# Projet informatique SICOM
# CAMUS-KAPLAN

## Livrable n°2 : Mise en place des files et début de l'analyse syntaxique

## Introduction

La dernière fois, nous avions réussi à construire un programme d'analyse lexicale, mais sans une file de lexèmes en sortie. Une fois cette dernière programmée, elle va nous permettre de faire l'analyse syntaxique.

Nous parlerons donc, dans ce rapport, de la mise en place des files dans notre code dans la première partie. Puis le traitement grammatical sera ensuite évoqué.

## 1. Mise en place des files de lexèmes

Pour pouvoir effectuer la vérification de la grammaire dans le programme, il a fallu mettre en place une file de lexèmes contenant les différents morceaux de la séquence d'instructions dans le fichier.s à compiler. Nous avons choisi une file pour pouvoir relire les différents lexèmes stockés dans l'ordre.

Au début nous avions des problèmes de pointeurs. Pour remédier à cela, pendant les vacances nous avons travaillé chacun de notre côté et avons finalement retenu une solution s'inspirant d'un code d'un tutoriel du site « *openclassrooms.com* », sous licence Creative Commons, et en l'adaptant à notre projet. (Le lien se trouve dans les sources en fin de rapport et dans le fichier Read_me). Cela donne, dans la figure 1, l'extrait suivant :

```
void enfiler(File *file, Lex_t lex)
{
   Element *nouveau = malloc(sizeof(*nouveau));
   if (file == NULL || nouveau == NULL)
   {
      exit(EXIT_FAILURE);
   }

   nouveau->lexeme = lex;
   nouveau->suivant = NULL;

   if (file->premier != NULL) /* La file n'est pas vide */
   {
      /* On se positionne à la fin de la file */
      Element *elementActuel = file->premier;
      while (elementActuel->suivant != NULL)
      {
         elementActuel = elementActuel->suivant;
      }
      elementActuel->suivant = nouveau;
   }
   else /* La file est vide, notre élément est le premier */
   {
      file->premier = nouveau;
   }
}
```

*Figure 1 : nouvelle version de la fonction enfiler se trouvant maintenant dans le fichier « file.c »*

Tests et résultats

Le test de ce programme s'est fait sur le fichier « *miam_sujet.s* ». La figure 2 montre une partie des résultats et le compte-rendu de la mémoire utilisée lors du programme, grâce au logiciel Valgrind :

```
nom :
ligne : 1
type de lexeme : [COMMENT]

nom :
ligne : 2
type de lexeme : [COMMENT]

nom : .set
ligne : 3
type de lexeme : [DIRECTIVE]

nom : noreorder
ligne : 3
type de lexeme : [SYMBOLE]

nom : Retour_ligne
ligne : 3
type de lexeme : [RETOUR_LIGNE]
```

```
==3449== HEAP SUMMARY:
==3449==     in use at exit: 4,449 bytes in 215 blocks
==3449==   total heap usage: 384 allocs, 169 frees, 12,817 bytes allocated
==3449==
==3449== Searching for pointers to 215 not-freed blocks
==3449== Checked 62,488 bytes
==3449==
==3449== LEAK SUMMARY:
==3449==    definitely lost: 416 bytes in 49 blocks
==3449==    indirectly lost: 4,033 bytes in 166 blocks
==3449==      possibly lost: 0 bytes in 0 blocks
==3449==    still reachable: 0 bytes in 0 blocks
==3449==         suppressed: 0 bytes in 0 blocks
==3449== Rerun with --leak-check=full to see details of leaked memory
==3449==
==3449== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==3449== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

*Figure 2 : Au-dessus, un extrait de la lecture des lexèmes des trois premières lignes de « miam_sujet.s », et en dessous, le résumé d'utilisation mémoire du test*

Si on analyse les lignes des résultats, le nom d'un lexème de type [COMMENT] est celle de la ligne lu. (Ici la ligne vide 25, en fin du test, prend le nom d'un lexème de type [COMMENT]). Nous pourrons corriger cet aspect, en ignorant les lignes ou les fins de lignes avec les commentaires, lors de la vérification grammaticale.

A part cela, le résumé de la mémoire utilisée n'indique aucune d'erreur. Mais il met en évidence des fuites de mémoire. On a tenté de corriger cela avec des free. Mais cela a provoqué des erreurs de segmentation et nous avons été obligé de laissé certaines fuites, pour éviter les erreurs.

## 2. Analyse grammaticale

L'analyse grammaticale doit permettre de vérifier que les lexèmes enfilés lors de l'analyse lexicale forment une séquence valide dans le langage assembleur MIPS.

Pour cela, on veut passer d'une file de lexèmes à quatre collections respectivement d'instructions .text, de données .data, de données .bss, et de symboles. Nous avons donc créé un module dédié aux listes (liste.c, liste.h) où nous avons défini les fonctions utiles et les nouvelles structures. Nous essayons d'utiliser des listes génériques. Nous avons aussi commencer à créer la machine à état qui doit décomposer la file de lexèmes. Un dictionnaire, *dico_instructions.txt*, contenant les instructions de l'assembleur MIPS.

## 3. Bilan et objectifs à venir

L'analyse lexicale se fait bien. La prochaine phase à réaliser complètement est l'analyse grammaticale où il faudra charger un dictionnaire des instructions de l'assembleur MIPS et comparer les lexèmes obtenus à la fin de l'analyse lexicale aux éléments de ce dictionnaire. Cela est en cours de réalisation pour l'instant, mais pas encore au point.

Nous avons eu quelques difficultés pour la réalisation de cette étape, en partie en raison de notre retard portant sur l'incrément 1. On était incertain par rapport à la réussite du programme final. On a donc créé notre répertoire à partir de tests pour éviter les dérives, quitte à perdre du temps.

## Sources


- Mathieu NEBRA, **Les piles et les files**, in **Apprenez à programmer en C !,** lien : **https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/19868-les-piles-et-les-files**