Converting Pascal To HTML

by Marco Cantú

While I was writing my latest book, *Mastering Delphi 3*, I thought it would be a nice idea to convert the source code files to HTML format, to let readers browse through them. Having 300 code examples on the CD-ROM makes it a little difficult to find what you are looking for! For this purpose I built a number of tools (using Delphi, of course):

- A Pascal to HTML converter, called PasToWeb, capable of reproducing the standard colour syntax highlighting of the Delphi editor. This is basically a parser which can analyse the source code and recognise keywords and comments.
- A tool to convert an entire project to HTML. This is basically a Wizard which can run the converter on any of the source files making up a project, converting it into HTML and adding a simple index of the files at the beginning of the HTML listing (in fact I decided to have only one HTML file per project).
- A tool to build a complete cross reference of all the identifiers (properties, methods, functions, procedures, objects, variables, fields, constants...) appearing in any of the source code files. This tool is another parser, which identifies all the elements of the source code which are not included in strings or comments.

In this article I'll focus only on the first two of these tools. *[I'm trying to twist Marco's arm to write a second article on the cross referencing tool! Editor]*

Building A Parser

The first thought I had was that, of course, I have to build a parser. After thinking about it, I decided I was too lazy to start from scratch. I knew about the undocumented `TParser` class inside the VCL, so I started exploring it. Unfortunately I found that this class is a little inadequate, but you can easily add capabilities to it. This is the approach I used. After a first rough version I decided to inherit a class from `TParser`, making the parser more powerful and flexible, and adding to it a virtual interface. The next step was to inherit from my class the code to handle the HTML generation, using the virtual interface. But let me go step by step.

The TParser Class

The starting point is the undocumented `TParser` class, defined in the `Classes` unit. I've found many simple examples demonstrating its usage, but it took me a while to realise how it actually works. In fact its name is totally misleading. `TParser` can be better described as a lexical tokeniser. It can divide a source file into its basic elements: symbols (or language identifiers, including keywords), strings, integers, floating point numbers, other characters and the end of file character.

The type of the token is indicated by the `Token` property of the class, which can assume one of the five values shown in Listing 1, or the value of an actual character (one of the language symbols and punctuation marks).

Besides the `Token` property, `TParser` has a `Create` constructor accepting the source stream as a parameter, a `NextToken` function you can use to move to the next token in the source file, a function and a property returning the actual position in the file (`SourcePos` and `SourceLine`) and some conversion functions including `TokenFloat`, `TokenInt` and `TokenString`.

The basic approach in using this class is to write a `while` loop scanning the source code file and to provide a `case` statement within the while loop to analyse the various items. You can see the structure of this code in Listing 2.

As you can see, there are a few big problems. First, comments are not recognised by this class. Second, `TParser` skips all the "white space" characters (spaces, tabs, new line characters and so on). Third, Pascal strings might also include special characters (such as `#13`) so `TParser` merges the special characters inside strings. We need to extract them again.

In the end we are probably doing more work subclassing the `TParser` class than creating a parser from scratch, but this is very interesting anyway and it is what I have actually done.

Extending TParser

My first step has been the definition of an abstract `TCodeParser` class, which has absolutely no knowledge of HTML. This class has

➤ *Listing 1*

```
toEOF     = Char(0);
toSymbol  = Char(1);
toString  = Char(2);
toInteger = Char(3);
toFloat   = Char(4);
```

➤ *Listing 2*

```
Parse := TParser.Create(SourceStream);
while Parse.Token <> toEOF do begin
  case Parse.Token of
    // main tokens
    toSymbol: ... // identifier or keyword (or comment)
    toString: ... // string (including special characters)
    toInteger: .. // numeric constant
    toFloat: ... // floating point constant
    // other tokens
    '{':
    '}':
    '/':
    ...
  end; // case
  // move to the next token
  Parse.NextToken;
end; // while
```

a special constructor, the main Convert procedure, a few local data fields used in different methods, plus a list of virtual methods. The class definition is in Listing 3.

