The 51st State

It's back to state machines to answer some reader questions

by Julian Bucknall

Oh boy, oh boy. I certainly hit a motherlode with my two articles on state machines and regular expressions (in the January and February 2001 issues). I can't remember when I had such a lot of email about one of my articles. From the messages I received, it certainly seems that a lot of my readers were interested in these topics and wanted to see more. So, gentle reader, with your permission, I'll devote this article to some of your questions and suggestions. Of course, you should re-read the two articles in question to get the background for what we'll be discussing this time: I won't be going over much of the previous information that I've already presented.

Any Comments?

One of the more popular topics was how to parse an Object Pascal source file. This is a large subject, and so we should approach it gradually.

The topic has been covered before by Marco Cantù in Issues 23 and 30, and by Paul Warren in Issue 38 *[So it's time to order that Collection 2000 back issues CD! Ed].*

We'll start off source file parsing with a simple automaton to extract all of the comments in a source file. This idea sounds a little bogus, but in reality it has some interesting applications. For example, we could be extracting specially formatted comments that will form the basis for a help file. I'm sure you've seen this kind of product; possibly the most familiar one is Time2Help. Another example, perhaps, would be to form the basis of a spell-checker for source code.

For our purposes, however, writing a program to extract comments from a source file is a good teaching aid in how to use state machines.

For this exercise, we shall assume that we will be reading a source file character-by-character and not line-by-line. Possibly one of the most important habits to break out of when using automata on a text file is the one of reading the file as a set of lines. The state machines we were discussing are all character-based, so we should get into the habit of viewing the source file as a sequence of characters. That's not to say we couldn't devise an automaton that was line-based, of course, but let's keep it simple. The first problem, then, is how to read a file character-by-character.

Over the past several years I've certainly come to believe that the most important hierarchy of classes in the Delphi VCL (or CLX, for that matter) is the `TStream` hierarchy. Borland R&D designed it well and, once you get used to the basic layout, it lends itself to some wonderful extensions. Normally, though, when developers have to use a stream, they pick one of the existing ones, or they write a new one that descends from one of them. There's nothing wrong with either of these choices, of course, but we'll do something different: we'll use delegation and write a stream *filter*. A stream filter is a `TStream` descendant that uses a pre-created `TStream` descendant internally to hold the stream data.

Input Character Stream

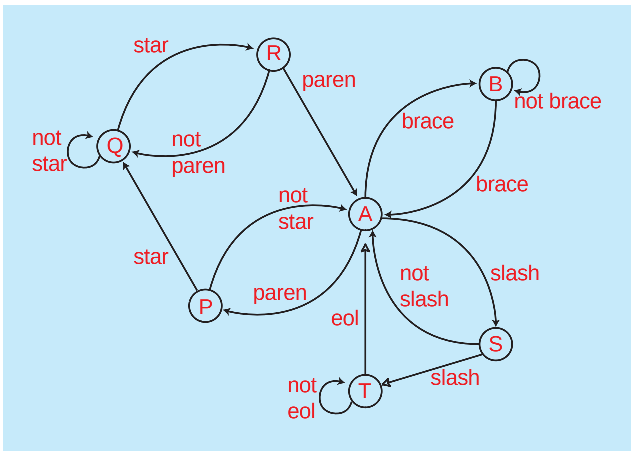
Consider my original initial problem: reading a file as a stream of

➤ *Listing 1: Input character stream.*

```
type
  TaaInCharStream = class(TStream)
    private
      FBufEnd : integer;
      FBuffer : PByteArray;
      FBufPos : integer;
      FStream : TStream;
    protected
      procedure icsGetBuffer;
    public
      constructor Create(aStream : TStream);
      destructor Destroy; override;
      function Read(var Buffer; Count : longint) : longint;
        override;
      function Write(const Buffer; Count : longint) :
        longint; override;
      function Seek(Offset : longint; Origin : word) :
        longint; override;
      function GetChar : char;
  end;
constructor TaaInCharStream.Create(aStream : TStream);
begin
  inherited Create;          {create the ancestor}
  FStream := aStream;        {save the stream}
  GetMem(FBuffer, BufSize);  {create the buffer}
end;
destructor TaaInCharStream.Destroy;
begin
  {free the buffer}
  if (FBuffer <> nil) then
    FreeMem(FBuffer, BufSize);
  inherited Destroy;
end;
function TaaInCharStream.GetChar : char;
begin
  repeat
    {make sure the buffer has data}
    if (FBufPos = FBufEnd) then
      icsGetBuffer;
    {if no more data, return #0 to signal end of stream}
    if (FBufEnd = 0) then
      Result := #0
    {otherwise return the current character}
    else begin
      Result := char(FBuffer^[FBufPos]);
      Assert(Result <> #0, 'TaaInCharStream.GetChar: ‘+
        ‘input stream is not text, read null');
      inc(FBufPos);
    end;
  until (Result <> CR);
end;
procedure TaaInCharStream.icsGetBuffer;
begin
  FBufPos := 0;
  FBufEnd := FStream.Read(FBuffer^, BufSize);
end;
```

characters. We could just use `TFileStream` and the `Read` method to get the characters one by one. This would be horrendously inefficient, mainly because the `TFileStream` is unbuffered: every call to `Read` is translated into a call to the Windows API to read from the encapsulated file. Calling the Windows API every time to read a single character is not my idea of fun. So, Plan A would be to write a `TFileStream` descendant that buffered the data from the file by reading the file buffer-by-buffer and doling out the characters one-by-one from the buffer. Nothing wrong with that, certainly, but it's rather limited. What if we wanted to buffer another type of stream? Well, we would have to write another buffered `TStream` descendant, copying much of the code we'd already written.

