



Rapport de Projet Linter en C

Tavernier ANTOINE
HUCHARD Théo
ESSAMAMI Hamza

10/12/2018

Table des matières

Introduction	3
Sujet	3
Membres de groupe	3
Analyse d'application	3
Structures principales	3
Structure Config_t	3
Structure Token_t	4
Structure Config_t	5
Fonctions principales	6
Fonction main	6
Fonction config	6
Fonction parse	7
Fonctions check	7
Choix de l'implémentation	8
Choix de l'environnement	8
Choix algorithmique	9
Installation de l'application	10
Utilisation de l'application	11
Bilan	12

Introduction

Sujet

Ce projet en C à pour but de fabriquer un Linter. Un Linter est un outil d'aide pour le développeur, dans notre cas il analyse le code source. Les utilisations primaires de ce Linter sont de détecter le non respects de conventions de codage mais aussi de souligner les possibles erreurs de syntaxe. Ce Linter à été codé en C pour analyser du codes en C.

Membres de groupe

Ce groupe est composé de trois personnes.

- Un ancien membre de l'ESGI, Théo HUCHARD
- Deux nouveaux arrivants, Antoine TAVERNIER et Hamza ESSAMAMI.

Analyse d'application

Cette section à pour but de présenter les structures et ainsi les fonctions principales de cette application. Vous y trouverez aussi le model et les choix d'implémentation.

Structures principales

Structure Config_t

Ci-dessous la structure qui va permettre de stocker en mémoire le fichier de configuration de l'utilisation. On y retrouve des options de coding stype tel que maxLineNumbers qui défini le nombre maximum de caractères sur une seule ligne. Nous retrouvons aussi des options de syntaxe comme noPrototype qui permet de détecter les fonctions qui n'ont pas de prototype.

```
1 typedef struct Config_t {  
2     char *extends;  
3     short arrayBracketEol;  
4     short operatorsSpacing;  
5     short commaSpacing;  
6     short indent;  
7     short commentsHeader;  
8     short maxLineNumbers;  
9     short maxFileLineNumbers;  
10    short noTrallingSpaces;
```

```

11     short NoMultiDeclaration;
12     short unusedVariable;
13     short undeclaredVariable;
14     short noPrototype;
15     short unusedFunction;
16     short undeclaredFunction;
17     short variableAssignmentType;
18     short functionParametersType;
19     char **excludedFiles;
20     int nbExcludedFiles;
21     short recursive;
22     char **configFileName;
23     int nbconfigFileName;
24 } Config;

```

Structure Token_t

Ci-dessous la structure qui va permettre de stocker en mémoire un Token. Un token est un élément du code source à qui ont à mis un type. Une ligne dans un fichier est donc une liste de Token. Un fichier source est donc une matrice de Token.

```

1 typedef struct Token_t {
2     Type type; /**< The type of the Token. */
3     char *value; /**< The string representation of the Token. */
4     int pos; /**< The position of the Token in the list of
5     characters of a line. */
6 } Token;

```

Le Type est une énumération qui donne un nom spécifique à chaque troçon de code. Voici la list, non exhaustive, de Type. En réalité elle comporte plus de 70 types.

```

1 typedef enum Type_e {
2     ...
3     Delimiter,
4     Numerical,
5     VOID,
6     CHAR,
7     SHORT,
8     INT,
9     LONG,
10    FLOAT,
11    DOUBLE,
12    SIGNED,
13    UNSIGNED,
14    ...

