RegEx Reflections

June 17, 2026

This will be the Tutorial 189.

We show different explanations how to use Regular Expressions in your code and how to optimize code blocks. As an example we took one single function called Commatize with a RegEx in it. This Delphi / maXbox function `Commatize` adds grouping separators (like commas, spaces, or periods) to a numeric substring inside a larger string, formatted similarly to how numbers are written with thousands separators.

The Tutorial has 4 parts:

- Code Explanation
- Regex explain step by step
- RegEx Optimization
- Practical Checklist for Regex

Code Explanation

Purpose in plain terms

Given:

- `s`: the full input string (e.g. `"The price is 1234567.89 dollars"`)
- `startIndex`: where in `s` the number starts (0-based or 1-based depending on your code's convention)
- `period`: how many digits per group (commonly `3` for thousands)
- `sep`: the separator character/string (e.g. `","`, `" "`, `"."`)

It will:

1. Validates parameters; if invalid, returns `s` unchanged.
2. Uses a regex (`TRegEx`) to match a number pattern in the substring starting at `startIndex`.
3. If the regex doesn't match, returns `s` unchanged.

4. Splits the matched number on `'.'` into:
    - `ip`: integer part
    - `dp`: decimal part (if present)
5. For each part whose length is greater than `period`:
    - Reverses the string
    - Inserts `sep` every `period` digits from the right
    - Reverses back
6. Reassembles integer and decimal parts with `'.'` if a decimal part exists.
7. Replaces the original matched number (`s1`) in the substring with the formatted number (`ip`), and returns:
    - the part before `startIndex`
    - a newline (`CRLF`)
    - the rest of the string with the number replaced by the comma-separated version.

Key operations

- `Copy(s, startIndex, Length(s))` – extracts the substring from `startIndex` to the end, using Delphi's 1-based `Copy`.
- `regex.Match(...)` – matches a number pattern (defined by `PATTERN`, not shown here).
- `flcStrSplit(s1, '.')` – splits the matched number into integer and decimal parts.
- `ReverseString` + manual loop inserting `sep` – implements the thousands-separator logic.
- `stringreplace(..., s1, ip, [rfReplaceAll])` – replaces the original number with the formatted one.

Example

If:

```
delphis := "Value: 1234567.89 end";
startIndex := 8;     // point to "1234567.89"
period := 3;
sep := ",";
```

Then the number part `1234567.89` becomes:

```
text1,234,567.89
```

And the result is roughly:

```
text "Value: " + CRLF + "1,234,567.89 end"
```

(With the exact prefix before `startIndex` preserved, then a newline, then the rest with the number reformatted.)

In short: `Commatize` formats a numeric substring with grouping separators every `period` digits, using `sep` as the separator, and replaces the original number in the string.

```
const
  PATTERN = '(\.[0-9]+|[1-9]([0-9]+)?(\.[0-9]+)?)';

  TEST0 ='pi=3.14159265358979323846264338327950288419716939937
  TEST1 ='The author has two Z$100000000000000 Zimbabwe notes
  TEST2 ='The land area of the earth is 57268900(29% of the s
```

```
  TEST3 ='-in Aus$+1411.8millions';
  TEST4 = 'Ain''t no numbers in this here words, nohow, no wa

var  TESTS: array[0..13] of string; // =

var regex: TRegEx;

function Commatize(s: string; startIndex, period: integer; se
var m: TMatch;
    s1, ip, pi, dp: string;
    splits: TStringArray; //array of string; //TArray<string>
    i: integer;
begin
 // define splits
  regex:= TRegEx.Create1(PATTERN);

  if (startIndex < 0) or (startIndex >= Length(s)) or (period
     result:=s;  writ('eout')
    exit; end;
  //m := regex.Match(s.Substring(startIndex, s.Length));
  m:= regex.Match(copy(s,startIndex, Length(s)));
  if not m.Success then begin
    result:= s; writ('not valid regex out:')
    exit; end;

  s1:= m.Groups[0].Value;
  //Func flcStrSplit( const S, D :Str) : TStringArray');
  splits:= flcStrSplit(s1,'.');
  writeln('debug size split:'+itoa(length(splits))+' ');

  ip:= splits[0];
  //writeln('debug split:'+ip);

  if Length(ip) > period then begin
    pi:= ReverseString(ip);
    i:= ((Length(ip)-1) div period) * period;
    while i >= period do begin
      //pi := pi.Substring(0, i) + sep + pi.Substring(i);
      pi:= substring(pi,0, i) + sep + substring(pi,i+1,0);
      i:= i - period;
    end;
    ip:= ReverseString(pi);
  end;

  //dStrContains( const aString : String; const aSubString :
  //if s1.Contains('.') then
  if dstrContains(s1,'.') then begin
    dp:= splits[1];
    if Length(dp) > period then begin
      i:= ((Length(dp)-1 ) div period) * period;
      while i >= period do begin
        //dp := dp.Substring(0, i) + sep + dp.Substring(i);
        dp:= substring(dp,0,i) + sep + substring(dp,i+1,0);
        i:= i - period;
      end;
    end;
    ip:= ip + '.' + dp;
  end;
  //Result := s.Substring(0, startIndex) + s.Substring(startI
  //writ(substring(s,startIndex,0))  //0 means  the rest from
  Result:= substring(s,0,startIndex) +CRLF+
             stringreplace(substring(s,startindex,0),s1,ip,[r
  regex.free;
end;
```