You might be wondering why I've added all those virtual abstract methods and a few other virtual methods. The idea is simple: I want this to be a generic extension of the TParser class which I can extend to produce HTML files, RTF files, or files in any other format I'm interested in. I want to write a single engine (the Convert method) and let it call the virtual methods which can be implemented by subclasses. Here is an excerpt of the code used when a keyword is found:

```
BeforeKeyword;
AppendStr(OutStr, TokenString);
AfterKeyword;
```

The first and last line call two virtual methods, which can be defined to add to the output string (OutStr) specific elements, such as the HTML <B> and </B> commands to start and terminate bold type. Listing 4 shows the code of the two methods for the THtmlParser subclass.

The core of this class is the Convert method, which has the structure indicated in Listing 2. The code of this method is actually quite long (about 150 lines) so you won't find a complete listing here, but only some noteworthy excerpts. All the code is on the companion disk of course. The entire conversion code works on an output string (OutStr). At the end this string is added to the output stream (Dest) by simply copying its actual data:

```
Dest.WriteBuffer(
  Pointer(OutStr)^,
  Length(OutStr));
```

Code Formatting

To handle code formatting properly I've added variables to store the current line and current position of the output file, called Line and Pos. Simply comparing these two values with the information of the input file (SourceLine and SourcePos) you can find out if you need to add extra white spaces or new lines. For example this code adds white spaces to the output string, to match the position in the input file:

```
while SourcePos > Pos do begin
  AppendStr(OutStr, ' ');
  Inc(Pos);
end;
```

Keywords Are Simple

Detecting keywords is actually one of the simplest things you can do. I prepared two string lists with all the Pascal keywords and the two keywords used by DFM files, object and end. When the program finds a symbol, it checks if the symbol is one of the keywords of the list, as shown in Listing 5.

Strings Are Complex

Handling keywords was simple, but I found handling strings to be a total nightmare! The problem is that the TParser class elaborates strings, doing operations you have to undo to make the code work properly. This is the reason you might want to get rid of this base class and write the entire code from scratch.

Basically a string can also be a single character, even one of those constant characters below ascii 32 which are generally added to a source code file using the # symbol, such as #13 for a new line. These special characters can also be the first or last character of a string, or other intermediate characters, and we must recognise them. For multi-character strings this operation is performed by the MakeStringLegal virtual function. Listing 6 shows the main code.

The details are quite complex, so I'm skipping them. Simply keep in mind that there is an extra check

➤ *Listing 3*

```
type
  TCodeParser = class(TParser)
  public
    constructor Create(SSource, SDest: TStream);
    procedure SetKeywordType(Kt: KeywordType);
    // conversion
    procedure Convert;
  protected
    // virtual methods(mostly virtual abstract)
    procedure BeforeString; virtual; abstract;
    procedure AfterString; virtual; abstract;
    procedure BeforeKeyword; virtual; abstract;
    procedure AfterKeyword; virtual; abstract;
    procedure BeforeComment; virtual; abstract;
    procedure AfterComment; virtual; abstract;
    procedure InitFile; virtual; abstract;
    procedure EndFile; virtual; abstract;
    function CheckSpecialToken(Ch1: char): string; virtual;
    function MakeStringLegal(S: String): string; virtual;
  protected
    Source, Dest: TStream;
    OutStr: string;
    FKeywords: TStrings;
    Line, Pos: Integer;
  end;
```

➤ *Listing 4*

```
procedure THtmlParser.BeforeKeyword;
begin
  AppendStr(OutStr, '<B>');
end;
procedure THtmlParser.AfterKeyword;
begin
  AppendStr(OutStr, '</B>');
end;
```

➤ *Listing 5*

```
toSymbol:
  if (FKeywords.IndexOf(TokenString) < 0) then
    // add the plain token
    AppendStr(OutStr, TokenString)
  else
    // add the keyword (see code above...)
```

required. Every character inside a string (and also outside of a string) must be checked for special cases. For example, you cannot use the < and > characters inside an HTML file. You must convert them to the corresponding HTML codes, as you can see in Listing 7.

