Searching PDF files: Indexing

Michaël Van Canneyt

December 28, 2022

Abstract

In a previous contribution, we've shown how to show a PDF in a Pas2JS program. We also showed how to search the PDF. In this article we'll show how to prepare for searching a bunch of PDF files, by indexing them.

1 Introduction

In a previous article (reference: ??) we have demonstrated how to show a PDF file in a Pas2JS program. The program allowed to show the PDF, and to search the contents of the PDF. But what if you have lots of PDF files - for instance, all issues of Blaise Pascal Magazine - and want to search them ? The program as it was developed can only search a PDF file that was loaded in the browser's memory. Clearly, this is not very efficient for a library of many files.

Another approach is needed for this. The PDF files need to be indexed: create a database of words in the PDF and record the pages on which each word is present. By cross-referencing the pages with articles, we can get a list of articles that contain a certain word in their text. When searching, the search term is looked up in the list of words, and a list of locations (or articles) can be retrieved.

To be able to do this, we need to read the PDF file and extract the words, and create a database with the words. Free Pascal and Lazarus contain all the tools needed to do this job: the search database tools exists since a long time and are used in the Free Pascal documentation at:

`https://www.freepascal.org/docsearch/docsearch.var`

Recently, a set of units to read a PDF file have been developed in Free Pascal. Combining both systems will allow us to create a database of words in PDF files and allow to search them. Since we need to read a file before we can extract the words from it, we'll start with discussing how to read a PDF file.

2 Dissecting a PDF file

The PDF file is ubiquitous. It is the de-facto standard for sending documents over the internet. The PDF format was created by Adobe, and bears some resemblance to the older Postscript format - also by Adobe. Postscript was a text-based format, and to a degree, so is the PDF format. However, for reasons of efficiency and reducing the size of the PDF files, binary data has entered the format. The format allows also to append new data to the file, without modifying the previous data in the file: new contents can simply be appended to the file.

The specification of the PDF format has been revised several times, each time adding more features; luckily, the format is made in such way that an application does not need to know all features in order to read a PDF file.

At its core, the PDF Format is simply a series of objects (called 'Indirect Objects') with dictionaries describing the properties of the objects. To find the objects you need, a cross-reference section is appended, with the positions of all the objects in the file, together with a small dictionary to find some key objects. In a simple PDF file, all this can be accomplished with text commands. However, the indirect objects and the cross-reference table can be encoded in binary streams, which can be compressed in various ways, and optionally encoded as well - a mechanism called 'Filters' in PDF parlance.

If content is appended to the file, the newly appended content needs its own cross-reference secton which must - obviously - refer to the previous cross-references section for existing content. This process can be repeated ad infinitum: every time when you open a file and add some annotations to a file, the annotations will be appended to the existing file using this mechanism. (this obviously depends on the software, it may decide to completely rewrite the file as well...)

All this makes reading a PDF file not an easy task: various forms of decompression must be supported, optionally encryption.

When reading a PDF file, you will be confronted with various kinds of objects in the file. These objects have been defined in a unit called `fppdfobjects`

TPDFXRef a cross reference to an indirect reference. This is used to refer to indirect objects, and is used in many places: for example, a page list is simply a list of `TPDFXRef` instances referring to the underlying pages (or another page list object)

TPDFDictionary This object encapsulates a PDF dictionary.

TPDFIndirect This is the base building block of the PDF file: an indirect object. It has a `ObjectID`, a number which uniquely identifies this object. By itself not very interesting, but each type of object in the PDF file is a descendant of this class. every `TPDFIndirect` object normally has a dictionary associated with it (available in the `ObjectDict` property).

TPDFDocumentInfo This object contains some meta information about the document: author, title etc. It is basically a dictionary object.

TPDFFontObject This object describes a font. The font may or may not be embedded in the file, usually in compressed form.

TPDFPageObject This object describes a page in the PDF document. It contains a list of resources needed to draw the page: fonts and images. It also contains the content stream: the content stream (which can be spread over various underlying streams) is the set of commands needed to draw the page. The `CommandList` property of the page object contains the actual commands that make up the page.

TPDFPagesObject This object describes a list of pages in the PDF document: Pages are organized in a tree, called the 'Page Tree'.

TPDFCommand This object represents a drawing command: a page is made up of commands that - all taken together - completely render the page.

TPDFCMap this object maps character codes to glyphs in a font. Or to unicode characters, as we will see below.

TPDFCMapData this class contains the actual data of a CMap object.

All these objects are owned by a `TPDFDocument` class, which represents the complete PDF document. It has the following properties:

PDFversion The PDF version in the header line

StartXRef The root of the cross-reference table.

TrailerDict Trailer dictionary, set during parsing

PageCount Page count in this document

PageNodes Indexed access to top-level indirect objects that represents a page tree node. 0 based. This can be another pages node or a page object

Page indexed access to a page by 0-based page number.

