Projet : Analyse d'un fichier au format PDF

Consignes

Différents rendus sont à faire sur http://exam.ensiie.fr :

- La partie 1 est à rendre pendant le TP du 30 mai dans le dépôt `proc_rendu1`,
- La partie 3.1 est à rendre pendant le TP du 7 juin dans le dépôt `proc_rendu2`,
- L'intégralité du projet est à rendre avant le **28 juin à 23h59**, dans le dépôt `proc_rendu_final`, sous la forme d'une archive au format `.tar.gz` contenant vos codes pour les différentes parties, vos fichiers de test éventuels et un rapport au **format PDF**.

L'objectif de ce projet est de développer une série d'outils permettant, étant donné un fichier PDF, d'extraire de l'information sur sa structure. Pour cela, nous allons 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 est composé de quatre parties :

1. récupération l'adresse de la *table de références*,
2. extraction de la liste des *objets PDF* depuis la *table de références*,
3. mise en place d'un parseur pour les *objets PDF*,
4. analyse d'un fichier PDF.

Vous devez faire au minimum les trois premières parties, et il faudra aborder certaines questions de la quatrième partie pour avoir une (très) bonne note.

0 Informations préliminaires

Pour commencer, les informations sur le format PDF dont vous aurez besoin pour faire le projet seront toutes introduites au fur et à mesure du sujet. Si vous souhaitez éclaircir un point ou en apprendre davantage, vous pouvez consulter le document plus complet mis à disposition dans `/pub/FISA_MATH24/PROC` (accessible aussi via `https://pydio.pedago.ensiie.fr`).

Dans chaque partie, on va vous demander d'écrire un code dont un des arguments est un chemin vers un fichier PDF. Traiter toutes les subtilités du format PDF serait beaucoup trop ambitieux, et nous ferons donc les hypothèses suivantes sur le fichier en argument :

- Le document n'est pas protégé par un chiffrement ;
- Le fichier ne contient qu'une seule *table de références* ;
- Cette *table de références* n'est pas compressée ;
- Les *objets PDF* sont définis directement dans le corps du document (et pas à l'intérieur d'un *flux*) ;
- Les *flux* (`stream`) ne contiennent jamais la suite de caractères `endstream`. Cette chaîne représentera donc toujours le marqueur de *fin de flux* ;
- Les parenthèses à l'intérieur des `chaînes de caractères PDF` sont toujours précédées du caractère `\`.

En pratique, cela signifie que vos codes seront en mesure de traiter quasiment [1] tous les fichiers PDF en version `1.4` (ou inférieure), non chiffré et sans modification incrémentale.

Tous les sujets de TP/TD de ce module, ainsi que ce sujet, respectent les hypothèses que nous venons de fixer. Vous pouvez donc utiliser ces fichiers afin de tester vos programmes. Vous pouvez aussi produire vos propres fichiers de tests en adaptant la commande :

```
$ qpdf --qdf --compress-streams=n --object-streams=disable input.pdf output.pdf
```

1. Seuls les fichiers ne vérifiant pas les deux dernières hypothèses poseront problème, mais cette situation est rare.

1 Récupération de l'adresse de la *table des références*

Un fichier PDF valide est constitué des éléments suivants [2], dans cet ordre :

1. un *header* d'une ligne, contenant un commentaire spécial de la forme `%PDF-xxx`, où `xxx` est la version du format PDF utilisée dans le fichier ;
2. un *corps*, qui est une succession de lignes vides, de commentaires (lignes commençant par `%`) et de définitions d'*objets PDF*. Ces *objets PDF* sont les éléments constitutifs du document PDF et feront l'objet de la partie 3 ;
3. une *table de références*, qui contient les adresses de chaque *objet PDF* du document et dont le format précis sera détaillé dans la partie 2 ;
4. un *trailer*, qui sera lui aussi détaillé en partie 2, et dont les trois dernières lignes sont :
   — une ligne avec uniquement le mot-clé `startxref`,
   — une ligne contenant l'adresse (un entier positif) du début de la *table de références*,
   — une ligne avec le commentaire spécial `%%EOF`.

L'objectif de cette partie est, étant donné un fichier, de vérifier qu'il s'agit bien d'un fichier au format `PDF` et d'extraire nos premières informations. Plus précisément, on souhaite :
— vérifier la présence du commentaire spécial en début de fichier,
— déterminer la version du format PDF utilisée dans le fichier,
— récupérer l'adresse à laquelle on peut trouver la *table des références*.
Pour cela, nous avons donc juste besoin de regarder la première et l'avant dernière ligne du fichier passé en argument.