Comments Don't Work

Handling comments is even more complex, because the TParser class ignores them. The program must identify comments of different kinds and must examine multiple tokens (actually multiple characters) to understand if a first slash or an open parenthesis is marking the beginning of a comment or not. When the program finds a slash character, it should basically wait until the next token, to determine whether to add the plain character to the output or call the BeforeComment virtual method before doing this.

The solution I used in a first version of this program was actually quite simple. When a slash was found the program produced no output and set the LastSlash variable to True. After the next token was read, if the new token was not another slash, the program simply produced the plain output:

```
if LastSlash and
  (Token <> '/') then begin
  AppendStr(OutStr, '/');
  LastSlash := False;
end;
```

otherwise it added both slashes, but only after the marking the beginning of a comment and turning the LineComm flag to True:

```
if LastSlash then begin
  LineComm := True;
  BeforeComment; // virtual
  AppendStr(OutStr, '//');
end
```

Since the parser doesn't recognise strings, I had to do an extra check to see if a keyword was part of a comment and if a string was part of a comment. This is easy. The problem is that a comment can also contain an invalid string, that is a single quote, as in don't. This is not legal in a Delphi string (you have to write it don''t) and the parser believes there is a string which doesn't terminate before the newline character. This generates a runtime error. How do you fix this? Either you remove all single quotes from comments (which is what I did for the book in just a few minutes) or you modify the code of the TParser class of the VCL.

Changing TParser

As a last resort, I decided to edit the TParser code to make it handle comments directly. This way a comment becomes a single token, as a string, indicated by a new constant:

```
const
  toComment = Char(5);
```

I copied the TParser source to the NewParse unit and renamed the class to TNewParser. Besides the class name the definition of the new class is exactly the same, as are most of its methods. The differences are in the function NextToken, which extracts tokens from the source code. You can see the initial portion of this function in Listing 8. In the code P is the pointer to the current character, while FSourcePtr indicates the beginning of the current token.

To fix the code we can simply add a new branch to the case statement to handle multiline comments. The program keeps scanning the code until it finds a closing brace, as you can see in Listing 9. For single line comments, though, we must use a different approach: we must add the code to the else branch of the case statement, check that we are not at the end of the file, and then look to the following character, (P+1)^.

You might want to edit the code further to also handle (* ... *) comments (not in the current version). I was now able to go back to

➤ *Listing 6*

```
BeforeString;
if (Length(TokenString) = 1) and
    (Ord(TokenString [1]) < 32) then
  AppendStr(OutStr, '#' +
    IntToStr(Ord(TokenString [1])))
else
  AppendStr(OutStr, MakeStringLegal(TokenString));
AfterString;
```

➤ *Listing 7*

```
function THtmlParser.CheckSpecialToken(Ch1: char): string;
begin
  case Ch1 of
    '<': Result := '&lt;';
    '>': Result := '&gt;';
    '&': Result := '&amp;';
    '"': Result := '&quot;';
  else
    Result := Ch1;
  end;
end;
```

➤ *Listing 8*

```
function TNewParser.NextToken: Char;
var
  I: Integer;
  P, S: PChar;
begin
  SkipBlanks;
  P := FSourcePtr;
  FTokenPtr := P;
  case P^ of
    'A'..'Z', 'a'..'z', '_':
      begin
        Inc(P);
        while P^ in ['A'..'Z', 'a'..'z', '0'..'9', '_'] do
          Inc(P);
        Result := toSymbol;
      end;
  ...
```

the original code of the `Convert` method and simplify it a little, since I don't need to handle comments there any more.

THtmlParser Class

From the basic class I've inherited the `THtmlParser` class, with the specific HTML conversion code. We have already seen some of the methods of this class, and you can see its definition in Listing 10.