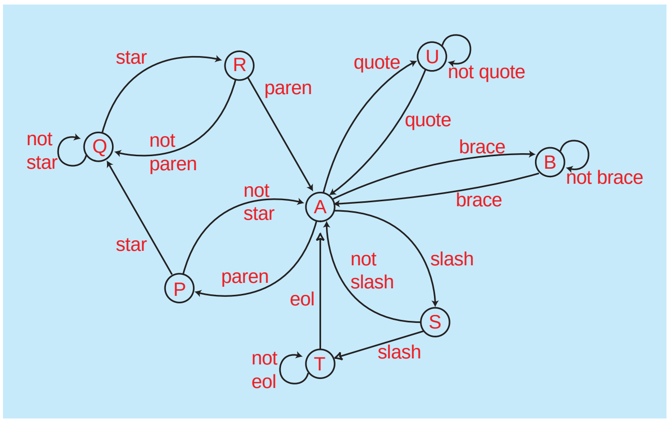
No, a much better plan, in my view, is to write a stream filter. Our stream filter will be a buffered stream (in other words, it will descend from `TStream` and inherit all its behavior) but it will read its buffers full of data from another stream, one that we pass to the filter's constructor. (The reason it's called a filter, by the way, is that all data from the original stream passes through the filter stream and we could, if needed, alter that data. Some examples: decompressing the data on reading, replacing CR/LF pairs by single LF characters, uppercasing all characters, etc.) The stream filter will be specially designed for text streams; in other words, we'll write a special `GetChar` method to return the next character from the stream. We could still use the `Read` method of course, but we will optimize the `GetChar` method to return a single character. It will return a null character when the stream is exhausted (and this forces us to accept that `GetChar` should only be used on text streams).



➤ *Figure 1: Automaton to recognize comments.*

Listing 1 shows the input character stream. I've deliberately made it a read-only stream (it's much easier to code that way and we generally use streams for pure input or output and not mixed I/O). As you can see from looking at this code, the class reads data from its underlying stream in chunks of 8Kb, and then doles it out as needed. Not only is there no write functionality, but there's no seek functionality either: for the reasons I'll be using the input character stream I don't need it. (This, by the way, is a tenet of Extreme Programming: don't write code until you need it. So, I shall implement the `Seek` method once I decide I really need it, and not before.) On this month's disk, you'll also find an output character stream, written to the same type of specification: pure output stream, no seeking.

Now we have an efficient input character stream, we can go back to our original problem: extracting comments from a source file. In Object Pascal, comments are defined in three separate ways: in between braces, `{}`, in between `(*` and `*)`, and finally the rest of a line after a `//` sequence. In fact, you can view the final definition as this: the comment starts with the `//` character pair and ends with the CR/LF pair. Of course, now that we can write source files with Kylix on Linux, we should accept lines that just end in a single LF character. (In fact, if you read the code for `GetChar` carefully in Listing 1, you'll see that it never returns a CR character, so the user of the character stream will only see LF characters delimiting lines.)



➤ *Figure 2: Better automaton to recognize comments.*

The Comment Automaton

A first attempt at a state machine to recognize comments is given in Figure 1. There's nothing too exciting about this particular automaton: if you trace the various possibilities you can see that comments are certainly recognized. Converting the figure into code would be fairly easy.

However, there's a bug. If you did convert the automaton into code and then ran it on the source file you'd just written, you'd see the problem straight away. Let's explain without writing the actual code. If you're like me, you'd write the code using the character constant `{` at least once. It's there so that you could recognize the start of one type of comment. The compiler knows that this open brace is not the beginning of a comment, but your code does not since the original automaton didn't.

We should therefore include an extra state in the automaton to recognize character and string constants and to ignore comment-like characters or pairs of characters in between the quotes. Figure

2 shows this revised automaton, and Listing 2 shows the equivalent code. This listing merely writes out the comments to an output character stream. In fact, if we *were* writing a spell checker for source code, the automaton we designed would enable us to spell check embedded string constants as well: a bonus.

Tokenizing

Having seen this simple case, let's approach parsing Pascal from a different angle. Although it would be nice to create a gigantic state machine for parsing a complete Pascal source file, unfortunately it's the wrong approach. One way to look at it would be to write a top-down parser from the Pascal grammar in the same way as we did for regular expressions. That's still a massive job, much greater than we can deal with in this small article (I like the approach taken by *Writing Compilers And Interpreters* by Ronald Mak: an 840 page behemoth of a book, now in its second edition). Better still is writing a bottom-up parser.

Instead we'll look at how to break down a Pascal source file into different tokens. From this we can do some interesting things like reformatting code, creating an HTML file to display the code, pretty print the file, and so on.

The job has been made fairly easy for us by the foresight of Pascal's designer, Niklaus Wirth. Unlike some languages, such as Fortran, scanning the source code and returning tokens is fairly independent of what those tokens actually mean syntactically. We can write the *tokenizer* without worrying about whether the code is syntactically correct; that's the job of the compiler. For example, our tokenizer will parse nonsense code like this:

```
not Identifier := 2.0 <=;
```

into the individual parts (`not`, `Identifier`, `:=`, `2.0`, `<=`, and `;`). It turns out that we can write the tokenizer as a finite state machine.

