Sweet Dreams

A better file search with regular expression matching

by Julian Bucknall

Time for a quick test. For those of you who haven't tried this before, the result is a little disconcerting and a bit of a nightmare. Listing 1 shows some code using the directory search routines `FindFirst`, `FindNext` and `FindClose`.

`FindFirst` sets up a data structure and finds the first file to match the path and attributes. `FindNext` continues the search, getting the remainder of the files, one at a time. `FindClose` clears up the data structure. All three routines are simple enough wrappers around the Windows or Win32 routines.

Analysing the code, we're asking for all subdirectories in the `e:\articles\filedir` directory. Simple enough, no? So what would the result be? All subdirectories in this particular directory, right? *Bzzzzt*, wrong. In actual fact, this code gives you all subdirectories (hooray!) together with all the normal files (huh?). Even in this day and age of 32-bit operating systems with APIs out the wazoo, it still gives you all these normal files as well. So every time you want to use these routines you have to remember to filter out all the files you don't want.

And there's another thing as well. The wildcards. Has it ever bugged you that we're still living with `?` and `*` as the only wildcards we can use with filenames? I mean, really, come off it. In these days of ActiveX and multi-tier applications we can still only specify groups of files by means of the same wildcards we used back in the early 80s with Turbo Pascal.

Enough already. Let me reveal what we'll be developing here in this article. We shall be designing replacements for the three `FindXXX` routines that will filter out the unwanted files according to our requested attributes, and we shall be writing a regular expression parser to enable better wildcard matching. By the end of the article we shall be writing code similar to that shown in Listing 2 which will find all files whose first three letters are `TST`, followed by zero or more digits, followed by `X.LOG`, and the files will have only their archive bit set or no attribute bits set at all. Sweet dreams, indeed. On the way, we shall be touching on parsers, state machines and greedy algorithms.

```
var
  SR : TSearchRec;
  FFRes : integer;
begin
  ...
  FFRes := FindFirst('e:\articles\filedir\*.*', faDirectory, SR);
  while FFRes = 0 do begin
    FileList.Add(SR.Name);
    FFRes := FindNext(SR);
  end;
  FindClose(SR);
  ...
```

➤ *Listing 1*

```
FindFirstEx(SomePath, 'TST[0-9]+X.LOG', [detFile], [deaNormal, deaArchive], SR);
while FFRes = 0 do begin
  FileList.Add(SR.srName);
  FFRes := FindNextEx(SR);
end;
FindCloseEx(SR);
```

➤ *Listing 2*

Let's Go!

First things first, and the easiest. The ultimate method we shall use is to retrieve all entries in the directory with `FindFirst` and `FindNext` and do the filtering ourselves. After all I'm sure we can do a better job ourselves and, as we have already seen, we certainly can't do worse. One of the main problems with the 'raw' routines is the silly bit values that mix types of directory entry (subdirectories, files, volume id) with attributes of the directory entries (hidden, system, readonly, changed). So let's create a couple of enumerated types (`TdeType` and `TdeAttr`) and then we can use sets of these types to pass attributes and entry types to our routines (Listing 3).

```
type
 TdeType = (detFile,
            detDirectory,
            detVolumeID);
  TdeTypeSet = set of TdeType;
  TdeAttr = (deaNormal,
             deaAltered,
             deaReadOnly,
             deaHidden,
             deaSystem);
  TdeAttrSet = set of TdeAttr;
```

➤ *Listing 3*

Every time we read a new filename with `FindFirst` and `FindNext` we check its attribute field to see if it matches the types and attributes we want. If they do match, we then try and match the filename itself with our regular expression engine. I won't go into the code for the first bit here: it's all pretty simple and you can check it out on the diskette.

Doubleplusgood

So, now onto the second part: regular expressions. What exactly is a regular expression? It is a way of defining a pattern that will be used to match text. The Delphi IDE's `Find` function has a regular expression search facility, for example: instead of trying to find some exact text you can specify a text pattern and the `Find` function

will try and match text to that pattern. Regular expressions come in many flavors. Borland used to ship a GREP program with their products that enabled you to do pattern matching across many files. Languages like Perl have built in regular expression matchers and the file wildcards are a poor man's regular expression language for files in DOS and Windows.

