# Arabic PDF Parser

The objective is to develop a pdf parser and Arabic text extractor.

Parsing PDF could be treated as 5-steps process, done respectively in the following order:

1. Traversing PDF logical tree and finding all ***Page*** objects - in a way you guarantee that pages are in the correct order.
   For each Page:
    2. Retrieve the Fonts information and ***ToUnicode*** Table.
    3. Retrieve the contents.
    4. Decode the contents.
    5. Position the text into their right orders.

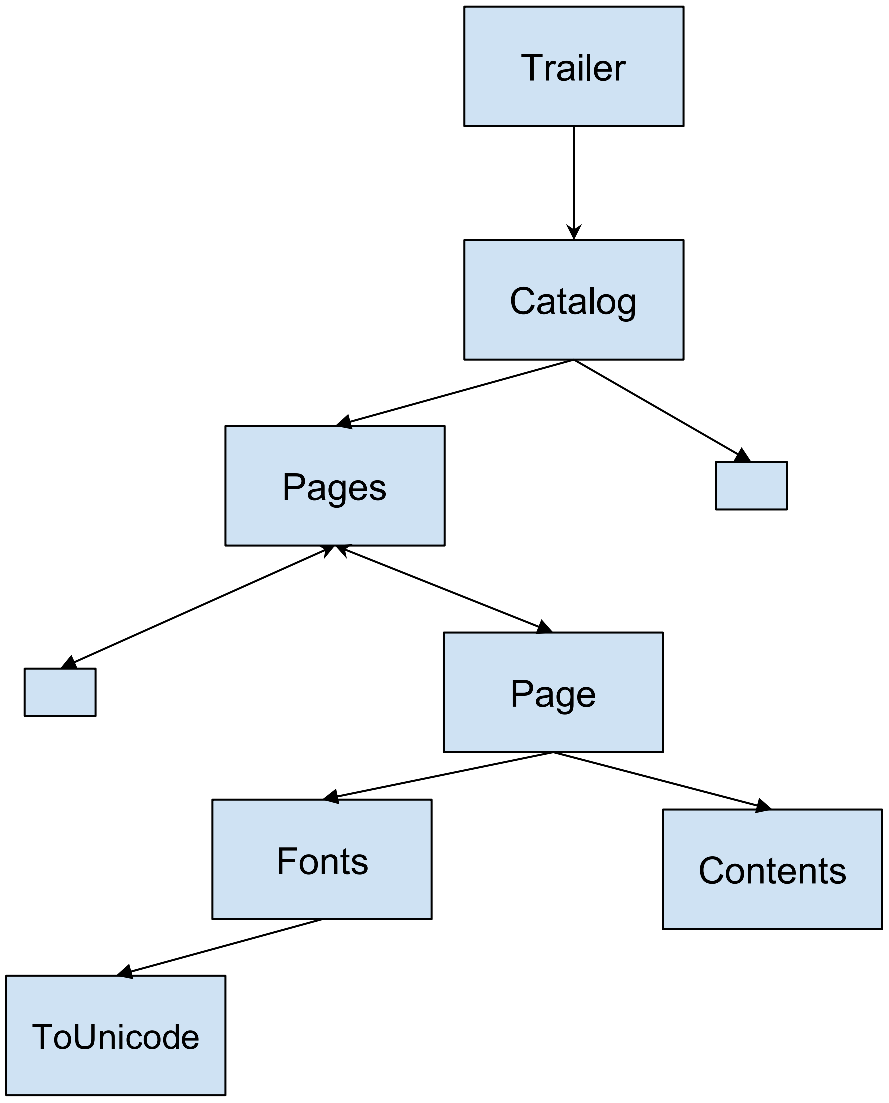
Before diving into the implementation details -provided in the notebook- it would be essential to have a background knowledge about pdf structure and how it works.

# Understanding PDF structure

Pdf files are composed of the following:

- a header
- a list of objects (for text extraction that direct attention will be on Pages, Page, content, Font, Cmap)
- across reference table
- a trailer

The relation between them is visualized as below:



Here is an example of a simple pdf.

# Objects In Details

## i. Page Object:

A ***Page*** object is referencing many objects, we care about ***/Contents*** and ***/Fonts***. ***/Contents*** could be a single value or an array of multiple contents. It points to an object that contains the text streams.
***/Fonts*** is a pointer in a dictionary that points to fonts specification and ToUnicode table that could be used to decode the contents

Other Page variables that might be helpful are ***/CropBox*** And /***Mediabox.*** Each Page will have a ***CropBox*** a rectangle that defines the visible region of default user space. Anything defined outside the CropBox then it's not visible to the user. If ***CropBox*** is not specified, then ***Mediabox*** is the default user space.

## ii. Contents Object:

a stream of string objects along with coordinate operators and other operators.
The way text is represented in the ***contacts*** is as follow,

| | |
|---|---|
| `BT` | → begin text |
| `/C2_0 1 Tf` | → font info |
| `15 0 0 15 -124.193 512.9449 Tm` | → transformation matrix |
| `<00BE00BE00BD00BC00C000BD00BB00BE00C1>Tj` | → text encoded "CIDs |
| `ET` | → ending text |

`BT` contains many `Tj` -or other showing operators- each followed by some operators such as font and positioning. CIDs which are the character code for the encoded text are encapsulated in `Tj`.