Pages an enumerator for the pages

XRefCount Count of elements in `XRefs` property.

XRefs Indexed access to all global XRefs (cross-references) in the PDF.

There are also some methods to find objects in the PDF File:

FindInDirectObject Find an indirect object by object ID.

FindFont Find a font object by object ID. May return Nil.

GetFont Similar to `FindFont` but raises an exception if the object is not found.

FindDocumentInfoObject Find the document information object. May return Nil.

FindDocumentInfo Find document information, resolved to a `TPDFDocumentInfo`, but can be Nil. You must free this object yourself.

GetDocumentInfo // Get the document information object. Raises exception if not found. You must free this object yourself.

FindCatalog Find the document catalog object. Can return Nil.

GetCatalog Similar to `FindCatalog` but raises an exception if not found.

FindPages Find the top-level pages object. Return Nil if none found.

GetPages Similar to `FindPages` but raises an exception if not found.

To read a document from file, a PDF parser is needed. This object will actually read the PDF file. The following is a (shortened) declaration of the PDF parser object:

```
TPDFParser = class
  Constructor Create(aFile : TStream; aBufferSize : Cardinal = PDFDefaultBufferSize); virtual
  Procedure ParseDocument(aDoc : TPDFDocument); virtual;
  Procedure ResolveToUnicodeCMaps(aDoc : TPDFDocument);
  Property LoadObjects : Boolean;
  Property ResolveObjects : Boolean;
  Property ResolveContentStreams : Boolean;
```

```
  Property OnUnknownFilter : TPDFFilterEvent;
  Property OnLog : TPDFLogNotifyEvent;
  Property OnProgress : TPDFProgressEvent;
end;
```

The constructor receives a stream with the contents of the PDF document. The `ParseDocument` method will actually read and parse the PDF Document. The PDF parser can function in several modes:

- it can simply read the PDF document, and make the cross-reference table available.
- Additionally it can also actually read all indirect objects, and make them available as plain `TPDFIndirect` instances.
- if the indirect objects are read it can also convert them to objects of the correct type: a TPDFFontObject for a font, a `TPDFPageObject` for a page, and so on.
- If the objects are converted to typed objects, the parser can also interpret the content stream of a page object, and fill the `CommandList` property of the various `TPDFPageObject` instances.
- If the objects are converted to typed objects, the parser can also interpret the `ToUnicode` CMap streams associated with fonts.

As you can see, each step means additional processing of the PDF data.

The various steps are controlled by some boolean properties:

LoadObjects load all objects when XRef is parsed. Default `True`.

ResolveObjects When loading objects, resolve objects to their actual class. Default `True`.

ResolveContentStreams Resolve content streams of pages to commands. The default is `True`.

ResolveToUnicodeCMaps Immediatly parse font `ToUnicode` CMap streams. Default `False`.

There are some events available:

OnUnknownFilter if an unknown filter is encountered you can manually handle the filter with this event.

OnLog a callback to which log messages are sent.

OnProgress a progress event.

So, to read a PDF File, the following code is sufficient:

```
{$mode objfpc}
{$h+}
uses fppdfobjects, fppdfparser, sysutils, classes;

procedure ReadPDF(const aStream: TStream; aDoc: TPDFDocument);
```

```
var
  aParser : TPDFParser;

begin
  aParser:=TPDFParser.Create(aStream);
  try
    aParser.ResolveToUnicodeCMaps:=True;
    aParser.ParseDocument(aDoc);
  finally
    aParser.Free;
  end;
end;

var
  F : TFileStream;
  Doc : TPDFDocument;

begin
  F:=TFileStream.Create('mydocument.pdf',fmCreate or fmShareDenyNone);
  try
    Doc:=TPDFDocument.Create;
    ReadPDF(F,Doc);
    // Do somethiing with your PDF.
  finally
    doc.free;
    f.free;
  end;
end.
```

Or, even easier: the `fppdfparser` unit has a type helper for the `TPDFDocument` class with `LoadFromFile` and `LoadFromStream` methods:

```
{$mode objfpc}
{$h+}
uses fppdfobjects, fppdfparser;

var
  Doc : TPDFDocument;

begin
  Doc:=TPDFDocument.Create;
  try
    Doc.LoadFromFile('mydocument.pdf');
    // Do somethiing with your PDF.
  finally
    doc.free;
  end;
end.
```

Which is about as easy as it gets. The `LoadFromFile` and `LoadFromStream` methods have an overloaded variant with which the various boolean properties of the parser can be set. The `OnLog` and `OnProgress` events can optionally also be specified.

The `pdfdump` example program demonstrates the use of the various objects by allowing you to dump certain information from a PDF file.

