Building A Parsing Component

by Paul Warren

The explosion of web and intranet technologies has resulted in file parsing becoming the most common task I have to do for my clients (after database work that is). I am asked to parse data out of log files, count and manipulate tags in html, read and write comma separated files and, of course, create html on the fly.

Until recently I would write parsing code as required, but clearly, for such a common task, a reusable, flexible parser is needed. You may recall Marco Cantù introduced TParser in Issue 23 and I used TParser in Issue 33. Both Marco and I commented on TParser's utility and deficiencies. Our esteemed Editor, somewhat tongue in cheek, challenged me to create an improved version. Since I had a pressing need for a reusable parser I took him up on the challenge.

Design Considerations

I wanted a parser class that could duplicate the action of TParser (so my old code would still work) but that I could extend to handle the comments in source code. In addition, I needed to create descendant classes that would be capable of other parsing tasks. As a final consideration, conversion to a component might also be desirable.

As is usually the case, I started by studying the code (found in the classes unit). One thing I will say about TParser, it is a lot easier to use than it is to understand.

Stripped to the bare essentials, TParser accepts a stream and pages it into a PChar buffer. Then, using a series of buffer pointers, it reads through the buffer to see if certain conditions are met. If they are, a token is returned indicating which condition has been met. I know this is an oversimplification but I'll go into more detail later.

The grunt work is done in a horrible looking function called NextToken. I noticed that if you remove all the case statements, while True do loops and if..then .. Break statements in NextToken the function reduces to Listing 1: NextToken increments FSourcePtr until it finds toEOF.

It looks like the stripped down TParser could serve as an ancestor for a hierarchy of parsers: precisely what we need. Before going to work, though, we should take a look at TParser in more detail.

Understanding TParser

In the constructor TParser accepts a stream for parsing, allocates memory for FBuffer and sets FBufPtr, FSourcePtr and FSourceEnd to point to FBuffer. FBufEnd is set to point to FBuffer + ParseBufSize. FBuffer[0] is set to #0 to serve as an end of buffer marker and finally NextToken is called once.

All these variables are of type PChar and as such can serve several functions. They can hold data as FBuffer does, they can point to data, they can be treated as operands and they can be de-referenced. Which proves even something as annoying as a PChar can be useful.

In the stripped down version of NextToken the first line is a call to SkipBlanks. SkipBlanks (Listing 2) uses a case statement inside an infinite loop to distinguish between different characters pointed to by the de-referenced FSourcePtr.

When called by the constructor via NextToken the character being pointed to is clearly #0. In this case a call is made to ReadBuffer and if the result is still #0, indicating an empty stream, then we Exit, otherwise we continue with the next iteration of while true do. Characters between #0 and #9 and #11 and #32 are ignored by incrementing FSourcePtr. Future calls to SkipBlanks read data into the buffer as needed.

ReadBuffer (Listing 3) is the stream paging engine. To really understand how ReadBuffer works you need a table of pointer addresses which I have generated using writeln output to a text file. As you can see from Table 1 FBuffer and FBufEnd are constant, note how FBuffer is used as both a data structure and a pointer (as discussed earlier).

➤ *Listing 1*

```
function TParser.NextToken: Char;
begin
  SkipBlanks;
  FTokenPtr := FSourcePtr;
  FToken := FSourcePtr^;
  if FToken <> toEOF then Inc(FSourcePtr);
  Result := FToken;
end;
```

➤ *Listing 2*

```
procedure TParser.SkipBlanks;
begin
  while True do begin
    case FSourcePtr^ of
      #0 :
        begin
          ReadBuffer;
          if FSourcePtr^ = #0 then
            Exit;
          Continue;
        end;
      #10 : Inc(FSourceLine);
      #33..#255 : Exit;
    end;
    Inc(FSourcePtr);
  end;
end;
```

```
procedure TParser.ReadBuffer;
var
  Count: Integer;
begin
  Inc(FOrigin, FSourcePtr - FBuffer); { used for SourcePos }
  FSourceEnd[0] := FSaveChar;
  Count := FBufPtr - FSourcePtr;
  if Count <> 0 then Move(FSourcePtr[0], FBuffer[0], Count);
  FBufPtr := FBuffer + Count;
  Inc(FBufPtr, FStream.Read(FBufPtr[0], FBufEnd - FBufPtr));
  FSourcePtr := FBuffer;
  FSourceEnd := FBufPtr;
  if FSourceEnd = FBufEnd then
  begin
    FSourceEnd := LineStart(FBuffer, FSourceEnd - 1);
    if FSourceEnd = FBuffer then Error(SLineTooLong);
  end;
  FSaveChar := FSourceEnd[0];
  FSourceEnd[0] := #0;
end;
```

