Everything Put Together Falls Apart

This month we look at parsing, compiling and using regular expressions

by Julian Bucknall

Just like an old Saturday morning movie serial, last month I deliberately left several plot threads hanging, just to make sure you renewed your subscriptions and turned up this month. I'd shown how to convey a regular expression as a table of special values, but hadn't shown you how to generate the table. I'd shown you how to show that a string was part of the language defined by a regular expression ('matching'), but had not said anything about how to do the same for a substring in a longer string (which is what the regular expression search program, grep, does). Most of all, I'd shown you the grammar for the regular expression syntax, but hadn't shown you how to parse a regular expression to validate it and compile it into a form we could use (the aforementioned table).

So there's a lot of ground to cover this month. Get seated comfortably and I'll load up the first reel.

Learn How To Fall

The first problem we'll attack is the one of parsing a given regular expression string (or *regex*, for short). In this process, our objective is merely to validate a regex string, to show that the regex follows the syntax defined by the grammar. To facilitate this, I'll reproduce the grammar from last month, especially as last month I'd forgotten to add one regex operator to the grammar: the `.` operator (which stands for a single instance of any character). Listing 1 shows the regular expression grammar. Recall that the `::=` operator means 'is defined as', `|` means OR, and that items in `<>` are identifiers in the grammar. So, the first line says that an `<expr>` is defined as either a `<term>` or as a `<term>`, followed by a `|` character, followed by another `<expr>`. The Object Pascal language is defined the same way in the Delphi Help file.

Given this grammar definition and a regex string, how can we read through the characters in the string and verify that the regex as a whole satisfies the grammar? The easiest way is to write a *top-down parser* (sometimes called a *recursive descent parser*). Providing the grammar is well defined this is a fairly easy task, as we'll see. I remember some time ago reading a message on a Delphi newsgroup. It was from someone who was complaining that he wanted to see another example of writing a recursive descent parser, apart from the standard example of reading and evaluating an arithmetic expression. Well, I hope he/she reads this magazine, because here it comes.

For top-down parsing, each of the *productions* in the grammar becomes a separate routine. (A production is one of the definitions in the grammar, that is, one of the lines that has a `::=` operator.) Take the first production in the grammar, the one for `<expr>`. Make it into a method called `ParseExpr`. (In the literature, a recursive descent parser is usually implemented as a set of interrelated routines. Because we are all ace object oriented programmers, we'll write a class instead. It actually makes things a lot easier.)

So what does `ParseExpr` do? Well, the production states that an `<expr>` is either a `<term>` on its own, or it's a `<term>` followed by the pipe character (`|`) followed by another `<expr>`. So let's assume that we have a method that parses a `<term>` called `ParseTerm`. The first thing we do either way is to call this routine to parse a `<term>`. If, on return from this routine, the current character is the pipe character, then we go ahead and call ourselves recursively (aha! the recursive bit). That's all there is to `ParseExpr`. Not too difficult, eh? Listing 2 shows this amazingly simple routine, as well as the class declaration.

Let's leave the implementation of `ParseTerm` for last (you'll see why in a moment) and proceed with `ParseFactor` to parse a `<factor>`. Again, the code is simple enough.

➤ *Listing 1: Grammar for regular expressions.*

```
<expr> ::= <term> |
           <term> '|' <expr>                  - alternation
<term> ::= <factor> |
           <factor><term>                     - concatenation
<factor> ::= <atom> |
             <atom> '?' |                     - zero or one
             <atom> '*' |                     - zero or more
             <atom> '+'                       - one or more
<atom> ::= <char> |
           '.' |                              - any char
           '(' <expr> ') |                    - parentheses
           '[' <charclass> ']' |              - normal class
           '[^' <charclass> ']'               - negated class
<charclass> ::= <charrange> |
                <charrange><charclass>
<charrange> ::= <ccchar> |
                <ccchar> '-' <ccchar>
<char> ::= <any character except metacharacters> |
           '\' <any character at all>
<ccchar> ::= <any character except '-' and ']'> |
             '\' <any character at all>
```

The first thing is to parse an <atom> by calling ParseAtom, and then check for one of the three metacharacters *, + or ?. (A *metacharacter* is a character that has special meaning within the grammar, for example, the asterisk, the plus sign, the parentheses, and so on. Other characters have no special meaning.)

The first part of Listing 3 shows this method. ParseAtom is again fairly trivial to code: it's either a <char> or it's a period; it's an open parenthesis, followed by an <expr>, followed by the close parenthesis; it's an open bracket, followed by a <charclass>, followed by a close bracket; or it's an open bracket, followed by a caret, followed by a <charclass>, followed by a close bracket. We code it in exactly that form as shown by the second method in Listing 3. The other methods that implement the other productions are equally as trivial. Notice that it's the very lowest methods that have the actual validation in them. For example, ParseAtom will check that a close parenthesis is present after parsing the open parenthesis and the <expr>. ParseChar checks that the current character is not a metacharacter. And so on, so forth.