Actually, it's going to get rather annoying to use the full phrase 'regular expression' all the time so, if it's all right with you, let's shorten it to 'regex.' It'll save on paper, and our Editor won't think I'm trying to pad out this article.

Let's in fact start with the DOS file wildcards. Here they are, with their accepted meanings:

?	Match any single arbitrary character
*	Match zero or more arbitrary characters

So the wildcard pattern `???.*` will be matched by `ABC.TXT` and `ABC.T` but not by `AB.TXT` or `ABCD.T`. The three `?`s force exactly three characters to be matched, whereas the `*` can match any number of characters. The period `.` in the regex is called a literal character and it must be matched exactly. The other characters in the pattern, those that have some subtle meaning, are generally called metacharacters.

As a side note, the behavior of the literal character `.` in our regex language will differ from that in DOS. In DOS, every file that had no extension was assumed to have an implied `.` at the end, ie `*.` as a wildcard pattern would match filenames without extensions like `ABC`. In our regex language it won't: if the regex has a `.` then it will match a filename with an explicit `.` in it and not otherwise.

Now let's introduce the regex language enhancements we shall be using:

\	Escape character
+	zero or more copies of the prior subpattern
[...]	character class

The escape character precedes another character that might otherwise be interpreted by the parser as something else. For example, you can create files with a `+` character in them but our regex language uses `+` as a metacharacter. So if we wanted to include a plus sign in a pattern, we would escape it with the backslash: `\+`.

The plus sign, when un-escaped, signifies that the previous subpattern can be repeated zero or more times. So `X+` in a regex should be read as 'the text to be matched at this point can have no `X`, or one or more `X`s.'

Lastly, the character class pattern defines a set of characters, exactly one of which must be matched in the text being tested. You specify the characters in the set inside the square brackets. For example `[abc]` means that the text being matched must have an `a`, `b`, or `c` at this point. And that's it. As a space saving device you can specify a range of characters like this: `[a-z]`, which should be read as any character from `a` to `z` inclusive. Of course you can mix and match: `[0-9abc]` means any digit or one of the characters `a`, `b` or `c`.

There's yet another refinement of character classes: you can negate them. Instead of saying 'the next character must be any one of a, b, or c,' you might like to say 'the next character can be anything *but* an a, b or c.' Rather than spelling this out longwindedly (giving the other 253 characters inside brackets, or as two or more ranges) you can specify a negated class: `[^abc]`. This should be read as 'anything but a, b or c.' The caret `^` must appear as the first character after the left bracket.

Some of you might have noticed that the `-` character is a metacharacter within the character class. So what would you do if you wanted the dash as an item in the class rather than its metacharacter meaning? Escape it of course, with the `\` escape character. The same goes for the `^` and `]` characters: `[\^\-\]]` means a character class that consists of a `^`, `-` or `]`.

Let's reinforce what we've just learnt, by interpreting the somewhat ridiculous

```
abc*[0-9]+[^xyz]?
```

Starting from the left this pattern will match text that starts with the three letters abc, followed by zero or more arbitrary characters, followed by zero or more digits, followed by a character that's not an x, y or z, followed by a single arbitrary character. Phew!

Let's be a little more creative and yet concrete. A test application I wrote recently created a series of snapshot log files. Their names were formatted as a `T`, followed by the time in `HH-MM` format followed by the extension `.LOG`. What would be the regex for this set of files? I started off with

```
T[0-9]+-[0-9]+.LOG
```

which translates to `T`, followed by zero or more digits, followed by `-`, followed by zero or more digits, followed by `.LOG`. However, it can easily be seen that a file called `T-.LOG` would be matched by this regex. (Why?) My next try was much better:

```
T[012][0-9]-[0-5][0-9].LOG
```

Try saying out loud what this regex says: the more you read them the more it helps you understand them. Anyway, apart from the fact that a file name of `T29-00.LOG` would match (which is invalid according to my formatting rules for these log files since 29 is not a valid hour value), this is about as far as we can get with the particular regex language I have defined. To properly exclude filenames like this, I would have to incorporate `OR` logic into my file regex language and coding that will have to wait for a rainy day .

Do You Want To Break Up?