3 Extracting text from a PDF file

Now that we're able to read a PDF file, we can start extracting words from it. Alas, this is not so easy as it may sound. PDF is a format suitable for reproducing a page exactly the same under all circumstances: It literally contains instructions to draw the page. a PDF reader program reads the instructions, and executes them, by for instance drawing them on the screen, or sending appropriate commands to the printer.

If text happens to be part of a page, then the PDF File simply has instructions to draw images representing the letters (glyphs) on certain locations on the page. The glyphs are part of a font, and the PDF File essentially says something like 'draw me glyphs numbers 12 23 45 at position (145,175) on the page.'

Traditionally (and simplifying a bit, we ignore codepages for a second), the glyph numbers were simply the byte code of the character they represented: the glyph for the letter 'A' was simply glyph 65 (the ASCII code for the letter A). Thus a text "Hello World" could easily be mapped to glyph numbers in a font, and vice versa.

With the appearance of Unicode encoding of characters, and fonts with only a subset of all characters, this mapping became more difficult. Since the PDF format has no idea of text and simply used glyph numbers, Adobe came with a solution: a font could specify a `ToUnicode` mapping: this was a mapping from glyph numbers to Unicode character numbers. Simply said, this is a mapping which allows an application to map font glyph numbers back to Unicode characters. The mapping is in CMap format, and a Font dictionary in a PDF file can contain an entry towards such a mapping, using the `ToUnicode` key of the dictionary.

This mechanism allows PDF-displaying software soch as Acrobat reader to offer a text-selection mechanism and a search mechanism: when selecting the text, the glyphs that you selected are translated back to letters. When searching, all text-displaying commands are scanned for the glyphs they display, the glyphs are translated to unicode text using the `ToUnicode` map, and this text is then searched.

This is what we need to do as well if we want to index the pdf.

As indicated earlier, the contents of a page exists of a series of drawing commands. All drawing commands are a descendant of `TPDFCommand`. Some of these commands will draw a text (for example, the `Tj` and `TJ` operators), they are all descendants of an abstract `TPDFTextCommand` class, which is defined as follows:

```
TPDFTextCommand = Class(TPDFCommand)
Public
  Function GetFullText(aUnicodeMap : TPDFCMap) : RawByteString;
  Function GetFullText : RawByteString; virtual; abstract;
end;
```

The `GetFullText` function returns all the glyphs for the text to be drawn. For simple fonts, these will map directly to characters and can be used as-is. For complicated fonts, a mapping of character glyph IDs to unicode characters is needed: The `aUnicodeMap` argument to `GetFullText`. Several descendants of this class exist, they implement these 2 methods.

A text drawing command just draws a text using the current font. If we want to know what unicode map we need to use, we also need to check out the font selection command that preceded the text drawing command: it will contain a font name. The font name can be mapped using the page resource dictionary to a font object (an indirect object). If that font has a `ToUnicode` map associated with it, this

will point to the unicode map to use for all text drawing commands that follow the font selection command. The font selection operator (`Tf`) is represented by the `TPDFTfCommand` class, defined as follows:

```
  TPDFTfCommand = class(TPDFCommand)
  property FontName : String Read GetFontName;
  Property FontSize : Integer Read GetFontSize;
end;
```

Armed with these classes, we can now implement an algorithm to extract the text from a PDF page. To show this algorithm, we'll create a small demo that loads a PDF and extracts the text of a page and shows it in a memo.

For this, we design a form with 5 elements on it:

FEPDF A filename edit to select a PDF file.

edtPageNo An edit to select a page number.

lblPageCount A label to show the page count of the loaded file.

btnShow a button to start the action: load a file and extract the text of the selected page.

mText a memo to show the text of the selected page.

The code for this application is quite simple. The process is started with the `OnClick` event handler of `btnSelect`: If the PDF file was not yet loaded, or a different file was chosen, first the file is loaded using `LoadPDF`:

```
procedure TMainForm.btnShowClick(Sender: TObject);

Var
  aPage : Integer;

begin
  if (FDoc=Nil) or (FEPDF.FileName<>FloadedFile) then
    LoadPDF(FEPDF.FileName);
  aPage:=StrToIntDef(edtPageNo.Text,0);
  if (aPage>0) and (aPage<FDoc.PageCount) then
    ShowPageText(aPage)
  else
    ShowMessage('Invalid page, valid values 1-'+IntTostr(FDOc.PageCount));
end;
```

When the file is loaded, the page number is examined, and if it is valid, the page text is shown using `ShowPageText`. If it is invalid, an error message is shown.

