The Man-Machine

This month we look at finite automata or state machines

by Julian Bucknall

There are some times when the weekend during which I've decided to write my next column looms and I have no idea what I want to write about. That dread condition called Writer's Block rears its ugly head. I flip through algorithm books to no avail; nothing seems to click that spark. It's not that I'm running out of algorithms and data structures, you understand (after all Knuth wrote three big volumes on the subject), it's just that nothing seems to grab my fancy. I don't want to drag out an article, word by painful word, and then have to get enough enthusiasm to design and write the code. No, you deserve better than that. When I *want* to write about something, the words flow, the code seems to write itself, and Our Esteemed Editor and you, my Dear Reader, get your money's worth.

And then, as it happens so often, several connected things happened at once, and the topic for this article was clear.

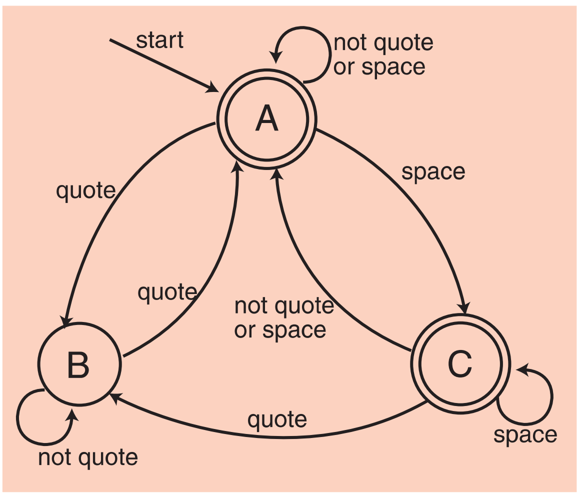
The first one was TurboPower's SysTools newsgroup. Someone wanted a particular string function, as it happens not covered by SysTools, and had tried to write it himself by using other SysTools routines. For some reason, the way to write this particular string routine came into my head fully formed: use a state machine. I wrote the function, together with a small test program, and posted it to the newsgroup. Fifteen minutes, tops.

Coincidentally, the next one was also to do with SysTools: this time, a bug in the regular expression code. This code is pretty nasty if you don't know what algorithm was being followed, so that necessitated a diversion into understanding the algorithm and then tracing the code to find the bug. It was to do with the backtracking algorithm used within the code.

And then, whilst browsing on eBay, I found a textbook with an intriguing title: *Compiler Construction: Theory and Practice* by Barrett et al. I bid on it for fun and won. When I got the book I flipped through it wondering what I'd bought on a whim and, by golly, it had the clearest explanation of finite state machines and automata that I'd ever read. Much easier to understand than the Dragon Book in my view. (The Dragon Book is of course *Compilers: Principles, Techniques, and Tools* by Aho et al, so called because it has a dragon on the cover.)

Suddenly the way was clear: an article on finite state machines.

Before you turn the page, thinking that dear old Bucknall has flipped and you'd never use anything like that, even if you did know what they were, let me reassure you. We will be taking it easy, working our way in step by step, and before long you'll be ready to convert an NFA to a DFA.



➤ *Figure 1: State machine to compress two or more spaces to one.*

Neon Lights

Let's go back to the original problem on the SysTools newsgroup. Paraphrasing, the programmer wanted this: a function that would take a string and return it with the same text, except that all runs of two or more spaces in the original string would be replaced by a single space, except if these multiple spaces occurred in a quoted part of the text. So,

```
The   cat said “Hello  there”
```

Would be converted to,

```
The cat said “Hello  there”
```

With the three spaces between 'The' and 'cat' replaced by one, and the two spaces between 'Hello' and 'there' left as they were (since they appear in between double quotes).

Think a little about how you would perform this conversion before reading on. Using a couple of SysTools routines, the programmer was trying to extract out each word from the original string and then join them together, separating them by a single space. To me, that kind of string manipulation rings alarm bells galore; visions of all the string allocations and frees floated before my eyes. Surely there must be a better way? At which point the answer came in a flash.

My solution was to use a state machine to read the input string character by character. There's that phrase again, *state machine*. A state machine is merely a system (usually digital) that moves from one *state* to another according to input it receives. The moves are called *transitions*. You can think of a state machine as a specialized flowchart and, indeed, Figure 1 shows the flowchart for my function. The state machine shown has three states: A, B and C. We enter the flowchart into state A. At this point, we read a character from the input string. If it is a double quote, we move to state B. If it is a space character we move to state C. If it is any other character we keep in the same state, A (this is shown by the loop).

When we make a move, we may also have an action to do. So, each of these moves just described causes a character to be output to the return string. Suppose we read a double quote and therefore moved to state B. We will have output the double quote. State B is even simpler than state A: the only character that it'll do something special for is a double quote. For any other character, it'll output the character to the output string and stay in the same state, B. For a double quote, it outputs the character and returns to state A. Notice that in state B, spaces have no significance whatsoever. There is no move that depends on a space, apart from the any-character-that-is-not-a-double-quote move of course.

Now suppose that we were in state A and we read a space character from the input string. We have to move to state C and emit a space in the process. State C is a little like state A: it's partitioning the input characters into a double quote, a space and any other character. For a double quote it will output the character and move to state B. For a space, it does not output a character at all, and merely stays in the same state. For any other character, it outputs the character and moves to state A.