First, though, we should work out what we might see as we scan source code. Identifiers and keywords start with a letter, and have zero or more letters, digits and underscores afterwards. Space characters separate identifiers and keywords. Numbers start with a digit (or maybe a minus sign or a plus sign) and have other digits, plus decimal radix points, afterwards. There's a whole slew of punctuation marks, the most obvious being the semicolon `;`, but also `=`, `:=`, `<=`, `^` and so on so forth. Notice that some of the punctuation marks are doubled-up. Other than that, there's nothing to trip us up.

Our tokenizer will work as a standalone routine. We call it with

➤ *Listing 2: Extracting comments from a Pascal source file.*

```
procedure ExtractComments(aInStm  : TaaInCharStream;
  aOutStm : TaaOutCharStream);
type
  TStates = (          {the types of states...}
    sScanNormal,       {normal scanning}
    sScanBraceComment, {scanning comment after getting
      '{'}
    sScanAfterSlash,   {scanning after getting first '/'}
    sScanSlashComment, {scanning comment after getting '//'}
    sScanAfterParen,   {scanning after getting '('}
    sScanPStarComment, {scanning comment after getting ‘(*'}
    sScanAfterStar,    {scanning after getting '*' in a
                        (**) comment}
    sScanString);      {scanning a string}
var
  State : TStates;
  Ch    : char;
begin
  State := sScanNormal;
  Ch := aInStm.GetChar;
  while (Ch <> #0) do begin
    case State of
      sScanNormal :
        begin
          case Ch of
            '''': State := sScanString;
            '(' : State := sScanAfterParen;
            '/' : State := sScanAfterSlash;
            '{' : begin
                    aOutStm.PutChar('{');
                    State := sScanBraceComment;
                  end;
          end;{case}
        end;
      sScanBraceComment :
        begin
          aOutStm.PutChar(Ch);
          if (Ch = '}') then begin
            aOutStm.PutChar(#10);
            State := sScanNormal;
          end;
        end;
      sScanAfterSlash :
        begin
          case Ch of
            '''': State := sScanString;
            '(' : State := sScanAfterParen;
            '/' : begin
                    aOutStm.PutChar('/');
                    aOutStm.PutChar('/');
                    State := sScanSlashComment;
                  end;
            '{' : begin
                    aOutStm.PutChar('{');
                    State := sScanBraceComment;
                  end;
          else
            State := sScanNormal;
          end;{case}
        end;
      sScanSlashComment :
        begin
          aOutStm.PutChar(Ch);
          if (Ch = #10) then
            State := sScanNormal;
        end;
      sScanAfterParen :
        begin
          case Ch of
            '''': State := sScanString;
            '(' : State := sScanAfterParen;
            '*' : begin
                    aOutStm.PutChar('(');
                    aOutStm.PutChar('*');
                    State := sScanPStarComment;
                  end;
            '/' : State := sScanAfterSlash;
            '{' : begin
                    aOutStm.PutChar('{');
                    State := sScanBraceComment;
                  end;
          else
            State := sScanNormal;
          end;{case}
        end;
      sScanPStarComment :
        begin
          aOutStm.PutChar(Ch);
          if (Ch = '*') then
            State := sScanAfterStar;
        end;
      sScanAfterStar :
        begin
          aOutStm.PutChar(Ch);
          if (Ch = ')') then begin
            aOutStm.PutChar(#10);
            State := sScanNormal
          end
          else
            State := sScanPStarComment;
        end;
      sScanString :
        begin
          if (Ch = '''') then
            State := sScanNormal;
        end;
    end;{case}
    Ch := aInStm.GetChar;
  end;
end;
```

a character stream primed and ready to release the next character, and it will return a token identifier and the token string (at least in the case of an identifier/keyword or a number). By token identifier, I mean a value from an enumerated type, so the first thing is to define the enumerated type. Listing 3 shows the various token identifiers we'll be recognizing. The character stream will be left at the next character to be read after the token was extracted and identified. The tokenizer will automatically jump over white space (blanks, tabs or LF characters).

There's still a little wrinkle, though. When we are scanning through the character stream, how do we know when we reach the end of a token? For example, if we were scanning `program Project1;` at the start of a .DPR project file, how do we know that `program` ends at the `m`? The answer is, of course, we don't: we have to read the next character (or at least peek at it). If we read it and discovered that we didn't need it, we shall have to give it back to the character stream somehow so that it can be returned next time. For white space that's not really required, but we might be scanning something like `i:integer` and the colon would have to be put back.

Is that it? Nope, I'm afraid not. There's yet another wrinkle. Let's suppose we were scanning

```
A : array [0..9] of byte;
```

and we'd got to the 0. As we're scanning along, it looks like we're reading a decimal number: zero, point. It's not until we reach the second period that we suddenly say whoops, it's actually a range. We would in fact have to put back two periods in that case. Luckily this is the worst it gets for Pascal. I altered the character stream class to enable us to put back up to two characters (the `PutBackChar` method) and for the `GetChar` method to release the characters which have been put back first, before continuing with the actual stream data.