The two class functions simply generate the initial and final portions of the HTML file, with the title and a default background colour at the beginning and a copyright at the end. What is more interesting is the `AddFileHeader` method, which adds to the HTML file a header with the name of the specific file and defines a bookmark inside the HTML file. This is the trick for the version of the program which allows you to have multiple source code files within a single HTML file and provides local jumps (as we'll see in a while). The core of this method is a `Format` statement with the `NAME` directive which defines the bookmark inside an HTML anchor:

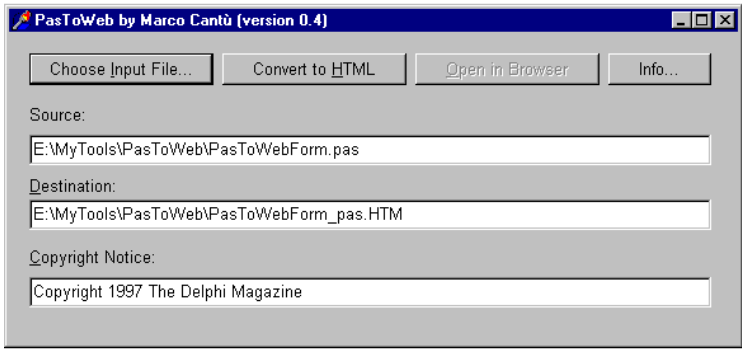
```
Format(
  '<A NAME=%s><H3>%s</H3></A>'
  + #13#10 + #13#10,
  [FName, FName]));
```

The User Interface

All the code we have seen up to now relates to the core of the program, in the `Convert` unit. This unit is then included in a program with a very simple user interface. The main form (see Figure 1) allows a user to operate in three steps.

The first step is the selection of a Pascal or DFM file, though an `OpenDialog` component. Once the file is selected the program suggests the name for a corresponding output file, for example `PasToWebForm_pas.HTM` in Figure 1. The user can modify the suggested name. The second step is the actual file conversion. To accomplish this the program must prepare the proper streams, create an instance of the `THtmlParser` class, initialise the object and call its `Convert` method. Key portions of this method are

➤ *Figure 1*



➤ *Listing 9*

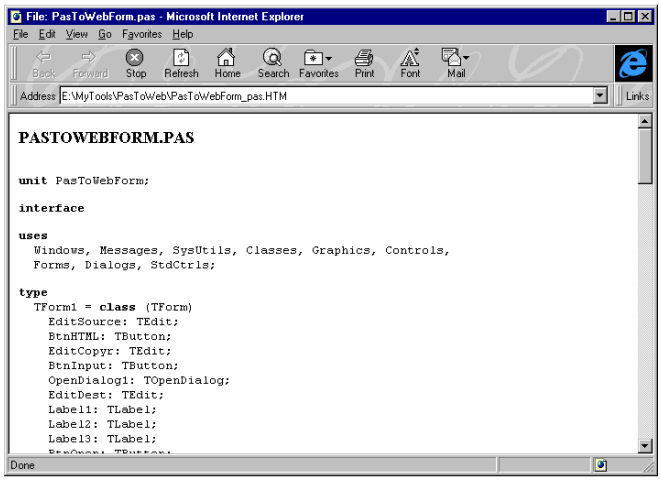
```
  '{':
    begin
      // look for closing brace
      while P^ <> '}' do
        Inc(P);
      // move to the next
      Inc(P);
      Result := toComment;
    end;
else
  if (P^ = '/') and (P^ <> toEOF) and ((P+1)^ = '/') then begin
    // single line comment
    while P^ <> #13 do
      Inc(P);
    Result := toComment;
  end else
    // original code of the else branch
```