As you can see, the state machine properly describes the function my programmer wanted. Multiple spaces are removed (this is state C's *raison d'être*), except in quoted text (this being done by state B). If we followed the state machine diagram for the start of our example sentence, we would perform the following actions:

```
Start at A
Read 'T', write 'T', stay in A
Read 'h', write 'h', stay in A
Read 'e', write 'e', stay in A
Read ' ', write ' ', move to C
Read ' ', stay in C
Read ' ', stay in C
Read 'c', write 'c', move to A
And so on
```

There is, however, one more property of the state machine in Figure 1 that I have ignored up to now. States A and C are circled with a double line, whereas B is not. No, this wasn't a mistake on my part when I was drawing the figure with Adobe Illustrator. It was deliberate. By convention, state machine diagrams use the double circle for a state to mean that this is a *terminating state* (also known as a *halt state* or an *accepting state*). When the input string is completely read, the state machine will be in a particular state (for my nonsensical example text above, the final state of the state machine is A). If that state is a terminating state, the state machine is said to *accept* the input string. No matter which characters (or, more strictly, *tokens*) were found in the input string, and no matter what moves were made, the state machine 'understood' the string. If, on the other hand, the state machine ended up in a non-terminating state, the string was not accepted and the state machine did not understand the string.

In our case, state B is not an accepting state. What does that mean in practical terms? Well, if we're in state B when the input string is exhausted then we've read one double quote, but not a second. The state machine has been reading a string containing text with an unbalanced double quote. Depending on how strict we were being, this could be viewed as an error or we could just ignore it. My state machine views it as an error.

Talking of errors, although our particular example doesn't show this possibility, we could get into a state that doesn't have a move for a particular character or token. This would cause an immediate error.

➤ *Listing 1: State machine code to remove multiple spaces.*

```
function aaRemoveSpaces1(const S : string) : string;
var
  Inx : integer;
  State : (ScanningNormal, ScanningQuoted, ScanningSpaces);
  ResultLen : integer;
  Ch : char;
begin
  if S = '' then begin
    Result := '';
    Exit;
  end;
  SetLength(Result, length(S));
  ResultLen := 0;
  State := ScanningNormal;
  for Inx := 1 to length(S) do begin
    Ch := S[Inx];
    case State of
      ScanningNormal :
        begin
          inc(ResultLen);
          Result[ResultLen] := Ch;
          if (Ch = ' ') then
            State := ScanningSpaces
          else if (Ch = '"') then
            State := ScanningQuoted;
        end;
      ScanningQuoted :
        begin
          inc(ResultLen);
          Result[ResultLen] := Ch;
          if (Ch = '"') then
            State := ScanningNormal;
        end;
      ScanningSpaces :
        begin
          if (Ch <> ' ') then begin
            inc(ResultLen);
            Result[ResultLen] := Ch;
            if (Ch = '"') then
              State := ScanningQuoted
            else
              State := ScanningNormal;
          end;
        end;
    end;
  end;
  if (State = ScanningQuoted) then begin
    Result := '';
    raise Exception.Create(
      'Unbalanced quotes in input string');
  end else
    SetLength(Result, ResultLen);
end;
```

It's More Fun To Compute

So far, we've just been looking at pretty pictures: time for some code. The first way to code a state machine is to write statements that do exactly what the state machine diagram shows. We tend to invert it slightly, so that reading the input string drives the state machine rather than each state having to read the next character from the input string. Listing 1 shows the code that implements the state machine from Figure 1. Notice that I've decided not to name the states unimaginatively as A, B and C to mimic the figure, but instead have given them descriptive names like `ScanningNormal`, `ScanningSpaces` and `ScanningQuoted`.

The code gets a character from the input string and then enters a `case` statement that switches on the current state. For each state, we have `if` statements to implement the actions and the moves depending on the value of the current character. At the end we signal an exception if we are left in the `ScanningQuoted` state.

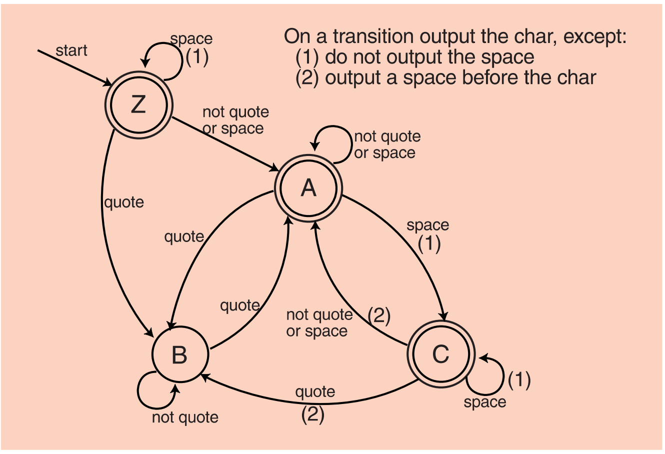
As you can see, the code implements the state machine perfectly. The code is even fairly simple to extend. Suppose, for example, we wanted to cover the use of single quotes as well. Simple enough: we create a new state, D, that functions in the same manner as state B except that the transitions to and from it use single instead of double quotes. In the code, this means a copy-and-paste so that we duplicate the state B functionality as state D.