Question 1. Écrivez les fichiers `parser1.y` et `lexer1.l`, dans le but de répondre aux trois besoins exprimés ci-dessus.

Le programme principal (fonction `main`) prendra comme unique argument le chemin vers le fichier à analyser. Si ce fichier n'est pas valide, on affichera un message d'erreur. S'il est valide, on affichera la version du format PDF et l'adresse de la *table de références* sur la sortie standard.

Votre parseur pourra commencer par une règle ressemblant à

```
S: VERSION lines LINE
```

où :
— `VERSION` est un lexème pour représenter les commentaires de la forme `%PDF-xxx`,
— `LINE` est un lexème pour représenter toute autre ligne du fichier,
— `lines` est un symbole non-terminal, représentant une suite de lignes et dont il faudra extraire la dernière ligne, vérifier qu'elle contient uniquement un entier positif, et afficher cet entier.

Question 2. Quelle(s) difficulté(s) rencontre-t-on si on essaie en plus de vérifier que l'adresse située à l'avant-dernière ligne est bien précédée du mot-clé `startxref` et suivie du commentaire spécial `%%EOF` ?

Question 3. Adaptez votre code afin, si le fichier est bien au format PDF, de récupérer le numéro de version et l'adresse de la *table de références* dans des variables.

note : Vous aurez besoin de l'option `%parse-param` de `bison`.

2 Extraction de la liste des `objets PDF` d'un fichier

Une fois l'adresse de la *table de références* connue, il est désormais possible d'analyser plus finement la fin d'un fichier `PDF`.

Une *table de références* valide est composée des éléments suivants, dans cet ordre :
— une ligne contenant le mot-clé `xref`,
— une ou plusieurs *tables*.

2. En réalité, si le ficher a subi des modifications incrémentales, il peut y avoir plusieurs *corps* avec pour chacun sa *table de références* et son *trailer*. Mais cette situation a été écartée précédemment.

Une *table* est constituée de :
- une ligne contenant deux entiers i et n séparés par un espace,
- n lignes de la forme `"aaaaaaaaaa ggggg x "` sans les guillemets et où l'espace final peut [3] être remplacé par le caractère `'\r'`.

Ces n lignes donnent les informations relatives aux *objets PDF* numéro i à $i + n - 1$. Ici, la partie `aaaaaaaaaa` est l'adresse de l'*objet PDF* sur 10 chiffres (en remplissant avec des `0` à gauche si besoin). La partie `ggggg` est le *numéro de génération* de l'objet sur 5 chiffres. Ce *numéro de génération* doit être compris entre 00000 et 65535. Enfin, `x` est une lettre parmi `n` (normal) ou `f` (free).

Nous pourrons ignorer les objets de type `f`. La plupart du temps, il n'y en a qu'un : l'objet numéro 0 à l'adresse `0000000000` et avec le *numéro de génération* `65535`.

Un trailer valide est composé des éléments suivants, dans cet ordre :
- une ligne avec le mot-clé `trailer`,
- un *dictionnaire* sur une ou plusieurs lignes, commençant par `<<` et finissant par `>>`,
- une ligne avec le mot-clé `startxref`,
- une ligne contenant l'adresse (un entier positif) du début de la *table de références*,
- une ligne avec le commentaire spécial `%%EOF`.

Nous n'avons pas besoin d'analyser le contenu du *dictionnaire* pour le moment. Son format précis sera décrit dans la partie suivante.

Question 4. Écrivez le code d'un programme qui, sur la donnée d'un fichier et d'une adresse :
- ouvre ce fichier en lecture seule,
- se positionne à l'adresse indiqué,
- vérifie que, à partir de cette adresse, le fichier contient bien une *table de références* valide, puis un *trailer* valide.

Pour cela, vous devrez au minimum créer les fichiers `parser2.y` et `lexer2.l`.

Question 5. Modifiez votre code pour stocker les informations relatives aux différents *objets PDF* dans une liste.

notes :
- Comme indiqué précédemment, il suffit de traiter les objets de type `n`.
- Vous aurez probablement besoin de définir une nouvelle structure pour stocker les informations d'un objet. Cette structure devrait au minimum contenir le numéro de l'objet, son adresse et son *numéro de génération*. Il est conseillé de définir la structure et les fonctions associées dans un fichier source à part.
- L'ordre des objets dans la liste n'a pas d'importance. Toutefois, avoir les objets triés par adresses croissantes peut s'avérer plus pratique pour la suite.
- Pour tester votre code, vous pouvez faire en sorte que votre programme affiche le contenu de la liste sur la sortie standard.