https://sourceforge.net/projects/maxbox5/files/examples/1488_commatize.txt/download

https://sourceforge.net/projects/maxbox5/files/examples/1488_commatize_regex2.txt/download

Writeln(commatize(TEST0, 0, 5, '.'));

//Writeln(commatize(TEST0, 0, 5, #32));

```
  WriteIn(commatize(TEST1, 0, 3, ‘.’));

  WriteIn(commatize(TEST2, 0, 3, ‘,’));

  WriteIn(commatize(TEST3, 0, 3, ‘,’));

  WriteIn(commatize(TEST4, 0, 3, ‘,’));
```

```
debug size split:2
pi=3.14159265358979323846264338327950288419716939937510582097494459231
pi=3.14159.26535.89793.23846.26433.83279.50288.41971.69399.37510.58209.74944.59231
debug size split:1
The author has two Z$100000000000000 Zimbabwe notes (100 trillion).
The author has two Z$100.000.000.000.000 Zimbabwe notes (100 trillion).
debug size split:1
The land area of the earth is 57268900(29% of the surface) square miles.
The land area of the earth is 57,268,900(29% of the surface) square miles.
debug size split:2
-in Aus$+1411.8millions
-in Aus$+1,411.8millions
not valid regex out:
Ain't no numbers in this here words, nohow, no way, Jose.
 mX5
```

```
executed: 16/06/2026 09:34:16 Runtime: 0:0:3.24 Memload: 80% use
```

In your `Commatize` function, the regex itself does not distinguish integer from decimal parts. It only matches the whole number (integer, possibly with a

decimal point and fractional digits) as a single string. The distinction happens after the match, in your own code.

REGEX_PATTERN = '(.[0-9]+|1-9?(.[0-9]+)?)';

Regex explain step by step

1. Regex matches the full number
   The pattern `PATTERN` (not shown in your code) is typically something like:text`\d+(\.\d+)?`ortext`[0-9]+(\.[0-9]+)?`This matches:
   - `123` (integer only)
   - `123.45` (integer + decimal part)

   But it treats the whole thing as one match, not as separate groups for integer and decimal.[stackoverflow]
2. You get the matched string delphi `s1 := m.Groups.Value;s1` is the full matched number, e.g. `"1234567.89"` or `"1234567"`.
3. You split on `'.'` to separate parts delphi `splits := flcStrSplit(s1, '.');`
   - If `s1` is `"1234567.89"`, then:
     - `splits[0]` = `"1234567"` → integer part (`ip`)`splits[1]` = `"89"` → decimal part (`dp`)

   If `s1` is `"1234567"` (no dot), then:
   - `splits[0]` = `"1234567"`There is no `splits[1]`.

   This splitting is your code, not the regex.[stackoverflow]
4. You check for the presence of `'.'`delphi
5. `if dstrContains(s1, '.') then begin dp := splits[1]; ... end;`
6. This condition determines whether there is a decimal part:
   - If `s1` contains `'.'`, you treat `splits[1]` as the decimal part.
   - If not, you only have an integer part.