Another example: as it stands at the moment, the state machine will replace leading and trailing blanks with a single space. Suppose we wanted to remove them from the returned string. Trailing blanks aren't too difficult. Instead of outputting the space character on a transition from state A to C, we output it when we leave state C, together with the character that caused us to leave state C. That means that, if the state machine ends up in state C when the input string is exhausted (indicating that the input text had trailing spaces), a space won't be output. The leading spaces seem a little more difficult until we realise that we can add a new state, call it Z (the `ScanningLeadSpaces` state), before A to weed out all the leading spaces. This state is only exited when the first non-blank character is found. For a space we stay in state Z, for a quote we move to state B, for any other character we transition to A. The new state machine is shown in Figure 2 and the code is shown in Listing 2. (The `Scanning-LeadSpaces` and the `Scanning-Spaces` states are virtually the same as far as code goes. It almost seems worthwhile to try and merge them by having an extra `Boolean` flag to denote whether we're between words or not. Resist this temptation: state machines are easy to understand because of their multiplicity of well-defined, independent states. Reducing their number by the addition of extra flags is a recipe for bugs and confusion.)

Now that we have seen a fairly complex state machine and are more familiar with the idea, I can introduce a couple of new terms. The first is *automaton* (plural: *automata*). This is nothing more than another fancy computer science name for a state machine. Simple enough. A *finite state machine* or *finite automaton* is merely a state machine whose number of states is countable; there is not an infinity of them.

Last new term for now: *deterministic*. Look at the updated state machine in Figure 2. No matter

➤ *Figure 2: State machine to compress spaces and remove leading/trailing spaces.*



➤ *Listing 2: New code to remove leading and trailing spaces as well.*

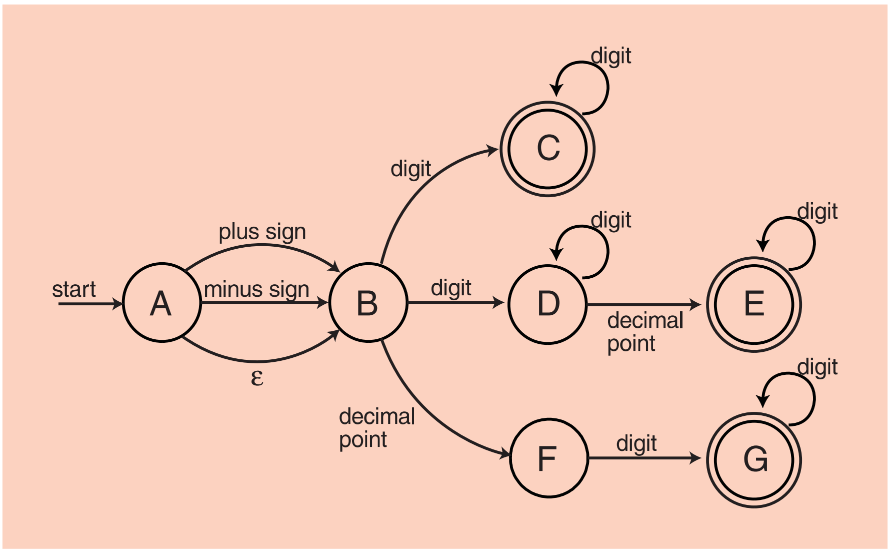
```
function aaRemoveSpaces2(const S : string) : string;
var
  Inx       : integer;
  State     : (ScanningLeadSpaces, ScanningNormal,
               ScanningQuoted, ScanningSpaces);
  ResultLen : integer;
  Ch        : char;
begin
  ..as before..
  State := ScanningLeadSpaces;
  for Inx := 1 to length(S) do begin
    Ch := S[Inx];
    case State of
      ScanningLeadSpaces :
        begin
          if (Ch <> ' ') then begin
            inc(ResultLen);
            Result[ResultLen] := Ch;
            if (Ch = '"') then
              State := ScanningQuoted
            else
              State := ScanningNormal;
          end;
        end;
      ScanningNormal :
        ..as before..
      ScanningQuoted :
        ..as before..
      ScanningSpaces :
        begin
          if (Ch <> ' ') then begin
            inc(ResultLen);
            Result[ResultLen] := ' ';
            inc(ResultLen);
            Result[ResultLen] := Ch;
            if (Ch = '"') then
              State := ScanningQuoted
            else
              State := ScanningNormal;
          end;
        end;
    end;
  end;
  ..as before..
end;
```

what state we're in, no matter what the next character is, we know without fail where to move to next (or we signal an error). We have no choice in the matter: it's not as if, for a double quote, we could either go to state C *or* state A. The move is well defined. This state machine is deterministic; it is known as a deterministic finite state machine (DFSM) or a deterministic finite automaton (DFA). The opposite is a state machine that involves some kind of choice with some of its states. In using this latter type of state machine we will have to make a choice as to whether to move to state X or state Y. As you might imagine, the processing of this kind of state machine involves some more intricate code. These state machines are known, not surprisingly, as non-deterministic finite state machines (NDFSM) or non-deterministic finite automata (NFA).

The Hall Of Mirrors

So, with our newfound confidence in state machines, let's now consider an NFA. Figure 3 shows an NFA that can convert a string containing a number in decimal format to a double value. Looking at it, you may be wondering what on earth that move is with the peculiar lowercase e (ε). This is a *no-cost* or *free move*, where you can make the move without using up the current character or token. So, for example, you can move from the start token A to the next token B by using up a '+' sign, using up a '-' sign, or by just moving there (the no-cost move). These free moves are a feature (I think that is the best word for them!) of non-deterministic state machines.

Take a moment to browse the figure and use it to validate strings such as '1', '1.23', '+.7', '-12.'. You'll see that the upper branch is for integer values (those without a decimal radix point); the middle one for strings that consist of at least one digit before the decimal point, but maybe none afterwards; the lower one for strings that do not have any digits before the decimal point but must have at least one afterwards. If you think about it for a while, you'll see that the state machine won't be able to accept the decimal point on its own.