➤ *Listing 3: Pascal tokens.*

```
type
  TaaPascalToken = ( {types of Pascal tokens...}
    ptInvalidToken,  {..some kind of error}
    ptEndOfFile,     {..end of file}
    ptKeyword,       {..keyword, eg, if, while, do, ...}
    ptIdentifier,    {..identifier}
    ptString,        {..string or character constant}
    ptHexNumber,     {..number in hex, starts with $}
    ptNumber,        {..sequence of digits, maybe with radix point}
    ptComment,       {..comment, any type}
    ptComma,         {..comma: ,}
    ptSemicolon,     {..semicolon: ;}
    ptColon,         {..colon: :}
    ptPeriod,        {..period: .}
    ptRange,         {..range: ..}
    ptEquals,        {..equals char: =}
    ptNotEquals,     {..not equals: <>}
    ptLess,          {..less than: <}
    ptLessEqual,     {..less than or equal: <=}
    ptGreater,       {..greater than: >}
    ptGreaterEqual,  {..greater than or equal: >=}
    ptAssign,        {..assignment: :=}
    ptOpenParen,     {..open parenthesis: (}
    ptCloseParen,    {..close parenthesis: )}
    ptOpenBracket,   {..open bracket: [}
    ptCloseBracket,  {..close bracket: ]}
    ptCaret,         {..caret: ^}
    ptHash,          {..hash: #}
    ptAddress,       {..ampersand: @}
    ptPlus,          {..addition: +}
    ptMinus,         {..subtraction: -}
    ptMultiply,      {..multiplication: *}
    ptDivide);       {..division: /}
```

➤ *Listing 4: Simple Pascal tokenizer.*

```
procedure ReadNumber(aInStm : TaaInCharStream;
  var aToken : string);
var
  Ch : char;
  State : (BeforeDecPt, GotDecPt, AfterDecPt, Finished);
begin
  State := BeforeDecPt;
  while (State <> Finished) do begin
    Ch := aInStm.GetChar;
    if (Ch = #0) then begin
      State := Finished;
      aInStm.PutBackChar(Ch);
    end else begin
      case State of
        BeforeDecPt :
          begin
            if (Ch = '.') then begin
              State := GotDecPt;
            end else if (Ch < '0') or (Ch > '9') then begin
              State := Finished;
              aInStm.PutBackChar(Ch);
            end else
              aToken := aToken + Ch;
          end;
        GotDecPt :
          begin
            if (Ch = '.') then begin
              aInStm.PutBackChar(Ch);
              aInStm.PutBackChar(Ch);
              State := Finished;
            end else begin
              aToken := aToken + '.';
              aToken := aToken + Ch;
              State := AfterDecPt;
            end;
          end;
        AfterDecPt :
          begin
            if (Ch < '0') or (Ch > '9') then begin
              State := Finished;
              aInStm.PutBackChar(Ch);
            end else
              aToken := aToken + Ch;
          end;
      end;
    end;
  end;
end;
procedure ReadHexNumber(aInStm : TaaInCharStream;
  var aToken : string);
var
  Ch : char;
  State : (NormalScan, Finished);
begin
  State := NormalScan;
  while (State <> Finished) do begin
    Ch := aInStm.GetChar;
    if (Ch = #0) then begin
      State := Finished;
      aInStm.PutBackChar(Ch);
    end else begin
      case State of
        NormalScan :
          begin
            if not (Ch in ['A'..'F', 'a'..'f', '0'..'9'])
              then begin
              State := Finished;
              aInStm.PutBackChar(Ch);
            end else
              aToken := aToken + Ch;
          end;
      end;
    end;
  end;
end;
```