All I've done for this particular parser is to write the current grammar item to the console and to raise an exception if we reach a point where we can determine whether the input regex string is invalid. Neither of these things

➤ *Listing 2: The parse <expr> method and class definition.*

```
type
  TaaRegexParser = class
    private
      FRegexStr : string;
      FPosn     : PAnsiChar;
    protected
      procedure rpParseAtom;
      procedure rpParseCCChar;
      procedure rpParseChar;
      procedure rpParseCharClass;
      procedure rpParseCharRange;
      procedure rpParseExpr;
      procedure rpParseFactor;
      procedure rpParseTerm;
    public
      constructor Create(const aRegexStr : string);
      destructor Destroy; override;
      function Parse(var aErrorPos : integer) : boolean;
  end;
procedure TaaRegexParser.rpParseExpr;
begin
  rpParseTerm;
  if (FPosn^ = '|') then begin
    inc(FPosn);
    writeln('alternation');
    rpParseExpr;
  end;
end;
```

➤ *Listing 3: Remaining simple parsing methods in parser class.*

```
procedure TaaRegexParser.rpParseFactor;
begin
  rpParseAtom;
  case FPosn^ of
    '?' : begin
            inc(FPosn);
            writeln('zero or one');
          end;
    '*' : begin
            inc(FPosn);
            writeln('zero or more');
          end;
    '+' : begin
            inc(FPosn);
            writeln('one or more');
          end;
  end;{case}
end;
procedure TaaRegexParser.rpParseAtom;
begin
  case FPosn^ of
    '(' : begin
            inc(FPosn);
            writeln('open paren');
            rpParseExpr;
            if (FPosn^ <> ')') then
              raise Exception.Create(
              'Regex error: expecting a closing '+
              'parenthesis');
            inc(FPosn);
            writeln('close paren');
          end;
    '[' : begin
            inc(FPosn);
            if (FPosn^ = '^') then begin
              inc(FPosn);
              writeln('negated char class');
              rpParseCharClass;
            end else begin
              writeln('normal char class');
              rpParseCharClass;
            end;
            inc(FPosn);
          end;
    '.' : begin
            inc(FPosn);
            writeln('any character');
          end;
  else
    rpParseChar;
  end;{case}
end;
procedure TaaRegexParser.rpParseCCChar;
begin
  if (FPosn^ = #0) then
    raise Exception.Create('Regex error: expecting a '+
    'normal character, found null terminator');
  if FPosn^ in [']', '-'] then
    raise Exception.Create('Regex error: expecting a '+
    'normal character, ie found a metacharacter');
  if (FPosn^ = '\') then begin
    inc(FPosn);
    writeln('escaped ccchar ', FPosn^);
    inc(FPosn);
  end else begin
    writeln('ccchar ', FPosn^);
    inc(FPosn);
  end;
end;
procedure TaaRegexParser.rpParseChar;
begin
  if (FPosn^ = #0) then
    raise Exception.Create('Regex error: expecting a '+
    'normal character, found null terminator');
  if FPosn^ in MetaCharacters then
    raise Exception.Create('Regex error: expecting a '+
    'normal character, ie found a metacharacter');
  if (FPosn^ = '\') then begin
    inc(FPosn);
    writeln('escaped char ', FPosn^);
    inc(FPosn);
  end else begin
    writeln('char ', FPosn^);
    inc(FPosn);
  end;
end;
procedure TaaRegexParser.rpParseCharClass;
begin
  rpParseCharRange;
  if (FPosn^ <> ']') then
    rpParseCharClass;
end;
procedure TaaRegexParser.rpParseCharRange;
begin
  rpParseCCChar;
  if (FPosn^ = '-') then begin
    inc(FPosn);
    writeln('--range to--');
    rpParseCCChar;
  end;
end;
```

would be done in a production environment, of course: the first because our goal is to build the transition table for the regex string (compile it, if you like), the second because we shouldn't use exceptions for validation as it's too inefficient. The code does show the structure and design of a simple top-down parser: you design the grammar and then convert it into code in this fairly trivial fashion.

Suddenly, writing a recursive descent parser doesn't seem all that mysterious after all!

The one fly in the ointment is the `ParseTerm` method. Compared with what we've just done, it's a little more complicated. The problem is that the production says that a `<term>` is either a `<factor>` or a `<factor>` followed by another `<term>` (that is, concatenation). There is no operator that links the two, such as the plus sign. If there were, we could easily write `ParseTerm` in the same manner as all the other `ParseXxx` methods. However, since there is no metacharacter for concatenation, we have to use another trick.

Consider the problem here. Suppose we were parsing the regular expression `ab`. We would parse it as an `<expr>`, which means parsing it as a `<term>`, then a `<factor>`, then an `<atom>`, then a `<char>`. That takes care of the `a` part. We go back up the grammar until we reach `<term>` again, which says that after the first `<factor>` we can have another `<term>`. Proceeding down the productions again, we parse the `b` as a `<char>` again, and we're done.

Sounds simple enough, so where's the problem? Do the same for `(a)`. This time we go down the productions until we reach the point where it says that an `<atom>` could consist of a `(`, followed by an `<expr>`, followed by a `)`. So the `(` is taken care of and we start over at the top of the grammar parsing an `<expr>`. Wander down again: `<expr>`, then `<term>`, then `<factor>`, then `<atom>`, then `<char>` and that takes care of the `a`. On the way up again, we encounter the alternative for the `<term>` production. So, why don't we take the alternative this time and try and parse a concatenation? Well, duh, because this time the current character is a `)`. In the first example we decided to parse a concatenation because the current character was a `b` and this time we don't because the current character is a `)`. Do you see the point I'm trying to make here? We take a quick peek at the current character before deciding whether to parse another concatenated `<term>` or not. If it could be counted as the start of another `<atom>` then we go ahead and parse it as such. If not, we assume that someone else (that is, a caller method) will do something with it and that there is no concatenation.