➤ *Figure 3: NFA to validate a string to contain a floating-point number.*

The problem still remains though: although the state machine will accept '1.2' how does it 'know' to take the middle path? An even more basic question: why bother with these bizarre NFAs and ε signs anyway? Let's just stick with nice simple DFAs: no choices equals no problems.

The second question is actually easier to answer than the first. NFAs are the natural state machines for evaluating regular expressions. Once we understand how to use an NFA, we are more than half way towards being able to apply regular expression matching to a string, the eventual goal of this article or two.

Back to the first question: how does it know? The answer is, of course, it doesn't. There are a couple of ways of processing a string with such a state machine, the simpler being a trial-and-error algorithm. (Note that we are only interested in finding *one* path through the state machine that accepts the string. There may well be others, but we're not interested in enumerating them all.) To help in this trial-and-error algorithm we make use of another algorithm: the backtracking algorithm.

Let's see how it works by tracing through what happens when we are trying to see whether the state machine accepts '12.34'.

We start off in state A. The first token is '1'. We can't make the '+' move to B, nor the '-' move. So we take the free move (the ε link). We're now at state B with the same token, the '1'. We now have two choices: move either to C or to D, consuming the token in the process. Let's take the first choice. Before we move, though, we make a note of what we are about to do, so that if it was wrong we know not to do it again. So we arrive at C, consuming the digit as we do so. We get the next token, the '2'. Simple enough: we stay in the same state, using the token.

We get the next token, the '.'. There are no possible moves at all. We're now stuck: there are no moves and yet we have a token to process. Enter the backtracking algorithm. We look back at our notes and see that in state B, when we were trying to use the '1' we made a choice. Maybe it was the wrong one, so we backtrack to find out. We reset the state machine back to state B, and we reset the input string so that we are at the '1'. Since the first choice resulted in a problem, we try the second choice: the move to D. We make the transition to state D, consuming the '1'. The next token is '2'; we use it up and stay in state D. The next token is '.': a move to state E, which, in fact, consumes the next two digits. We're finished with the input string, and we're in a terminating state, E, and so we can say the NFA accepts the string '12.34'.

Computer World

Great stuff with the hand waving, but now we need to consider how to code this pretty little mess in Delphi. We've a lot to discuss: implementing the state machine algorithm first (is it the same as coding up a DFA?) and then worrying about the backtracking algorithm (all this mysterious 'taking notes' malarkey).

The first thing to note is that we can no longer have a simple `for` loop to cycle through the characters in the string. In the DFA case, every character read from the input string resulted in a move and there was no possibility of backtracking, or going back to a character we'd already visited. So, we'll have to replace the `for` loop with a `while` loop instead and make sure we increment the string index variable when we need to.

The next thing to notice is that we cannot have a simple `case` statement on the input character for some states. We have a plurality of 'move choices' to worry about. Certain of these choices will be rejected immediately (the current character doesn't match the condition for the move). Some will be followed, with some of these being rejected at a later stage and the next choice being followed. For now we'll simply enumerate the possible moves and make sure we follow them in order. We'll use an integer variable for that purpose.

All very well, but now we must consider the final piece: the backtracking implementation. What we want to do is this: whenever we choose a move that is valid (compare this with rejecting a move because the current character doesn't match the conditions for the move) we save the fact that we made that particular move. Then, if we need to backtrack to the same state, with the same input character, we can easily select the next move and try that. Of course, at any stage of the game we may be making choices about our moves, so we must save them all and revisit them in reverse order; the backtrack goes to the most recent choice we made. In other words, a last-in first-out type structure, a stack.

What shall we save on the stack? Well, we need to save the state where we made the choice, the move number we were making (so we know which is the next one we have to try), and finally the character index where we made the choice. Using these three items of information we can easily rewind the state machine to a previous point so that we can make the next, and possibly better, choice for a move.

Looking at the figure for this state machine, we can see that the longest path through states is not that long (the longest is four states), so our stack doesn't have to be that big. There are no cycles to worry about either. (A *cycle* is a path you can take from a particular state that would return you to the same state. A state machine is merely a directed graph and so we use the same terminology.) We'll pre-allocate the stack to hold 10 items, which is quite ample for our purposes.

➤ *Listing 3: Awfully complex NFA code to validate a floating-point number (continues on next page).*

```
function NFAValidateNumber(const S : string) : boolean;
const
  Digits =
    ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'];
type
  TState = (StartScanning,        {state A in figure}
            ScannedSign,          {state B in figure}
            ScanInteger,          {state C in figure}
            ScanLeadDigits,       {state D in figure}
            ScannedDecPoint,      {state E in figure}
            ScanLeadDecPoint,     {state F in figure}
            ScanDecimalDigits);   {state G in figure}
  TChoice = packed record
    chInx   : integer;
    chMove  : integer;
    chState : TState;
  end;
var
  i     : integer;
  State : TState;
  Ch    : char;
  Move  : integer;
  ChoiceStack : array [0..9] of TChoice;
  ChoiceSP    : integer;
begin
  {assume the number is invalid}
  Result := false;
  {initialize the choice stack}
  ChoiceSP := 0;
  {prepare for scanning}
  Move := 0;
  i := 1;
  State := StartScanning;
  while i <= length(S) do begin
    Ch := S[i];
    case State of
      StartScanning :
        begin
          case Move of
            0 :
              begin
                if (Ch = '+') then begin
                  with ChoiceStack[ChoiceSP] do begin
                    chInx := i;
                    chMove := Move;
                    chState := State;
                  end;
                  inc(ChoiceSP);
                  State := ScannedSign;
                  Move := 0;
                  inc(i);
                end else
                  inc(Move);
              end;
            1 :
              begin
                if (Ch = '-') then begin
                  with ChoiceStack[ChoiceSP] do begin
                    chInx := i;
                    chMove := Move;
                    chState := State;
                  end;
                  inc(ChoiceSP);
                  State := ScannedSign;
                  Move := 0;
                  inc(i);
                end else
                  inc(Move);
              end;
            2 :
              begin
                with ChoiceStack[ChoiceSP] do begin
                  chInx := i;
                  chMove := Move;
                  chState := State;
                end;
                inc(ChoiceSP);
                State := ScannedSign;
                Move := 0;
              end;
          else
            if (ChoiceSP = 0) then
              Exit; // error
            dec(ChoiceSP);
            with ChoiceStack[ChoiceSP] do begin
              i := chInx;
              Move := succ(chMove);
              State := chState;
```

