École nationale supérieure d'informatique pour l'industrie et l'entreprise

---

Rapport de projet d'analyse d'un fichier au format PDF

« Projet Compilation »

Élève : **Adam Ouzegdouh**

Responsable pédagogique : **Christophe MOUILLERON**

Table des matières

0. Introduction

L'objectif de ce document est de présenter l'ensemble de mes travaux sur le projet de compilation.

Ce projet a pour but de développer une série d'outils permettant, étant donné un fichier PDF, d'extraire de l'information sur sa structure. Pour cela, nous avons développer différents parseurs dans le but d'analyser le contenu d'un fichier. Idéalement, ces outils doivent pouvoir servir de base au développement d'un code permettant de réparer, voire même d'éditer, un fichier PDF.

Le sujet étant composé de quatres parties, ce rapport présentera mon travail sur chacune des ces parties tout en étant très synthétique accompagné d'explications sur la méthodologie mis en place, les problèmes rencontrés et les cas d'usages.

N'ayant pas eu le temps nécessaire de traiter l'ensemble des parties, tout au long de ce rapport, je présenterai les parties suivantes :

1) Récupération du numéro de version PDF et de l'adresse de la table de références
2) L'extraction de la liste des objets PDF depuis la table de références
3) Mise en place d'un parseur pour les objets PDF

Pour chacune des parties, un script de test python *script_test.py* existe afin de tester notre grammaire sur un fichier texte réunissant un/des cas de tests ou un fichier pdf de référence (qui représente le sujet du projet).

Ces scripts s'auto-suffisent, ils compilent et tests les grammaires, il suffit donc de l'executer avec cette commande :

$ *python3 script_test.py*

De plus, pour chaque partie et question, un répertoire existe contenant :

- Un fichier parseur ***parser.y*** comprenant l'arbre syntaxique qui vérifie que la séquence de tokens respecte bien la grammaire.
- Un lexer ***parser.l*** qui définie la grammaire lexicale (je l'ai nommé ainsi pour simplifier le makefile)
- Un script python de test, un/des fichiers de texte pour les tests
- Un ***Makefile***.

1. Récupération du numéro de version PDF et de l'adresse de la table de références

1.1. Question 1

Pour cette question, je n'ai pas utilisé la règle conseillé dans le sujet :
S : VERSION lines LINE

En effet, j'ai eu des difficultés à récupérer la version du PDF et le *startxref*.

J'ai donc mis en place une grammaire simple qui a pour but de détecter uniquement les éléments qui nous intéresse puis arrête le programme (et ignore le reste des caractères).

Vous pouvez retrouver cette logique mise en place depuis les fichiers parseur et lexer du répertoire *Partie_1/Question_1*.

1.2. Question 2

Comme je n'ai pas utilisé la grammaire conseillé, je n'ai pas eu de problèmes sur la vérification de l'adresse située à l'avant dernière ligne.

1.3. Question 3

Afin de récupérer la version du PDF et l'adresse de la table de référence dans des variables, j'ai modifié assez peu mon code de la *Question 1* de la manière suivante :

- Initialiser 2 variables qui pourront être utilisé dans l'arbre syntaxique

```
%parse-param {char **version}
%parse-param {int *startxref}

%%
S: VERSION STARTXREF {
        *version = $1;
        *startxref = $2;
    }
 ;
%%
```

- Modification des paramètres avec *yyparse*() et de *yyerror*()

```
extern int yyparse(char **version, int *startxref);
```

```
extern void yyerror(char** pdf_version, int* startxref_adress, const
char *msg);
```

2. L'extraction de la liste des objets PDF depuis la table de références

2.1. Question 4

Pour cette question, afin de détecter une table de références et un trailer, j'ai mis en place 2 struct pour chacun (dans un *code requires* sinon cela ne marchait pas) :

```
%code requires {
typedef struct {
    char *addr;
    int gen;
    char status;
} Entry;

typedef struct {
    int i;
    int n;
} Header;
}
```