➤ *Listing 3*

```
{ TTextParser }
function TTextParser.NextToken: Char;
begin
  SkipBlanks;
  FTokenPtr := FSourcePtr;
  case FSourcePtr^ of
    'A'..'Z', 'a'..'z', '_':
      begin
        Inc(FSourcePtr);
        while True do
          case FSourcePtr^ of
            'A'..'Z', 'a'..'z', '0'..'9', '_': Inc(FSourcePtr);
            '''': begin  { apostrophies }
                if (FSourcePtr+1)^ in ['A'..'Z', 'a'..'z', '0'..'9', '_'] then
                  Inc(FSourcePtr)
                else Break;
              end;
            '-': begin  { hyphenated words }
                if (FSourcePtr+1)^ in ['A'..'Z', 'a'..'z', '0'..'9', '_'] then
                  Inc(FSourcePtr)
                else Break;
              end;
            else Break;
          end;
        FToken := toSymbol;
        Result := FToken;
      end;
    '-', '0'..'9':
      begin
        Inc(FSourcePtr);
        while FSourcePtr^ in ['0'..'9'] do Inc(FSourcePtr);
        FToken := toInteger;
        Result := FToken;
        while FSourcePtr^ in ['0'..'9', '.', 'e', 'E', '+', '-'] do
        begin
          Inc(FSourcePtr);
          FToken := toFloat;
          Result := FToken;
        end;
      end;
    else Result := inherited NextToken;
  end;
end;
```

➤ *Listing 4*

Fbuffer	FBufPtr	FSrcPtr	FSrcEnd	FBufEnd	Pos	Occurred
9194444	9194444	9194444	9194444	9198540	0	On create
9194444	9194444	9194444	9194444	9198540	0	Before read
9194444	9198540	9194444	9194444	9198540	4096	After read
9194444	9194508	9198476	9198476	9198540	4096	Before read
9194444	9198540	9198476	9198476	9198540	8128	After read
9194444	9194454	9198530	9198530	9198540	8128	Before read
9194444	9198282	9198530	9198530	9198540	11956	After read
9194444	9194444	9198282	9198282	9198540	11956	Before read
9194444	9194444	9198282	9198282	9198540	11956	After read

➤ *Table 1*

FBufPtr, FSourcePtr and FSourceEnd initially point to FBuffer, as already mentioned, and remain in this state until after the first call to FStream.Read. After the stream read, FBufPtr points 4096 bytes (ParseBufSize) into the buffer and FStream.Position is 4096 as well. FSourcePtr and FSourceEnd still point to the beginning of the buffer.

Next, SourceEnd is moved to coincide with FBufPtr and is then checked against FBufEnd. If they are equal FSourceEnd is backed up to the last CRLF in the buffer by the function LineStart (from the Classes unit). If there is no CRLF pair a LineTooLong exception is raised.

Finally, the character at FSourceEnd[0] is saved and replaced with a #0 marker. When the next call to ReadBuffer occurs the saved character is replaced and the whole operation starts over. You can follow these pointer movements in the remaining lines of Table 1.

All this probing into TParser's inner workings was necessary to create a custom base class that preserves the critical functionality of TParser but allows us to modify the behaviour in descendant classes. All the source is naturally included on this months disk so I won't reproduce TCustomParser here.

TTextParser

The next logical step is to create a descendant that can parse a text file into individual words. If you're asking why at this point, shame on you: this would be perfect for a spell checker or for counting words in text files.

Only NextToken has to be changed to parse text files into their individual words.

In overriding NextToken you have to realise we are examining characters to see if they meet certain criteria and only if they *don't* are we then going to call inherited. Note that, unlike a procedure, you must set Result := inherited NextToken when you do call inherited.

Listing 4 shows the TTextParser.NextToken function. FTokenPtr is set equal to FSourcePtr so the TokenString function can return the string FSourcePtr-FTokenPtr. The first level of case statements checks to see whether a character is a number or letter and thus of interest. If not it is passed on to inherited. For characters that are of interest we increment FSourcePtr and keep examining characters inside a loop. When we finally hit a character not of interest we set FToken appropriately, break the loop and exit.