There are two steps to using a regex pattern string. The first one is compiling the string into some representation that suits us: this will also help us validate the pattern. The second one is applying

the compiled regex against various text strings. Sometimes both these steps are combined into one, but it's generally a waste of time to continually validate the same string over and over. So we'll be compiling the regex and then applying it in two separate steps: the first to be done in our replacement for `FileFirst` and the second in `FileFirst` and `FileNext`. The compiled regex will be disposed of in the `FileClose` routine. Great, I love it when a plan comes together!

Onto validation and compiling. The data structure I'm going to propose for the compiled regex is a linked list. Each node in the list will be a separate subpattern or token in the string. The basic node will look like this:

```
type
  PBaseNode = ^TBaseNode;
  TBaseNode = packed record
    NextNode  : PBaseNode;
    TokenType : TTokenType
  end;
```

The `TokenType` field will indicate what type of token the node represents: a literal character, a `?`, a `*`, a character class or whatever. Now obviously for some nodes or tokens we shall require some more information. For a `?` token, we don't have to store anything else in the node. For a literal character token we'll store an extra character with the node. For a character class token we'll store a set (ie `set of char`, a 32-byte variable) with the node. What about the `*` and `+` tokens? The first thing to realize is that `?+` (any character zero or more times) is a synonym for `*` (zero or more arbitrary characters). So in fact we don't need to store a separate token for `*`, we can use `?+`. The other tokens we need to store are a literal character followed by `+`, and a character class followed by `+`. There are no other tokens we need worry about in our regex language.

So, we have three tokens for single characters and three tokens for the same token types, but zero or more times. The `+` metacharacter is called a closure (I'm not really sure why), so we end up with an any character token and its closure, a literal character and its closure, and a character class and its closure.

One last thing before I let you off for a deep breath and some coffee. File names in DOS and Windows are case-insensitive, in other words, `ABC.DEF` is the same as `abc.def`. It makes sense to convert characters to the same case when we compile to save us having to do any continual changing of case later on.

Normally, whether we use uppercase or lowercase makes no difference. However, I keep getting caught out with this: when I say 'normally' I usually mean 'here in the US.' Elsewhere in the world it is not so clear-cut, and with some other code pages there are more lowercase characters than uppercase ones. So, we shall convert every character to lowercase before we use it and, of course, we shall use the specific Windows API to do it. Beware though: if you write `[A-z]` as a character class in the middle of a pattern string, the parser will in fact read it as `[a-z]`, and not as every character from `A` all the way though `z` with all the punctuation characters in between.

So how do we compile the pattern? Well, we read the string character by character, looking for our special metacharacters. If we find a `?` or `*` character, we simply create a new node and add it to the linked list (Listing 4).

```
Ch := LowerCaseChar(aPattern[Inx]);
case Ch of
  c_frxAnyChar :
    AllocPatternNode(aBinPattern, c_binAnyChar);
  c_frxClosure :
    AllocPatternNode(aBinPattern, c_binAnyClosure);
```

➤ *Listing 4*

```
c_frxEscape :
  begin
    if (Inx = PatLen) then begin
      FRXFreeBinPattern(aBinPattern);
      Result := frxcrMissingChar;
      Exit;
    end;
    Token := AllocPatternNode(aBinPattern, c_binLiteral);
    inc(Inx);
    Token^.bpnChar := LowerCaseChar(aPattern[Inx]);
  end;
```

➤ *Listing 5*

```
c_frxPatClosure :
  begin
    if not CloseLastPatternToken(aBinPattern) then begin
      FRXFreeBinPattern(aBinPattern);
      Result := frxcrNoSubpattern;
      Exit;
    end;
  end;
```

➤ *Listing 6*

`AllocPatternNode` allocates a new node of the correct type and adds it to the end of the `aBinPattern` linked list. If we find a `\` we look at the next character and create a literal character node for it. Of course, there's an error should the `\` be the last character in the pattern string (Listing 5).

The `FRXFreeBinPattern` routine frees the entire linked list created so far. If we find a `+` we have to close the last token on the linked list; there is an error if the last token is already closed (eg, we can't have `++` in the string) or there is no last token (ie, `+` was the first character in the pattern string). The `CloseLastPatternToken` routine shown in Listing 6 does all this, it returns `False` if there was an error.