We are, at this point, going to have to 'break the grammar' to see what to do next. We are going to have to assume that, if there is concatenation, the current character will serve as the starting character for an `<atom>`. In other words, if the current character is a `.`, a `(`, a `[`, or an ordinary character, we shall parse another `<term>`. If not, we assume there is no concatenation and exit the `ParseTerm` method. We are using the information for the `<atom>` production, a 'lower' production, to determine what to do about the `<term>` production, a 'higher' production. Again, it bears repeating that this is only necessary because we don't have a concatenation metacharacter.

Listing 4 shows the resulting `ParseTerm` method.

```
procedure TaaRegexParser.rpParseTerm;
begin
  rpParseFactor;
  if (FPosn^ = '(') or (FPosn^ = '[') or (FPosn^ = '.') or
    ((FPosn^ <> #0) and not (FPosn^ in MetaCharacters)) then
    rpParseTerm;
end;
function TaaRegexParser.Parse(var aErrorPos : integer) : boolean;
begin
  Result := true;
  aErrorPos := 0;
  FPosn := PAnsiChar(FRegexStr);
  try
    rpParseExpr;
    if (FPosn^ <> #0) then begin
      Result := false;
      aErrorPos := FPosn - PAnsiChar(FRegexStr) + 1;
    end;
  except
    on E:Exception do begin
      Result := false;
      aErrorPos := FPosn - PAnsiChar(FRegexStr) + 1;
    end;
  end;
end;
```

➤ *Listing 4: The parse <term> method and the interfaced Parse method.*

Run That Body Down

So, at this point in the game, we have seen how to parse a regular expression by taking its grammar definition and doing some fairly trivial conversion steps to implement it in code. Now we need to look at the next step: generating the transition table for the non-deterministic finite automaton (NFA) that represents the regular expression; that is, compiling it.

Since we don't know beforehand how big the transition table is going to be, we shall use a `TList` to hold it. Doing so means that we can take advantage of its growing capabilities and don't have to worry about predefining the table's size and getting it wrong. For efficiency's sake, we shall preset the list's capacity to 64 elements: this will avoid too much growing when parsing simple regex strings.

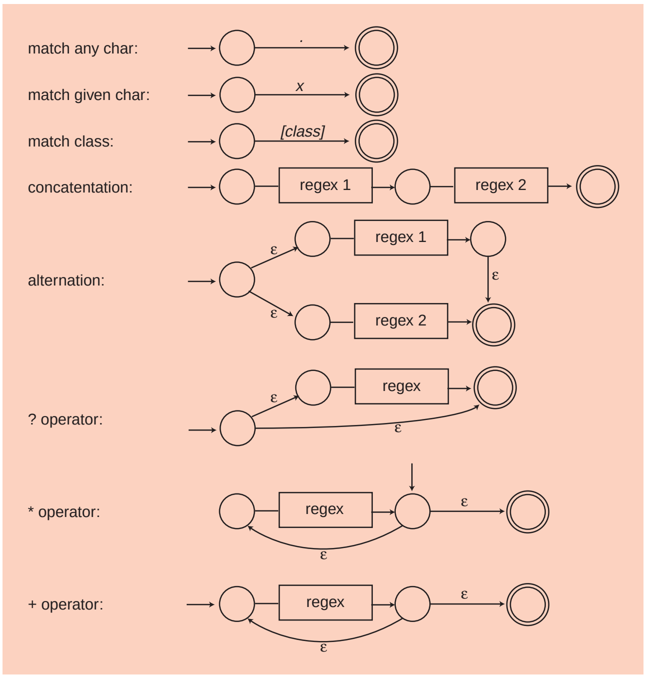
Let's take this slowly. We shall use last month's figures on constructing NFA diagrams as our guide, which I've reproduced as Figure 1.

The simplest case is the expression that recognizes a single character. As you see from the first image in Figure 1, we need a start state, which will recognize the character, and it will have a single link to the end state, we'll need one of those too. We'll write a simple routine that will create a new state (as a record) and append it to our transition table. Listing 5 shows this simple method: as you can see, it takes in a match type, a character, a pointer to a character class, and two links to other states. Not all of these parameters will be required for every state we want to create of course, but it makes it easier to have one method that can create any type of state record than a whole bunch of them, one for each type of state we may need.

From the figure it seems as if we need to create two new states for this simple character recognizer. Actually, we can get away with only creating one, the start state, and assume that the end state is the next state to be added to the list. We leave it as a 'virtual' end state. If we do this with every parsing routine, we may be able to get away with making the end state equal to the start state of another subexpression. Let's see how we do; from now on all parsing routines will return their start state, and we'll assume that the end state, if it really existed, would be the next state to be added to the transition table.

From Listing 5, if we pass the special state number `NewFinalState` as a next state number, you can see that we actually set the link to the index of the next item to be added to the transition table. This item doesn't exist yet, of course, but we're assuming that it will or that something else will come along and patch a new link in.