{ Continued on following page...}

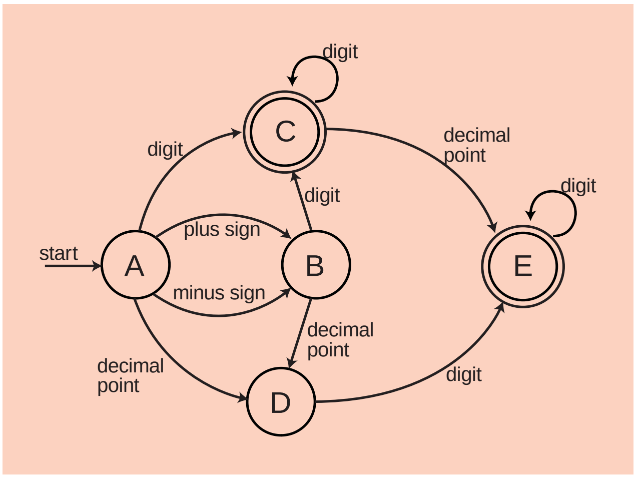
```
{ Continued from previous page...}

            end;
          end;{Move case}
        end;
      ScannedSign :
        begin
          case Move of
            0 :
              begin
                if (Ch in Digits) then begin
                  with ChoiceStack[ChoiceSP] do begin
                    chInx := i;
                    chMove := Move;
                    chState := State;
                  end;
                  inc(ChoiceSP);
                  State := ScanInteger;
                  Move := 0;
                  inc(i);
                end else
                  inc(Move);
              end;
            1 :
              begin
                if (Ch in Digits) then begin
                  with ChoiceStack[ChoiceSP] do begin
                    chInx := i;
                    chMove := Move;
                    chState := State;
                  end;
                  inc(ChoiceSP);
                  State := ScanLeadDigits;
                  Move := 0;
                  inc(i);
                end else
                  inc(Move);
              end;
            2 :
              begin
                if (Ch = DecimalSeparator) then begin
                  with ChoiceStack[ChoiceSP] do begin
                    chInx := i;
                    chMove := Move;
                    chState := State;
                  end;
                  inc(ChoiceSP);
                  State := ScanLeadDecPoint;
                  Move := 0;
                  inc(i);
                end else
                  inc(Move);
              end;
          else
            if (ChoiceSP = 0) then
              Exit; // error
            dec(ChoiceSP);
            with ChoiceStack[ChoiceSP] do begin
              i := chInx;
              Move := succ(chMove);
              State := chState;
            end;
          end;{Move case}
        end;
      ScanInteger :
        begin
          case Move of
            0 :
              begin
                if (Ch in Digits) then
                  inc(i)
                else
                  inc(Move);
              end;
          else
            if (ChoiceSP = 0) then
              Exit; // error
            dec(ChoiceSP);
            with ChoiceStack[ChoiceSP] do begin
              i := chInx;
              Move := succ(chMove);
              State := chState;
            end;
          end;{Move case}
        end;
      ScanLeadDigits :
        begin
          case Move of
            0 :
              begin
                if (Ch in Digits) then
                  inc(i)
                else
                  inc(Move);
              end;
            1 :
              begin
                if (Ch = DecimalSeparator) then begin
                  with ChoiceStack[ChoiceSP] do begin
                    chInx := i;
                    chMove := Move;
                    chState := State;
                  end;
                  inc(ChoiceSP);
                  State := ScannedDecPoint;
                  Move := 0;
                  inc(i);
                end else
                  inc(Move);
              end;
          else
            if (ChoiceSP = 0) then
              Exit; // error
            dec(ChoiceSP);
            with ChoiceStack[ChoiceSP] do begin
              i := chInx;
              Move := succ(chMove);
              State := chState;
            end;
          end;{Move case}
        end;
      ScannedDecPoint :
        begin
          case Move of
            0 :
              begin
                if (Ch in Digits) then
                  inc(i)
                else
                  inc(Move);
              end;
          else
            if (ChoiceSP = 0) then
              Exit; // error
            dec(ChoiceSP);
            with ChoiceStack[ChoiceSP] do begin
              i := chInx;
              Move := succ(chMove);
              State := chState;
            end;
          end;{Move case}
        end;
      ScanLeadDecPoint :
        begin
          case Move of
            0 :
              begin
                if (Ch in Digits) then begin
                  with ChoiceStack[ChoiceSP] do begin
                    chInx := i;
                    chMove := Move;
                    chState := State;
                  end;
                  inc(ChoiceSP);
                  State := ScanDecimalDigits;
                  Move := 0;
                  inc(i);
                end else
                  inc(Move);
              end;
          else
            if (ChoiceSP = 0) then
              Exit; // error
            dec(ChoiceSP);
            with ChoiceStack[ChoiceSP] do begin
              i := chInx;
              Move := succ(chMove);
              State := chState;
            end;
          end;{Move case}
        end;
      ScanDecimalDigits :
        begin
          case Move of
            0 :
              begin
                if (Ch in Digits) then
                  inc(i)
                else
                  inc(Move);
              end;
          else
            if (ChoiceSP = 0) then
              Exit; // error
            dec(ChoiceSP);
            with ChoiceStack[ChoiceSP] do begin
              i := chInx;
              Move := succ(chMove);
              State := chState;
            end;
          end;{Move case}
        end;
    end;{case}
  end;
  {if we reach this point, the number is valid
   if we're in a terminating state}
  if (State = ScanInteger) or
     (State = ScannedDecPoint) or
     (State = ScanDecimalDigits) then
    Result := true;
end;
```

