Thibaut Patel 3IF3 - 2010/2011
Stanislas Signoud Binôme **B3310**

Document de Spécification et de Conception

TP-AC – Références croisées

Table des matières

Présentation de l'application

L'application CrossedRefs permet de rechercher rapidement et efficacement les occurrences de certains identificateurs dans des fichiers donnés.

En entrée, cette application prends plusieurs paramètres :

- Une option `-k ficherkw` indiquant un fichier dans lequel sont indiqués les identificateurs à rechercher (*fichier mot clé*),
- Une option `-e` indiquant si le programme ne doit chercher que les identifiants indiqués, au lieu de chercher tous les identificateurs sauf les identifiants donnés ;
- Une liste de fichiers dans lequel il va rechercher les identificateurs.

Si l'option `-k` n'est pas renseignée, les identifiants sont les mots clés du C++.

En sortie, l'application affiche le résultat de la recherche dans un format spécifique :

Chaque ligne de sortie correspond à un mot clé recherché ; chaque nom de fichier où le mot clé apparait est ensuite affiché suivi des différents numéros de ligne correspondante séparés par des espaces, le tout étant séparés par des tabulations (affichés ici par une flèche).

```
motcleX→fileA 1 5→fileB 3
motcleY→fileB 8
motcleZ→fileD 4 8 15 16 23 42
```

Lexique

Dans ces spécifications et dans le programme, les termes suivants sont utilisés :

- **Mot clé ou identifiant :** Suite de caractères choisie pour être recherchée (ou être exclue de la recherche).
- **Identificateur :** Correspond à une chaîne qui est un identificateur au sens C++ du terme : plus précisément, il s'agit d'un *caractère de début d'identificateur* suivi, optionnellement, de *caractère de milieu et de fin d'identificateur* ; le nombre total de caractères d'un identificateur ne peut dépasser 255.
- **Caractère de début de l'identificateur :** Une lettre de l'alphabet simple (A-Za-z) ou un underscore (_).
- **Caractère de milieu et de fin de l'identificateur :** Un chiffre (0-9), une lettre de l'alphabet simple (A-Za-z) ou un underscore (u).
- **Fichier mot clés :** Un fichier en texte brut qui contient plusieurs *identificateurs*. Par convention, ces *identificateurs* sont séparés par des retours chariots[1].
- **Parseur :** Classe de l'application qui s'occupe d'analyser un fichier.
- **Index :** Classe de l'application qui permet de stocker à travers ses objets le résultat d'une recherche.
- **Analyseur :** Classe de l'application chargée de coordonner les *parseurs* pour effectuer une recherche sur plusieurs fichiers, et pour enregistrer le résultat de ces recherches dans un *Index*.
- **Liste de Mots Clés :** Classe de l'application chargée de stocker les *mots clés ou identifiants* à rechercher, offrant une abstraction permettant de savoir rapidement si un *identificateur* donné est un *mot clé* ou non.

Choix généraux

Le fichier est parcouru caractère par caractère. Le parseur ne renvoit à l'Analyseur que les identificateurs C++ valides. Si il y a plusieurs occurences d'unmême identifiant sur la même ligne, on affiche une seule fois la ligne (les doublons sont ignorés).

Si le fichier contenant la liste de mots clés n'est pas accessible ou s'il ne contient pas d'identificateur valide, le programme affiche une erreur dans la sortie d'erreur et retourne une valeur non nulle. Si un des fichiers dans lequel on doit chercher n'est pas accessible, le programme fait de même. Si le résultat total est vide, on en avertit l'utilisateur sur la sortie d'erreur (warning) mais le retour du programme reste nul.

Pour la gestion de la ligne de commande, on renvoie une erreur particulière pour chaque problème dans la ligne de commande.

1 Le programme acceptera toutefois un fichier comprenant des *identificateurs* séparés par n'importe quel séparateur n'étant pas en soi un *identificateur* (espaces, tabulations, ponctuation...).