If the character is a right bracket we shall mark this as an error (Listing 7). The character class parser

```
c_frxClassRight :
  begin
    FRXFreeBinPattern(aBinPattern);
    Result := frxcrMissingLeft;
    Exit;
  end;
```

➤ *Listing 7*

```
case Ch of
  ..the metacharacters..
else
  {any other character is a literal}
  Token := AllocPatternNode(aBinPattern, c_binLiteral);
  Token^.bpnChar := Ch;
end;{case}
```

➤ *Listing 8*

code we'll show in a minute (Listing 9) parses the right bracket as part of the character class and therefore in our outer loop we should never see it.

Leaving aside the left bracket for a moment, any other character we see is a literal character (Listing 8).

Take Your Pain Away

Now the left bracket (Listing 9). This, as we know, signifies the start of a character class definition, and we must read all the characters until we reach the right bracket (or run off the end of the string, which is an error). To make this easy on myself when I first wrote this, and also to keep the character class checking code entirely separate from the outer loop, I made it a separate routine called `ParseCharClass`.

```
c_frxClassLeft :
  begin
    {it can't appear at the end of the pattern}
    if (Inx = PatLen) then begin
      FRXFreeBinPattern(aBinPattern);
      Result := frxcrBadClass;
      Exit;
    end;
    {create a new token as if everything was OK}
    Token := AllocPatternNode(aBinPattern, c_binClass);
    {parse the character class}
    Result := ParseCharClass(aPattern, PatLen, Inx, Token);
    if (Result <> frxcrSuccess) then begin
      FRXFreeBinPattern(aBinPattern);
      Exit;
    end;
  end;
```

➤ *Listing 9*

```
c_frxNegate :
  begin
    {the class negation can only be the first character}
    if (aInx <> FirstInx) then
      Exit;
    {make a note that we have a negated class}
    NegatedClass := true;
  end;
```

➤ *Listing 10*

The code in `ParseCharClass` looks much the same as the outer code: a loop and a case statement looking for metacharacters, the metacharacters being a different set than that in the outer loop. The loop finishes when either we find the right bracket to close off the class definition, or we run out of characters in the pattern string (which will be an error, missing right bracket).

The easiest metacharacter to parse is `^`. It must appear as the first character in the pattern string after the left bracket, otherwise it's an error. If we find one we make a note that we must negate the class (Listing 10).

Before considering literal and escaped characters, we need to think a little about ranges. A range consists of a literal character, followed by -, followed by another literal character. The end of the range cannot be immediately followed by another -, as in `a-c-e`, because this is invalid. So when we're dissecting a range we have to be very careful. In fact, we can see that there are three states to parsing a range: (a) the first literal character has been read, (b) the dash character has been read, (c) the second literal character has been read.

If we are in state (a), we can move to state (b) or we can remain in state (a) if the next character is another literal (as in xyz). If we are in state (b) we *have* to move to state (c), there is no other choice: the dash must be followed by a literal character. Once we are in state (c) we *have* to move back to state (a) again (ie, we cannot move to state (b) by reading another dash). This is a very simple state machine that will be correctly terminated by reading a right bracket either in state (a) or in state (c). To code this we create an enumerated type:

```
type
  TRangeState = (CouldStart,
    Started, Completed);
```

State (a) is the value `CouldStart`: we could start a range if we wanted to as we've just read a literal character. State (b) is `Started`: we've just started a range by reading the dash character. State (c) is `Completed`: we've just completed a range. The whole loop starts initially with the state set to `Completed`, in other words we have to move to state (a) first thing. The code for a literal character looks similar to that shown in Listing 11, which neatly encapsulates this state machine.

Notice that when we move to state (c) we add all of the characters in the range to our set. For state (a) we include that single literal in our character set. Escaped characters have exactly the same code.

The code for parsing the - metacharacter closes off the state machine (note that we allow a - as the first character in the class definition, in which case it becomes a literal character, Listing 12).

When we encounter a right bracket, it's all over (Listing 13).