Anyway, Listing 6 shows the parsing method to recognize a single character. Notice how we've reengineered the original character parsing method. The first thing is that we don't raise any exceptions on errors any more, instead we return a special state number: `ErrorState`. We also track the error code for any error that occurred. If there were an error, we would add a new state to the transition table and return it as the function result. This is, of course, the start state for this expression.



➤ *Figure 1: Constructing an NFA from a regular expression.*

That was easy enough, so let's look at another, more complex, parsing method: the one that parses an atom. The first case, the parenthesized expression, is pretty much the same as before: we don't need to add any states for this. The second case, the character class or the negated one, is definitely one that needs a new state machine. We parse the character class as before (by treating it as a set of ranges, each of which can be a single character or two characters separated by a dash). This time, however, we must record the characters in the class. We use a set of characters allocated on the heap for this purpose. The final step is to add a new state to the transition table that recognizes this character class, much as we did for the character recognizer. The final case, apart from the single character we've already discussed, is the state machine for the 'any character' operator, the period. This is pretty simple: create a new state that matches any character. The complete

➤ *Listing 5: Adding a state record to the transition table.*

```
function TaaRegexCompiler.rcAddState(aMatchType : TaaNFAMatchType; aChar : char;
  aCharClass : PaaCharSet; aNextState1: integer; aNextState2: integer) : integer;
var
  StateData : PaaNFAState;
begin
  {create the new state record}
  New(StateData);
  {set up the fields in the state record}
  if (aNextState1 = NewFinalState) then
    StateData^.sdNextState1 := succ(FTable.Count)
  else
    StateData^.sdNextState1 := aNextState1;
  StateData^.sdNextState2 := aNextState2;
  StateData^.sdMatchType := aMatchType;
  if (aMatchType = mtChar) then
    StateData^.sdChar := aChar
  else if (aMatchType = mtClass) or (aMatchType = mtNegClass) then
    StateData^.sdClass := aCharClass;
  {add the new state}
  Result := FTable.Count;
  FTable.Add(StateData);
end;
```

➤ *Listing 6: Creating the state machine for a single character.*

```
function TaaRegexCompiler.rcParseChar : integer;
begin
  {if we hit the end of the string, it's an error}
  if (FPosn^ = #0) then begin
    Result := ErrorState;
    FErrorCode := recSuddenEnd;
    Exit;
  end;
  {if the current char is one of the metacharacters, it's an error}
  if FPosn^ in MetaCharacters then begin
    Result := ErrorState;
    FErrorCode := recMetaChar;
    Exit;
  end;
  {otherwise add a state for the character}
  {..if it's an escaped character: get the next character instead}
  if (FPosn^ = '\') then
    inc(FPosn);
  Result := rcAddState(mtChar, FPosn^, nil, NewFinalState, UnusedState);
  inc(FPosn);
end;
```

```
function TaaRegexCompiler.rcParseAtom : integer;
var
  MatchType : TaaNFAMatchType;
  CharClass : PaaCharSet;
begin
  case FPosn^ of
    '(' :
      begin
        {move past the open parenthesis}
        inc(FPosn);
        {parse a complete regex between the parentheses}
        Result := rcParseExpr;
        if (Result = ErrorState) then
          Exit;
        {if the current character is not a close
          parenthesis, there's an error}
        if (FPosn^ <> ')') then begin
          FErrorCode := recNoCloseParen;
          Result := ErrorState;
          Exit;
        end;
        {move past the close parenthesis}
        inc(FPosn);
      end;
    '[' :
      begin
        {move past the open square bracket}
        inc(FPosn);
        {if the first character in the class is a '^' then
          the class if negated, otherwise it's a normal one}
        if (FPosn^ = '^') then begin
          inc(FPosn);
          MatchType := mtNegClass;
        end else begin
          MatchType := mtClass;
        end;
        {allocate the class character set and parse the
          character class; this will return either with an
          error, or when the closing square bracket is
          encountered}
        New(CharClass);
        CharClass^ := [];
        if not rcParseCharClass(CharClass) then begin
          Dispose(CharClass);
          Result := ErrorState;
          Exit;
        end;
        {move past the closing square bracket}
        inc(FPosn);
        {add a new state for the character class}
        Result := rcAddState(MatchType, #0, CharClass,
          NewFinalState, UnusedState);
      end;
    '.' :
      begin
        {move past the period metacharacter}
        inc(FPosn);
        {add a new state for the 'any character' token}
        Result := rcAddState(mtAnyChar, #0, nil,
          NewFinalState, UnusedState);
      end;
  else
    {otherwise parse a single character}
    Result := rcParseChar;
  end;{case}
end;
```

➤ *Listing 7: Creating the state machine for an <atom>.*

listing for the atom parser is shown in Listing 7. Again, the start state for these expressions is returned as the function result and the end state is the virtual end state.

We seem to be going great guns here but, to be honest, none of it so far is difficult. Let's attack a more challenging state machine now: the alternation operator, `|`.

So far we've been creating states without any reference to each other, but if you look at the NFA construction diagram for the `OR` operator, you'll see that we need to finally do some joining up, *à la* Lego. We need to save the start states for each subexpression. We need to create a new start state that will have no-cost links to each of these two start states. The final state of the first subexpression must be linked to the final state of the second, which then becomes the final state of the alternation expression.