Summary

- The regex matches the entire number as one string (integer ± optional `.fraction`).
- The integer vs. decimal distinction is made by:
  - Splitting the matched string on `'.'`
  - Using `dstrContains(s1, '.')` to check if a decimal part exists.

So the regex does not directly separate integer and decimal; your code does that after the match.

In Delphi, plain string parsing (e.g. `Pos`, `IndexOf`, `Copy`, manual loops) is almost always faster than using regular expressions, especially for simple tasks like splitting on a fixed delimiter or extracting a number. The performance difference becomes significant in hot paths or when called many times.

Key performance differences

Aspect	Regex (e.g. `TRegEx`)	Plain string parsing (`Pos`, `Copy`, `IndexOf`, loops)
Speed for simple patterns	Slower: must parse pattern, compile, and run regex engine	Faster: direct character search, no extra overhead
Speed for complex patterns	Often better trade-off: much less code, reasonably fast	Can be slow or very complex to implement efficiently
** reuse of pattern**	Very slow if you create `TRegEx` on every call	Fast if you reuse logic; no per-call “pattern” overhead
Backtracking cost	Can be exponential for poorly written patterns [blog.codinghorror]	No backtracking; predictable behavior
Maintainability	High: complex patterns in one line	Lower: more code, more subtle logic

Delphi-specific benchmark from the community:

- Task: check if `FunctionX(...)` exists in a string, allowing up to 4 spaces before `(`.
- Implementations:
  - Unoptimized regex: `TRegEx` created on every call.
  - Optimized regex: one compiled `TRegEx` reused.
  - Plain `Pos` + loop.

Timings (ms) for 100,000 iterations on a “no hit” string:

- Regex (unoptimized, new per call): very slow (bottleneck)
- Regex (optimized, reused): much faster, but still not best
- Pos + loop: fastest of the three[en.delphipraxis]

In general:

“using Pos and string iteration is faster of course”[en.delphipraxis]

For your specific use case (splitting a number on `'.'`)

You’re doing:

```
delphi splits := flcStrSplit(s1, '.');
```

This is a simple split on a single character. For this:

- A manual split using `Pos('.', s1)` and `Copy` will be:
  - Faster than regex-based splitting.
  - Very simple to implement.
- Regex is unnecessary overhead here; you already have the matched number as `s1`.

Example of a fast manual split (integer/decimal):

```
delphi procedure SplitNumber(const s: string; out IntegerPart,
DecimalPart: string);
var
  p: Integer;
begin
  p := Pos('.', s);
  if p = 0 then
  begin
    IntegerPart := s;
    DecimalPart := '';
  end else begin
    IntegerPart := Copy(s, 1, p - 1);
    DecimalPart := Copy(s, p + 1, MaxInt);
  end;
end;
```

```
1  procedure SplitNumber(const s: string; out IntegerPart, Decim
2  var p: Integer;
3  begin
4    p:= Pos('.', s);
5    if p = 0 then begin
6      IntegerPart:= s;
7      DecimalPart:= '';
8    end else begin
9      IntegerPart:= Copy(s, 1, p - 1);
10     DecimalPart:= Copy(s, p + 1, MaxInt);
11   end;
12 end;
```

This avoids:

- Regex engine overhead
- Array allocation if you only need two parts
- Any regex-related backtracking

When regex is worth it in Delphi /maXbox

Use `TRegEx` when:

- The pattern is complex: multiple alternatives, optional groups, character classes, etc.
- You need matching with groups and don't want to manually track positions.
- The code is not in a hot path, or the performance gain from simplicity outweighs the cost.

For something like:

- "Find a number with optional decimal part": `(\d+)(\.\d+)?`
- "Validate a more complex format" (e.g. with optional signs, ranges, etc.)

regex is convenient and usually acceptable.

Practical guidelines for Delphi /maXbox

1. Hot path + simple pattern (e.g. split on `'.'`, `','`, check prefix/suffix):
    - Use `Pos`, `IndexOf`, `Copy`, `StringReplace`, manual loops.

    - Avoid regex.
2. Repeated regex use:
    - Reuse a single `TRegEx` instance (create once, store as a field/class variable).
    - Avoid `TRegEx.Create` inside the loop.
3. Complex patterns, non-hot path:
    - Regex is fine; performance cost is acceptable vs. code complexity.
4. Uncertain:
    - Measure with `TStopWatch` on realistic data.
    - Optimize only if you see a real bottleneck.