After the loop is over, we have been successful in parsing a valid character class if (1) we found a right bracket (duh!), and (2) the

```
if (RangeState = Started) then begin
  if (Ch <= FirstChar) then
    Exit;
  for ChInx := succ(FirstChar) to Ch do
    Include(aToken^.bpnCharClass, LowerCaseChar(ChInx));
  RangeState := Completed;
end else begin
  Include(aToken^.bpnCharClass, Ch);
  FirstChar := Ch;
  RangeState := CouldStart;
end;
```

➤ *Listing 11*

```
c_frxClassRange :
  begin
    {if this is the first character in the class then it's a literal character}
    if (aInx = FirstInx) then
      Include(aToken^.bpnCharClass, c_frxClassRange)
    {otherwise it's a range character, so we must be able to start a range}
    else if (RangeState <> CouldStart) then
      Exit
    {make a note that we're in a range}
    else
      RangeState := Started;
  end;
```

➤ *Listing 12*

```
c_frxClassRight :
  begin
   {the right bracket cannot be the first character}
    if (aInx = FirstInx) then
      Exit;
   {make a note that we found the right bracket and break out of the loop}
    FoundRightBracket := true;
    Break;
  end;
```

➤ *Listing 13*

range state is not Started (ie, is not in state (b)). If we have a valid character class, then all we then need to do is to check to see whether the class was negated, and if so negate it.

Please do refer to the entire code on this month's diskette for all the missing glue code that brings all this pattern string parsing together.

The First Cut

Once we've successfully parsed a pattern string and created a compiled version of it, we can try and apply it to various file names. This matching process will either return true (the file name does match the compiled regex) or false (the file name does not match the regex).

So what does it really mean to 'match the regex'? What happens is that we read through the regex node by node and read through the filename string character by character and see if the next node matches the next character. For the any character, literal character and character class tokens this is easy. The any character token matches... (you guessed it) any character; the literal character token has to match exactly that character and nothing else; the character class token matches the character if the character appears in the node's character set.

We finish successfully when we run out of regex tokens at exactly the same time we run out of characters in the filename. If there are tokens left over after we've matched all the characters, for example: `t[0-9]` matching `t`, or there are more characters left over after we've run out of tokens, for example `t[0-9]` matching `t1a`, then the regex as a whole did not match the filename.

Piece of cake! We're really motoring now! Unfortunately we then fetch up on the closure problem, which I've been momentarily avoiding. Recall that 'closure' means the zero or more matches ability. How on earth do we match zero or more characters? What does that mean exactly? When do we match zero and when do we match more? When do we stop matching characters, if there are possibly more than one? And so on.

How Long?

Let us look at a specific regex pattern string and a Win32 long file name:

```
Regex: t*.1*
File:  tabc.def.123
```

From our initial discussion in this article you can recognize that the regex says: the filename must start with a `t`, followed by zero or more arbitrary characters, followed by a period, followed by a `1`, followed by zero or more arbitrary characters. Let's try and manually match the filename to the regex. It starts with a `t`, which the regex requires, so we are OK so far. Now we have to match zero or more characters. Mmm. Do we match the minimum allowed (ie, zero) and then move on? Do we match the maximum allowed (ie, the rest of the characters in the filename)? Do we match one or two characters or something else instead?

For our first try, let's take the minimal viewpoint: always match as little as we can with a closure. Assume we match no characters. Advance the regex token to the literal period, and get the next character: it's an `a`. They don't match. So are we done? No, we're not: we made an assumption that we only needed to match zero characters. Maybe we were wrong and we needed to match one character with the * closure. OK, retrace our steps and match one character. Advancing again, we are trying to match the . token with a `b`. Doesn't match again. Retrace our steps and match two characters with the * closure. You can see that this still isn't going to work. So, we retreat, match three characters (the `abc`) and suddenly the next token matches the first period in the filename. Great! Advance to the next token and character, and we are trying to match a `1` literal with a `d`. They don't, so back we go to our previous closure and match 4

characters. Without filling this magazine with the laborious steps in between, I'm sure you can see that eventually we match abc.def to the first * closure, and 23 to the second. Using the second * closure meant matching no characters (error, more characters to go), matching one character (error, more characters to go) and finally matching two characters. But getting there was so long-winded.

For our second try, let's take the maximal viewpoint: always match as much as we can with a closure. This is known as the greedy algorithm: closures try and grab as many characters as they can. OK, here goes: assume the first * closure token matches all the remaining characters in the filename, wham bam. We're not done because we have some more tokens to use. So back up one character (ie, the first * matches all but one character). Advance to the next token and character. They don't match. Back up another character. Still no go. Back up one more. We're there. The period literal token matches the second period in the filename, the 1 literal token matches the 1, and the second * closure matches the rest of the filename. Boom, done.