```

```
15 } Type;
```

Structure Stack_t

La structure Stack_t est une structure customisées, elle reprend le principe de la stack mais couplé avec une simple liste d'indices de position.

```
1 typedef struct Stack_t
2 {
3     int capacity;    /**< The capacity in Token Array */
4     int top;         /**< The position of our top pointer */
5     Token *stack;    /**< The stack of Tokens */
6 } Stack;
```

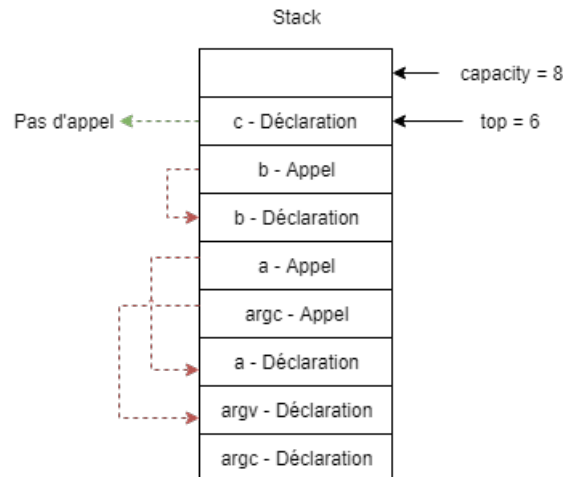
On a donc le principe d'une stack classique où les deux actions principales sont le Push et Pop. Dans cette stack on va Push des noms de variables. La liste d'indices sert à délimiter la portée (scope) d'une fonction.

Prenons cette exemple :

```
1 int main(int argc, int **argv)
2 {
3     int a = argc;
4     int b = a++;
5     b++;
6     int c;
7 }
```

Quand on arrive sur une déclaration de fonction, on y push tous ses arguments dans la stack. On parcourt le corps de la fonctions et on y push toutes les variables déclarées et appelées. Une fois que l'on sort de la fonction on regarde s'il existe une déclaration pour un appel de variable. De même on regarde si une variable déclarée a au moins un appel, sinon nous avons une variables inutilisées

Voici la représentation de la stack une fois toutes les lignes parcourus.



Fonctions principales

Décrire les Fonctions ICI.

Fonction main

La fonction main est le point d'entré du programme. C'est elle qui possède toute la logique du programme.

Tout d'abord elle va essayer de charger le ou les fichiers de configurations.

Puis elle va lister tous les fichiers dont on peu appliquer l'analyse syxtaxique.

On y retrouve uniquement les fichiers avec l'extention .c qui ne sont pas exclus selon le fichier de configuration. On peu aussi aller chercher les fichiers dans les dossiers selon l'option recursive dans la configuration.

On va lire chacun des fichiers et les chargers dans la mémoires grâce au parser.

Le parser va découper les mots clés et leur assigner un type.

On se retrouve avec des listes de Token représentant les lignes d'un fichier.

On va faire des traitements sur ses listes de Token respectivement des options de la configuration. Les traitements vont avertir l'utilisateur des erreurs qu'il peut commettre.

Fonction config

La fonction config, comme dit précédemment, va ouvrir un fichier .lconf et assigner chaque règle à un champ de la structure Config_t.

La options extends et recursive va prendre les valeurs dans la ligne suivante, alors que les règles sont sur la même ligne.

Si on se rend compte que la configuration hérite d'un autre fichier, nous allons l'ouvrir, le charger dans la structure Config_t et le fusionner avec la configura-

tion initiale. On stock un historique des fichiers chargés pour éviter de charger les mêmes indéfiniment.

Fonction parse

Le parsing a pour but de découper les mots lu dans un fichier et de leur affecter à un type. On va donc pouvoir résonner sur un type plutôt qu'un mot. On gagne en abstraction. Voici un exemple :

```
1 if (!a) // Comment
2 {
3     printf("Hello World\n");
4 }
```

```
1 IF DELIMITER OpenPar Operator IDENTIFIER ClosedPar Nothing
2 OpenBra
3 DELIMITER IDENTIFIER OpenPar STRING ClosedPar SEMICOLON
4 ClosedBra
```

Ici le type Nothing est assigné à bout de code inutile à la compilation, tel qu'un commentaire.

Fonctions check

Nous avons réalisé une fonction par règles. Ce qui rend chaque règle indépendante l'une de l'autre. Les règles sont déclenchées grâce aux membres de la structure Config_t. Les opérations sur la coding style se font via les Tokens. On va créer des conditions sur des cas précis.