Below a brief explanation of the operators:

## Text state parameters:

*TL* It specifies the vertical distance between the baselines of adjacent lines of text, and it used only by *T**, *′*, *″*

## Text-positioning operators:

*tx ty Td* → Move to the start of the next line, offset from the start of the current line by (*tx* , *ty*)

More precisely, this operator performs the following assignments:

Tm=Tlm= [[1 0 0][ 0 1 0][*tx ty 1* ]] × Tlm

*tx ty TD* → Move to the start of the next line, offset from the start of the current line by (*tx* , *ty*).

As a side effect, this operator sets the leading parameter in the text state.

Same as

***−ty TL***

*tx ty Td*

*abcdef Tm* → The initial value for Tm is the identity matrix, [1 0 0 1 0 0]. This update it after each movement.The matrix specified by the operands is not concatenated onto the current text matrix, but replaces it.

Tm=Tlm= [[a b 0][ c d 0][*e f 1* ]]

Where a: horizontal scale, b: vertical scale, c: horizontal rotation, d: vertical rotation, e: horizontal position, f: vertical position

___ *T** → Move to the start of the next line. This operator has the same effect as the code

*0 Tl Td* where Tl is the current leading parameter in the text state.

## Text-showing operators:

the concern is only on these two:

- *<XXXX>Tj* → Show string at the current position.
- *[<XXXX><XXXX>]TJ* → This operator allows a text string to be shown with adjustments for individual glyph positions

## iii. Coordinates System and Text Space:

The transformation from text space to user space is defined by the text matrix *Tm* in combination with several text states that we can ignore if all we looking for is arranging the text without the exact pixel position.

The transformation from user space to device space is defined by the current transformation matrix (CTM) and it set by *cm* the coordinate transformation operator.Typically, this is done within a pair of q and Q operators to isolate the effect of the transformation. This transformation could be

- Translation : are specified as [ 1 0 0 1 tx ty ], where tx and ty are the distances to translate the origin of the coordinate system in the horizontal and vertical dimensions, respectively.
- Rotation,
- Reflection,
- and skew.

## iv. Font Object:

There are 3 main font technologies used in PDF font files:

1- Postscript/Type1

is an Adobe font format originally for use with PostScript. The standard 14 fonts are defined as Type 1 fonts.

2- Truetype

based on Apple's True- Type font format (also frequently used in Microsoft Windows).

3 - CID fonts "Type0"

These are composite fonts, intended to support multibyte character sets (where a font has a huge number of glyphs, such as Arabic).The main features that CID fonts add are the ability to have 16bit values (so 65535 separate glyphs rather than 256) and much more sophisticated and more flexible unicode settings for extraction.CID fonts are also better at allowing for text which does not have a left to right flow.

Font object has many attributes to specify, we are ignoring all of these for now and only retrieving ToUnicode which points to a stream containing instructions for the extraction of text content.

## v. ToUnicode CMaps :

ToUnicode CMaps It is a suitable scheme for representing the information content of text, but not its appearance. It maps CID (which begins at decimal 0,and expressed as 0x0000 in hexadecimal notation) to Unicode that identifies characters.
This arises as viewer applications sometimes need to determine the information content of text during operations such as searching, indexing, and exporting of text to other applications.
There are some situations where text is not extractable,But the glyphs can still be shown.
There are 3 methods for mapping a character code to a Unicode value, and in our situation is when the font dictionary contains a ToUnicode CMap

CMaps Operands:

- *`begincodespacerange`* and *`endcodespacerange`* operators that define codespace ranges—the valid input character code ranges—by specifying a pair of codes of some particular length giving the lower and upper bounds of each range . in case of ToUnicode thy are consistent with the encoding that the font uses
- *`Beginbfchar`* and *`endbfchar`*, In case of ToUnicode they define the mapping from character codes to 2-byte Unicode values, in- terpreted high-order byte first

Syntax :

```
 n beginbfchar
srcCode dstString

srcCode2 dstString2
endbfchar
```

example :

```
 2 beginbfchar
<07> <03C0>
<07> <03C0>
endbfchar
```

So character 7 mapped to unicode hex value 03C0 ..etc

- *`beginbfrange`*, and *`endbfrange`*, the same as *`beginbfchar`* and *`endbfchar`* , but for ranges of input codes, and they have been extended to handle cases where a single character code maps to one or more Unicode values.

Syntax:

```
n beginbfrange
srcCode1 srcCode2  dstString
srcCode1 srcCoden  [dstString1
dstString2 ... dstStringn]
endbfrange
```

Example:

```
2 beginbfrange
<0000> <005E> <0020>
<005F> <0061> [<00660066> <00660069>
<00660066006C>]
endbfrange
```

<0000> to <005E> are mapped to the Unicode values U+0020 to U+007E.
The character codes <005F> mapped to <00660066> which is ff and  <0060> to fi  ….etc.