But hold on there. The final state for the first expression *does not exist*, remember? So we'll have to create one. But if we create one, won't it mess up another state machine since we'll then have one or more states pointing to a state that they're not supposed to? Not if we're careful.

The first thing we must do, of course, is to parse the initial `<term>`. We'll get back the start state (so we save it in a variable) and we know that the final state is the virtual end state just beyond the end of the list. If the next character is a | then we know that we're parsing an alternation clause and that we should be parsing another `<expr>`. It's now that we have to take things carefully. The first thing we do is to create a state for the end state of that initial `<term>`. We don't care at present where its links point, we'll patch that up in a moment. Creating this end state now also means that whichever states in the `<term>` point to the virtual end state will in fact point to the state we just made real. Now we shall create the alternation start state. We know one of the links, it's to the initial `<term>`, but we don't know the other yet; after all, we haven't parsed the second `<expr>` yet. Now we can parse the second `<expr>`. We'll get back a

➤ *Listing 8: Creating the state machine for an <expr>.*

```
function TaaRegexCompiler.rcParseExpr : integer;
var
  StartState1 : integer;
  StartState2 : integer;
  EndState1   : integer;
  OverallStartState : integer;
begin
  {assume the worst}
  Result := ErrorState;
  {parse an initial term}
  StartState1 := rcParseTerm;
  if (StartState1 = ErrorState) then
    Exit;
  {if the current character is *not* a pipe character, no
    alternation is present so return the start state of the
    initial term as our start state}
  if (FPosn^ <> '|') then
    Result := StartState1
  {otherwise, we need to parse another expr and join the two
    together in the transition table}
  else begin
    {advance past the pipe}
    inc(FPosn);
    {the initial term's end state does not exist yet
      (although there is a state in the term that points to
      it), so create it}
    EndState1 := rcAddState(mtNone, #0, nil, UnusedState,
      UnusedState);
    {for the OR construction we need a new initial state: it
      will point to the initial term and the second
      just-about-to-be-parsed expr}
    OverallStartState := rcAddState(mtNone, #0, nil,
      UnusedState, UnusedState);
    {parse another expr}
    StartState2 := rcParseExpr;
    if (StartState2 = ErrorState) then
      Exit;
    {alter the state state for the overall expr so that the
      second link points to the start of the second expr}
    Result := rcSetState(OverallStartState, StartState1,
      StartState2);
    {now set the end state for the initial term to point to
      the final end state for the second expr and the
      overall expr}
    rcSetState(EndState1, FTable.Count, UnusedState);
  end;
end;
```

start state that we use to patch up the second link in the alternation start state. The new virtual end state can be used to link up from the initial `<term>`'s end state.

And that's it! After all these shenanigans, we had to create two new states (the first being the start state for the alternation, the second being the end state for the initial `<term>`), and we were careful enough so that the virtual end state of the second `<expr>` was the virtual end state of the overall alternation. Listing 8 shows this bit of intricacy (notice I wrote another method to help out that sets the links for a state after it was created).

Take Me To The Mardi Gras

Having seen this particular construction, creating the state machines for the three closures (the `*`, `+` and `?` operators) is equally simple, providing we are careful about the order in which we create the states. Follow along in Listing 9.

For the zero or one closure (`?` operator), we need to create the end state for the atom's expression to which we're applying the operator, and we need to create a start state for the overall state machine. These new states are linked up as shown in Figure 1.

For the zero or more closure (`*` operator), it's even easier: we just need to create the end state for the atom. This then becomes the start state for the overall expression. The virtual end state is the end state for the expression.

For the one or more closure (`+` operator), create the end state for the atom and link it to the start state for the atom (which is also the start state for the expression). The virtual end state is again the end state for the expression.

So, what's left? Our old friend, concatenation, that's what. It looks easy in Figure 1: the end state for the first regex becomes the start state for the second and, bingo, they're linked. In practice, it's not quite so easy. The end state for the first expression is the virtual end state, and there's no guarantee that this will be equal to the start state of the next expression (in which case, they would be automatically linked). No, there's nothing for it but to create the end state for the first expression and link it to the second's start state. Listing 10 shows the final piece of the jigsaw, including the creation of the terminal state.

So, now we have it: we can parse a regular expression and build up a transition table that implements the NFA for that regular expression.

Hobo's Blues

Once I got to this point in my article, I felt some dissatisfaction with the code. Why? Well, mainly it was because of the problem of adding that extra state to solve the concatenation problem. If you recall, we had to do that so that the first expression could be linked to the second. We could not be sure that the number of the second expression's start state was equal to the number of the first expression's end state. Essentially we added a 'do-nothing' state consisting of a single no-cost move to another state. When we use the transition table to match a string, using the code I presented last month, we would then have a whole bunch of useless pushes of these 'do nothing' states onto our deque; by 'useless' I mean they really serve no purpose since, once popped, all they would do is cause another state to be pushed. So I thought about getting rid of them as an optimization measure.