{ CONTINUES ON FACING PAGE...}

{ ... LISTING 4 CONTINUED FROM FACING PAGE}

```
procedure ReadIdentifier(aInStm : TaaInCharStream;
  var aToken : string);
var
  Ch : char;
begin
  Ch := aInStm.GetChar;
  while Ch in ['A'..'Z', 'a'..'z', '0'..'9', '_'] do begin
    aToken := aToken + Ch;
    Ch := aInStm.GetChar;
  end;
  aInStm.PutBackchar(Ch);
end;
procedure ReadString(aInStm : TaaInCharStream;
  var aToken : string);
var
  Ch : char;
begin
  Ch := aInStm.GetChar;
  while (Ch <> '''') and (Ch <> #0) do begin
    aToken := aToken + Ch;
    Ch := aInStm.GetChar;
  end;
  if (Ch = '''') then
    aToken := aToken + Ch
  else
    aInStm.PutBackchar(Ch);
end;
procedure ReadBraceComment(aInStm : TaaInCharStream;
  var aToken : string);
var
  Ch : char;
begin
  Ch := aInStm.GetChar;
  while (Ch <> '}') and (Ch <> #0) do begin
    aToken := aToken + Ch;
    Ch := aInStm.GetChar;
  end;
  if (Ch = '}') then
    aToken := aToken + Ch
  else
    aInStm.PutBackchar(Ch);
end;
procedure ReadSlashComment(aInStm : TaaInCharStream;
  var aToken : string);
var
  Ch : char;
begin
  Ch := aInStm.GetChar;
  while (Ch <> #10) and (Ch <> #0) do begin
    aToken := aToken + Ch;
    Ch := aInStm.GetChar;
  end;
  aInStm.PutBackchar(Ch);
end;
procedure ReadParenComment(aInStm : TaaInCharStream;
  var aToken : string);
var
  Ch : char;
  State : (NormalScan, GotStar, Finished);
begin
  State := NormalScan;
  while (State <> Finished) do begin
    Ch := aInStm.GetChar;
    if (Ch = #0) then begin
      State := Finished;
      aInStm.PutBackChar(Ch);
    end else begin
      aToken := aToken + Ch;
      case State of
        NormalScan :
          if (Ch = '*') then
            State := GotStar;
        GotStar :
          if (Ch = ')') then
            State := Finished
          else
            State := NormalScan;
      end;
    end;
  end;
end;
procedure AAGetToken(aInStm : TaaInCharStream;
  var aTokenType : TaaPascalToken; var aToken : string);
var
  Ch : char;
begin
  {assume we have an invalid token}
  aTokenType := ptInvalidToken;
  aToken := '';
  {ignore any whitespace prior to the token}
  Ch := aInStm.GetChar;
  while (Ch <> #0) and (Ch <= ' ') do
    Ch := aInStm.GetChar;
  {if we've reached end-of-file, exit returning that token
    type}
  if (Ch = #0) then begin
    aTokenType := ptEndOfFile;
    Exit;
  end;
  {parse the token based on the current character}
  case Ch of
    '#' : aTokenType := ptHash;
    '$' : begin
            aTokenType := ptNumber;
            aToken := Ch;
            ReadHexNumber(aInStm, aToken);
          end;
    '''': begin
            aTokenType := ptString;
            aToken := '''';
            ReadString(aInStm, aToken);
          end;
    '(' : begin
            Ch := aInStm.GetChar;
            if (Ch <> '*') then begin
              aInStm.PutBackChar(Ch);
              aTokenType := ptOpenParen;
            end else begin
              aTokenType := ptComment;
              aToken := '(*';
              ReadParenComment(aInStm, aToken);
            end;
          end;
    ')' : aTokenType := ptCloseParen;
    '*' : aTokenType := ptMultiply;
    '+' : aTokenType := ptPlus;
    ',' : aTokenType := ptComma;
    '-' : aTokenType := ptMinus;
    '.' : begin
            Ch := aInStm.GetChar;
            if (Ch = '.') then
              aTokenType := ptRange
            else begin
              aInStm.PutBackChar(Ch);
              aTokenType := ptPeriod;
            end;
          end;
    '/' : begin
            Ch := aInStm.GetChar;
            if (Ch <> '/') then begin
              aInStm.PutBackChar(Ch);
              aTokenType := ptDivide;
            end else begin
              aTokenType := ptComment;
              aToken := '//';
              ReadSlashComment(aInStm, aToken);
            end;
          end;
    '0'..'9' :
          begin
            aTokenType := ptNumber;
            aToken := Ch;
            ReadNumber(aInStm, aToken);
          end;
    ':' : begin
            Ch := aInStm.GetChar;
            if (Ch = '=') then
              aTokenType := ptAssign
            else begin
              aInStm.PutBackChar(Ch);
              aTokenType := ptColon;
            end;
          end;
    ';' : aTokenType := ptSemicolon;
    '<' : begin
            Ch := aInStm.GetChar;
            if (Ch = '=') then
              aTokenType := ptLessEqual
            else if (Ch = '>') then
              aTokenType := ptNotEquals
            else begin
              aInStm.PutBackChar(Ch);
              aTokenType := ptLess;
            end;
          end;
    '=' : aTokenType := ptEquals;
    '>' : begin
            Ch := aInStm.GetChar;
            if (Ch = '=') then
              aTokenType := ptGreaterEqual
            else begin
              aInStm.PutBackChar(Ch);
              aTokenType := ptLess;
            end;
          end;
    '@' : aTokenType := ptAddress;
    'A'..'Z', 'a'..'z', '_' :
          begin
            aTokenType := ptIdentifier;
            aToken := Ch;
            ReadIdentifier(aInStm, aToken);
          end;
    '[' : aTokenType := ptOpenBracket;
    ']' : aTokenType := ptCloseBracket;
    '^' : aTokenType := ptCaret;
    '{' : begin
            aTokenType := ptComment;
            aToken := '{';
            ReadBraceComment(aInStm, aToken);
          end;
  end;
end;
```

Listing 4 shows the tokenizer. I haven't shown the automaton that it implements (it's very long and thin with a billion-and-one terminating states), but I'm sure you get the idea by following the code. Notice that the piece of code that scans an identifier is itself an automaton, as is the code for recognizing a number, a hex number, and the various types of comments. (For recognizing a floating-point number, I have followed the Pascal definition: there must be at least one digit before and after the decimal radix point.) The tokenizer also returns comments: we don't want to discard anything at this level.

Adding Keyword Detection

The code as written does not recognize any keywords yet. All sequences that start with a letter and are followed by letters, digits or underscores are just lumped together as 'identifier' tokens. However, it's still usable. Listing 5 shows a simplistic code obfuscator, one that just removes all the indentation, comments and human-required white space and leaves a single blank between tokens that need it (for example, between identifiers and numbers). It outputs lines that are about 70 characters long. The Delphi compiler doesn't care about white space, of course, and will happily compile obfuscated code generated by this program. (To save space, I've not shown all the token types in Listing 5.)

So what about those keywords, then? Well, the first thing we need to do is get a list of them. The Delphi help file provides that (search for 'reserved words' and 'directives'). For each identifier we scan in the tokenizer we shall have to check whether the identifier exists in this list of keywords and hence is one. I imagine that my regular readers already have their hands in the air, saying 'hash table', for this is the obvious choice. Luckily, we already have one in our *Algorithms Alfresco* toolbox and can reuse it.

Before we use the tokenizer, then, we need to create the keyword hash table and feed it the keywords. After we've finished using the tokenizer we need to free this hash table. Sounds to me as if a class is required. Listing 6 shows this class.

From this tokenizer we could start to write a pretty formatter; however, it's going to take some doing and I'll defer it until a later article.

Infinite Regexes

Let's put aside Pascal tokenizing for now and go back to regular expressions. In February 2001's article, I presented a complete engine for parsing, compiling and then matching regular expressions. The class worked very well, at least for the regular expressions I was trying it out on. About three weeks ago, however, one of my customers was pointing out a problem with the regular expression engine in TurboPower's SysTools product. Since this is my area of expertise, I experimented with the regex and, there was no doubt about it, the engine was at fault. Since the SysTools regex engine is written using a different algorithm than the one I presented in these pages, I thought it would be instructive to try the regex out on the *Algorithms Alfresco* engine as well.