Question 6. Modifiez votre code afin de rendre le deuxième argument (l'adresse) facultatif.

Si cet argument n'est pas fourni, vous réutiliserez le code de la partie précédente pour déterminer vous-même l'adresse à utiliser.

note : Comme vous allez avoir deux fichiers `.y` et deux fichiers `.l`, vous ne pourrez plus utiliser systématiquement les noms de variables/fonctions par défaut de `bison` et `flex`. Le dernier exemple fourni sur la page web du cours montre comment changer ces noms.

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

Grâce aux deux parties précédentes, nous sommes désormais en mesure de déterminer la position des différents *objets PDF* d'un fichier. Notre objectif est maintenant de mettre en place un outil permettant de détecter un *objet PDF* valide et de sauvegarder les informations relatives à un tel objet.

3. C'est typiquement le cas avec un PDF généré depuis le système d'exploitation `Windows`.

3.1 Parseur pour les *objets PDF*

Un *objet PDF* valide possède l'une des formes suivantes.

1. une constante :
   — la valeur `null`,
   — un booléen = `true` ou `false`,
   — un entier signé,
   — un nombre réel signé (avec au moins un chiffre avant le `.`)
2. une *chaîne de caractères* PDF :
   — classique = une suite quelconque [4] de caractères entourée par `(` et `)`,
   — hexadécimale = une suite de chiffres, de lettres allant de `A` à `F` (majuscule ou minuscule), et de blancs (espaces, tabulations et sauts de lignes) entourée par `<` et `>`.
   On rappelle que nous avons fait l'hypothèse [5] que toute parenthèse dans une chaîne de caractères classique est précédée du caractère `\`.
3. un *nom*, commençant par le caractère `/` et pouvant ensuite contenir tous les caractères sauf les blancs et les séparateurs (parenthèses, crochets, accolades, `/`, `%`, `<` et `>`).
4. une *référence* vers une autre *objet PDF*, sous la forme de deux entiers positifs suivis de la lettre `R`.
5. une *liste* d'*objets PDF*, séparés par des blancs et entourés par `[` et `]`.
6. un dictionnaire commençant par `<<`, finissant par `>>` et composé d'une succession d'entrées. Ces entrées sont formées d'un *nom*, suivi d'un *objet PDF* quelconque, et sont séparées par des blancs.

Question 7. Écrivez les fichiers `parser3.y` et `lexer3.l`, afin de reconnaître les *objets PDF* valides.
Dans un premier temps, le programme principal (fonction `main`) se contentera de lire sur l'entrée standard. Pour chaque objet lu, le programme affichera soit un message sur la sortie standard indiquant que l'objet est valide, soit un message d'erreur.

Question 8. Ajoutez des traitements à vos règles de grammaire afin de conserver un maximum d'information sur chaque *objet PDF* valide lu.
Pour tester votre code, vous afficherez ces informations sur la sortie standard.

3.2 Vérification de la présence d'un *objet PDF*

Notre but est désormais de vérifier les informations fournies par la *table de références* d'un fichier PDF. Plus précisément, on souhaite vérifier que, pour chaque entrée de la table, il y a bien un `objet PDF` correctement défini à l'adresse indiquée.

On va noter a l'adresse associée à l'*objet PDF* dont on souhaite vérifier l'existence. Et on notera b la plus petite adresse dans la *table de références* parmi les adresses strictement plus grandes que a. En l'absence [6] d'adresse strictement plus grande que a, on prendra l'adresse de la *table de références* comme valeur pour b.

Ainsi, notre objectif est de lire le contenu du fichier de l'adresse a incluse à l'adresse b exclue, et de vérifier que cette partie du fichier contient bien la *définition d'un objet PDF*. Celle ci doit être constituée des éléments suivants, dans cet ordre :
— une ligne contenant deux entiers n et g, suivis du mot-clé `obj`. La valeur n est le numéro de l'objet, et g est le *numéro de génération*. Ces valeurs doivent correspondre au contenu fourni par la *table de références* ;
— une ou plusieurs lignes contenant l'*objet PDF* en lui-même. La syntaxe est celle présentée dans la partie 3.1, au détail qu'un *dictionnaire* peut ici être suivi d'un *flux*. Ce flux commence par le mot-clé `stream`, suivi d'une suite quelconque de caractères, et enfin du mot-clé `endstream` ;
— une ligne avec le mot-clé `endobj`.
De plus, la *définition d'un objet PDF* peut contenir des lignes vides et des commentaires (lignes commençant par le caractère `%`), que vous devrez ignorer (règles à ajouter dans le fichier `.l`).

