.yy.c
flex -p zz ./second.l
gcc ./c-grammar.tab.c ./lex.zz.c -o ./parser
rm lex.zz.c
```

Les parseurs sont définies de la façon suivante :

yy 1er parseur

zz 2e parseur

7.2 Second parseur

Ce parseur va parser notre fichier ligne par ligne.Ce dernier n'étant pas finis, il est donc impossible de le compiler.

Cependant un algorithme a été écrit. Ce dernier est décrit dans la première partie des Bugs et Evolutions.

Chapitre 8

Bugs et évolutions

Le problème qui s'est posé à nous c'est le second parsage, par manque de temps et d'expèrience en FLEX nous n'avons pas eu le temps de la finir. Cette partie était pourtant à notre portée. L'algorithme proposé était de prendre chaque ligne du fichier du premier pars récupérer les informations, les stockés dans une variable et une fois toute la documentation de la fonction faite faire un bloc html correspondant aux informations dans les commentaires.

8.0.1 Second parsage qui ne marche pas

Le second parsage n'est pas fini. Ceci implique qu'il n'est pas possible de générer le fichier html correspondant. Ceci c'est produit par manque de temps et par manque d'efficacité sur ce dernier parsage.

Deuxième partie

Visualiser et interargir avec la documentation

Chapitre 9

Etude des objectifs

Cette partie du projet consiste à mettre en forme les fichiers obtenus via le parseur et à développer des fonctionnalités permettant à l'utilisateur de facilement utiliser la documentation.

On peut séparer l'ensemble des tâches de cette section en 2 parties :
— Affichage d'une documentation épurée
— Utilisation du code source

9.1 Objectifs

Une majeure partie du travail demandé consiste à "augmenter" du code HTML avec des fonctionnalités JavaScript. L'essentiel du travail de cette partie est donc réalisée avec ce langage.
Cependant, une autre partie (décrite précédemment) consiste à mettre en place une syntaxe des fichiers HTML afin de faciliter l'intégration des deux modules Parseur et Web.

En plus de ces tâches, il est aussi nécessaire de faire un minimum de travail de design, dans le but d'obtenir une documentation épuré et organisé. Un développeur doit pouvoir trouvé rapidement ce qu'il veut, et ne doit pas se fatiguer en lisant une documentation.

9.2 Technologies utilisées

Afin d'obtenir un développement efficace, plusieurs librairies sont utilisées :

Less Compilateur permettant de créer des fichiers CSS avec des fonctionnalités supplémentaires (encapsulation de balises, variables, ...)

JQuery Lib JS - Permet de manipuler des documents HTML plus simplement en JavaScript

Bootstrap Framework CSS - Permet d'utiliser une grille pour placer des élements HTML

MathJax Lib JS - Permet d'afficher des formules mathématiques à partir de formules LaTeX, MathML, etc.

Chapitre 10

Affichage du code source

10.1 Définition des balises HTML

Pour utiliser la mise en page du code C, il est nécessaire d'encapsuler tout le code entre des balises ** **.

Il suffit alors de mettre le code C avec des balises permettant de connaître la syntaxe du code. Par exemple :

```
1 int myFunction(char* a)
```

s'écrirait avec le bout de code suivant :

```
1 <declaration><type>int</type>
2         <identifier class="declaration"> myFunction</identifier>(
3         <declaration>
4                 <type>char*</type>
5                 <identifier class="declaration">a</identifier>
6         </declaration>)
7 </declaration>
```

Voici un index des balises reconnues pour la mise en page du code :

<type> Est un type

<directive> Est une directive de précompilation

<value> Est une valeur simple

<identifier> Est un nom de fonction/variable (si **class="declaration"**, cette identifiant est dans une déclaration d'une fonction/variable)

<keyword> Est un mot-clé

<comment> Est un commentaire

<string> Est une chaîne de caractères

10.2 Coloration syntaxique

A partir de ces balises, le travail de coloration syntaxique est extrêmement simple : il suffit d'appliquer une mise en forme à chaque balise via l'utilisation des feuilles de styles (CSS).

Par exemple, on souhaite obtenir un affichage des types avec une couleur et en gras, ce qui est fait simplement par :