Summary

- Regex splitting vs. string parsing in Delphi:
    - For simple splits like `flcStrSplit(s1, '.')`: string parsing is faster.
    - For complex patterns: regex is more maintainable and usually acceptable unless in a tight loop.
- In your `Commatize` function, replacing the regex-based match + split with:
    - A simple `Pos` + `Copy` for integer/decimal split, and
    - Optionally a simpler `Pos`-based number search instead of regex, would likely improve performance, especially if this function is called frequently.

RegEx Optimization

In Delphi and maXbox (which uses a backtracking regex engine in `TRegEx`, similar to .NET/PCRE), the most common regex pitfalls that cause exponential backtracking (catastrophic backtracking) are:

---

1. Nested Quantifiers Over Overlapping Patterns

Patterns where repetition operators are nested and can match the same characters in multiple ways:

```
text (a+)+$
(.*)+
([a-z]+)+$
(.*).*
```

Problem:
For a long string like `aaaaaaaa…b` (many `a` then a `b`), the engine tries an exponential number of ways to split the `a`s between the inner and outer quantifiers before failing.

Delphi-specific risk:

- `TRegEx` does not support possessive quantifiers like `++` or atomic groups `(?>...)` in all versions, so you can't easily "lock" these.

Safe rewrite:

```
text a+$        # instead of (a+)+$
[a-z]+$    # instead of ([a-z]+)+$
```

Rule:

When nesting repetition operators, make sure there is only one way to match the same text.[blog.codinghorror]

---

2. Using .* / .+ Without Boundaries or Specificity

Patterns like:

```
text .*foo.*bar.*
(.*){n}
.*(\d+).*
```

Problems:

- `.*` is greedy and matches everything, then backtracks character by character when the rest of the pattern fails.
- On long input that doesn't match, this can cause huge backtracking chains.

Delphi usage risk:

```
delphiTRegEx.IsMatch(input, '.*foo.*bar');  // can be very slow on long non-matching input
```

Safer patterns:

- Replace `.*` with explicit character classes that match only what you expect:text`[^<]*foo[^<]*bar # if you know delimiters are '<'`
  `\d*foo\d*bar # if only digits are expected`
- Anchor when possible:text`^.*foo.*bar$ # full-string match, less scanning`

[testregex]

---

3. Multiple Alternatives That Can Match the Same Text

Patterns where several alternatives overlap:

```
text (\d+|[\p{N}]*|0)*
(a|ab|abc)*
([0-9]+|[1-9][0-9]*)*
```

Problem:

- The engine tries every combination of which alternative matches each chunk.
- For long inputs near a mismatch, this becomes a “tug-of-war” over the same characters, leading to exponential paths.[stackoverflow]

Example from Java that applies to Delphi too:

```
text []([\p{N}]*|[\p{N}]*|[\p{N}]*)*
```

Each `[\p{N}]*` competes over the same digits.

Safe rewrite:

- Use a single alternative that covers the case:text`[\p{N}]+`
- Remove unnecessary optional alternatives that overlap.

---

4. Too Many Optional Parts and Nested * / + with Overlap

Patterns with almost everything optional:

```
text \[([^\]]*|]*)*\]
(a|b|c|d|e|f|g|h|i|j)*
```

Problems:

- The engine can match the same prefix with many different combinations of alternatives.
- When the final part fails, it backtracks through all those combinations.

Rule:

If it’s going to fail, you want it to fail as quickly as possible. Avoid structures where many paths lead to the same partial match.]

Simplify:

- Remove redundant alternatives.
- Avoid * where + is what you mean if emptiness is not intended.

---

5. Unbounded Lazy Quantifiers on Long Text

Lazy patterns like:

```
text [\s\S]*?foo
.*?foo
```

On very long strings that don’t contain `foo`, the engine expands one character at a time and re-tests the rest of the pattern repeatedly.

This is not always “exponential” in the strict sense, but on 10k–100k line inputs it can still be extremely slow.]

Better:

- Use a more specific pattern that skips large chunks:text`[^f]* (?:f[^o]*|fo[^o])*foo`
- Or unwrap the lazy pattern if you know delimiters:text`NEM12[^<]*(?: <(?!/CSVIntervalData>)[^<]*)*</CSVIntervalData>`

]

---

6. Nested Lookaheads / Lookbehinds

Patterns like:

```
text (?!.*foo)(?!.*bar).*
(?=(.*))(?=(.*))...
```

Problems:

- Each lookahead can scan the entire remaining string.
- Nested lookaheads multiply that cost.
- On large inputs, this can explode in time.