The `LoadPDF` is not substantially different from what we've shown earlier:

```
procedure TMainForm.LoadPDF(const aFileName : string);

begin
  FreeAndNil(FDoc);
  FDoc:=TPDFDocument.Create;
  Try
    FDoc.LoadFromFile(aFilename);
```

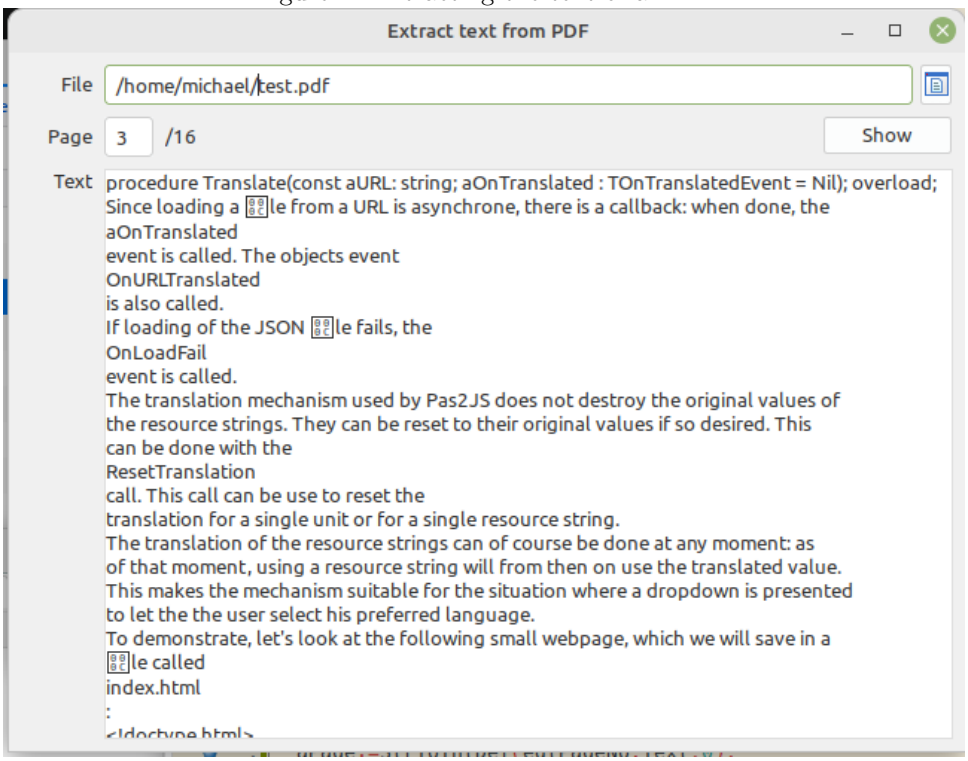
```
    FLoadedFile:=aFileName;
    lblPageCount.Caption:='/'+IntToStr(FDoc.PageCount);
  except
    on E : Exception do
      ShowMessage('Error loading PDF file :'+E.Message);
  end;
end;
```

The `ShowPageText` is where the real work happens. It is in fact a simple loop, with 2 cases: When a font selection command is encountered, the CMAP for the font's `ToUnicode` mapping is extracted and saved.

```
procedure TMainForm.ShowPageText(aPageNo : Integer);

Var
  aPage : TPDFPageObject;
  Cmd : TPDFCommand;
  FontName,Rawtext : RawByteString;
  aFontRef : TPDFRefData;
  UnicodeMap : TPDFCMap;
  aFontObj : TPDFFontObject;

begin
  mText.Clear;
  aPage:=FDoc.Page[aPageNo];
  for Cmd in aPage.CommandList do
     begin
     if Cmd is TPDFTfCommand then
       begin
       FontName:=TPDFTfCommand(Cmd).FontName;
       if (FontName<>'') and (FontName[1]='/') then
         Delete(FontName,1,1);
       aFontRef:=aPage.FindFontRef(FontName);
       aFontObj:=FDoc.FindFont(aFontRef); // TPDFFontObject
       if Assigned(aFontObj) then
         UnicodeMap:=aFontObj.UnicodeCMap
       else
         UnicodeMap:=nil;
       end
     else If cmd is TPDFTextCommand then
      begin
      rawText:=TPDFTextCommand(Cmd).GetFullText(UnicodeMap);
      SetCodePage(RawText,CP_UTF8);
      mText.Lines.Add(RawText);
      end;
    end;
end;
```

When a text drawing command is encountered, the `GetFullText` method is used to extract the full text. If there was an active unicodemap, then that is used to interpret the text. If there was no unicodemap, then the font is assumed to be a simple font and the raw text should be readable (this is not 100% correct, as we'll see) The codepage of the text is set to UTF8 because that is what the Lazarus LCL framework expects, and then the text is added to the memo. The result can be seen

Figure 1: Extracting the text of a PDF



in figure figure 1 on page 9. There are 2 things to note in the screenshot:

- The page text contains some characters that cannot be displayed. This is the letter combination `fi` for which a special glyph is used, and which cannot be displayed in the memo.
- The text contains linebreaks at unexpected places. This is a consequence of the simple mechanism we used to extract the text.