Spécification des tests fonctionnels

Groupe A (tests proposés par le sujet)

Listing de file1.cpp

```
// affiche le message "Hello world"
int main(){
    cout << "Hello world" << endl;
    return 0;
}
```

Listing de file1.h

```
int main();
```

Listing de keywords.txt

```
int
world
template
```

Test A1

Fonctionnement classique avec fichier de mots clé, mots clés exclus

Ligne de commande :

```
crossedrefs -k keywords.txt file1.cpp file1.h
```

Sortie attendue :

```
Hello→file1.cpp 1 3
affiche→file1.cpp 1
cout→file1.cpp 3
endl→file1.cpp 3
le→file1.cpp 1
main→file1.cpp 2→file1.h 1
message→file1.cpp 1
return→file1.cpp 4
```

Test A2

Fonctionnement classique avec fichier de mots clé, mots clés recherchés uniquement

Ligne de commande :

```
crossedrefs -e -k keywords.txt file1.cpp
file1.h
```

Sortie attendue :

```
int→file1.cpp 2→file1.h 1
world→file1.cpp 1 3
```

Groupe B (extension des tests proposés par le sujet)

Test B1
Fonctionnement classique avec mots clés du C++, mots clés exclus

Ligne de commande :

```
crossedrefs file1.cpp file1.h
```

Sortie attendue :

```
Hello→file1.cpp 1 3
affiche→file1.cpp 1
cout→file1.cpp 3
endl→file1.cpp 3
le→file1.cpp 1
main→file1.cpp 2→file1.h 1
message→file1.cpp 1
world→file1.cpp 1 3
```

Test B2
Fonctionnement classique avec mots clés du C++, mots clés uniquement

Ligne de commande :

```
crossedrefs -e file1.cpp file1.h
```

Sortie attendue :

```
int→file1.cpp 2→file1.h 1
return→file1.cpp 4
```

Groupe C (cas inédit)

Listing de helloworld.py

```
1.  #!/usr/bin/env python2
2.
3.  def helloworld():
4.      print "Hello world !"
5.
6.  def bonjourlemonde():
7.      print "Bonjour le monde !"
8.
9.  def hello_i18n(language):
10.     lang = {"en": helloworld, "fr":
     bonjourlemonde}
11.     if language not in lang.keys():
12.         return lang[en]()
13.     else:
14.         return lang[language]()
```

Listing de testall.py

```
#!/usr/bin/env python2
import helloworld

malangue = raw_input("Langue ?")
helloworld.hello_i18n(malangue)
```

Test C1
Fonctionnement classique avec fichier mot clé, mots clés uniquement

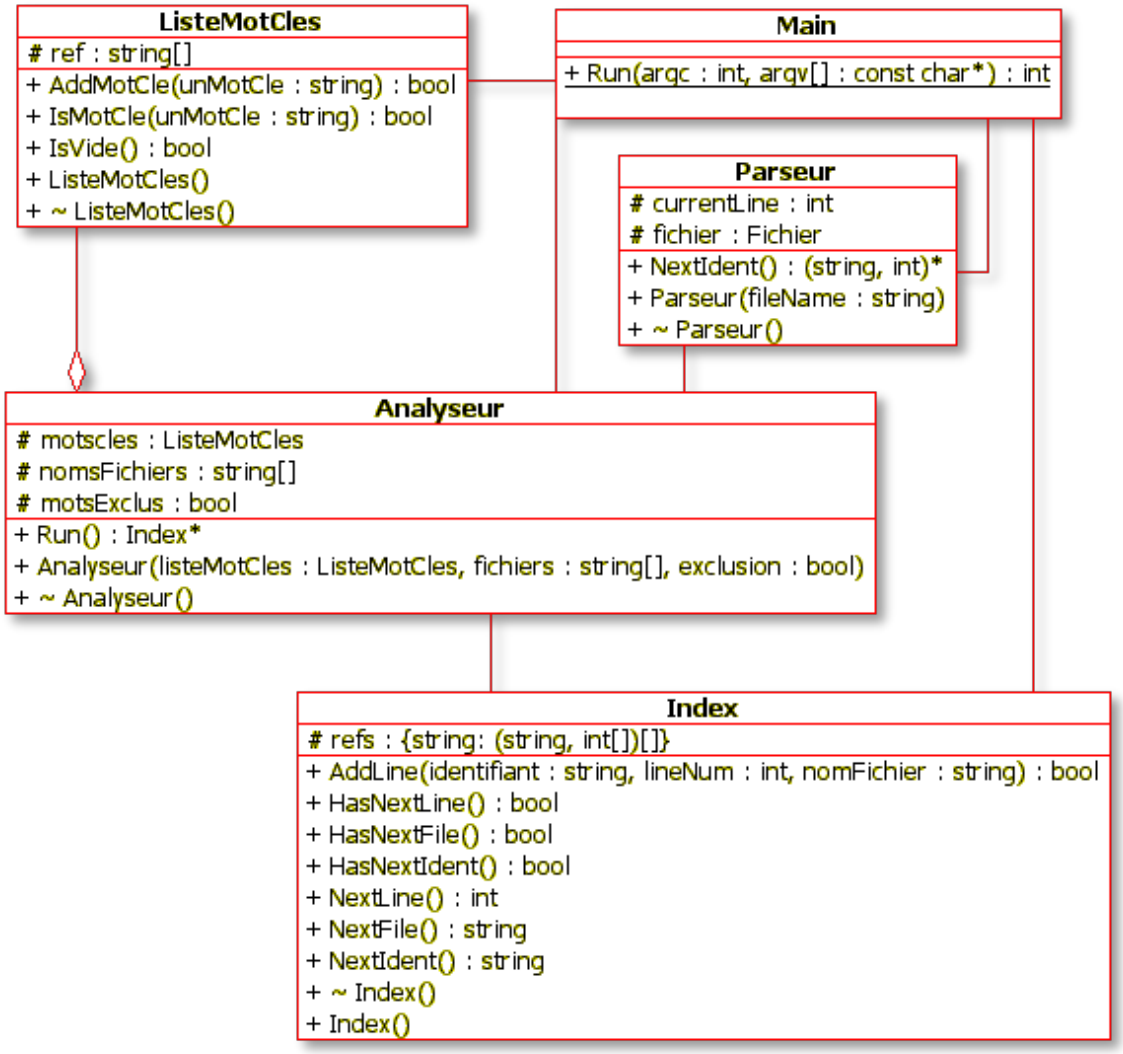
Ligne de commande :

```
crossedrefs -e -k
mesfonctions.txt helloworld.py
testall.py
```