Now we can write Listing 3 to incorporate the backtracking algorithm. Again the state machine will accept a string when the string is exhausted and the automaton is in a terminating state. It will fail a string if the string is exhausted and we're not in a terminating state, or if we reach a state and the current character cannot be matched against a move. The second condition has a further caveat for the NFA case: the backtracking stack must be empty.

If you look at this code and compare it to the code for our DFA in Listing 2 (even though this latter code is doing something else), you can see that it's much more complicated. It's prone to error as well (we have to worry about the stack, about rewinding the state machine, about selecting another move, and so on). We could simplify it somewhat by extracting out pushing/popping choice information, initializing for a new state, etc, but it will still be more complicated. In general, if we need a fixed, predefined automaton we would devise and use a deterministic one. We try and leave non-deterministic ones well alone.

It's instructive to devise a DFA for the number matching example. Figure 4 shows such a deterministic state machine. I shall leave it to the interested reader to convert the figure into code. Once you have done so, you will see that it is impressively simpler code than the NFA case, although it performs exactly the same job.



➤ *Figure 4: DFA to validate a string to contain a floating-point number.*

Of course, with this NFA example (and the DFA figure), all we're doing is validating a string to be the textual representation of a floating-point number. We should also add in some actions for each move in order to calculate the `double` value that's equivalent to the string. For the DFA, that's pretty easy. We set an accumulator variable to 0. As we decode each digit before the decimal point, we multiply the accumulator by 10.0 and then add the new digit value. For digits after the decimal point, I would maintain a counter for the decimal place, incrementing it by one for each digit after the point. For each such digit, we add that digit value multiplied by the power of one tenth that we've reached for that decimal place. Easy enough.

What about the NFA? Well, it could be pretty bad, let me tell you. The problem all lies in the backtracking algorithm. At any time, we could suddenly find the state machine rewinds to a previous position. For the string to floating-point number example, it's not too bad: we just save the current accumulator value on the stack as we push a move. When we backtrack, we'll pop off the accumulator value as well as the data for the point where we made the bad choice. But, consider another hypothetical NFA, where the action for a particular move actually does something irreversible: we just cannot backtrack and recover a previous state. We will have to do a *lot* more work to try and avoid this situation, and in general it's just not worth it. Stick to deterministic automata.

In fact, it can be shown that, if you have a non-deterministic finite automaton, you can convert it into a deterministic one by following a simple-to-describe algorithm.

Before we do so (it'll have to be in next month's article by the way), let's discuss the primary application of NFAs: regular expression matching.

Techno Pop

Let's recap what regular expressions are. Essentially they're a mini-language for describing, in a simple way, a pattern for searching text (or, more rigorously, matching text). At its most basic, a regular expression merely consists of a word or set of characters. However, using the standard metacharacters (or regular expression operators), you can search for more complex patterns. The standard metacharacters are '.' (matches any character except newline), '?' (matches zero or one occurrence of the previous subexpression), '*' (matches zero or more occurrences of the

➤ *Figure 5: Regular expression grammar.*

```
<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> |
           '(' <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>
This grammar means that parentheses have maximum precedence, followed by square
brackets, followed by the closure operators, followed by concatenation, finally
followed by alternation.
```

previous subexpression), '+' (matches one or more occurrences of the previous subexpression), '|' (the OR operator, which matches either the left subexpression or the right one). You can also define a character class to match one of a set of characters.

Figure 5 shows the grammar for the regular expressions we'll be dealing with. It's written in standard BNF form. The '::=' means 'is defined as' and the '|' means 'OR'. Hence the first line says that an 'expr' is either a 'term', or is a term, followed by the pipe character, followed by an 'expr' again. The second line says that a 'term' is either a 'factor' or is a factor followed by a term; and so on so forth. This grammar definition (it's called a *grammar* because it defines a language; if you search in the Delphi help you will find the grammar for Object Pascal: it's defined in the same way) can be used to generate a routine to evaluate a regular expression; we'll see how next month. But for now, be aware that we could use the grammar to desk-check that a given regular expression were valid or not.

This month, however, I'd like to describe how to create an NFA from a regular expression. We can then use the NFA to match input strings against the regular expression. For example, if we could generate the NFA for the regular expression '(a|b)*bc' (read this as 'an *a* or a *b*, repeated zero or more times, followed by a *b*, followed by a *c*'), we could then see whether the input strings 'abc', 'abbc', 'bac', etc were matched by the regular expression (in computer science terms, we ask whether the input string is part of the *regular language* defined by the regular expression).