TPasParser

TTextParser comes close to duplicating the functionality of Inprise's TParser. It already tokenizes integers and floating point numbers. The handling of symbols is different though, and there is no support for tokenizing Pascal strings.

To correct the behaviour of TTextParser we have to declare TPasParser as a descendant of TTextParser and override NextToken. Since TTextParser already handles integers and floats we can let these pass through to inherited. Unlike TParser, TTextParser can accept single quotes so we can't pass the remaining characters along. If you look at Listing 5 and compare it to TParser.NextToken you'll see the code is the same except for a bug fix and the comments around the case selector.

TPasParser can be used in place of TParser without any changes to your code, except of course the instantiation and including NewParse in the uses clause.

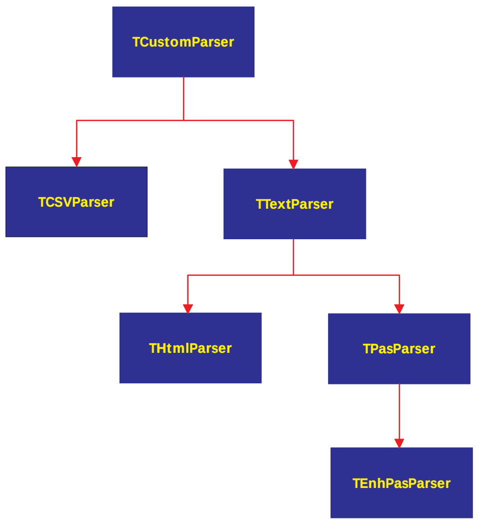
A Component Wrapper

Before I introduce the other parsers shown in Figure 1, I want to create a component to simplify using the parser class. ThgsParser is a component 'wrapper' for a parser class. The most important feature of this rather simple component is the OnParse event, a custom event triggered on every call to NextToken. This avoids the user having to write a code block to use the parser.

The other important feature is the use of a class reference so the desired parser can be created. One thing I don't want is five more components on my palette. I love components but you can have too much of a good thing. Using a class reference ThgsParser can create any parser type descended from TCustomParser. Listing 6 is the complete code for ThgsParser.

As a point of interest notice the SetParseStream method. This is a property handling method normally used to set the property value. In this case setting the property actually executes the parser. While this is unusual it is also necessary.

Since the TCustomParser class takes the stream in its constructor and executes NextToken once, we can't create an instance of a