Sortie attendue :

```
bonjourlemonde→helloworld.py 6 10
hello_i18n→helloworld.py 9→testall.py 5
helloworld→helloworld.py 3 10→testall.py 3 5
```

Schéma de l'architecture générale de l'application



Ce diagramme montre clairement la structure de l'application, chacune des classes effectuant le rôle indiqué dans le lexique. Pour plus de lisibilité, les attributs et méthodes privées d'Index utiles à la réalisation des méthodes HasNext et Next ne sont pas affichées, tout comme les constructeurs de copies et les opérateurs d'affectation.

La classe Main ne contient qu'une méthode publique : Run. Elle est chargée d'analyser les arguments du programme, de construire un Analyseur avec les bons paramètres et d'afficher l'Index qui en retourne dans le format indiqué ; ces opérations sont vraiment spécifiques à l'application et leur généralisation n'a pas en soi d'intérêt.

La classe Index contient toute la complexité nécessaire au parcours de la structure de données utilisée pour le stockage des occurrences des identifiants.

On notera que la classe Parseur est utilisée également par Main : grâce au travail de réutilisabilité, il est aisé de réutiliser cet objet.

Liste des algorithmes principaux

<chaîne_caractère , entier>* Parseur::NextIdent()

```
Le flux f (attribut de classe) est déjà ouvert. L'entier ligne a été initialisé à 1
lors de l'ouverture de ce flux.
caractère c
booléen est_dans_identifiant ← faux
chaîne_caractère buffer ← chaîne_vide
entier nbre_caractère ← 0

tant_que ( (c ← f.getcaractère()) n'est_pas 'fin_de_fichier' ) alors

   si c est retour_à_la_ligne alors
        ligne ← ligne + 1
   fin_si

   si est_milieu_identifiant(c) = vrai et est_dans_identifiant = vrai alors
        nbre_caractère ← nbre_caractère + 1

        si nbre_caractère > 256 alors
            buffer ← chaîne_vide
            est_dans_identifiant ← faux
            entier nbre_caractère ← 0
        sinon
            buffer ← buffer.concaténé_à(c)
        fin_si

   sinon_si est_milieu_identifiant(c) = faux et est_dans_identifiant = vrai alors
        est_dans_identifiant ← faux
        renvoyer <buffer, ligne>

   sinon si est_début_identifiant(c) = vrai et est_dans_identifiant = faux alors
        buffer ← c
        est_dans_identifiant ← vrai
        entier nbre_caractère ← 0
   fin_si

fin_tant_que

renvoyer pointeur_nul
```

Index* Analyseur::Run()