Whenever a font change happens in the displayed text (another font, or simply the same font printed bold or italic), the PDF will contain a paint instruction for each part of the text: the text leading up to the font change, the text in the alternative font, and the rest of the text, which can be in the original font or yet another font. Note that there is no guarantee that the instructions will appear in the order that the words appear in the text: the layouting mechanism could decide to first draw all words in one font, then words in another font, thus saving on font selection commands.

In order to reconstruct actual lines or paragraphs of text, the positioning of the text would also need to be examined: all text drawing commands that are on the same baseline can be expected to form a single line of text, although there is no guarantee that this is actually so.

For the purposes of extracting words from a text, the above mechanism is sufficient: disregarding words for which a different font was selected halfway a word, or hyphenized words, a text drawing command will contain complete words.

4 A class to index words

Free Pascal comes with a unit called `fpIndexer`. This contains some abstract classes which serve as the base class for a word indexing and searching mechanism.

```
TFPIndexer = class(TComponent)
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  function IndexStream(const AURL: UTF8String;
                       ADateTime: TDateTime;
                       S: TStream;
                       Reader: TCustomFileReader): int64;
  function IndexFile(AURL: UTF8String; AllowErrors: boolean;
                     const ALanguage: UTF8String = ''): int64;
  function Execute(AllowErrors: boolean): int64;
  property ErrorCount: int64 read FErrorCount;
published
  property Language: UTF8String;
  property OnProgress: TIndexProgressEvent;
  property UseIgnoreList: boolean;
  property IgnoreNumeric: boolean;
  property CommitFiles: boolean;
  property Database: TCustomIndexDB;
  property ExcludeFileMask: UTF8String;
  property FileMask: UTF8String;
  property SearchPath: UTF8String;
  property SearchRecursive: boolean;
  property DetectLanguage: boolean;
  Property CodePage : TSystemCodePage;
  Property StripPath : String;
end;
```

This class can be used to index the contents of files: the `IndexStream` and `IndexFile` methods will create a database of words from the contents of a text file. The various properties allow to control the search for files on disk and how to index them. The meaning of most of these properties is clear: only 'CommitFiles' needs explanation: the default behaviour is to enter all data in one bug trancaction in the database. When this property is set to `True`, a commit will be done after each indexed file.

The indexer uses a factory to create a tokenizer per file type (determined by its extension): the tokenizer will return all the words in a file for a certain file type. You can register a handler for a file type, and this handler will then be used to read the words and fill the database. The thing to do would be to create a handler for PDF files.

For educational reasons, we'll take a slightly different approach here.

The interesting property is the `Database` property: this determines where the indexer writes the words and matches. The `TCustomIndexDB` class is defined as follows:

```
TCustomIndexDB = class(TComponent)
public
  procedure CreateDB; virtual; abstract;
  procedure Connect; virtual; abstract;
```

```
  procedure Disconnect; virtual;
  procedure CompactDB; virtual; abstract;
  procedure BeginTrans; virtual; abstract;
  procedure CommitTrans; virtual; abstract;
  procedure DeleteWordsFromFile(URL: UTF8String); virtual; abstract;
  procedure AddSearchData(ASearchData: TSearchWordData); virtual; abstract;
  procedure FindSearchData(SearchWord: TWordParser;
                           FPSearch: TFPSearch;
                           SearchOptions: TSearchOptions); virtual; abstract;
  Function GetAvailableWords(out aList : TUTF8StringArray;
                             aContaining : UTF8String;
                             Partial : TAvailableMatch) : integer;virtual; abstract;
  procedure CreateIndexerTables; virtual; abstract;
end;
```

Most of these methods are self-explaining.

CreateDB Create the index database.

CreateIndexerTables Create the tables needed by the indexer.

Connect Connect to the index database.

Disconnect Disconnect from the index database.

CompactDB Clean up the DB.

BeginTrans Start a transaction (if available in the backend).

CommitTrans Commit a transaction (if available in the backend).

DeleteWordsFromFile Delete all word matches from a file.

AddSearchData Add a word match.

FindSearchData find word matches.

GetAvailableWords Get a list of available words matching a pattern.

The FPIndexer package contains several descendants of this class:

TPGIndexDB Is a descendant that writes the data to a PostgreSQL database. Implemented in the unit `pgindexdb`

TFBIndexDB Is a descendant that writes the data to a Firebird/Interbase database. Implemented in the unit `fbindexdb`

TSQLiteIndexDB Is a descendant that writes the data to a SQLite database. Implemented in the unit `sqliteindexdb`

TMemIndexDB Is a descendant that keeps the data in memory. Implemented in the unit `memindexdb`

TFileIndexDB Is a descendant that keeps the data in memory, but can additionally save the data to a file. Implemented in the unit `memindexdb`

Rather than create a file type handler for PDF files, we can use one of the descendants of the `TCustomIndexDB` class directly to fill a database with search words.