In fact, research has shown that the greedy algorithm is usually the best in terms of speed, since in general there is less backtracking to do. For our implementation of regular expressions, it probably doesn't make too much difference, after all filenames tend to be short, but I'm sure that you would agree that trying to apply the standard *.* as a non-greedy algorithm in your head will convince you that Greed is Good.

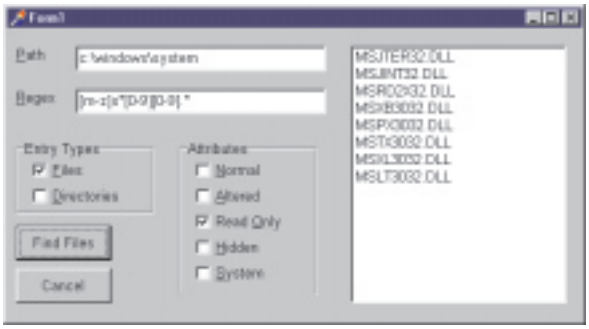
The Walk

Having made that little digression into how to match closures, we're still left with a problem: how are we to implement them? In the literature, closures are usually implemented using recursion. There's some routine that takes a token and a substring and returns true if the token (and subsequent tokens) match the substring. The routine basically keeps calling itself with the next token and a shorter substring, until we run out of tokens or substrings or both.

Anything recursion can do, we can do better. We'll set up a stack to hold closure tokens as we get to them. If we get to a point where we have a mismatch between a token and a character from the filename,

➤ *Listing 14*

```
function FRXMatchesPattern(aBinPattern : PfrxBinPattern;
  const aFileName   : string) : boolean;
type
  TCheckPoint = packed record
    cpToken : PBinPatNode;
    cpStart : word;
    cpInx   : word;
  end;
var
  FNLen   : integer;
  Inx     : integer;
  StartInx: integer;
  Token   : PBinPatNode;
  BadSimpleMatch : boolean;
  TokenSP : integer;
  TokenStack : array [0..127] of TCheckPoint;
begin
  {assume that we'll fail}
  Result := false;
  {if the pattern is empty, there's no match}
  if (aBinPattern = nil) then
    Exit;
  {if the filename is the empty string, there's no match}
  FNLen := length(aFileName);
  if (FNLen = 0) then
    Exit;
  {prepare closure token stack to be empty}
  TokenSP := -1;
  {prepare for loop}
  Token := PBinPatNode(PBinPatHeader(aBinPattern)^.bphData);
  Inx := 1;
  while True do begin
    BadSimpleMatch := false;
    case Token^.bpnToken of
      c_binAnyClosure :
        begin
          {push it onto the stack as a greedy token}
          inc(TokenSP);
          with TokenStack[TokenSP] do begin
            cpToken := Token;
            cpStart := Inx;
            cpInx := succ(FNLen);
          end;
          {indicate we've matched everything}
          Inx := succ(FNLen);
          {advance the token}
          Token := Token^.bpnNext;
        end;
      c_binLitClosure,
      c_binClsClosure :
        begin
          {match as many chars as we can}
          StartInx := Inx;
          while (Inx <= FNLen) and
            MatchOneChar(Token, aFileName[Inx]) do
            inc(Inx);
          {if we matched at least one char...}
          if (StartInx < Inx) then begin
            {push it onto the stack as a greedy token}
            inc(TokenSP);
            with TokenStack[TokenSP] do begin
              cpToken := Token;
              cpStart := StartInx;
              cpInx := Inx;
            end;
          end;
          {advance the token}
          Token := Token^.bpnNext;
        end;
    else {the current token is a simple token}
      {if there is a current character and it matches the
       current token, advance}
      if (Inx <= FNLen) and
        MatchOneChar(Token, aFileName[Inx]) then begin
        Token := Token^.bpnNext;
        inc(Inx);
      end
      {otherwise there is no current character
       or it did not match}
      else begin
        {if there is no closure to revert to,
         we're done but failed}
        if (TokenSP = -1) then
          Exit;
        {make a note we failed to match: this'll trigger an
         operation at the end of the loop to revert to a
         previous closure}
        BadSimpleMatch := true;
      end;
    end;{case}
    {we're finished and successful if the current token is
     nil (ie, we ran out of tokens) and the current
     character index is greater than the length of the
     string (ie, we ran out of string)}
    if (Token = nil) and (Inx > FNLen) then begin
      Result := true;
      Exit;
    end;
    {if the current token is nil or there was a bad simple
     match, we need to revert to a previous closure and back
     up one character}
    if (Token = nil) or BadSimpleMatch then begin
      while (TokenSP <> -1) do begin
        with TokenStack[TokenSP] do begin
          if (cpInx > cpStart) then begin
            dec(cpInx);
            Token := cpToken^.bpnNext;
            Inx := cpInx;
            Break;{out of while loop}
          end;
          dec(TokenSP);
        end;
      end;
      {if there are no more closures or the current token is
       still nil, we're done but failed}
      if (TokenSP = -1) or (Token = nil) then
        Exit;
    end;
  end;{of forever loop}
end;
```



