Projet : Langage C

3A ESGI

L'objectif de ce projet est de construire un Linter; Un Linter est un outil d'analyse de code source. Il doit permettre de détecter des erreurs de syntaxe mais aussi du non-respect de convention de codage pour les programmes C uniquement.

Partie 1 : Fichier de configuration pour activer des règles

Construire un ensemble de fonctions permettant d'activer des options de votre Linter à partir d'un fichier de configuration.
ATTENTION il ne faut pas coder les règles dans cette partie mais uniquement traiter les options à activer pour les prochaines parties.

Voici le format du fichier: (extension .lconf)
=KEY
VALEUR # Valeur de KEY

=KEY2
- VALEUR # Éléments de KEY2
- VALEUR
- VALEUR

=KEY3
- E1 = VALEUR
- E2 = VALEUR
- E3 = VALEUR

KEY, KEY2 et KEY3 sont les clés correspondantes aux options de votre Linter, c'est une chaine de caractères uniques permettant d'identifier une option.

VALEUR correspond au contenu derrière la première clé, elle peut être soit une chaine de caractères, soit un entier, soit un boolean.

E1, E2, E3 correspondent aux éléments contenus derrière une autre clé.

Ci-dessous l'ensemble des clés principales :
- **extends**: Héritage possible d'un autre fichier lconf (Permet de récupérer des options depuis un autre fichier de configuration)
- **rules**: Les règles de votre 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.

Exemple de fichier lconf:

```
=extends
main.lconf

=rules
- rule1 = on
- rule2 = off
- rule3 = 50

=excludedFiles
- bonjour.c
- test.c

=recursive
true
```

Partie 2 : Les règles de convention de codage

Lorsque vous allez démarrer votre Linter, il va charger ses préférences grâce au fichier .lconf (par défaut: **default.lconf**)
Puis il va exécuter l'ensemble des règles que vous avez activées depuis le fichier de configuration sur l'ensemble des fichiers C du répertoire courant, excepté les fichiers exclus dans la configuration.
Le traitement se fera de manière récursive par défaut, cela pourra être bloqué depuis la configuration.

En cas d'erreur sur une règle, afficher **AU MINIMUM** le fichier, la règle et le numéro de la ligne qui posent problème.

ATTENTION la règle doit se déclencher **UNIQUEMENT** si elle est activée dans la configuration.

Coder les règles suivantes :

- **array-bracket-eol** = on | off

L'accolade doit se trouver en fin de ligne pour les fonctions, if, boucles, …

Exemple:

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

// NO GOOD
int main(int argc, const char* const * argv)
{
  return 0;
}
```

- **operators-spacing** = on | off

Il doit avoir un espace de chaque côté entre tous les opérateurs binaires.

Exemple:

```
// GOOD
int a = 5 + 3;
```

```
// NO GOOD
int a =5 +4;
```

- **comma-spacing** = on | off

Il doit avoir un espace à droite d'une virgule.

Exemple:

```
// GOOD
printf("%d", a);
```

```
// NO GOOD
printf("%d",a);
```

- **indent** = n | off

L'indentation doit être respectée entre les différents blocs, **n** correspond au nombre d'espaces du décalage.

Exemple:

```
// GOOD
for(int a = 0; i < 10; i++) {
  printf("%d\n", a);
}
```

```
  // NO GOOD
  for(int a = 0; i < 10; i++) {
printf("%d\n", a);
  }
```

- **comments-header** = on | off

Test la présence d'un commentaire multi-ligne en entête de fichier.

- **max-line-numbers** = n | off

Les lignes ne doivent pas dépasser n caractères.

- **max-file-line-numbers** = n | off

Les fichiers ne doivent pas dépasser n lignes.

- **no-trailing-spaces** = on | off

Il ne doit pas avoir d'espace en fin de ligne. (en dehors de l'indentation classique)

Exemple:

```
// GOOD
if (1) {

}
```

```
// NO GOOD
if (1) {

}
```

Partie 3 : Les règles sur les variables et fonctions

- **no-multi-declaration** = on | off

Il ne doit pas avoir plusieurs déclarations de variable sur une même ligne.

Exemple:

```
// GOOD
int a;
int b;
```

```
// NO GOOD
int a, b;
```

- **unused-variable** = on | off

Il ne doit pas avoir de variable inutile dans le code.

Exemple:

```
// GOOD
int main(int argc, const char* const * argv) {
  int a = 10;
  if (a > 5) {
    printf("I<3C");
  }
  return 0;
}
```

```
// NO GOOD
int main(int argc, const char* const * argv) {
  int a = 10;
  return 0;
}
```

- **undeclared-variable** = on | off

Il ne doit pas avoir de variable non déclarée dans le code et manipulée quand même.
ATTENTION aux variables globales qui sont disponibles partout dans le code.

Exemple:

```
// GOOD
int func() {
  int a = -1;
  a++;
  return 0;
}
```

```
//  NO GOOD
int func() {
  a++;
  return 0;
}
```

- **no-prototype** = on | off

Les fonctions doivent toutes avoir un prototype dans le fichier.

Exemple:

```
// GOOD
void func(int*);
void func(int* p) {
  *p += 1;
}
```

```
//  NO GOOD
void func(int* p) {
  *p += 1;
}
```

- **unused-function** = on | off

Il ne doit pas avoir de fonction inutile dans le code.

Exemple:

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

```
// NO GOOD
int main(int argc, const char* const * argv) {
  return 0;
}

void func() {
  
}
```

- **undeclared-function** = on | off

Il ne doit pas avoir de fonction non déclarée dans le code et déclenchée quand même.

Partie 4 : Les vérifications de type

- **variable-assignment-type** = on | off

Il ne doit pas avoir de problème de type lors de l'affectation d'une valeur dans une variable.

Exemple:

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

```
//  NO GOOD
int main(int argc, const char* const * argv) {
    int a = argv;
    return 0;
}
```

- **function-parameters-type** = on | off

Il ne doit pas avoir de problème de type lors de passage de paramètres dans une fonction.

Exemple:

```
// GOOD
void swap(int* a, int* b) {
  *a -= (*b = (*a += *b) - *b);
}
int main(int argc, const char* const * argv) {
    int k = 10;
    int c = 30;
    swap(&k, &c);
    return 0;
}
```

```
// NO GOOD
void swap(int* a, int* b) {
  *a -= (*b = (*a += *b) - *b);
}
int main(int argc, const char* const * argv) {
    int k = 10;
    int c = 30;
    swap(k, c);
    return 0;
}
```

Règles de rendu de projet

L'évaluation du projet sera effectuée sur les points suivants :

- la réalisation de l'application en elle-même : les différentes parties doivent correspondre exactement à l'énoncé, et ne pas donner lieu à des plantages à l'exécution lors d'éventuelles mauvaises manipulations : **l'utilisation de bibliothèques extérieures est interdite.**

  Préférer rendre une partie incomplète plutôt qu'erronée.
- la construction d'un rapport de projet, fourni sous la forme d'un fichier pdf, contenant au minimum les informations suivantes :
  - une introduction, rappelant le sujet du projet, et la liste des étudiants y ayant participé,
  - une analyse rapide de l'application : liste des structures de données, description des fonctions principales, des choix d'implémentation et de tout détail technique important pour la compréhension du sujet,
  - un dossier d'installation de votre application,
  - un dossier d'utilisation précis (considérer que l'utilisateur final n'est pas un informaticien)
  - un bilan du projet, listant notamment les points non résolus, les difficultés techniques (ou humaines) rencontrées.

Prévoir de rendre l'intégralité des fichiers sources et compilés de l'application sur MYGES