4. En réalité, il y a quand même une restriction sur l'utilisation des parenthèses à l'intérieur de la chaîne. En pratique, les générateurs de PDF protègent toujours ces parenthèses en plaçant un caractère `\` devant. Seule l'utilisation de parenthèses
5. C'est ce que fait tout générateur de `PDF` raisonnable, même si le standard est un peu plus laxiste.
6. C'est le cas uniquement pour le dernier *objet PDF* défini dans le *corps* de notre fichier PDF.

On rappelle que nous avons fait l'hypothèse que la chaîne `endstream` apparaît uniquement comme marqueur à la fin d'un `flux`.

Question 9. En repartant de ce qui a été fait pour `parser3.y` et `lexer3.l`, créez les fichiers `parser4.y` et `lexer4.l` afin de reconnaître une *définition d'objet PDF* valide.

notes :
- Contrairement à `parser3.y`, nous souhaitons vérifier ici que le contenu fourni en entrée contient **une et une seule** *définition d'objet PDF* valide. Vous devrez donc adapter le début de la grammaire dans `parser4.y` en conséquence.
- Si vous souhaitez sauvegarder le contenu d'un *flux* dans une variable, il faudra utiliser dans votre fichier `lexer4.l` la fonction `memcpy` et la variable `yyleng` (indiquant le nombre de caractères dans `yytext`). Utiliser la fonction `strdup` pour cet usage serait incorrect [7].

Question 10. Modifiez votre programme afin de :
- prendre trois arguments = le chemin vers un fichier f et deux adresses a et b (avec $a < b$),
- récupérer le contenu de f de l'adresse a incluse à b exclus (par exemple à l'aide de `fread`) dans une variable `buffer`,
- ajouter **deux** caractères `'\0'` à la fin du `buffer`,
- appeler le code [8]

```
yy_scan_buffer(buffer, b-a+2);
yy_parse();
```

afin de demander à `bison` d'analyser le contenu de `buffer` au lieu de l'entrée standard.

Question 11. Testez votre nouveau programme.

4 Analyse d'un fichier PDF

Question 12. Utilisez ce que vous avez fait aux parties 2 et 3 afin d'écrire un programme qui, sur la donnée d'un fichier, vérifie qu'il s'agit d'un fichier PDF valide, analyse la *table des références* pour dresser la liste des adresses des différents *objets PDF*, et enfin détermine la nature de chacun de ces objets.

Question 13. Écrivez une fonction qui dresse la liste des fontes de caractères définies dans un fichier PDF. Vous pourrez pour cela regarder le contenu des *objets PDF* de type *dictionnaire* où la clé `/Type` est associée à la valeur `/FontDescriptor`.

note : Pour tester votre code, vous pourrez comparer votre résultat à celui de la commande `pdffonts`.

Question 14. Écrivez une fonction qui calcule le nombre de pages définies dans un fichier PDF.

Pour cela, vous pouvez compter les *objets PDF* de type *dictionnaire* où la clé `/Type` est associée à la valeur `/Page`.

Question 15. Écrivez une fonction qui calcule l'arborescence des pages d'un fichier PDF.

Il s'agit dans un premier temps de consulter le *dictionnaire* situé dans le *trailer* du fichier. Il contient une entrée `/Root` dont la valeur est une référence vers un *objet PDF*. Cet objet est un *dictionnaire*, qui doit contenir au minimum la clé `/Type` avec la valeur `/Catalog`, et la clé `/Pages` dont la valeur est une *référence* vers l'*objet PDF* servant de racine à l'arborescence des pages. Cette racine est elle aussi un *dictionnaire*, contenant au minimum :
- la clé `/Type` avec la valeur `/Pages`,
- la clé `/Count` avec comme valeur le nombre de fils de la racine,
- la clé `/Kids` avec comme valeur une *liste* de *références* vers les fils de la racine.

Les fils sont tous des *dictionnaires*, où la clé `/Type` a pour valeur soit `/Pages` (noeud interne de l'arborescence suivant les mêmes règles que la racine), soit `/Page` (feuille de l'arborescence, contenant la définition d'une page).

7. Pourquoi ?
8. N'oubliez pas de modifier le `yy` en fonction du nom que vous avez donner à votre parseur.