Rapport de projet

I- Sujet :

L'objectif de ce projet est de construire un Linter. Un Linter est un outil d'analyse de code comme on peut en retrouver dans les langages compilés/pseudo-compilés comme le C ou le Java, en général l'IDE utilisé comprend un Linter qui lui nous indique les erreurs dans notre code.

Dans un IDE se trouve donc un Linter auquel est ajouté un débugueur, un profileur et d'autre outils permettant d'aider le développeur dans la quête du code « parfait ».
Pour résumé, le but d'un Linter est d'améliorer la qualité du code (erreurs de syntaxe, non-respect de convention de codage) et de permettre la reprise et la maintenabilité de celui-ci.
Nous sommes chargés, lors de ce projet, de la création d'un Linter en langage C.

II- Mise en œuvre :

Vous l'aurez compris, un Linter n'est que le reflet d'un certain nombre de conventions de codage ou de syntaxe qui viendront corriger un code.
Pour ce faire, le Linter possède des options que l'on appellera des règles. Ces règles peuvent être activées/désactivées selon la volonté de l'utilisateur via l'utilisation d'un fichier de configuration qu'il nous faudra lire afin de récupérer les règles en question.

Une série de fonctions permettront ensuite de répondre à ces différentes règles et de permettre au Linter d'avertir l'utilisateur des erreurs qu'il a pu commettre dans son code.

La **première partie** du projet consistait à pouvoir utiliser un fichier de configuration qui se présenterait sous la manière suivante :

```
=extends
nop.lconf


=rules
- array-bracket-eol = on
- operators-spacing = on
- comma-spacing = off
- indent = off
- comments-header = off
- max-line-numbers = 25


=excludedFiles
- titi.c

=recursive
false
```

Le fichier de configuration fonctionne avec un format de :
= KEY
VALEUR # Valeur de la KEY (string,int ou bool)
Les valeurs de KEY sont les suivantes :
- extends: Héritage possible d'un autre fichier lconf
(Permet de récupérer des options depuis un autre fichier de configuration)
- rules: Les règles de notre Linter
- excludedFiles: Les fichiers à ne pas inclure lors de la lecture du dossier en cours
- recursive: Le parcours se fera de manière récursive, le Linter devra s'exécuter
aussi dans les sous dossiers.
Petite précision sur l'héritage, toute valeur de règle déjà établie par le premier fichier de conf se verra remplacée par le fichier de conf hérité.
L'ensemble de ces KEYS représentent les options de notre Linter, et le traitement final est entièrement dépendant des différentes valeurs des différentes KEYS.

La récupération des données au sein des fichiers de configuration s'est faite pas à pas. Nous avons d'abord cherché à extraire simplement les données de règles, de fichiers exclus et de récursivité. Cela ne nous a pas pris longtemps avant de pouvoir travailler avec un seul fichier de configuration donné, nous avons ensuite pris en compte la possibilité que le fichier de configuration et les données qui le constituait pouvait être hérité d'un autre fichier (et ainsi de suite). La prise en compte de l'héritage complet des fichiers de configuration s'est faite via des fonctions d'agrégats qui permettent de "fusionner" les données constituant les différents fichiers de configuration, en prenant en compte les priorités d'écrasement en fonction de l'arborescence de l'héritage.

La **deuxième partie** du projet se concentrait sur le respect des règles de convention de codage, suite bien sûr à la lecture de(s) fichier(s) de configuration. Bien que les premières règles, concernant par exemple l'accolade en fin de ligne ou l'espace à droite d'une virgule, paraissaient plutôt simples à vérifier ; d'autre règles quant à elles nécessitent beaucoup de stockages temporaires et un travail assez lourd en mémoire, le principal problème étant la gestion de portée de variables, les soucis de déclarations liés à cette portée etc…

Une analyse plus approfondie du sujet nous a vite amenés à penser qu'il serait très compliqué de mettre en place certaines règles demandées par le sujet (affectation de type, variable non utilisée) en pratiquant des méthodes de développement "standard".

Le fait que les variables dépendent de la portée des niveaux dans lesquels elles se trouvent nous aurait énormément compliqué la tâche si nous n'avions pas décidé de parser le fichier en mettant en place différentes structures.