The experiment was dreadful. Just prior to running out of memory, the machine went into some big league disk thrashing as it used the swap file to try and fulfill my memory requirements. What went wrong?

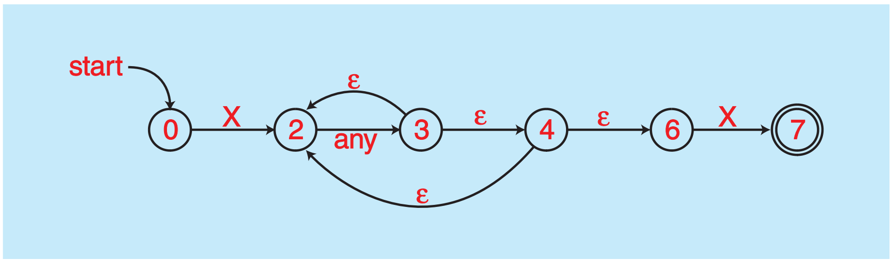
The regex concerned is `X(.+)+X` and the string we try to match is `XX` followed by about 50 non-`X` characters. The match should fail. (The test comes from *Mastering Regular Expressions* by Jeffrey E F Friedl, O'Reilly & Associates, Inc, 1997.) The regex is a little 'unusual' to say the least, but still valid. It says that a matching string should consist of an `X`, followed by one or more sets of one or more characters, followed by an `X`. Using the *Algorithms Alfresco* regex engine, the test program runs out of memory before it

➤ *Listing 5: Simple Pascal source code obfuscator.*

```
var
  F        : TFileStream;
  InF      : TaaInCharStream;
  T        : system.text;
  PrevType : TaaPascalToken;
  TokenType : TaaPascalToken;
  Token    : string;
  Count    : integer;
begin
  F := nil;
  InF := nil;
  try
    F := TFileStream.Create('AAPasTok.pas', fmOpenRead or
      fmShareDenyWrite);
    InF := TaaInCharStream.Create(F);
    System.Assign(T, 'c:\ZZPasTok.pas');
    System.Rewrite(T);
    try
      Count := 0;
      PrevType := ptEndOfFile;
      aaGetToken(InF, TokenType, Token);
      while (TokenType <> ptEndOfFile) do begin
        case TokenType of
          ptKeyword, ptIdentifier, ptNumber :
            begin
              if (PrevType = ptKeyWord) or
                 (PrevType = ptIdentifier) or
                 (PrevType = ptNumber) then begin
                write(T, ' ');
                inc(Count);
              end;
              write(T, Token);
              inc(Count, length(Token));
            end;
          ptString :
            begin
              write(T, Token);
              inc(Count, length(Token));
            end;
          ptComma :
            begin
              write(T, ',');
              inc(Count);
            end;
          ..other token types..
        end;
        if (Count >= 70) then begin
          writeln(T);
          Count := 0;
        end;
        if (TokenType <> ptComment) then
          PrevType := TokenType;
        aaGetToken(InF, TokenType, Token);
      end;
    finally
      System.Close(T);
    end;
  finally
    InF.Free;
    F.Free;
  end;
```

```
type
  TaaPascalParser = class
    private
      FInStrm   : TaaInCharStream;
      FKeywords : TaaHashTableLinear;
    protected
      procedure ppInitKeywords;
    public
      constructor Create(aInStm : TaaInCharStream);
      destructor Destroy; override;
      procedure GetToken(var aTokenType : TaaPascalToken;
var aToken : string);
  end;
const
  KeywordCount = 106;
  KeywordList : array [0..pred(KeywordCount)] of string = (
    {reserved words}
    'AND', 'ARRAY', 'AS', 'ASM', 'BEGIN', 'CASE', 'CLASS',
    'CONST', 'CONSTRUCTOR', 'DESTRUCTOR', 'DISPINTERFACE',
    'DIV', 'DO', 'DOWNTO', 'ELSE', 'END', 'EXCEPT',
    'EXPORTS', 'FILE', 'FINALIZATION', 'FINALLY', 'FOR',
    'FUNCTION', 'GOTO', 'IF', 'IMPLEMENTATION', 'IN',
    'INHERITED', 'INITIALIZATION', 'INLINE', 'INTERFACE',
    'IS', 'LABEL', 'LIBRARY', 'MOD', 'NIL', 'NOT', 'OBJECT',
    'OF', 'OR', 'OUT', 'PACKED', 'PROCEDURE', 'PROGRAM',
    'PROPERTY', 'RAISE', 'RECORD', 'REPEAT',
    'RESOURCESTRING', 'SET', 'SHL', 'SHR', 'STRING', 'THEN',
    'THREADVAR', 'TO', 'TRY', 'TYPE', 'UNIT', 'UNTIL',
    'USES', 'VAR', 'WHILE', 'WITH', 'XOR',
    {directives}
    'ABSOLUTE', 'ABSTRACT', 'ASSEMBLER', 'AUTOMATED',
    'CDECL', 'CONTAINS', 'DEFAULT', 'DISPID', 'DYNAMIC',
    'EXPORT', 'EXTERNAL', 'FAR', 'FORWARD', 'IMPLEMENTS',
    'INDEX', 'MESSAGE', 'NAME', 'NEAR', 'NODEFAULT',
    'OVERLOAD', 'OVERRIDE', 'PACKAGE', 'PASCAL', 'PRIVATE',
    'PROTECTED', 'PUBLIC', 'PUBLISHED', 'READ', 'READONLY',
    'REGISTER', 'REINTRODUCE', 'REQUIRES', 'RESIDENT',
    'SAFECALL', 'STDCALL', 'STORED', 'VIRTUAL', 'WRITE',
    'WRITEONLY',
    {others}
    'AT', 'ON'
    );
constructor TaaPascalParser.Create(aInStm :
  TaaInCharStream);
begin
  inherited Create;
  FInStrm := aInStm;
  FKeywords := TaaHashTableLinear.Create(199, AAELFHash);
  ppInitKeywords;
end;
destructor TaaPascalParser.Destroy;
begin
  FKeywords.Free;
  inherited Destroy;
end;
procedure TaaPascalParser.GetToken(var aTokenType :
  TaaPascalToken; var aToken : string);
var
  DummyObj : pointer;
begin
  AAGetToken(FInStrm, aTokenType, aToken);
  if (aTokenType = ptIdentifier) then
    if FKeywords.Find(UpperCase(aToken), DummyObj) then
      aTokenType := ptKeyword;
end;
procedure TaaPascalParser.ppInitKeywords;
var
  i : integer;
begin
  Assert(FKeywords <> nil,
    'ppInitKeywords cannot be called with nil hash table');
  for i := 0 to pred(KeywordCount) do
    FKeywords.Insert(KeyWordList[i], nil);
end;
```