Initially, I was concerned about deleting state records from the transition table and messing up all the various transitions between states. I'd even started thinking about having a linked list structure for the transition table. Then it became clear: leave the do-nothing states where they were and just skip over them. In the end, implementing this optimization turned out to be remarkably easy. I visited each state record in the transition table. For each of these records I checked to see whether the first next state was a do-nothing state (that is, a no matching state with a single no-cost move). If so, I'd replace the link to the do-nothing state with the do-nothing state's next link. Repeat this process for a given state until the first next state is no longer a do-nothing state. If the state's second next state is in use, do the same for that. Listing 11 shows this routine. Notice that the

➤ *Listing 9: Creating the state machine for a <factor>.*

```
function TaaRegexCompiler.rcParseFactor : integer;
var
  StartStateAtom : integer;
  EndStateAtom   : integer;
begin
  {assume the worst}
  Result := ErrorState;
  {first parse an atom}
  StartStateAtom := rcParseAtom;
  if (StartStateAtom = ErrorState) then
    Exit;
  {check for a closure operator}
  case FPosn^ of
    '?' : begin
            {move past the ? operator}
            inc(FPosn);
            {the atom's end state doesn't exist yet, so
              create one}
            EndStateAtom := rcAddState(mtNone, #0, nil,
              UnusedState, UnusedState);
            {create a new start state for the overall regex}
            Result := rcAddState(mtNone, #0, nil,
              StartStateAtom, EndStateAtom);
            {make sure the new end state points to the next
              unused state}
            rcSetState(EndStateAtom, FTable.Count,
              UnusedState);
          end;
    '*' : begin
            {move past the * operator}
            inc(FPosn);
            {the atom's end state doesn't exist yet, so
              create one; it'll be the start of the overall
              regex subexpression}
            Result := rcAddState(mtNone, #0, nil,
              NewFinalState, StartStateAtom);
          end;
    '+' : begin
            {move past the + operator}
            inc(FPosn);
            {the atom's end state doesn't exist yet, so
              create one}
            rcAddState(mtNone, #0, nil, NewFinalState,
              StartStateAtom);
            {the start of the overall regex subexpression
              will be the atom's start state}
            Result := StartStateAtom;
          end;
  else
    Result := StartStateAtom;
  end;{case}
end;
```

```
function TaaRegexCompiler.rcParseTerm : integer;
var
  StartState2 : integer;
  EndState1   : integer;
begin
  {parse an initial factor, the state number returned will
    also be our return state number}
  Result := rcParseFactor;
  if (Result = ErrorState) then
    Exit;
  if (FPosn^ = '(') or (FPosn^ = '[') or (FPosn^ = '.') or
    ((FPosn^ <> #0) and not (FPosn^ in MetaCharacters))
       then begin
    {initial factor's end state doesn't exist yet
      (although there is a state in the term that points to
      it), so create it}
    EndState1 := rcAddState(mtNone, #0, nil, UnusedState,
      UnusedState);
    {parse another term}
    StartState2 := rcParseTerm;
    if (StartState2 = ErrorState) then begin
      Result := ErrorState;
      Exit;
    end;
    {join the first factor to the second term}
    rcSetState(EndState1, StartState2, UnusedState);
  end;
end;
function TaaRegexCompiler.Parse(var aErrorPos : integer;
  var aErrorCode: TaaRegexError) : boolean;
begin
  rcClear;  {clear the current transition table}
  {empty regex strings are not allowed}
  if (FRegexStr = '') then begin
    Result := false;
    aErrorPos := 1;
    aErrorCode := recSuddenEnd;
    Exit;
  end;
  {parse the regex string}
  FPosn := PAnsiChar(FRegexStr);
  FStartState := rcParseExpr;
  {if error occurred or we're not at end of regex string,
   clear transition table, return false and error position}
  if (FStartState = ErrorState) or (FPosn^ <> #0) then begin
    if (FStartState <> ErrorState) and (FPosn^ <> #0) then
      FErrorCode := recExtraChars;
    rcClear;
    Result := false;
    aErrorPos := succ(FPosn - PAnsiChar(FRegexStr));
    aErrorCode := FErrorCode;
  end else begin
    {otherwise add a terminal state, optimize, return true}
    rcAddState(mtTerminal, #0, nil, UnusedState,
      UnusedState);
    Result := true;
    aErrorPos := 0;
    aErrorCode := recNone;
  end;
end;
```

➤ *Listing 10: Creating the <term> state machine and the interfaced calling method.*

final part of the routine is not strictly necessary (the part that sets the do-nothing states to 'unused') but it makes it easier for assertion code to be inserted into the string matching method, these states should not be reached.

Was A Sunny Day

Now that we can create a transition table from a regular expression, we can use the code from last month to see whether an input string matches the regular expression exactly. One of the things we would like to do, however, is not match the *entire* string to the regular expression but to only match *part* of the string and to obtain the position of that matching substring. There are a couple of things we need to do to enable this particular functionality.

The first of these is to recognize that sometimes we *would* like to match the whole string. We therefore introduce two new regex operators to enable us to do just that: the anchor operators `^` and `$`. The caret means that the matching must occur from the beginning of the string. The dollar sign means that the matching must occur all the way to the end of the string. Thus, for example, the regular expression `^function` means 'match the word *function* at the beginning of the string,' whereas `^end.$` means 'the entire string should just consist of the characters *e*, *n*, *d* and the full stop. No spaces, no other characters'. The `^` and `$` can only appear at the start and at the end of the regular expression respectively; they cannot occur anywhere else within the regex.