➤ *Figure 1: Class hierarchy diagram.*

➤ *Listing 5*

```
function TPasParser.NextToken: Char;
var I: integer;
begin
  SkipBlanks;
  FTokenPtr := FSourcePtr;
  case FSourcePtr^ of
    'A'..'Z', 'a'..'z', '_':
      begin
        Inc(FSourcePtr);
        while FSourcePtr^ in
          ['A'..'Z', 'a'..'z', '0'..'9', '_'] do
          Inc(FSourcePtr);
        FToken := toSymbol;
        Result := FToken;
      end;
    '#', '''':
      begin
        FStringPtr := FSourcePtr;
        while True do
          case FSourcePtr^ of
            '#':
              begin
                Inc(FSourcePtr);
                I := 0;
                while FSourcePtr^ in ['0'..'9'] do begin
                  I := I * 10 + (Ord(FSourcePtr^) -
                    Ord('0'));
                  Inc(FSourcePtr);
                end;
                FStringPtr^ := Chr(I);
                Inc(FStringPtr);
              end;
            '''':
              begin
                Inc(FSourcePtr);
                while True do begin
                  case FSourcePtr^ of
                    #0, #10, #13:
                      Error(SInvalidString);
                    '''':
                      begin
                        Inc(FSourcePtr);
                        if FSourcePtr^ <> '''' then Break;
                      end;
                  end;
                  FStringPtr^ := FSourcePtr^;
                  Inc(FStringPtr);
                  Inc(FSourcePtr);
                end;
              end;
          else
            Break;
          end;
        FToken := toString;
        Result := FToken;
      end;
    '$':
      begin
        FToken := FSourcePtr^;  { assume NOT an integer }
        Result := FToken;
        Inc(FSourcePtr);
        while true do begin
          case FSourcePtr^ of
            '0'..'9', 'A'..'F', 'a'..'f': Inc(FSourcePtr);
            else Break;
          end;
          FToken := toInteger;
          Result := FToken;
        end;
      end;
  (*  '-', '0'..'9':
      begin
        Inc(FSourcePtr);
        while FSourcePtr^ in ['0'..'9'] do Inc(FSourcePtr);
        FToken := toInteger;
        Result := FToken;
        while FSourcePtr^ in
          ['0'..'9', '.', 'e', 'E', '+','-'] do begin
          Inc(FSourcePtr);
          FToken := toFloat;
          Result := FToken;
        end;
      end;  *)
    else Result := inherited NextToken;
  end;
end;
```

```
unit Parsecmp;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Controls, NewParse;
type
  TParserType = (ptCSV, ptText, ptPascal, ptEnhPas, ptHtml);
  TOnParseEvent = procedure(Sender: TObject; Token: char) of
    object;
  ThgsParser = class(TComponent)
  private
    ParserRef: TParserClass;
    FParser: TCustomParser;
    FParserType: TParserType;
    FParseStream: TStream;
    FOnParse: TOnParseEvent;
    procedure SetParseStream(Value: TStream);
    procedure SetOnParse(Value: TOnParseEvent);
  protected
  public
    constructor Create(AOwner: TComponent); override;
    property ParseStream: TStream write SetParseStream;
  published
    property ParserType: TParserType read FParserType
      write FParserType default ptHtml;
    property OnParse: TOnParseEvent read FOnParse
      write SetOnParse;
  end;
procedure Register;
implementation
{$IFDEF WIN32}
  {$R PARSECMP.D32}
{$ELSE}
  {$R PARSECMP.D16}
{$ENDIF}
constructor ThgsParser.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FParserType := ptHtml;
end;
procedure ThgsParser.SetParseStream(Value: TStream);
begin
  { assign stream - note must be freed by caller }
  FParseStream := Value;
  case FParserType of { check the parser type }
    { set class reference according to parser type }
    ptCSV: ParserRef := TParserClass(TCSVParser);
    ptText: ParserRef := TParserClass(TTextParser);
    ptPascal: ParserRef := TParserClass(TPasParser);
    ptEnhPas: ParserRef := TParserClass(TEnhPasParser);
    ptHtml: ParserRef := TParserClass(THtmlParser);
  end;
  { create parser instance }
  FParser := ParserRef.Create(FParseStream);
  try
    { parse file }
    while FParser.Token <> toEOF do begin
      FParser.NextToken;
      if Assigned(FOnParse) then
        FOnParse(FParser, FParser.Token);
    end;
  finally
    FParser.Free; { free parser }
  end;
end;
procedure ThgsParser.SetOnParse(Value: TOnParseEvent);
begin
  FOnParse := Value;
end;
procedure Register;
begin
  RegisterComponents('HomeGrown', [ThgsParser]);
end;
end.
```

➤ *Listing 6*

TCustomParser class until we're ready to use it. So without changing the constructor, which I didn't want to do, legacy code and all that, this was the best solution.

Adding More Parsers: TEnhPasParser

TPasParser still suffers from the limitations pointed out in the earlier articles. Much more useful is TEnhPasParser which I have made aware of comments. This is the class you should use if you want to upgrade my packing list utility from Issue 33 or Marco Cantù's PasToWeb utility: see the sidebar for details.

The secret to handling comments is to recognize that they can span multiple lines. All the tokenization we've looked at so far has to occur on one line so SkipBlanks gets called at least once per line. Since ReadBuffer always moves FSouceEnd to a CRLF pair NextToken can never attempt to read beyond the buffer end. When tokenizing comments this changes. Now we have the chance of reading beyond the buffer end and generating an access violation. To avoid this problem we can check for a #0 FSourceEnd marker, call ReadBuffer, and Break if we find one.

Unfortunately this isn't enough since the #0 can also represent the end of a stream. Therefore we need to check FSourcePtr^ again after the call to ReadBuffer. If it is still #0 we can break execution. As you can see in Listing 7 we also have to increment FSourceLine at line feeds and break execution when the comment is closed.

THtmlParser

THtmlParser descends from TTextParser since it doesn't need the extra functionality of the source parsers.

This class returns a token of toOpenTag or toCloseTag when it encounters tags. When not encountering html tags THtmlParser returns toSymbol, toInteger and toFloat the way TTextParser would. When Token is toOpenTag or

Pascal To HTML, Source Cross-Referencing And File Packing List Projects

To use the new parsers in Marco Cantù's Crossref Delphi source cross-referencing project (see Issue 30) and for my own packing list project (see Issue 33), all you need to do is substitute the newparse unit and redeclare TParser as TEnhPasParser.