➤ *Listing 6: Parsing keywords in Pascal source.*



➤ *Figure 3: The problematic NFA.*

can report that the string does not match. Why?

The first thing to note is that the algorithm isn't at fault. If I'd had enough memory the algorithm would have finished eventually. But the amount of memory required would have been much more than I could ever have slotted into the machine.

The next thing to have a look at is the NFA that is auto-generated. I have reproduced this in Figure 3. It looks simple enough: only 6 states (note that I've used the state numbers generated by the regex engine, where states 1 and 5 were optimized away). The problem comes from the fact that state 3 has a no-cost move back to 2 and also one to 4. State 4 *also* has a no-cost move back to 2 as well as one to 6. Whenever the matching logic pops state 3, it will push state 2 and state 4 to the deque. Popping state 3 will enqueue state 3. Popping state 4 will push states 6 and 2. Popping state 2 will enqueue state 3. Therefore, every time state 3 is popped, it results in state 3 being enqueued twice more. This is an exponential process. The first time we hit state 3, we enqueue state 3 twice. The next time round, those two state 3s in the deque result in four state 3s being enqueued. In the next cycle, those four multiply to eight. And so on.

Essentially, every cycle is another character from the input string. If I had 50 non-X characters, I would end up with 2^{50} state 3s enqueued in the deque. Each item in the deque is 4 bytes, so the algorithm would require 2^{52} bytes just for the deque. That's four million gigabytes or so. No wonder my machine was thrashing the swap file.

What can we do? The answer is to remove those pesky no-cost moves and convert the NFA into a DFA.

Consider the problem. State 0 in Figure 3 just goes to state 2 when the current character is an X. End of story, no problem. State 2, however, can go to state 2, or 6 when the current character is any character at all, just by following all the no-cost moves until we end up at a state that needs to match something. States 3 and 4 aren't really needed at all; they just serve as stopping points as we follow the various no-cost moves.

What we need to do is to optimize our transition table a little further. For every state in the original transition table, we need to work out all of the states it can reach via the no-cost moves. We just need to make a note of the states it can reach that actually match something. Each state will then have a list of other states, not just one, that can be reached with a match on the current character. There will be no states left that have no-cost moves. The slight problem is that we don't want to enter an infinite or exponential loop as we are analyzing the transition table: we need to mark the states as having been reached in some way.

Let's take it slowly, and optimize each state. State 0, as I have said, can only reach state 2 and that's on a match with an X. There are no no-cost moves from state 2, so there are no optimizations possible with state 0.

State 2 reaches state 3 with an any character match. Using no-cost moves from state 3, we can go back to state 2, or reach state 4. From state 4 we can reach state 2 (again) or state 6. So, ignoring the duplicate state 2, we can see that from state 2, with an any character match, we can reach state 2 or state 6.

State 3 is merely a no-cost move state. We ignore it; similarly with state 4.

State 6 just goes to state 7 on a match with an X, and there are no no-cost moves to follow from state 7 (it's a terminating state, after all). So there are no optimizations possible with state 6.

State 7 is a terminating state and therefore can have no optimizations.

We therefore have the following transition table:

```
State 0
  match X
    goto state 2
State 2
  match any char
    goto state 2
    goto state 6
State 6
  match X
    goto state 7
State 7
  stop
```

Not quite a DFA, but pretty close. The reason it's not a complete DFA is that we still have to make a choice at state 2 for an any character match: either go back to state 2 or move on to state 6. Nevertheless, it's much better than our original NFA in that there are no longer any no-cost moves, and it was the no-cost moves that were our downfall with this simple, yet bizarre, regex.

As you can see, we need to change the elements in our transition table slightly. Before this analysis, we had two 'next state' fields in the transition record. Now we have to have an array of them: we may have one, or two, or many. We could use a bare `TList` for this purpose, and typecast integers into pointers (or vice versa) when we need to, but we are likely to require an integer list class at some stage, so our best bet is to write one now (just as we did with the integer deque, in fact).