This entails a change to our grammar: not too drastic but, as we've seen, codifying the grammar properly makes writing the code much easier. The new rule is shown in Listing 12, together with the relevant parsing method. The interfaced `Parse` method is changed to call this method instead of the original, of course.

We can now change the string matching code from the previous article to match substrings as well as complete strings. If the regex starts with a `^` then we just try and match the entire string. If not, then we try and match each of the substrings formed from the original string. This is simple enough: we change the matching code to accept not only the string itself, but also a starting position. The initial routine in Listing 13 shows the interfaced method for matching a string. As you can see, depending on the presence of the start anchor, we either call the

➤ *Listing 11: Optimizing a transition table.*

```
procedure TaaRegexCompiler.rcLevel1Optimize;
var
  i : integer;
  Walker : PaaNFAState;
begin
  {cycle through all state records, except for last one}
  for i := 0 to (FTable.Count - 2) do begin
    {get this state}
    with PaaNFAState(FTable.List^[i])^ do begin
      {walk the chain pointed to by the first next state,
        unlinking the states that are simple single no-cost
        moves}
      Walker := PaaNFAState(FTable.List^[sdNextState1]);
      while (Walker^.sdMatchType = mtNone) and
        (Walker^.sdNextState2 = UnusedState) do begin
        sdNextState1 := Walker^.sdNextState1;
        Walker := PaaNFAState(FTable.List^[sdNextState1]);
      end;
      {walk the chain pointed to by the first next state,
        unlinking the states that are simple single no-cost
        moves}
      if (sdNextState2 <> UnusedState) then begin
        Walker := PaaNFAState(FTable.List^[sdNextState2]);
        while (Walker^.sdMatchType = mtNone) and
              (Walker^.sdNextState2 = UnusedState) do begin
          sdNextState2 := Walker^.sdNextState1;
          Walker := PaaNFAState(FTable.List^[sdNextState2]);
        end;
      end;
    end;
  end;
  {cycle through all the state records, except for the last
    one, marking unused ones--not strictly necessary but
    good for debugging}
  for i := 0 to (FTable.Count - 2) do begin
    with PaaNFAState(FTable.List^[i])^ do begin
      if (sdMatchType = mtNone) and
         (sdNextState2 = UnusedState) then
        sdMatchType := mtUnused;
    end;
  end;
end;
```

workhorse method to match the entire string, or we go through the original string trying a match at every position.

What about the `$` operator, then? Here, we need to change the matching code. The method in Listing 14 shows the modified matching code. How has this changed from before? For a start we are now passing in the starting position in the string. Also we have an explicit terminating state this time (last time, it was merely a no-match state with no next state links). Take a look at the case switch clause for this terminating state. We only accept the terminating state as indicating a match if the regular expression had no ending anchor, or if we managed to reach the end of the string. If either of these conditions were not met, the terminating state would be ignored.

What else? All the code I've presented so far is case-sensitive. If the regex has a subexpression matching an a, then only `a` will match it, `A` will not. A simple change then is to allow for case-insensitive regular expression matching. This is not too arduous a task, but has a hidden catch that could cause a stumbling block. An initial stab at this would be to uppercase the regular expression string just before parsing and compiling it, and to uppercase the match string just before attempting to match it against the compiled regex. For the vast majority of regexes this would work perfectly well. However, consider the regex `[A-z]+`. If we uppercased this regular expression prior to parsing it we would be parsing `[A-Z]+`, a very different animal than before. (Why? Well there are punctuation marks in the range `A-z`, but there aren't in the range `A-Z`.)

A better strategy would be to uppercase individual characters as we came across them in the parsing process. When we populate a character set to represent a character class, we not only add all the characters in the original range, we add the uppercased versions of them as well. For the match string, we can uppercase letters as we come across them, or we could uppercase the whole string; it doesn't matter much.

On this month's disk, you will find the complete regular expression compiler code to parse a regular expression, to build the transition table, to optimize the table, and finally to match given strings to the regular expression.

```
<anchorexpr> ::= <expr> |
                 '^' <expr> |
                 <expr> '$' |
                 '^' <expr> '$'
function TaaRegexCompiler.rcParseAnchorExpr : integer;
begin
  {check for an initial '^'}
  if (FPosn^ = '^') then begin
    FAnchorStart := true;
    inc(FPosn);
  end;
  {parse an expression}
  Result := rcParseExpr;
  {if we were successful, check for the final '$'}
  if (Result <> ErrorState) then begin
    if (FPosn^ = '$') then begin
      FAnchorEnd := true;
      inc(FPosn);
    end;
  end;
end;
```

➤ *Listing 12: Parsing the anchors.*

➤ *Listing 13: The interfaced method for matching a string.*

```
function TaaRegexCompiler.MatchString(const S : string) : integer;
var
  i : integer;
  ErrorPos  : integer;
  ErrorCode : TaaRegexError;
begin
  {try and see if the string matches (empty strings don't)}
  Result := 0;
  if (S <> '') then
    {if the regex specified a start anchor, then we only need to check
     the string starting at the first position}
    if FAnchorStart then begin
      if rcMatchSubString(S, 1) then
        Result := 1;
    end else begin
      {otherwise we try and match the string at every position and
       return at the first success}
      for i := 1 to length(S) do
        if rcMatchSubString(S, i) then begin
          Result := i;
          Break;
        end;
    end;
end;
```

Something So Right

If you take a look at this source, you'll see a lot more code than has been printed in this article. Not extra functionality, mind you, but debugging code. A couple of months ago I read *Debugging Applications* by John Robbins and much of what he said made a lot of sense to me, especially with regard to assertions and tracing code helping you debug. His view was that assertions were good to have, not only during the original coding process, but also afterwards, way afterwards when you've forgotten how the code works. Assertions are like active comments in a way: they encode your thought processes about how the code should work, what invariants you're assuming, the invalid values you can't accept for parameters, and so on.