In fact, creating a state machine diagram for a particular regular expression is pretty easy. The language basically states that a regular expression consists of various sub-regular-expressions arranged or joined together in various ways. Each subexpression has a single start state and a single terminating state and, like Lego, we fit these simple building blocks together to show the entire regular expression. Figure 6 has the most important constructions. The first one is a state machine for recognizing a single character in the alphabet. The second is equally simple: a state machine for recognizing *any* character in the alphabet (the '.' operator, in other words). The third construction shows you how to draw concatenation (one expression followed by another). We simply merge the start state of the second subexpression to the terminating state of the first subexpression. Simple, eh? The next one is a state machine for the '?' operator: here we have an ε path as well as the path through the subexpression from the start state to the terminating state. The most complicated constructions are probably for the '+' and '*' operators.

Anyway if you look at Figure 6, you'll notice some interesting properties. Some constructions define and use extra states in order to create their state machine, but they do it in a well-defined way: every state has either one or two moves coming from it and, if there are two moves, both are no-cost moves. There's a reason for this: it just makes it simpler to code.

If we take our little regular expression example '(a|b)*bc', we can build up its NFA step by step. Figure 7 shows how. Notice that at every step, we have an NFA with one start state and one terminating state, and every new state we create, we make sure that there are at most two moves from it.

Because of the construction method we used, we can create a very simple tabular representation for each state. The state will be represented by a record in an array of such records (the state number being the index of the record in the array). Each state record will consist of something to match, and two state numbers for the next state (NextState1, NextState2). The something is a character pattern to match; it can be ε, an actual character, the '.' operator for any character, a character class (ie a set of characters, one of which must match the input character), or a negated character class (the input character cannot be part of the set to match). This array is known as the transition table.

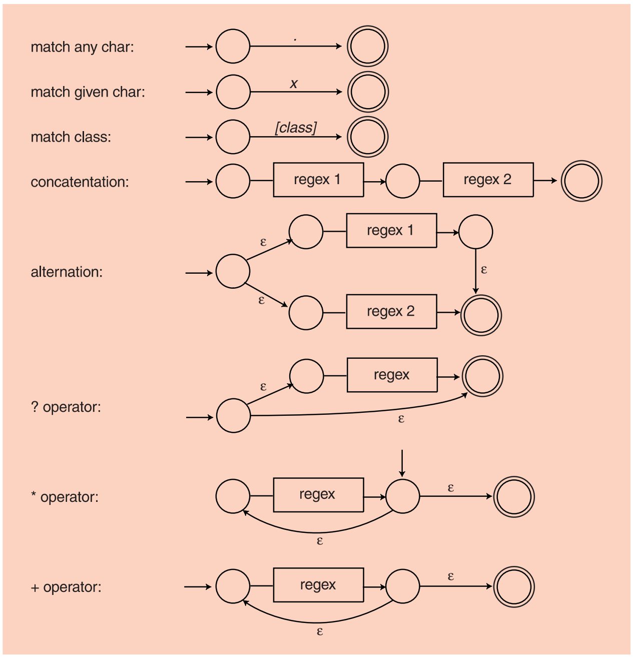
We can now show the transition table for '(a|b)*bc' in Table 1. We start off in state 0, and move through, matching each character in the input string, until we reach state 7. As you can see, the transition table describes the state machine in Figure 7 perfectly, and furthermore it's very easy to write code to traverse the state machine with a given input string.

Instead of writing a backtracking algorithm, we'll go one better. I did say that there were two ways of traversing an NFA, the trial-and-error method coupled with the backtracking algorithm being the first. The other one seems even more bizarre and fantastic: we shall traverse the NFA with the input string, tracing every possible path through the state machine simultaneously. We shall make no choices since we're following every possible path at every single step. Eventually we shall run out of string, and have one or more paths that got us there, or we shall run out of possible paths part way through the string.

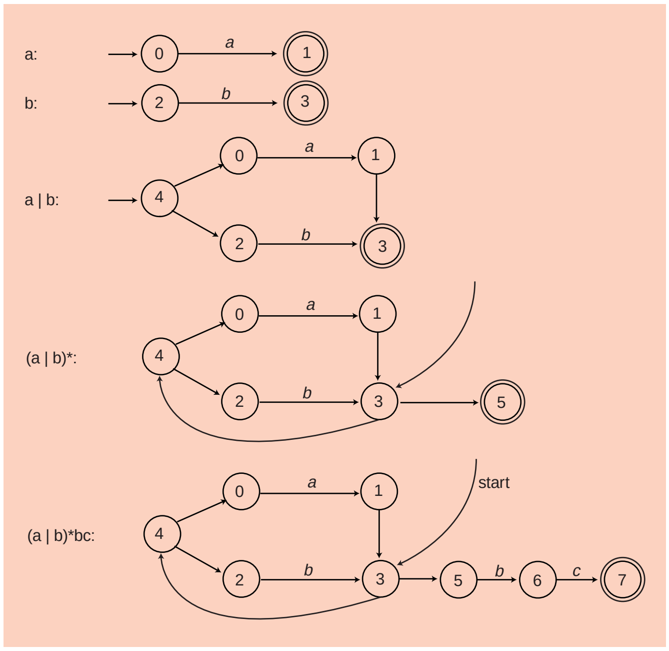
Scary stuff? Maybe: just follow along carefully and let's see if I can't persuade you how easy it is. I shall assume that we have a transition table for the NFA. I shall also assume that we have a deque implementation. A *what*? A *deque* (usually pronounced *deck*) is a double-ended queue. You can add items to either the front or rear of the queue, and similarly, remove items from both places as well. In a way, it acts like a mix between a queue (add to rear, remove from front) and a stack (add to front, remove from front). For this article we shall only need the operations Push (add to the front of the queue), Pop (remove from the front), and Enqueue (add to the back of the queue). The deque we'll use will be a deque of integers (actually state numbers).