➤ *Figure 1*

we shall back up to the previous closure token (it'll be at the top of the stack), decrement the number of characters it refers to and continue from that point. If the closure token at the top of the stack runs out of characters it can match, we pop it off the stack and continue with the new closure token that's at the top of the stack. Eventually, we either run out of closure tokens on the stack (in which case we didn't match the regex to the filename) or we match all the tokens to the filename at some stage.

The stack is implemented as a simple array of records with an integer variable to act as the stack pointer. Each record has enough information to checkpoint a particular closure: the token itself, the point in the string where it was applied and the end point in the string to which it applies. If the stack pointer is `-1` the stack is empty, otherwise the stack pointer is the index in the array of the top of the stack. Popping means decrementing the stack pointer, pushing means incrementing the stack pointer and storing the checkpoint at that element.

Look at the `FRXMatchesPattern` code in Listing 14: it implements the matching logic. The preliminaries show that the compiled regex must have at least one token to match and the filename must have at least one character. Then, after some minor setup, we're in the loop! It's an infinite loop and we'll be exiting it and the routine at various points within the code in the loop itself. We start the loop off with a `case` statement looking at the current token type.

For an any character closure (the * metacharacter) we push the closure onto the stack, and match the rest of the filename string (ie, set the current character index to one more than the length of the string). Then we advance to the next token.

For a literal character closure or a character class closure, we match as many characters in the filename as we can. If we match at least one, push the closure token onto the stack. Then advance to the next token.

For any other token (it must be a simple one) try and match it to the current character. If we do, advance to the next token and advance the index of the current character in the filename string. If it doesn't match, have a look at the closure token stack. If it's empty, we've failed to match the regex to the filename, so bail out now. Otherwise we set a `boolean` flag to say that we failed to match, and it will trigger some code at the end of the loop.

After this `case` statement, we check the current token and the current character. If the current token is `nil`, we've run out of tokens. If the current character is greater than the length of the string, we've run out of filename. If both are true, we've succeeded: the regex matches the filename.

If we're still in the loop after this test, we check the current token. If it is `nil`, we've run out of tokens, so what should we do now? If we failed a simple token match then we need to do something else as well. In either case what we need to do is backtrack. Look at the top of the stack. If the closure token there can be backed up by one character, do so. Get the next token from this closure token, and set the character index. Break out of the stack-checking loop. If, on the other hand, the token at the top of the stack has no more characters to back up, pop it off, and repeat this algorithm on the token that's now at the top of the stack. If we manage to pop off all the closure tokens from the stack, we've failed to match the regex to the filename and so we bail out.

Wrap It Up

And that's it. At long last! We have all the pieces together to write our own extended `FindFirstEx`, `FindNextEx` and `FindCloseEx` routines. After the experience of parsing and applying regexes to filenames, a discussion of how to wrap it all up into these new replacement routines would be somewhat of an anticlimax because of its sheer simplicity. So I will leave you to have a look at the code on the diskette. As usual you are free to use it in your own applications as you wish, but I retain all copyright to the source code. I've also included a little test program called EXAMPLE (Figure 1).

Julian Bucknall works for TurboPower Software. He is eurhythmically regular in his expressions. He can be reached by e-mail at julianb@turbopower.com or on CompuServe at 100116,1572.