Rapport de Projet

`Linter en C`

Tavernier ANTOINE
HUCHARD Théo
ESSAMAMI Hamza

11/12/2018

Table des matières

Introduction

Sujet

Ce projet en C a 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 les non-respects de conventions de codage mais aussi de souligner les possibles erreurs de syntaxe. Ce Linter a 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 a pour but de présenter les structures et ainsi les fonctions principales de cette application. Vous y trouverez aussi le modèle 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 style tel que maxLineNumbers qui définit 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.

```
typedef struct Config_t {
    char *extends;
    short arrayBracketEol;
    short operatorsSpacing;
    short commaSpacing;
    short indent;
    short commentsHeader;
    short maxLineNumbers;
    short maxFileLineNumbers;
    short noTrallingSpaces;
```

```
    short NoMultiDeclaration;
    short unusedVariable;
    short undeclaredVariable;
    short noPrototype;
    short unusedFunction;
    short undeclaredFunction;
    short variableAssignmentType;
    short functionParametersType;
    char **excludedFiles;
    int nbExcludedFiles;
    short recursive;
    char **configFileName;
    int nbconfigFileName;
} 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 on a mis un type. Une ligne dans un fichier est donc une liste de Token. Un fichier source est donc une matrice de Token.

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

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

```
typedef enum Type_e {
    ...
    Delimiter,
    Numerical,
    VOID,
    CHAR,
    SHORT,
    INT,
    LONG,
    FLOAT,
    DOUBLE,
    SIGNED,
    UNSIGNED,
    ...
```

```
15 } Type;
```

Structure Stack_t

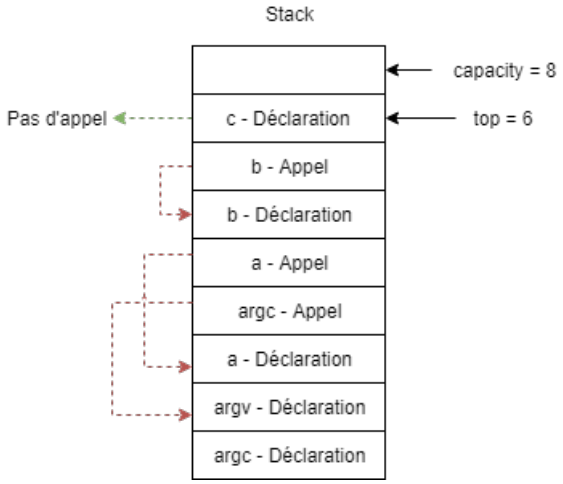
La structure Stack_t est une structure customisées, elle reprend le principe de la stack mais couplé avec deux simple listes d'indices de position, Top et Base.

```
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 l'action principale est de Push un item. Dans cette stack on va Push des noms d'identifiant et leurs propriétés, déclaration, appel, variable, function. La liste d'indices sert à délimiter le portée (scope) d'une fonction.
Prenons cet 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 fonction 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 variable inutilisée
Voici la représentation de la stack une fois toutes les lignes parcourues.



Fonctions principales

Décrire les Fonctions ICI.

Fonction main

La fonction main est le point d'entrée 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 configuration.
Puis elle va lister tous les fichiers dont on peut appliquer l'analyse syntaxique. On y retrouve uniquement les fichiers avec l'extension .ce qui ne sont pas exclus selon le fichier de configuration. On peut aussi aller chercher les fichiers dans les dossiers selon l'option récursive dans la configuration.
On va lire chacun des fichiers et les chargers dans la mémoire 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.
L'option Extend 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 confi-

guration initiale. On stocke un historique des fichiers chargés pour éviter de charger les mêmes indéfiniment.

Fonction parse

Le parsing a pour but de découper les mots lus 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 :

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

```
IF DELIMITER COMMENT
OpenBra
DELIMITER IDENTIFIER OpenPar STRING ClosedPar SEMICOLON
ClosedBra
```

Ici le type Comment 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 cherchait 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 fonctions pour éviter les conflits.

```
int getType(Token **tokens, int *pos, int nbNode)
{
    if (eatToken(tokens, LONG, pos, nbNode))
    {
        if (eatToken(tokens, LONG, pos, nbNode))
        {
            if (eatToken(tokens, INT, pos, nbNode))
                return funcOrVar(tokens, pos, nbNode, "LONG LONG
  INT");
```

```
            return funcOrVar(tokens, pos, nbNode, "LONG LONG");
        }
        if (eatToken(tokens, INT, pos, nbNode))
            return funcOrVar(tokens, pos, nbNode, "LONG INT");
        return funcOrVar(tokens, pos, nbNode, "LONG");
    }
    if (eatToken(tokens, INT, pos, nbNode))
        return funcOrVar(tokens, pos, nbNode, "INT");
    return 0;
}
```

Cet exemple de code permet de détermiter 5 combinairons 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 semblable de grammaire Gauche à Droite (LR(0)). 0 correspond au nombre du token qu'on regarde en avance pour éviter les conflits. Cette méthode a été trouvée dans un livre parlant de conception des compilateurs [1]

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 avions 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 Holub [2].
Pour le parser nous avions 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 autres délimiteur. Une fois le mot encadré par nos deux pointeurs nous pouvions 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 chaine de caractères.
L'assignation de type est un simple switch ou une cascade d'if pour trouver les mots-clés comme les types de variables.

```
for (int i = 0; i < nbNodes; i++) {
  switch (listToken[i]->value[0]) {
    case '=':
          listToken[i]->type = EQ_OP;
            break;
        case '+':
            /* + - / * % ^ ! */
        case '*':
          listToken[i]->type = Operator;
            break;
        case ' ':
        case '\t':
        case '\n':
            listToken[i]->type = Delimiter;
            break;
        case ')':
            listToken[i]->type = ClosedPar;
            break;
        case '(':
            listToken[i]->type = OpenPar;
            break;
        default:
            listToken[i]->type = IDENTIFIER;
            break;
    }
    if (strcmp("int", listToken[i]->value) == 0)
        listToken[i]->type = INT;
        /* Cascade de if pour chaque mot cle: int, long, void,
   short, return ... */
    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 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 "Fonctions 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 My Ges, vous allez devoir le Dézipper sous dans un Linux.
Sinon vous pouvez obtenir les fichiers sources via le repo Git. Il vous faut simplement un environement 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 paquets 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 peut vous servire à créer le votre.

```
=extends
default.lconf

=rules
- array-bracket-eol = on
- operators-spacing = on
- indent = 4
- comma-spacing = on
- 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 = off
- unused-function = on
- undeclared-function = on
- variable-assignment-type = off
- function-parameters-type = off

=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

Pour conclure, nous pouvons dire que ce projet fut une expérience enrichissante. Nous avons fait face à certaines difficultés d'architecture de projet et de travail d'équipe.
En effet les premières règles du Litter ont été conçues de façon naïve. Suite à la recherche d'informations sur la création d'un compilateur, nous avons mieux compris la méthode pour gérer les tokens de façon efficace et claire.