Pour les règles sur la syntaxe, le système de Token est très efficace. En effet si l'on voit un Token d'un certain type, alors on s'attend à retrouver un autre type après. La logique de cette fonction chercher à manger un Token d'un certain Type, s'il y arrive, alors il avance dans la liste de Token et cherche un autre Type. C'est une recherche dite paresseuse. Il faut donc bien faire attention à nos appels de fonction pour éviter les conflits.

```
1 int getType(Token **tokens, int *pos, int nbNode)
2 {
3     if (eatToken(tokens, LONG, pos, nbNode))
4     {
5         if (eatToken(tokens, LONG, pos, nbNode))
6         {
7             if (eatToken(tokens, INT, pos, nbNode))
8                 return funcOrVar(tokens, pos, nbNode, "LONG LONG
INT");
```

```

9         return funcOrVar(tokens, pos, nbNode, "LONG LONG");
10    }
11    if (eatToken(tokens, INT, pos, nbNode))
12        return funcOrVar(tokens, pos, nbNode, "LONG INT");
13    return funcOrVar(tokens, pos, nbNode, "LONG");
14 }
15 if (eatToken(tokens, INT, pos, nbNode))
16     return funcOrVar(tokens, pos, nbNode, "INT");
17 return 0;
18 }

```

Cet exemple de code permet de déterminer 5 combinaisons de type :

- LONG LONG INT
- LONG LONG
- LONG INT
- LONG
- INT

Ce projet permet de déterminer 29 combinaisons de type, sans le type CONST.

Nous avons donc ici, un semblage de grammaire Gauche à Droite (LR(0)). 0 correspond au nombre de token qu'on regarde en avance pour éviter les conflits. Cette méthode a été trouvée dans un livre parlant de conception des compilateurs¹

Choix de l'implémentation

Ayant déjà une connaissance sur la théorie des langages et des compilateurs dans le groupe, le choix de l'implémentation a été de se rapprocher le plus possible du mécanisme de Flex et Bison. Un système de parsing et de traitement d'une grammaire simple.

Choix de l'environnement

Le choix de l'environnement n'a pas été vraiment discuté au sein du groupe, cependant la majorité des programmes C sont développés sous Linux. C'est une plateforme gratuite et simple d'installation. De plus c'est le choix le plus pratique des autres plateformes. Il a suffi de créer une VM pour chaque membre du groupe afin d'avoir le même environnement. Avoir un même environnement permet aussi la facilité d'aider nos collègues. Nous avons tous le même compilateur ; GCC sous linux.

1. Compiler Design in C by Allen Holub

Choix algorithmique

Il a été dit que nous avions repris le choix d'implémentation par Allen Holub².

Pour le parser nous avons une liste de caractères à découper, Nous avons donc créé deux pointeurs sur cette liste, le pointeur Gauche et le pointeur Droit. Nous avançons le pointeur Droit tant que nous ne trouvons pas un espace, une parenthèse, un opérateur ou autre délimiteur. Une fois le mot encadré par nos deux pointeurs nous pouvons découper le mot et lui assigner à un type. Le pointeur Gauche se met au niveau du Droit pour traiter le mot suivant, jusqu'à arriver à la fin de la chaîne de caractères.

L'assignation de type est un simple switch ou une cascade de if pour trouver les mots clés comme les types de variables.

```
1 for (int i = 0; i < nbNodes; i++) {
2     switch (listToken[i]->value[0]) {
3         case '=':
4             listToken[i]->type = EQ_OP;
5             break;
6         case '+':
7             /* + - / * % ^ ! */
8         case '*':
9             listToken[i]->type = Operator;
10            break;
11        case ' ':
12        case '\t':
13        case '\n':
14            listToken[i]->type = Delimiter;
15            break;
16        case ')':
17            listToken[i]->type = ClosedPar;
18            break;
19        case '(':
20            listToken[i]->type = OpenPar;
21            break;
22        default:
23            listToken[i]->type = IDENTIFIER;
24            break;
25    }
26    if (strcmp("int", listToken[i]->value) == 0)
27        listToken[i]->type = INT;
28    /* Cascade de if pour chaque mot cle: int, long, void,
29    short, return ... */
30    if (isNum(listToken[i]->value))
```

2. Compiler Design in C by Allen Holub

```

30     listToken[i]->type = Numerical;
31 }

```

Une fois les Types assignés aux Token trouvés, nous pouvions mettre en place des règles. Des règles soit basées sur l'attente.