Les règles qui étaient indépendantes du parsing (pour la plupart appartenant à la partie numéro 2 du projet) ont été traitées à l'aide de fonctions mises en place en début de projet. Dans un premier temps, nous avons récupérés l'intégralité du fichier et nous somme passés par un tableau temporaire ('**line**') qui stocke de façon temporaire le contenu de chaque ligne afin d'effectuer tous les traitements si et seulement si la règle en question est bien active. Ces règles ont été traités ligne par ligne.

Après avoir discuté du problème au sein du groupe, nous avons décidé de partir sur un système de **parsing** de notre code, qui permettrait ensuite un traitement bien plus simple sans avoir à parcourir éternellement notre fichier et à stocker ce que l'on cherche dans plusieurs tableaux temporaires. En effet ce parsing nous permet un premier tri du code afin qu'il soit plus « logiquement » lisible pour nos traitements futurs liés aux règles du Linter. La logique ressemblerait donc à ceci :

- Lecture du/des fichier(s) de configuration afin d'établir les règles activées
- Parsing du code en question afin de faciliter son traitement futur
- Application des règles via le traitement du code «post-parsing »

Bien que compliqué à mettre en place, autant dans la réflexion et l'approche du problème que dans la réalisation concrète de cette tâche, le parsing du fichier a eu l'effet escompté. Le stockage, pour chaque niveau (structure ***lineLevels***), de leurs différents paramètres (notamment ***levelNumber*** et ***identifier***, un couple de valeurs uniques permettant d'identifier avec précision la structure) nous a permis un parcours totalement contrôlable du fichier.

```
1  typedef struct lineLevels lineLevels;
2  struct lineLevels{
3      int identifier;             // Identifiant numérique (dadAddr*10+i)
4      int levelNumber;            // Numero de niveau (globale = 0, int main() = 1)
5
6      int startLevel;             // Indice de debut de niveau
7      int startingLine;           // Ligne de debut de niveau
8      int endLevel;               // Indice de fin de niveau
9      int endingLine;             // Ligne de fin de niveau
10     char*** declaredVars;       // Variables de chaque type
11     int* nbVars;                // Nombre de variables de chaque type
12     lineLevels* sonStructs;     // Tableau des niveaux fils
13     int nbSons;                 // Nombre de fils
14
15     char*** globalVars;         // Tableau des variables globales de chaque type
16     int* nbGlobal;              // Nombre de variables globales de chaque type
17
18     int dadAddr;                // identifier du parent du niveau
19 };
20
```

Le parsing des fichiers sources nous permet de repérer les différents niveaux au sein de chacun des codes. Ce repérage est accompagné d'un stockage systématique de différents paramètres relatifs au niveau que l'on traite. Ces différents paramètres tels que la ligne de début, la ligne de fin, les variables contenues selon leurs types, et j'en passe, nous permettent une lecture du fichier de manière partitionnée. Ce type de lecture nous permet de retrouver, avec plus ou moins de simplicité, les différentes "portées" possibles pour chacunes des variables du fichier en cours de traitement en identifiant les niveaux auxquels elles appartiennent.
Chaque niveau possède également un tableau de ses niveaux fils, et ainsi de suite, de manière à stocker tous les types de portées possibles ainsi que leurs paramètres attribués.

CONCLUSION

Pour conclure ce projet fut un véritable défi. Autant sur la réflexion, l'anticipation et la gestion des très nombreuses possibilités que dans la complexité algorithmique mise en oeuvre. Le parsing plus particulièrement et les règles qui l'accompagnent nous ont véritablement amenés à réaliser des traitements de plus en plus complexes et cela avec de plus en plus d'aisance.

Dossier d'installation

Pour Windows :

Le fichier exécutable issue de la compilation (Debug) doit être lancé à la racine du dossier du projet. Il faut que le .exe soit auprès de son fichier de conf pour une exécution en mode graphique.

Pour le mode console, on peut passer par la commande :
./nomProgramme dossierConfig dossierCibleExec

Pour Linux :

Il faut utiliser la commande **gcc PATHSRC.c -o *EXEDEST*** afin de compiler le main.c, où PATHSRC représente le chemin correspondant à l'emplacement de main.c.

On peut ensuite lancer le programme par la commande :
./nomProgramme dossierConfig dossierCibleExec