We'll do this in a class `TPDFindexer` (implemented in the unit `fppdfindexer`) with the following interface, it can be used to index a single PDF file.

```
TPDFIndexer = Class(TComponent)
Public
  Procedure Connect;
  Procedure CreateTables;
  Procedure Disconnect;
  Procedure IndexPDF(const aFileName : string); overload;
  Procedure IndexPDF(const aStream : TStream); overload;
  Property PDFURL : String;
  Property Language : String;
  Property Indexer : TSQLDBIndexDB;
  Property MinWordLength : Integer;
  Property IgnoreWords : TStrings;
  Property OnLog : TPDFIndexLogEvent;
end;
```

The `Connect`, `CreateTables` and `Disconnect` methods simply call their counterparts in the `Indexer`, which is a descendant of `TCustomIndexDB` that writes to a postgres or firebird database. Which of these three databases is used depends on a define in the beginning of the file:

```
{ $DEFINE USEFIREBIRD}
{$DEFINE USEPG}
```

Since the class only indexes a single document, the URL to write in the database can be specified in the `PDFURL` property. It will be set to the PDF filename if nothing was set. The `Language` property has a similar purpose: it determines the language written into the database.

The `IgnoreWords` strings property allows to provide a list of words to ignore, and the `MinWordLength` is set by default to 3, meaning that words with length less than 3 will not be inserted in the database. The log event serves to log progress.

The main method is `IndexPDF`, which starts the whole indexing process. The method accepts a filename or a stream. It is simple enough:

```
procedure TPDFIndexer.IndexPDF(const aStream: TStream);

begin
  DoLog('Start indexing PDF %s',[PDFURL]);
  Connect;
  try
    DoIndexPDF(aStream);
    DoLog('Done indexing PDF %s',[PDFURL]);
  finally
    Disconnect;
  end;
end;
```

The protected `DoIndexPDF` reads the PDF file and indexes the words by calling the `IndexPDFPage` for each page:

```
procedure TPDFIndexer.DoIndexPDF(const aStream: TStream);

var
  aPageNo : integer;
```

```
  aPage : TPDFPageObject;
  aParser : TPDFParser;
  Doc : TPDFDocument;

begin
  aPageNo:=0;
  Doc:=Nil;
  aParser:=TPDFParser.Create(aStream);
  try
    Doc:=TPDFDocument.Create();
    aParser.ResolveToUnicodeCMaps:=True;
    aParser.ParseDocument(Doc);
    For aPage in Doc.Pages do
      begin
      Inc(aPageNo);
      IndexPDFPage(aPage,aPageNo);
      end;
  finally
    doc.free;
    aParser.Free;
  end;
end;
```

The `IndexPDFPage` simply makes sure that each page is handled in a separate transaction, the real work is done in the `DoIndexPDFPage` method:

```
procedure TPDFIndexer.IndexPDFPage(const aPage: TPDFPageObject; aPageNo: Integer);

begin
  Indexer.BeginTrans;
  try
    DoIndexPDFPage(aPage,aPageNo);
    Indexer.CommitTrans;
    DoLog('Indexed page %d',[aPageNo])
  except
    On E : exception do
      DoLog('Error %s while indexing page %d: %s',[E.ClassName,aPageNo,E.Message]);
  end;
end;
```

The `DoIndexPDFPage` method works similar to what our PDF page text display program did, it scans the commands of the page for texts. Every found text is split into words, which are then saved in the database:

```
procedure TPDFIndexer.DoIndexPDFPage(const aPage: TPDFPageObject; aPageNo: Integer);

Var
  aData: TSearchWordData;
  aCmd : TPDFCommand;
  aWord,FontName,Rawtext : RawByteString;
  aFontRef : TPDFRefData;
  UnicodeMap : TPDFCMap;
  aFontObj : TPDFFontObject;
  aDoc : TPDFDocument;
```

```
begin
  aDoc:=aPage.Document;
  With aData do
    begin
    FileDate:=Date;
    Language:='EN';
    Position:=aPageNo;
    URL:=PDFURL;
    end;
  for aCmd in aPage.CommandList do
    begin
    if aCmd is TPDFTfCommand then
      begin
      FontName:=TPDFTfCommand(aCmd).FontName;
      if (FontName<>'') and (FontName[1]='/') then
        Delete(FontName,1,1);
      aFontRef:=aPage.FindFontRef(FontName);
      aFontObj:=aDoc.FindFont(aFontRef); // TPDFFontObject
      if Assigned(aFontObj) then
        UnicodeMap:=aFontObj.UnicodeCMap
      else
        UnicodeMap:=nil;
      end
    else If aCmd is TPDFTextCommand then
      begin
      rawText:=TPDFTextCommand(aCmd).GetFullText(UnicodeMap);
      aData.Context:=Rawtext;
      SetCodePage(RawText,CP_UTF8);
      for aWord in DoSplit(RawText) do
        begin
        aData.SearchWord:=aWord;
        Indexer.AddSearchData(aData);
        end;
      end;
    end;
end;
```

The `TSearchWordData` record used to insert data in the index database is defined in the `fpIndexer` unit, and is used by the indexer class both for inserting data as when returning results when searching for data:

```
TSearchWordData = record
  Context: UTF8String;
  FileDate: TDateTime;
  Language: UTF8String;
  Position: int64;
  Rank: integer;
  SearchWord: UTF8String;
  URL: UTF8String;
end;
```

The meaning of these fields are:

Context some context around the word: usually the line on which the word ap-

pears.