Ensuite, après mis en place ma grammaire dans le lexer, j'ai d'ailleurs utilisé $\%x$ afin de déclarer les états de départs de d'arrivée pour le dictionnaire.

```
"<<"                    { yylval.str = strdup(yytext); BEGIN(DICTIONARY);
return DICT_START; }
<DICTIONARY>{
    ">>\n"               { yylval.str = strdup(yytext); BEGIN(INITIAL); return
DICT_END; }
    .|\n                { /* Ignorer le contenu du dictionnaire */ }
}
```

Le fonctionnement du programme est le suivant :

- Détection d'une table de références correcte
  - Vérification de la présence de tous les éléments d'une table de références dans l'ordre
  - Vérification du nombre de lignes indiqué par les deux entiers i et n (il y en a $n - i + 1$) tout en **prenant en compte que seulement les status à n**
- Détection d'un trailer correcte
  - Vérification de la présence de tous les éléments d'un trailer dans l'ordre en s'arrêtant au *EOF*

Tous les éléments détectés sont affichés sur la sortie standard.

Le programme s'arrête directement à la lecture d'un caractère qui n'est pas prévu dans la grammaire.

2.2. Question 5

Afin de récupérer les informations relatives aux différentes objets PDF dans la liste, j'ai utilisé la structure précédente en la transformant en une liste chainée.

```
typedef struct Entry {
    char *addr;
    int gen;
    char status;
    struct Entry *next;
} Entry;
extern Entry *head;
```

En effet, j'ai ajouté 2 fichiers *myparser.h* et *myparser.c* dans lesquels on peut retrouver les définitions des fonctions suivantes :

```
void add_entry(Entry new_entry);
void print_entries();
void free_entries();
```

Ces fonctions servent à ajouter un objet PDF de type n dans la liste tout triant cette liste dans l'ordre croissant des adresses, puis, 2 fonctions qui ont pour but de respectivement afficher et libérer la liste chainée.

Donc à présent, au lieu d'afficher directement dans l'arbre syntaxique, j'appelle *add_entry* pour ajouter l'objet PDF à la liste, et j'affiche cette dernière ultérieurement dans le *main*.

Note : J'ai aussi réecrit le *Makefile* pour qu'il inclus les 2 nouveaux fichiers dans la compilation.

2.3. Question 6

Pour cette question, je pense avoir fait beaucoup trop compliqué mais passons.
Tout d'abord, j'ai voulu relier cette question à la logique mis en place dans la partie 1. Pour cela, j'ai mis en place un *Makefile* qui va appeler celui de la partie 1, puis, j'ai ajouté une définition de fonction dans mon *myparser.h* :

```
int get_startxref_from_external(const char* filename);
```

Cette fonction a pour but de d'executer le *parser.bin* de la partie 1, puis, récupérer le *startxref* du document PDF sur la sortie standard et de le renvoyer.

Cette méthode (assez fastidieuse) m'a permis de récupérer l'adresse de la table de références au début du *main* car le Yacc le permet pas d'avoir des conditions dans l'arbre syntaxique.

3. Mise en place d'un parseur pour les objets PDF

3.1. Question 7 & 8

Pour ces questions, vous pouvez retrouver la grammaire et ma logique dans mes fichiers *parser.y* & *.l* dans le répertoire *Partie_3.1*.

J'ai fait en sorte que ma liste et mon dictionnaire soit recursive pour pouvoir gérer tous les cas possibles.

En restant sur ce sujet, j'ai initialisé une variable qui pour chaque objet PDF, vérifie si il est correct, sinon, elle passe à 0 et la liste/dictionnaire devient invalide.

De plus, j'ai eu beaucoup de problèmes sur la mise en forme d'un dictionnaire, je suis conscient que ma gestion d'erreurs dans ce cas est peu rigoureuse.

Vous pouvez retrouver une série de tests dans *test.txt* à executer avec le script python de test associé.

Le programme continue d'ailleurs même après une ou plusieurs erreurs.

La détection d'erreurs se fait au caractère près.

4. Conclusion

Pour conclure, le temps ne m'a pas permis de toucher à l'ensemble des parties du projet, en particularité la partie 4 où l'objectif était de tout assembler.
D'ailleurs, je pense que j'aurai agit de la même manière que pour la Question 6, c'est-à-dire que j'aurai utilisé les programmes précédents et exploiter la sortie standard pour pouvoir lié un peu toutes mes parties.