➤ *Listing 10*

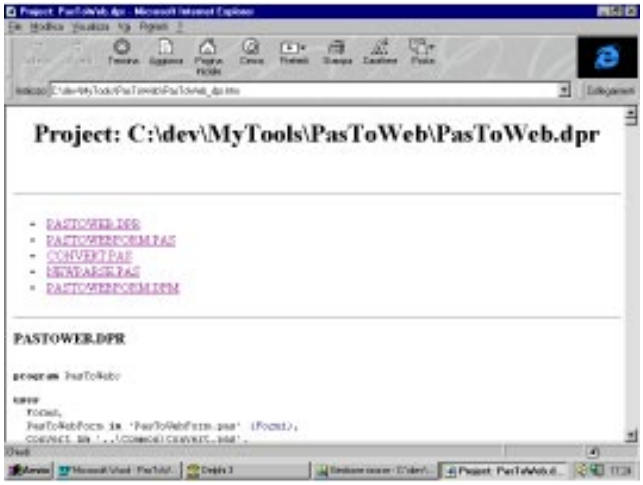
```
type
  THtmlParser = class(TCodeParser)
  public
    FileName: string;
    Copyright: string;
    Alone: Boolean;
    procedure AddFileHeader(FileName: string);
    class function HtmlHead(Filename: string): string;
    class function HtmlTail(Copyright: string): string;
  protected
    procedure BeforeString; override;
    procedure AfterString; override;
    procedure BeforeKeyword; override;
    procedure AfterKeyword; override;
    procedure BeforeComment; override;
    procedure AfterComment; override;
    procedure InitFile; override;
    procedure EndFile; override;
    function CheckSpecialToken(Ch1: char): string; override;
  end;
```

➤ *Listing 11*

```
procedure TForm1.BtnHTMLClick(Sender: TObject);
var
  Source, BinSource, Dest: TStream;
  Parser: THtmlParser;
begin
  // create the two streams
  Dest := TFileStream.Create(EditDest.Text, fmCreate or fmOpenWrite);
  if ExtractFileExt(EditSource.Text) = '.dfm' then begin
    // convert the DFM file to text
    BinSource := TFileStream.Create(EditSource.Text, fmOpenRead);
    Source := TMemoryStream.Create;
    ObjectResourceToText(BinSource, Source);
    Source.Position := 0;
  end else begin
    Source := TFileStream.Create(EditSource.Text, fmOpenRead);
    BinSource := nil;
  end;
  // parse the source code
  Parser := THtmlParser.Create(Source, Dest);
  Parser.Alone := True;
  Parser.Filename := EditSource.Text;
  Parser.Copyright := EditCopyr.Text;
  if ExtractFileExt(EditSource.Text) = '.dfm' then
    Parser.SetKeywordType(ktDfm);
  Parser.Convert;
  // free the objects
  ...
```



➤ *Figure 2*



➤ *Figure 3*

```
function TPrjToWebWizard.GetState: TExpertState;
begin
  if ToolServices.GetProjectName <> '' then
    Result := [esEnabled]
  else
    Result := [];
end;
procedure TPrjToWebWizard.Execute;
var Copyr, ResFile: string;
begin
  Copyr := 'Source code copyright ...';
  if InputQuery('PrjToWeb Wizard', 'Enter Copyright notice:', Copyr) then begin
    ResFile := CurrProjectToHTML(Copyr);
    if MessageDlg('HTML file generated.'#13 +
     'Do you want to open it in your browser?', mtConfirmation,
     [mbYes, mbNo], 0) = idYes then
     ShellExecute(ToolServices.GetParentHandle, 'open', PChar(ResFile),
       '', '', sw_ShowNormal);
  end;
end;
```

➤ *Listing 12*

shown in Listing 11. Notice the code used to convert the binary DFM file into the textual description by calling the `ObjectResourceToText` procedure and the following statement used to reset the stream.

The third step is to press the last button to load the newly generated HTML file in the default browser. The program accomplishes this simply by calling the `ShellExecute` API function, passing the filename and the `Open` command:

```
ShellExecute(Handle, 'open',
  PChar(EditDest.Text), '',
  '', sw_ShowNormal);
```

You can see the effect of pressing this third button (and an example of the source code of the PasToWeb program converted to HTML) in Figure 2.