For Marco's PasToWeb utility (see Issue 23) do the same as above and change the MakeCommentLegal function to the following:

```
function TCodeParser.MakeCommentLegal (S: String): string;
var
  I: Integer;
begin
  Result := '{'; {prepend a brace}
  // for each character of the string
  for I := 1 to Length (S) do
    AppendStr (Result, CheckSpecialToken(S[I]));
  AppendStr (Result, ' }'); {append a brace}
end;
```

You could also change the constructor from Create to CreateNew and call inherited Create if you want to avoid the compiler warning, although not doing so has no adverse effects that I could detect.

```
function TEnhPasParser.NextToken: Char;
begin
  SkipBlanks;
  FTokenPtr := FSourcePtr;
  case FSourcePtr^ of
    '{':
      begin { comment or compiler directive... }
        FStringPtr := FSourcePtr;
        Inc(FSourcePtr);  { check next char... }
        while true do begin
          case FSourcePtr^ of
            #0: begin
              ReadBuffer;
              FStringPtr := FSourcePtr;
              if FSourcePtr^ = #0 then Break;
             {$IFDEF DEBUG}
              writeln(Log, 'in comment');
             {$ENDIF}
            end;
            #10: Inc(FSourceLine);
            '}':
              begin
                Inc(FSourcePtr);
                Break;      { end comment... }
              end;
          end;
          FStringPtr^ := FSourcePtr^;
          Inc(FStringPtr);
          Inc(FSourcePtr);
        end;
        FToken := toComment;
        Result := FToken;
      end;
    '(', '/':  { possible comment or compiler directive... }
      begin
        FStringPtr := FSourcePtr;
        Inc(FSourcePtr);  { check next char... }
        case FSourcePtr^ of
          '*':  { is a comment }
            begin
              Inc(FSourcePtr);  { check next char... }
              while True do begin
                case FSourcePtr^ of
                  #0: begin
                    ReadBuffer;
                    FStringPtr := FSourcePtr;
                    if FSourcePtr^ = #0 then Break;
                   {$IFDEF DEBUG}
                    writeln(Log, 'in comment');
                   {$ENDIF}
                  end;
                  #10: Inc(FSourceLine);
                  '*':
                    begin
                      Inc(FSourcePtr);
                      if FSourcePtr^ = ')' then begin
                        Inc(FSourcePtr);
                        Break; { end of comment }
                      end;
                    end;
                end;
                FStringPtr^ := FSourcePtr^;
                Inc(FStringPtr);
                Inc(FSourcePtr);
              end;
              FToken := toComment;
              Result := FToken;
            end;
          '/':  { is a comment }
            begin
              Inc(FSourcePtr);
              while (FSourcePtr^ <> #10) do begin
                { end of line, hence comment }
                FStringPtr^ := FSourcePtr^;
                Inc(FStringPtr);
                Inc(FSourcePtr);
              end;
              FToken := toComment;
              Result := FToken;
            end;
        end;
      end;
    else
      Result := inherited NextToken;
  end;
end;
```

➤ *Listing 7*

```
procedure TForm1.hgsParser1OnParse(Sender: TObject; Token: Char);
begin
  if (Token = toOpenTag) and ((Sender as THtmlParser).TokenString
    not in ['P','p','BR','br']) then
    inc(counter);
  if Token = toCloseTag then
    dec(counter);
  if counter <> 0 then
    Label1.Caption := IntToStr(counter)+' unbalanced tags';
end;
```

➤ *Listing 8*

toCloseTag the function TokenString returns the tag parameters without the enclosing brackets.

Listing 8 shows the code in an ThgsParser.OnParse event handler which counts open and close tags and reports discrepancies. Note: There are other standalone html tags besides <BR> and <P> but this gives you the general idea for searching out unbalanced tags.

TCSVParser

Finally, TCSVParser descends directly from TCustomParser and extracts the fields from comma separated text files. I use this class to read comma separated log files from an intranet server and report the results via dynamic html.

Conclusion

Though strange in appearance and of limited use Inprise's TParser is a surprisingly useful tool when slightly restructured and wrapped in a component.

I've provided a simple demo along with all the source on this month's free disk. Hopefully you'll find the class hierarchy as useful as I do and if you need additional functionality it should be fairly easy to add.

Paul Warren runs HomeGrown Software Development in Langley, British Columbia, Canada and can be contacted by email at
hg_soft@uniserve.com