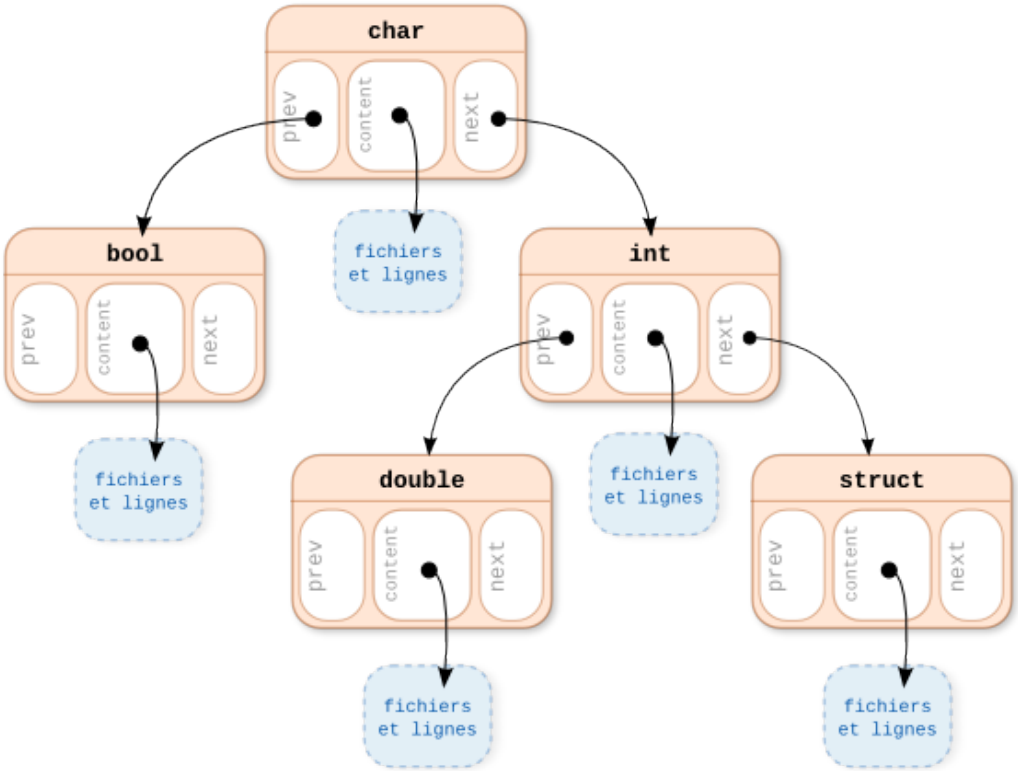
```
L'index index est déjà créé. On dispose du booléen exclureMotCles.
Pour chaque fichier f
   si f non_accessible ou f vide alors
        ! Avertir l'utilisateur !
   sinon
        créer le Parseur p sur le fichier f
        <chaîne_caractère , entier> paire
        tant_que ( (paire ← p.NextIdent()) n'est_pas pointeur_nul) alors
            booléen est_mot_clé ← listeMotCles.IsMotCle(paire->premier)
            si (exclureMotCles = vrai et est_mot_clé = faux) ou
                        (exclureMotCles = faux et est_mot_clé = vrai) alors
                   Ajouter paire à l'Index index
            fin_si
        fin_tant_que
   fin_si
fin_pour
```

bool Index::AddLine(string identifiant, int lineNum, string fichier)

```
si (identifiant) existe_pas  alors
      ajoutIdentificateur(identifiant)
fin_si
si index[identifiant].dernierfichier différent_de  fichier
      ajoutFichier(identifiant, fichier)
fin_si
si index[identifiant].dernierfichier.ligne_existe(lineNum) = faux alors
      Liste_fichiers liste ← index[identifiant]
      Fichier f ← liste.dernierfichier
      f.ajouteLigne(lineNum)
fin_si
```