Exemple : Je ne m'attends pas retrouver une espace avant chaque fin de ligne.

```

1 void checkSpace(Token **listToken, int nbToken, int line, char *
  fileName) {
2     if (nbToken == 1)
3         return;
4     for (int i = 0; i < nbToken; i++) {
5         if (strcmp(listToken[i]->value, "\n") == 0 && strcmp(
listToken[i - 1]->value, " ") == 0)
6             print_warning("Extra space", line, listToken[i - 1]->
pos, fileName);
7     }
8 }

```

Soit par une pseudo grammaire vu dans la partie "Functions Check". Cette grammaire va permettre de trouver l'emplacement de déclaration et l'appel de fonctions/variables. Nous stockons les informations dans une Stack puis l'analysons pour vérifier les règles.

Installation de l'application

Pour installer l'application il vous faut télécharger le binaire ou les fichiers sources pour créer le binaire.

Si vous téléchargez le ZIP sur la plateforme MyGES, vous aller devoir le Dézipper sous dans un Linux.

Sinon vous pouvez optenir les fichiers sources via le repo Git. Il vous faut simplement un environnement Linux avec l'application "git" installé.

```

$ git clone https://github.com/Uranium2/CLinter.git
$ cd CLinter

```

Pour pouvoir compiler le projet et la documentation il vous faut au total 3 applications. GCC, make et Doxygen. Voici comment les installer via le gestionnaire de packet APT :

```

$ sudo apt get install gcc
$ sudo apt get install make
$ sudo apt get install doxygen

```

Vous avez maintenant tous les outils nécessaires pour générer le binaire de l'application. Placez vous dans le dossier "CLinter" contenant le projet.

```
$ cd CLinter
$ make all
```

Votre application se trouve dans le dossier "test/", où vous pourrez déposer vos propres fichiers à analyser par le Linter.

Utilisation de l'application

Maintenant que vous savez où se trouve le binaire de l'application (CLinter/test/linter), il faut l'exécuter. Pour cela il vous faut au préalable fournir un fichier de configuration dans ce même dossier. Un fichier de configuration par défaut est déjà présent dans le dossier, nommé "default.lconf".

Pour lancer l'application avec la configuration par défaut il vous suffit de déposer vos fichiers C dans le même dossier que le binaire "linter" et que son fichier de configuration.

```
$ cd test/
$ ./linter
```

Dans cet exemple, vous lancerez l'application avec la configuration par défaut. Vous pouvez éditer les règles dans le fichier "default.lconf". Ou alors en créer un vous même avec vos propres règles. Pour lancer l'application avec votre propre fichier de configuration :

```
$ ./linter MaConfiguration.lconf
```

Voici le contenu et les règles d'un fichier de configuration qui peu vous servir à créer le votre.

```
=extends
default.lconf

=rules
- array-bracket-eol = on
- operators-spacing = on
- indent = 4
- comments-header = on
- max-line-numbers = 80
```

```
- max-file-line-numbers = 80
- no-trailing-spaces = on
- no-multi-declaration = on
- unused-variable = on
- undeclared-variable = on
- no-prototype = on
- unused-function = on
- undeclared-function = on
- variable-assignment-type = on
- function-parameters-type = on

=excludedFiles
- file.c

=recursive
false
```

Pour lire la documentation technique, allez dans le dossier "doc/".

```
$ cd CLinter/doc/
$ doxygen Doxyfile
$ firefox html/index.html
```

Bilan

Décrire Ule Bilan ICI.