The first thing is to enqueue -1 onto the deque. This is a special value that says we need to advance by one through the input string. We then enqueue the number of the initial state onto the deque. Now, we enter a loop. The first thing we do is to pop the top state number from the deque. If it is -1, as it will be initially, we getthe next character from the input string and enqueue a -1. Otherwise it is an actual state number. We check to see if the current input character matches that state's character pattern. If it does, we enqueue the state's first 'next state' value, NextState1. If the state's character pattern is ε, we push the first next state value on to the deque, and if the second next state value is set, we push that as well.

➤ *Figure 6: Constructions used to create an NFA from a regular expression.*



➤ *Figure 7: Constructing an NFA for (a|b)*bc.*



```
type
  PaaCharSet = ^TaaCharSet;
  TaaCharSet = set of char;
  TaaNFAMatchType = ( {types of matching performed...}
     mtNone,          {..no match (an epsilon no-cost move)}
     mtAnyChar,       {..any character}
     mtChar,          {..a particular character}
     mtClass,         {..a character class}
     mtNegClass);     {..a negated character class}
  TaaNFAStateData = record
    sdNextState1: integer;            {-1 means "not used"}
    sdNextState2: integer;            {-1 means "not used"}
    sdMatchType : TaaNFAMatchType;
    case integer of
      0 : (sdChar  : char);
      1 : (sdClass : PaaCharSet);
  end;
  PaaNFAStateTable = ^TaaNFAStateTable;
  TaaNFAStateTable = packed record
    stStartState: integer;
    stFinalState: integer;
    stTable     : array [0..9999] of TaaNFAStateData;
  end;
function aaMatchRegEx(aTable  : PaaNFAStateTable;
  const S : string) : boolean;
const
  MustScan = -1;
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
    {push the special value to start scanning}
    Deque.Enqueue(MustScan);
    {enqueue the first state}
    Deque.Enqueue(aTable^.stStartState);
    {prepare the string index}
    StrInx := 0;
    {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 the deque is empty at this point, we might as
          well give up since there are no states left to
          process new characters}
        if not Deque.IsEmpty then begin
          {if we haven't run out of string, get the
            character, and enqueue the "must scan" state
            again}
          inc(StrInx);
          if (StrInx <= length(S)) then begin
            Ch := S[StrInx];
            Deque.Enqueue(MustScan);
          end;
        end;
      end
      {otherwise, process the state}
      else with aTable^.stTable[State] do begin
        case sdMatchType of
          mtNone :
            begin
              {for free moves, push the next states onto the
                deque}
              if (sdNextState2 <> -1) then
                Deque.Push(sdNextState2);
              if (sdNextState1 <> -1) then
                Deque.Push(sdNextState1);
            end;
          mtAnyChar :
            begin
              {for a match of any character, enqueue the
                next state}
              Deque.Enqueue(sdNextState1);
            end;
          mtChar :
            begin
              {for a match of a character, enqueue the next
                state}
              if (Ch = sdChar) then
                Deque.Enqueue(sdNextState1);
            end;
          mtClass :
            begin
              {for a match within a class, enqueue the next
                state}
              if (Ch in sdClass^) then
                Deque.Enqueue(sdNextState1);
            end;
          mtNegClass :
            begin
              {for a match not within a class, enqueue the
                next state}
              if not (Ch in sdClass^) then
                Deque.Enqueue(sdNextState1);
            end;
        end;
      end;
    end;
    {if we reach this point we've either exhausted the deque
      or we've run out of string; we need to check the
      states left on the deque (if there are any) to see if
      one is the terminating state; if so the string matched
      the regular expressionn defined by the transition
      table}
    while not Deque.IsEmpty do begin
      State := Deque.Pop;
      if (State = aTable^.stFinalState) then begin
        Result := true;
        Exit;
      end;
    end;
  finally
    Deque.Free;
  end;
end;
```

➤ *Listing 4: Code to match a string against a transition table for (a|b)*bc.*

State:	0	1	2	3	4	5	6	7
Match char:	a		b			b	c	
Next state 1:	1	3	3	5	0	6	7	-1
Next state 2:	-1	-1	-1	4	2	-1	-1	-1

➤ *Table 1: Transition table for (a|b)*bc.*

The loop terminates once the deque is empty (no paths match the input string) or we extract all the characters (the deque then contains the set of states we've reached; we can pop them off until we find the one-and-only terminating state or not, as the case may be).

The overall effect is this: we have a 'get next character' value (-1) on the deque. To the 'left' of it is a set of states that we still need to test the current character against (we're continually popping these off and pushing states we can reach via the no-cost move). To its 'right' is a set of states derived from states that have already matched the current character. We'll be getting to them once we have popped the -1 and retrieved the next character.

Listing 4 shows this matching routine. I've not shown the deque code (it's not that interesting, honestly!) but it is on this month's disk, as is the driver program that tests the matching routine with our simple regular expression.

Ohm Sweet Ohm

Next month, we'll look at how to automatically generate the transition table from the regular expression itself (so far, I've miraculously provided it). This will involve converting the regular expression grammar into a top-down or recursive descent parser. We'll then look at how to convert an NFA to a DFA (hint: you've already seen part of the process!). Until then...

Julian Bucknall works craftily, but can be reached at julianb@turbopower.com