Tracing is a different kettle of fish. Originally I wrote the regex parser class before the compiler class. In running the parser class, it was obvious if something went wrong: I could see the debug print on the screen. For the compiler class, I took the parser and removed the `writeln`s to the console window and replaced them with the code that created state records for the transition table. Needless to say something went wrong during development, and I had to put equivalents back in to help me debug. Better still this time, I wrote them to a log file that I could peruse at leisure. I also added code that dumped the transition table to the same log file on a

```
function TaaRegexCompiler.rcMatchSubString(const S : string;
  StartPosn : integer) : boolean;
var
  Ch     : char;
  State  : integer;
  Deque  : TaaIntDeque;
  StrInx : integer;
begin
  {assume we fail to match}
  Result := false;
  {create the deque}
  Deque := TaaIntDeque.Create(64);
  try
    {enqueue the special value to start scanning}
    Deque.Enqueue(MustScan);
    {enqueue the first state}
    Deque.Enqueue(FStartState);
    {prepare the string index}
    StrInx := StartPosn - 1;
    {loop until the deque is empty or we run out of string}
    while (StrInx <= length(S)) and not Deque.IsEmpty
      do begin
      {pop the top state from the deque}
      State := Deque.Pop;
      {process the "must scan" state first}
      if (State = MustScan) then begin
        if not Deque.IsEmpty then begin
          inc(StrInx);
          if (StrInx <= length(S)) then begin
            Ch := S[StrInx];
            Deque.Enqueue(MustScan);
          end;
        end;
      end else with PaaNFAState(FTable.List^[State])^
        do begin
        {otherwise, process the state}
        case sdMatchType of
          mtNone :
            begin
              Deque.Push(sdNextState2);
              Deque.Push(sdNextState1);
            end;
          mtAnyChar :
            begin
              Deque.Enqueue(sdNextState1);
            end;
          mtChar :
            begin
              if (Ch = sdChar) then
                Deque.Enqueue(sdNextState1);
            end;
          mtClass :
            begin
              if (Ch in sdClass^) then
                Deque.Enqueue(sdNextState1);
            end;
          mtNegClass :
            begin
              if not (Ch in sdClass^) then
                Deque.Enqueue(sdNextState1);
            end;
          mtTerminal :
            begin
              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;
      end;
    end;
    while not Deque.IsEmpty do begin
      State := Deque.Pop;
      with PaaNFAState(FTable.List^[State])^ do begin
        case sdMatchType of
          mtNone :
            begin
              Deque.Push(sdNextState2);
              Deque.Push(sdNextState1);
            end;
          mtTerminal :
            begin
              if (not FAnchorEnd) or (StrInx > length(S))
                then begin
                Result := true;
                Exit;
              end;
            end;
        end;{case}
      end;
    end;
  finally
    Deque.Free;
  end;
end;
```

➤ *Listing 14: The final code for matching a substring.*

successful parse. This extra code is called tracing code. The code dumps information to the screen or a file, detailing various values of importance, tracing the code flow. It's well worth keeping it in there, isolated by a compiler define, for that point in the future when something goes wrong and you need it again.

Needless to say I shall be making sure that my future coding has more of this kind of thing, and I encourage you to experiment and do the same.

Before I sign off, I've an extra comment to make about the matching process that I've been concentrating on over these last couple of articles. Those of you who've used regexes for a while will be used to thinking about the greediness of the algorithm. Greediness? Let's explain it like this. Suppose you are using the regex `[a-z]*a`, that is, zero or more letters followed by an `a`. The string you wanted to test for matching is `banana`. Would you match `ba`, `bana`, or the whole word? Does it actually matter?

In one sense, no, it doesn't matter in the least. All you get is an indication that the string matched, which (generally) is all you wanted. In another sense, maybe it does. Suppose you were writing a regex search and replace. Here it would matter if you were talking about matching `ba`, `bana` or `banana`. The normal algorithms implemented by regex searchers (like grep, or the one embedded in the search dialog in the Delphi IDE) are called 'greedy': they will try and find the longest string that matches. The algorithm greedily gobbles up letters until it finds the longest match. This is the classic backtracking algorithm I described last week. Our algorithm, on the other hand, is not greedy: it favours speed over finding the longest match. It will terminate as soon as it identifies a path from the start state to the end state. In our example, it will terminate after finding `ba`, preferring to report a success after this short string instead of waiting to see if it could find a longer one. Altering the code so that it becomes greedy is just a matter of calculating *all* possible matches and selecting the longest. However, doing *that* is liable to make things very slow.

That's all for now. I hope you enjoyed these two forays into automata and string searching through regular expressions.

Julian Bucknall exhorts you not to call him Al, Paul, or Simon. 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.
© Julian M Bucknall, 2001