EISTI CERGY - PAU	**Cycle ingénieur 1ère année** **Projet de parcours GSI** *Florent* Devin, *Bernard* Gloneau, *Nga* Nguyen	
Matière : **Projet de parcours GSI**		*Date* : **13 Juin 2014**
		Durée de l'épreuve : **1 semaine**
		Nombre de pages du sujet : **3**

1 Introduction

L'objectif de ce projet est de permettre la comparaison de deux codes et de donner un pourcentage de similitude. Vous coderez votre programme en C. Les groupes sont composés d'au maximum 3 personnes.
Il existe plusieurs stratégies pour modifier un code source : changer les commentaires, les types de données, le nom des variables, l'ordre des opérandes, ajouter des instructions redondantes, modifier les structures de contrôle (échanger les 2 branches d'un test), remplacer l'appel d'une fonction par son code … Pour faire face à toutes ces imaginations, les outils pour détecter la similarité entre 2 codes sources utilisent plusieurs techniques différentes : attribut-counting (nombre de variables, nombre d'opérateurs uniques, …), ranking measures (métriques sur les structures de contrôle tel que le nombre de chemins d'exécution, …), comparaison de chaîne de caractères (algorithme de Karp-Rabin, checksum, …), … ou une combinaison de toutes ces méthodes.
Le projet commence par une étude bibliographique sur les méthodes existantes. À cause de la durée limitée du projet, nous allons développer l'approche basée sur la structure du programme. Bien entendu, vous êtes libre d'enrichir votre programme avec d'autres méthodes heuristiques afin d'avoir une meilleure précision de similitude.

2 Objectifs

Afin de réaliser cet objectif, vous devrez comparer deux codes sources. Il ne s'agit pas de faire de la comparaison textuelle, mais plutôt de faire une comparaison entre deux structures de programme. Nous ne nous intéresserons uniquement au programme qui compile (c'est à dire sans erreur de syntaxe). Pour vous simplifier le travail, nous vous recommandons de regarder ce que fait la commande `gcc -E` *`fichierC.c`*. Ceci peut être une bonne base pour commencer l'analyse de code source, puisque de nombreuses actions sont effectuées lors de cette phase de préprocessing.
Regardez aussi, la librairie `glib` [1] qui vous simplifiera grandement le travail à faire...
Plusieurs sortes de structure peuvent être envisagées, dans ce projet nous nous intéresserons uniquement aux deux structures suivantes : AST (*Abstract Syntax Tree*), graphe d'appel de fonctions.
Dans un premier temps, pour vous simplifier les traitements, nous vous recommandons de reformater le code en entrée. Pour chaque élément syntaxique important, rajouter un espace avant et après. Essayer dans le même temps de ré-indenter correctement le code.
Par exemple :

```c
int main(int argc,char** argv) {
  int a;
  if(argc==2){
  a=3;}else a=5;
  printf("%d\n",a);
  return(0);
}
```

1. `https://developer.gnome.org/glib/2.40/`

Donnera :

```
int main ( int argc , char** argv ) {
  int a ;
  if ( argc == 2 ) {
    a = 3 ;
  } else {
    a = 5 ;
  }
  printf ( "%d\n" , a ) ;
  return ( 0 ) ;
}
```

2.1 AST (*Abstract Syntax Tree*)

Une fois ceci, vous extrairez l'arbre syntaxique du programme. Voici deux exemples de ce que pourrait être un arbre syntaxique à partir du programme précédent :

```
Fonction(main, List (ParametreFormel (argc, int), ParametreFormel (argv, char**)),
    Declaration (a, int),
    Alternative ( (argc, ==, 2), Affectation (a,3), Affectation (a, 5))
    Fonction (printf, List(Parametre("%d\n"), Parametre(a)))
    Fonction (return, List(Parametre(0)))
)
```

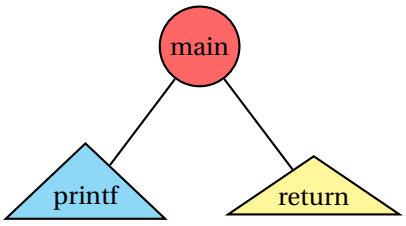
ou plus (trop ?) simple :

```
Fonction( Alternative () )
```

Bien évidemment, ceci n'est qu'un exemple, vous avez le droit de penser à une autre représentation. Il faut donc élaguer le programme afin de ne garder que les éléments structurants (mots réservés du langage). Ces éléments structurants devront être défini dans un fichier séparé, afin de pouvoir par exemple traiter d'autres langages (de même type).
Nous vous conseillons de ne pas obligatoirement faire un arbre syntaxique rigoureux, mais plutôt de vous concentrer sur les structures de contrôle qui sont plus faciles à trouver.
Une fois, l'arbre extrait il vous faudra le comparer afin de déterminer le taux de similarité. Plus votre arbre syntaxique sera précis, plus la distance entre deux codes sera précise.

2.2 Graphe d'appel de fonction

Par ailleurs, on veut aussi avoir un graphe d'appel des fonctions. Il faut donc extraire les fonctions définies. Puis réaliser un graphe d'appel de fonctions (c'est à dire quelle fonction appelle quelle fonction). Dans notre exemple, le graphe est l'arbre suivant :



Une fois extrait les deux graphes, il vous faudra donner une distance entre ces deux graphes afin de déterminer le taux de similarité.

3 Rendu

Voici quelques contraintes techniques à respecter :

- Le développement est en C.
- Il faut respecter la norme de programmation C.
- Pour le travail en équipe, vous pouvez utiliser Gitlab (`gitlab.etude.eisti.fr`) comme application de gestion de versions.
- Un fichier `Makefile` est indispensable pour automatiser les différentes tâches de votre projet.
- L'interface graphique n'est pas encouragée pour embellir votre logiciel.

À la fin du projet (deadline : vendredi 13 juin 2014 au soir), on vous demande de produire les rendus suivants sur Arel :

- une archive qui contient le code source, le fichier `Makefile` ainsi qu'un `README` ;
- un rapport .pdf.

Un code clair, structuré et bien commenté sera apprécié. Quant au rapport, il servira :

- à remercier avec émotion votre équipe pédagogique ^^ ;
- à présenter votre bibliographie (Google et autres Wikipedia sont vos meilleurs collaborateurs, à la condition expresse de citer vos sources) ;
- à critiquer les algorithmes étudiés ;
- à présenter les limites de votre programme, ainsi que les améliorations possibles ;
- à présenter vos jeux d'essai.