PDF Lexical Analysis

Boujamaa ATRMOUH

June 18, 2024

1 Introduction

The objective of this project is to develop a series of tools that, given a PDF file, can extract information about its structure. To do this, we will develop different parsers with the aim of analyzing the content of a file. Ideally, these tools should be able to serve as a basis for the development of code that can repair, or even edit, a PDF file.

2 PDF Structure

Addressing all the subtleties of the PDF format would be far too ambitious, and we will therefore make the following assumptions about the argument file:

- The document is not protected by encryption;
- The file contains only one reference table;
- This reference table is not compressed;
- PDF objects are defined directly in the body of the document (and not within a stream);
- Streams never contain the sequence of characters "endstream". This string will therefore always represent the end of stream marker;
- Parentheses within PDF character strings are always preceded by the character.

In practice, this means that our code will be able to handle almost all PDF files in version 1.4 (or lower), unencrypted and without incremental modification.

I

Files `parser1.y` and `lexer1.l`. The main program takes as its only argument the path to the file to be analyzed. If this file is not valid, an error message will be displayed. If it is valid, the version of the PDF format and the address of the reference table will be displayed on the standard output.

- `VERSION` is a lexeme to represent comments of the form %PDF-x,y,
- `LINE` is a lexeme to represent any other line of the file,
- `lines` is a non-terminal symbol, representing a sequence of lines from which the last line must be extracted, verify that it contains only a positive integer, and display this integer.

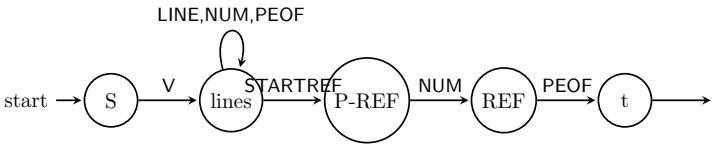

Figure 1: automata I

$$S \rightarrow \text{VERSION lines PEOF}$$

$$\begin{aligned} \text{lines} &\rightarrow \text{STARTREF NUM} \\ &\mid \text{LINE lines} \\ &\mid \text{NUM lines} \\ &\mid \text{PEOF lines} \end{aligned}$$

Figure 2: I: Grammar

$$S \rightarrow \text{VERSION lines}$$

$$\begin{aligned} \text{lines} &\rightarrow \text{LINE lines} \\ &\mid \text{NUM lines} \\ &\mid \text{PEOF lines} \\ &\mid \text{STARTREF ref} \\ \text{ref} &\rightarrow \text{NUM end\_line} \\ \text{end\_line} &\rightarrow \text{PEOF} \end{aligned}$$

Figure 3: I: Right-linear grammar

$$\begin{aligned} \text{VERSION} &= \texttt{\%PDF-[0-9]+\.[0-9]+} \\ \text{LINE} &= \texttt{[\^{}\textbackslash n]+} \\ \text{NUM} &= \texttt{[0-9]+} \\ \text{PEOF} &= \texttt{\%\%EOF} \end{aligned}$$

STARTREF = `startxref`

Command	Description
`make`	Compile the parser and lexer.
`make debug`	Compile in debug mode.
`make test`	Generate PDF files and test them.
`make testsNoLog`	Similar to `make test`, but with a clean console output.

II

Once the reference table address is known, further analysis of the end of a PDF file becomes possible. This involves validating a reference table and a trailer, which are essential components of a PDF file structure.

Reference Table Structure

A valid reference table in a PDF file consists of the following elements:

1. **Header Line (`xref`)**: Indicates the beginning of the reference table.
2. **Table Entries**: Each entry in the table provides information about a PDF object. The structure of each entry includes:
   - Object Address: A 10-digit address.
   - Generation Number: A 5-digit number indicating the generation of the object.
   - Type: Either 'n' (normal) or 'f' (free).

   These entries are typically organized in a format similar to:

   ```
   0000000015 00002 n
   0000000025 00000 n
   ```

   Here, `0000000015` is the object address, `00002` is the generation number, and `n` indicates it's a normal object.

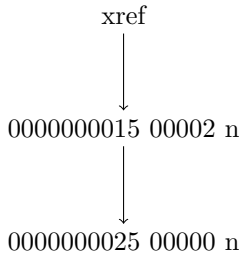

Figure 4: Schematic Representation of a Reference Table

Trailer Structure

After the reference table, a valid PDF file trailer includes:

1. **Trailer Keyword**: Marks the start of the trailer section.
2. **Dictionary**: A structured set of key-value pairs enclosed between `<<` and `>>`.
3. **Startxref**: Indicates the position of the start of the cross-reference table.
4. **Cross-reference Table Start Address**: An integer pointing to the beginning of the reference table.
5. **End-of-File Marker (`%%EOF`)**: Special comment indicating the end of the PDF file.

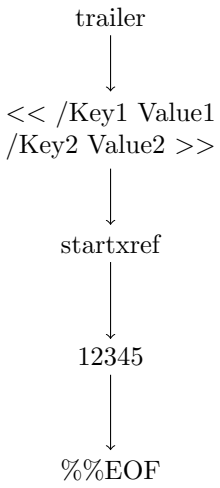

Figure 5: Schematic Representation of a PDF Trailer

Data Structures: A `PDFObject` structure is defined to store information about PDF objects, including their address, object ID, and generation number. This structure is managed through functions like `createPDFObject`, `insertPDFObject`, `printPDFObjects`, and `freePDFObjects`.

Command	**Description**
`make`	Compile the parser and lexer.
`make debug`	Compile in debug mode.
`make testSingle`	Run a single test.

III

The objective of this section is to develop a parser for valid PDF objects. According to the PDF specification, a valid PDF object can take several forms, each with specific characteristics that must be recognized and validated.

To parse and recognize different types of PDF objects, we define a grammar and corresponding lexical rules using `parser3.y` and `lexer3.l`. These files work together to identify and classify PDF objects based on their format and content.

In `lexer3.l`, lexical rules are defined using regular expressions to match various types of PDF objects:

- **Null, Boolean, Integer, Real**: Identified using specific patterns (`null`, `boolean`, `signedInt`, `signedReal`).
- **Strings**: Regular and hexadecimal strings enclosed in parentheses (`string`, `hexString`).
- **Names**: Start with `/` and can include any character except whitespace and specific delimiters (`name`).
- **References**: Composed of two integers followed by `R` (`ref`).
- **Lists**: Enclosed in square brackets (`startList`, `endList`).
- **Dictionaries**: Enclosed in double angle brackets (`startDict`, `endDict`).

The grammar in `parser3.y` specifies how these tokens are structured to form valid PDF objects:

Command	Description
`make`	Compile the parser and lexer.
`make debug`	Compile in debug mode.