Delphi tip:

- Avoid deep nesting of lookaheads.
- Prefer simpler patterns with explicit character classes and anchors.

]

---

7. Reusing `TRegEx` Without Re Compilation

In Delphi, creating `TRegEx` on every call is a major performance pitfall:

```
delphi // Bad: creates and compiles regex every call
if TRegEx.IsMatch(input, '\d+') then ...

// Better: reuse a compiled instance
var
  re: TRegEx;
begin
  re := TRegEx.Create('\d+');
  try
    if re.IsMatch(input) then ...
```

```
  finally
    re.Free;
  end;
end;
```


Community benchmark:

- Unoptimized regex (new per call): very slow, bottleneck.
- Optimized regex (reuse compiled instance): much faster, but still not as fast as `Pos` + loop for simple patterns. [][page:1]

---

8. Not Anchoring When You Expect Full-String Matches

Patterns like:

```
text \d+foo
```

Without ^ and $, the engine may try to match starting at many positions, especially with `.*` around them:

```
text .*\d+foo.*
```

Anchors reduce search scope and prevent scanning from every position:

```
text ^\d+foo$
```

[]]

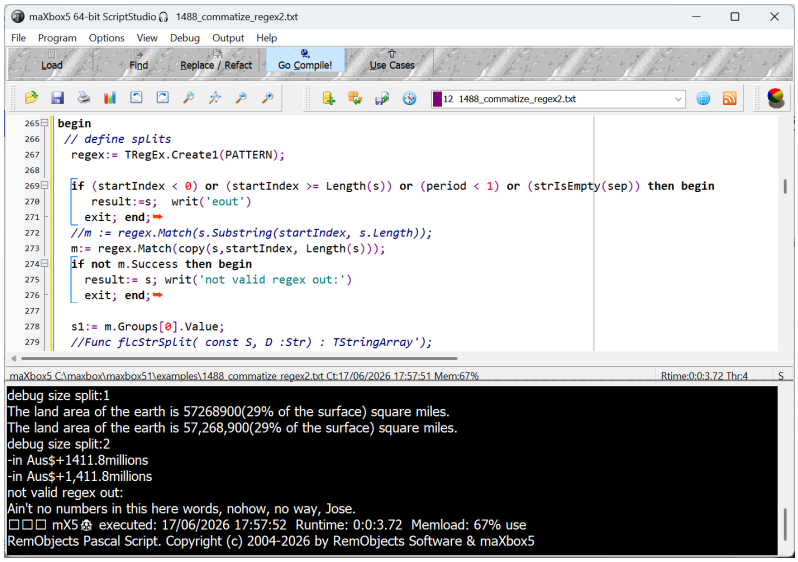
Practical Checklist for Regex

To avoid catastrophic backtracking in Delphi:

1. Avoid nested quantifiers like `(a+)+`, `(.*)+`, `([a-z]+)+`.
2. Replace `.*` with explicit character classes when you know delimiters.
3. Anchor patterns with ^ and $ when matching full strings.
4. Remove overlapping alternatives that can match the same text.
5. Prefer + over * when you don't need to allow empty matches.
6. Reuse `TRegEx` instances; don't create them in loops.
7. Test with worst-case inputs (long, near-matching, malformed) in benchmarks.
8. Limit input size and optionally add a timeout or length check before regex.
9. use ^ (start of string/line) and $ (end of string/line) if you need to match the *entire* target
10. escape metacharacters like `.`, `?`, `*`, `+`, `(`, `)`, `[`, `{`, or `$` using a backslash (\)?

11. verify brackets `[]` for doing what you want? (e.g., `[a-z]` matches letters, while `[^a-z]` negates them)
12. use `{n}` for an exact match, `{n,m}` for a range, or `{n,}` for n or more occurrences? [1, 2, 3, 4, 5]

[][][]



https://sourceforge.net/projects/maxbox5/files/examples/1488_commatize_regex2.txt/download

Posted in Code, Models, Tools

Leave a comment

← WorldNewsAPI