My recommendation with `TList` is to never descend from it. It was originally designed that way (in fact Danny Thorpe makes exactly the same recommendation in his excellent book, *Delphi Component Design*, now sadly out of print), and to my eyes it always looks awkward making a descendant from it. Better is to use delegation as the inheritance mechanism: write a class that has an internal `TList` instance and delegate all the list manipulation to this internal instance. On this month's companion disk is an implementation of an integer list which follows this methodology. I have not fleshed out all the methods yet, so it's a little light compared to `TList` itself; nevertheless it will be ample for our purposes.

(I was annoyed and mortified to find that Borland, in Delphi 5, decided to eviscerate the lean-and-mean `TList` in order that they could create a `TObjectList` descendant. And eviscerate it they did: `Clear` is now an O(n) operation instead of the original O(1) operation it used to be. Delete or set an item and it takes longer. And so on, and so forth. Everyone's use of `TList` suffers because someone at Borland decided to follow some misguided object oriented ideal. And don't get me started on the rest of the classes in the `Contnrs` unit: it should just be thrown away and rewritten properly. Maybe someone there will buy my book. Harrumph!)

I did implement one extra thing that `TList` does not have, though, and that is the ability to maintain the list in sorted order. If the current list is sorted, and you add another integer to the list using the `Add` method, it gets inserted into the correct place in the sequence. A further wrinkle is that the class has a property that enables or disables duplicate items (we don't want them in this application of the class).

A state record now contains the following: a transition type, a match character and character set, and a list of next states. Since we're optimizing insitu, we won't actually change the state record in any great way, we'll just redefine the second next state value to also be an integer list.

The optimization we perform is fairly easy. We visit each state in turn. We ignore states that have no-cost moves (in other words, we ignore states that don't attempt to match anything). For a state that matches something (be it an any character match, a single character match, or a match against a class, both normal and negated), we follow the link, and any no-cost links from states we reach. For each match state that we reach by this process, we add it to the 'next state' list for the state we're at.

There's an interesting catch to this, though. The no-cost move states we visit will have two possible moves (we've already unlinked the no-cost states with only one move, remember). We have to visit the possible chains from both links. But these chains may also start off with no-cost move states, and these in turn may have chains that start with no-cost states. How do we make sure that we visit *all* of them? Think of the two possible no-cost moves from such a state as being called 'left child' and 'right child'. Ring any bells? Indeed: the chains form a binary tree, and we just need to write a recursive visit method to do the work for us. Listing 7 shows this optimization code, together with the recursive walk method.

What else do we need to change? The matching code, obviously. Here, things simplify in one area, but get a little more complex in another. The good news, first: we no longer have to push any no-cost move states; there aren't any of them any more. The bad news:

```
procedure TaaRegexCompiler.rcLevel2Optimize;
var i : integer;
begin
  {cycle through all the state records,
   except for the last one}
  for i := 0 to (FTable.Count - 2) do begin
    {get this state}
    with PaaNFAState(FTable.List^[i])^ do begin
      {if it's not a no-cost move state...}
      if (sdMatchType <> mtNone) then begin
        {create the state list}
        sdNextList := TaaIntList.Create;
        {walk the chain pointed to by the first next
         state, adding the non-no-cost states to the list}
        rcWalkNoCostTree(sdNextList, sdNextState1);
      end;
    end;
  end;
end;
procedure TaaRegexCompiler.rcWalkNoCostTree(aList :
  TaaIntList; aState : integer);
begin
  {look at this state's record...}
  with PaaNFAState(FTable.List^[aState])^ do begin
    {if it's a no-cost state, recursively walk the
     first, then the second chain}
    if (sdMatchType = mtNone) then begin
      rcWalkNoCostTree(aList, sdNextState1);
      rcWalkNoCostTree(aList, sdNextState2);
    end
    {otherwise, add it to the list}
    else
      aList.Add(aState);
  end;
end;
```

➤ *Listing 7: Optimizing out all the no-cost moves.*

```
case sdMatchType of
  mtNone :
    begin
      Assert(false, 'no-cost states shouldn''t be seen');
    end;
  mtAnyChar :
    begin
      {for a match of any character, enqueue next states}
      for i := 0 to pred(sdNextList.Count) do
        Deque.Enqueue(sdNextList[i]);
    end;
  mtChar :
    begin
      {for a match of a character, enqueue next states}
      if (Ch = sdChar) then
        for i := 0 to pred(sdNextList.Count) do
          Deque.Enqueue(sdNextList[i]);
    end;
  mtClass :
    begin
      {for a match within a class, enqueue next states}
      if (Ch in sdClass^) then
        for i := 0 to pred(sdNextList.Count) do
          Deque.Enqueue(sdNextList[i]);
    end;
  mtNegClass :
    begin
      {for match not within a class, enqueue next states}
      if not (Ch in sdClass^) then
        for i := 0 to pred(sdNextList.Count) do
          Deque.Enqueue(sdNextList[i]);
    end;
  mtTerminal :
    begin
      {for a terminal state, the string successfully
       matched if the regex had no end anchor, or we're
       at the end of the string}
      if (not FAnchorEnd) or
        (StrInx > length(S)) then begin
        Result := true;
        Exit;
      end;
    end;
  mtUnused :
    begin
      Assert(false, 'unused states shouldn''t be seen');
    end;
end;
```

➤ *Listing 8: Matching with no no-cost moves.*

when we match something we have to enqueue all the next states in the state list. Actually, the bad news isn't so bad: a simple `for` loop will do the trick. The new string match code is shown in Listing 8.

After these changes, the *Algorithms Alfresco* regex engine no longer thrashed my swap file. Matching against the weird regex is now instantaneous, the way it should be.

Julian Bucknall has finally finished his book, and it should be available in all good bookstores in May 2001. He can be reached at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.