The Wizard

Having written the main program, which turns a single source code file into its HTML equivalent, I can now show you how to incorporate the conversion code into a Wizard. This needs to be able to convert an entire project. A Delphi Wizard, in fact, can access information such as the structure of a project using the `ToolServices` class. I'm not going to explain how to write a wizard in Delphi *[check articles in Issues 3, 7, 9, 13, 16, 17 and 21 for details. Editor]*, I'll just describe the specific features of this one.

The `GetState` method of this wizard returns the `enabled` style or nothing, depending on whether a project is currently open in the Delphi environment (as returned by the `GetProjectName` method of the `ToolServices` global object). You can see the source code of this method in Listing 12.

The same listing includes also the code of the `Execute` method of the wizard, which simply asks for the copyright string, executes the `CurrProjectToHTML` function and then suggests opening the file in the current HTML browser.

At the heart of this wizard is the `CurrProjectToHTML` function, defined in the `Convert` unit. The function generates an HTML file using the project name and replacing its extension with the string `_DPR.HTM` (this file name is also returned by the function). After a generic HTML header, the function generates a list of the units and forms of the project. Each of the items in this list is actually a link to the name of the file, which is an internal name generated along with the file itself. As a result, the HTML file of the project shows the list of the internal files (with the exclusion of a couple of special files) used as an index. You can see the effect in Figure 3.

The code of the `CurrProjectToHTML` function is quite long, so I've included in Listing 13 only two small excerpts: the code used to generate the list of units and the code used to generate the portions of the HTML file for each unit.

The `PrjToWeb` wizard is the only element of the `PrjWebW` package, available on the companion diskette, along with the entire source code of the examples discussed in this article.

Conclusion

As I mentioned, at the end of this project I was really not sure if using the existing `TParser` class has really been a good choice. Certainly I've had a chance to explore it and realise its strengths and many

```
...
// list of units...
for I := 0 to ToolServices.GetUnitCount - 1 do begin
  Ext := Uppercase(ExtractFileExt(ToolServices.GetUnitName(I)));
  FName := Uppercase(ExtractFilename(ToolServices.GetUnitName(I)));
  if (Ext <> '.RES') and (Ext <> '.DOF') then
    AppendStr (HTML, '<LI> <A HREF=#' + FName + '> ' + FName + '</A>'#13#10);
end;
...
// generate the HTML code for the units
for I := 0 to ToolServices.GetUnitCount - 1 do begin
  Ext := Uppercase(ExtractFileExt(ToolServices.GetUnitName(I)));
  if (Ext <> '.RES') and (Ext <> '.DOF') then begin
    Source := TFileStream.Create(ToolServices.GetUnitName(I), fmOpenRead);
    Parser := THtmlParser.Create(Source, Dest);
    try
      Parser.Alone := False;
      Parser.Filename := ToolServices.GetUnitName(I);
      Parser.Convert;
    finally
      Parser.Free;
      Source.Free;
    end;
  end; // if
end; // for
...
```

➤ *Listing 13*

weaknesses for parsing Pascal files. But I've had to change its original source code, extend it and modify it so much that probably starting with its code and editing it directly would have been a better choice.

After thinking a while about it, I've realised that the `TParser` class works great for the DFM files (which have no comments, for example). This is actually how it is used inside the Delphi environment. After searching through the VCL source code files, I found that the `TParser` class is used only in the `ObjectTextToBinary` procedure, and also in an HTTP related procedure in the client/server version of Delphi 3.

However, I've achieved my two objectives: firstly, to add HTML versions of the source code for all the projects in my book, spending only an evening to write the conversion program, and secondly, to write an article teaching you how to use the undocumented `TParser` class!

Marco Cantú is the author of *Mastering Delphi 3* and *Delphi Developer's Handbook*. Besides writing books he teaches advanced Delphi seminars and speaks at conferences worldwide. He is based in Italy and can be reached as marcocantu@compuserve.com or though his Web site at:
http://www.marcocantu.com