FileDate The file date of the file in which the word appeared.

Language A 2-letter language code.

Position A position in the file. In our application, we will insert the page number here.

Rank an integer signifying the "rank": this can be used to score the word.

SearchWord The word actually to be inserted. Will be returned on search.

URL the file in which the word was found. This will be the name of the PDF file.

The `DoSplit` method used to split the text into words deserves some attention. It is a virtual method, so it can be overridden to implement a different mechanism.

The `DoSplit` method as written here only accepts the latin letters 'A'..'Z' as words: if a different language (cyrillic or an eastern language) is needed, then this method needs to be overridden.

```
function TPDFIndexer.DoSplit(const aText: RawbyteString): TStringDynArray;

Var
  aCount : Integer;
  c : AnsiChar;
  Cleaned : String;

  Procedure MaybeAdd;

  begin
    if (Cleaned<>'') and AllowedWord(Cleaned) then
      begin
      Result[aCount]:=Cleaned;
      inc(aCount);
      end;
    Cleaned:='';
  end;

begin
  aCount:=0;
  Cleaned:='';
  Result:=[];
  SetLength(Result,Length(aText) div MinWordLength);
  for C in aText do
    if Upcase(C) in ['A'..'Z'] then
      Cleaned:=Cleaned+C
    else
      MaybeAdd;
  MaybeAdd;
  SetLength(Result,aCount);
end;
```

The method is not very difficult to understand. Note that here the check on allowed words is performed: if a word is not allowed, it is not returned. The `AllowedWord` is quite simple:

```
function TPDFIndexer.AllowedWord(aWord : String) : Boolean;

begin
  Result:=Length(aWord)>MinWordLength;
  if Result then
    Result:=FHash.Find(aWord)=Nil;
end;
```

The `FHash` variable is a hash list built from the list of words in the `IgnoreWords` property.

With this, the indexer class is ready. All that needs to be done is to use it in a program!

5 The PDF indexer program

The indexer program is a simple program with inputs for the `TPDFIndexer` class. It allows to index a single file or multiple files: it can scan a directory for PDF files. It also has a button to create the needed tables for the indexer, and a memo to show log output and progress messages.

The form is depicted in figure 2 on page 19. Usage is as follows:

- Enter the database connection info. The database for the index must already exist somewhere, and you must supply the location plus the user credentials to connect to it.

  Reminder: The `fppdfindexer` unit is by default set up to use Postgres database. If you wish to use another database (firebird), you must change the defines in the start of the fppdfindex unit.

- Press the 'Test connection button'. You will get a message if the connection failed or succeeded. The 'Create tables' and 'Index PDFs' buttons will become active.

- If you did not yet create the index tables, use the 'Create tables' to create the tables.

- Set the other parameters: a file with words to ignore, the language code.

- Select a PDF file or a directory with PDF files to index.

- Press the 'Index PDFs' button to start the indexing process.

When the form is created, it creates an instance of the `TPDFIndexer` class:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FIndexer:=TPDFIndexer.Create(Self);
  FIndexer.OnLog:=@DoIndexLog;
  if FileExists(SIniFile) then
    ConfigDB(SIniFile)
end;
```

If a config file (`SIniFIle`) exists, it is read and used to restore the contents of the various inputs: this way the user does not need to fill in this data every time. This

is simply reading the contents of an .ini file, and the interested reader can consult the sources of the sample program.

The operation of the program is quite simple: All connection data must be filled in. Using the `Test Connection` button, the connection can be tested. When the test is successful, the database tables can be made, or, if the tables are known to exist, the index button can be used to start indexing files.

The test connection button executes the `TestConnection` method:

```
procedure TMainForm.TestConnection;

begin
  ConfigConnection;
  Try
    FIndexer.Indexer.Connect;
    FIndexer.Indexer.Disconnect;
    FCanConnect:=True;
    SaveDBConfig(SIniFile);
  except
    on E : Exception do
      ShowMessage('Could not connect to index database:'#10+E.Message);
  end;
end;
```

The configconnection method simply transfers the data entered in the edits to the various properties of the indexer:

```
procedure TMainForm.ConfigConnection;
begin
    With Findexer.Indexer do
      begin
      DatabasePath:=edtDatabaseName.Text;
      Hostname:=edtHostname.Text;
      UserName:=edtUserName.Text;
      Password:=edtPassword.Text;
      end;
end;
```

The main method of this program is the `actIndexExecute` method. This the `OnExecute` event handler of the `actIndex` action.

It starts by setting the language property and reading the list of words to ignore. After that it either indexes the requested file using `IndexFile`, or calls `IndexDirectory`: a routine that retrieves all PDF file names from the selected directory, and calls `IndexFile` using the found names.