```
1 type {
2     color: #5999B7;
3     font-weight: bold;
4 }
```

10.3 Blocs : Indentation et masquage

Pour mettre en place l'indentation, des balises ** ** sont ajoutées avant chaque début d'un bloc de code qui doit être indenté.

Ces blocs permettent aussi de mettre en place une seconde fonctionnalité : le repliement et le dépliement des blocs de code. Une flèche est ajoutée au début de la ligne contenant un bloc de code (via **<img class="code-expander" src="img/collapse.png">**, et un clic sur une de ces flèches va faire déplier/replier le premier bloc de code qui suit cette flèche.

10.4 Surbrillance et info-bulle

10.4.1 Surbrillance - Recherche des identifiants

La mise en place de la surbrillance est réalisée via le script JavaScript. Tous les identifiants possèdent un appel à la fonction **highlightIdentifiers(id)** lorsque la souris passe au-dessus d'un identifiant.

Cette fonction ne fait que parcourir tous les identifiants de la page et une couleur de fond est ajoutée à un identifiant si il correspond à l'identifiant survolé.

10.4.2 Info-bulle - Recherche de la déclaration

Afin d'ajouter une info-bulle avec la déclaration de l'identifiant, il est d'abord nécessaire de retrouver cette déclaration.

La fonction **getDeclaration(id)** a été développée pour cette usage : cette fonction va faire une recherche bloc par bloc afin de retrouver la première déclaration de l'identifiant. On commence par le bloc dans lequel est notre identifiant, puis, si aucune bonne déclaration n'est trouvé, on remonte d'un bloc, et on réitère le processus.

Il ne reste plus qu'à ajouter une petite info-bulle avec cette déclaration (et les informations contenus dans les champs **description** et **return** de notre **<identifier>**) via l'ajout d'un élément de classe **identifierTooltip** à notre page.

10.5 Etiquette

Pour accéder directement à une déclaration depuis l'URL de la page, des ancres ont été disposées dans la page pour chaque identifiant via le code JavaScript. Très simplement, il suffit d'ajouter le nom de l'identifiant dans le champ **id** de celui-ci afin de pouvoir accéder directement à sa localisation dans la page.

Par exemple :

monCode.c.code.html#main

permet d'accéder directement à la fonction **main()** du fichier **monCode.c**.

10.6 Numérotation des lignes

Une remarque simple à faire avec la mise en page du code actuel est qu'il est difficile de savoir où nous sommes dans notre fichier.

Pour corriger ce souci, les numéros de lignes ont été ajoutées sur le côté gauche du code, via un élément ** **. De base, cet élément est vide, et va être compléter de la façon suivante :

- On parcourt toutes les lignes du fichiers. Pour chaque ligne :
  - On ajoute le numéro dans notre élément **lineNumber**
  - Si la ligne est un début de bloc, on ajoute un début de bloc dans notre **lineNumber** via la syntaxe **<block></block>**. Chaque **<block>** possède deux champs de données : **beginning** (pour la ligne de début du bloc) et **end** (pour la ligne de fin du bloc). On continue ensuite dans ce bloc, jusqu'à qu'il soit terminé.

Avec les deux champs de données **beginning** et **end**, il est simple d'ajouter le dépliement et le repliement des blocs de lignes à tous les éléments de classe **code-expander**.

Chapitre 11

Affichage de la documentation

11.1 Design

Le but d'une documentation est de fournir un visuel simple et organisé à partir de commentaires placés dans les fichiers de code.

La documentation est donc basé sur un flat design avec des couleurs pastels, permettant de mettre en valeur l'organisation de la documentation tout en évitant de fatiguer la lecture de celle-ci avec des couleurs trop vives.

11.2 Syntaxe générale des éléments

Pour présenter les éléments, il est nécessaire de les organiser.

Par exemple, pour présenter une fonction, on définit d'abord le bloc de cette fonction, et on ajoute le prototype, la description, les paramètres, etc. dans ce bloc.

On obtient la syntaxe suivante :

```
<div class="doc-function">
        <div class="prototype" id="main">
                Prototype here
        </div>
        <div class="doc-text">
                Description here
        </div>
        <div class="category">Parameters</div>
        <div class="doc-text">
                Parameters here
        </div>
        <div class="category">Return</div>
        <div class="doc-text">
                Return value and description here
        </div>
</div>
```

Pour la documentation de variables ou le sommaire de la page, la même logique est utilisée. Seules les classes des balises changent.

11.3 Référence vers le code

A toute expression contenue entre des balises **<identifier class="declaration">** et étant dans des balises de classe **prototype** va être assignée une fonction permettant, lors d'un clic, de consulter le code de la fonction.

Cette fonction correspond à une recherche de l'ancre portant le nom de la fonction dans le fichier **.code.html**.

Un clic similaire sur une fonction dans le sommaire va déplacer la page vers la documentation de cette fonction.

11.4 Tableaux et listes

Les tableaux sont disponibles via les balises standards HTML **<table>**, **<td>** et **<tr>**.
Ces mêmes balises sont utilisés pour simuler des listes.

— Pour les listes standards, la première colonne contient le caractère **-** et la deuxième colonne les objets de la liste.
— Pour les listes numérotées, la première colonne contient le numéro de l'élément et la deuxième colonne les objets de la liste.

11.5 Formules mathématiques

Les formules mathématiques sont gérées via la librairie MathJax qui va créer les comportements demandés via l'inclusion de la librairie JavaScript.

Chapitre 12

Problèmes et bugs

12.1 Navigateur utilisé

D'après les derniers tests, Mozilla Firefox ne produit pas les mêmes résultats que Google Chrome. Par exemple, la numérotation des lignes marche correctement sous Chrome tandis qu'elle possède des bugs sous Firefox.

Ce problème vient sûrement d'une interprétation Javascript différente entre les deux navigateurs. Cependant, par manque de temps, ce problème n'a pas pu être corrigé.

12.2 Animations de la numérotation des lignes

Lorsque l'utilisateur clique sur un étendeur de code pour "refermer" des balises de code, les numéros de lignes vont se déplacer sur le côté pour effectuer l'animation. Il en est de même pour l'animation inverse.

Après quelques recherches, la solution serait de coder une animation plus complexe et plus spécifique à notre documentation.

12.3 Gestion du contexte

Le surlignement des identifiants provient d'une recherche de l'identifiant sur toute la page. Il en est de même pour visualiser la déclaration de cette identifiant.

Si une variable est locale à deux fonctions, la variable qui sera utilisé pour les recherches sera la première variable de la page. Pour obtenir une meilleure documentation, il serait nécessaire de faire ces recherches en prenant en compte le contexte de la variable.