Analyse critique complète des structures de données

Structure stockant les identifiants avec leurs occurences

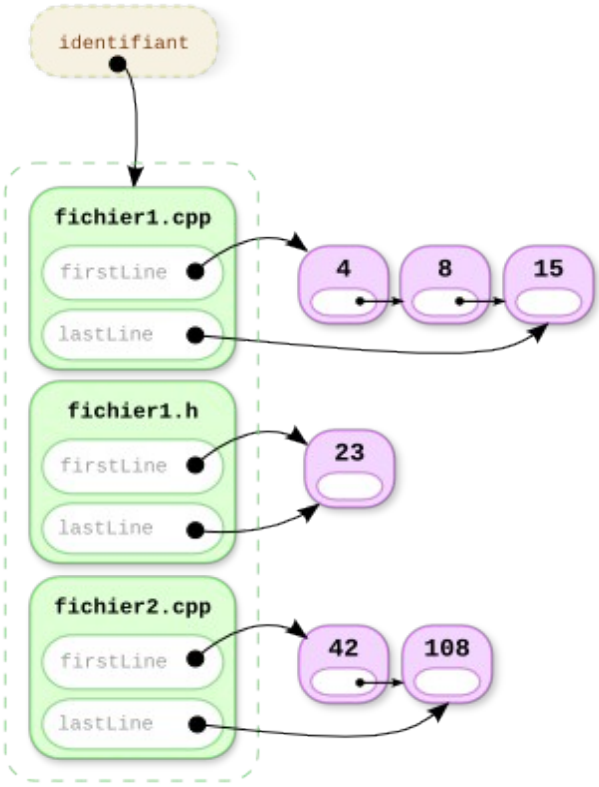


L'utilisation d'un arbre binaire (comme représenté ci-contre) pour stocker les identifiants trouvés semble une évidence. Ainsi, la complexité de la recherche doit être la plus faible possible, et l'utilisation d'un arbre binaire, dès lors qu'il est équilibré, permet une insertion et une recherche en O(log(n)), ce qui correspond à nos besoins.

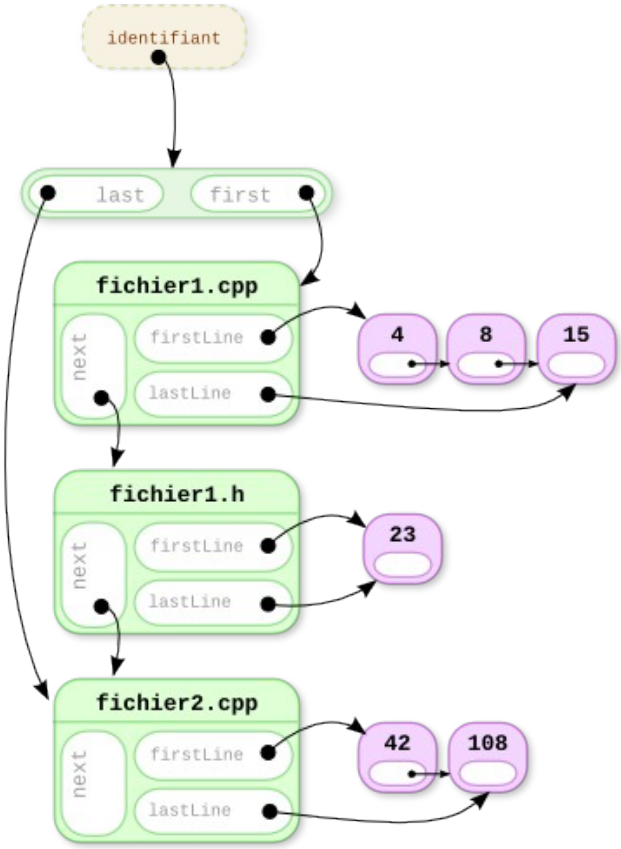
Une alternative plus farfelue, basée sur un tableau de pointeur vers une structure contenant les fichiers et les lignes, auraient eu des conséquences désastreuses aux niveau des performances, dès lors que le nombre d'identifiants est important, la recherche (en O(n)) se déroulant à chaque rencontre d'un identificateur dans les fichiers de recherches.

Structure stockant les occurences (fichiers et numéros de lignes)

Le choix d'une structure pour stocker les occurrences (les associations fichiers/lignes) est plus complexe.



L'utilisation d'un tableau de paire de chaîne et de liste chaînée d'entier a tout d'abord été trouvée (comme illustré ci-contre), ce tableau ayant tout le temps la même taille, celle du nombre total de fichier concernés par la recherche. Elle semblait à la fois performante et efficace ; cependant, elle possédait un problème majeur : l'occupation mémoire inutile pouvait être importante si des identificateurs étaient peu présents dans un grand nombre de fichiers (typiquement, en C++, les structures de langage tels que if ou for sont absents de tous les fichiers d'entête).



La solution illustrée ci-contre (une liste chaînée de paire de chaîne et de liste chaînée d'entiers) permet un accès direct au fichier en cours d'analyse, une insertion rapide d'un nouveau numéro de ligne, tout en permettant également de garder une taille nulle en cas d'identifiant absent de tous les fichiers. Cette solution, idéale, a été retenue pour cet utilitaire, en supposant qu'il puisse être utilisé dans le cadre d'une analyse de centaines de fichiers dans un grand projet.