```
procedure TMainForm.actIndexExecute(Sender: TObject);

Var
  Msg : String;

begin
  if (FEIgnoreWords.FileName<>'') and FileExists(FEIgnoreWords.FileName) then
    FIndexer.IgnoreWords.LoadFromFile(FEIgnoreWords.FileName);
  FIndexer.Language:=edtLanguage.Text;
```

```
  FFileCount:=0;
  FLastError:='';
  if RBFile.Checked then
    IndexFile(FEPDF.FileName)
  else
    IndexDirectory(DEPDFs.Directory);
  Msg:=Format('Done indexing %d files.',[FFileCount]);
  if FlastError<>'' then
    Msg:=Msg+#10'There were errors:'#10+FLastError;
  ShowMessage(Msg)
end;
```

At the end of the routine, some statistics are shown. The `IndexFile` is simplicity itself. The main logic consist of catching possible errors and displaying them in the output log.

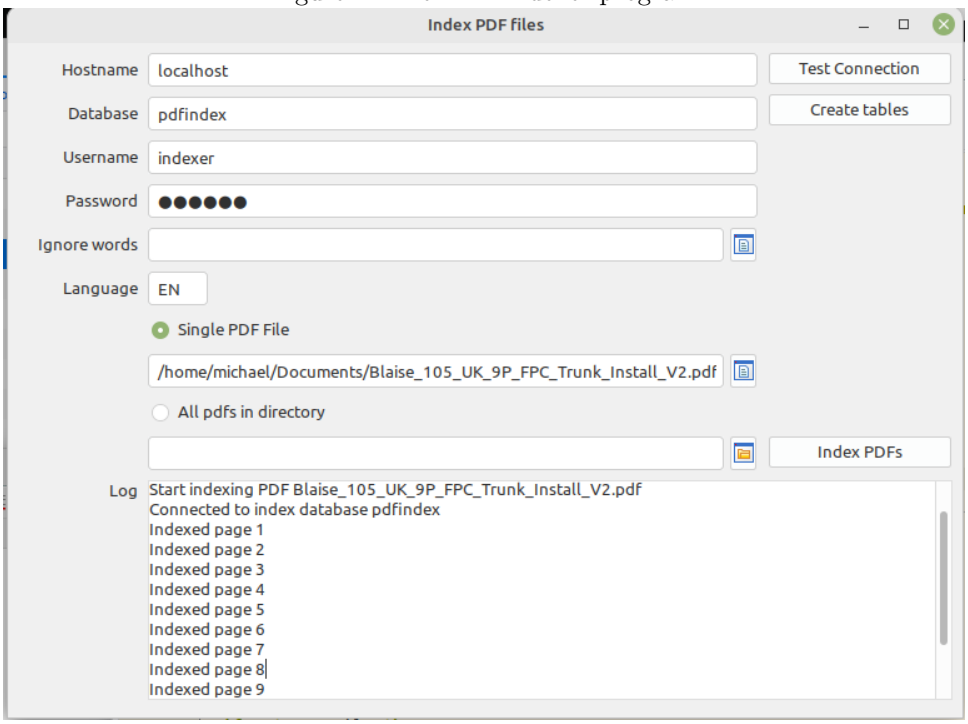
```
  procedure TMainForm.IndexFile(const aFile : string);

Const
  ErrMsg = 'Error %s while indexing %s: %s';

begin
  try
    FIndexer.Connect;
    try
      Inc(FFileCount);
      FIndexer.IndexPDF(aFile);
    finally
      Findexer.Disconnect;
    end;
  except
    On E : Exception do
      begin
      FLastError:=Format(ErrMsg,[E.ClassName,aFile,E.Message]);
      DoLog(FLastError);
      end;
  end;
  Application.ProcessMessages;
end;
```

These are the main methods of the program. The indexer is shown in figure 2 on page 19.

6 Conclusion

In this article, we constructed a PDF indexing program using units provided with Free Pascal and Lazarus. Along the way, we touched upon many subjects: the structure of a PDF file, the many objects needed to extract text from a PDF file. We also showed the classes made available by Free Pascal to create an indexing database. This database is now ready to be used in a searching program: this can be a native program or a pas2js program. That is the subject of a second article. Note that the indexer created by Free Pascal is pretty straightforward and the design is limited: only full words can be searched. For more powerful searching,

Figure 2: The PDF indexer program



a full-text indexer such as manticore search must be used. We'll investigate that in a separate contribution.