Intermediate Dev .NET C#

Solving complex parsing tasks with RegexTreeer

Sergiy Stoyan

3.75/5 (8 votes)

Oct 23, 2008 LGPL3 9 min read 34023 445

Solving complex parsing tasks by utilizing Regular Expression trees built with RegexTreeer.

Download sample - 160 KB

Annotation

This article can be helpful for those who potentially need to solve complex parsing tasks that require using more than one Regular Expression.

Problem background: Superposed regexes

Often it is impossible to write a single regex to parse exactly what you need from a text. In those cases, if even a single regex can be written, it may get a grotesque form that is hard to write and even harder to perform. Hence to solve such a parse task, you have to write several regexes so that they are applied in turn one after another to the text or to the results of the previous regex. This brings us to the issues listed below.

Those who have written code for such parsing knows that debugging of superposed regex constructions is dull pastime. It is so because although there are a lot of regex debug tools, in the case of superposed regexes, they become not too handy as we can debug only one regex at once. That means you have to intercept captures of the previous regex in order to debug the regex that is applied after. Keeping in mind that debugging should be performed on many matches to get confidence in the regex, it appears like a real headache. Yet again the same problem arises while updating the parser when new peculiarities in the input text are found that occurs often enough.

Another problem is that the parsing code becomes non-readable and intricate because of the presence of many regexes and superposed operations on parsing results. The code around regexes has to conform to the logic that they dictate, so in most cases, you will not be able to change the regexes without changing the code, and vice versa.

As a result, you lose your time and end up with obscure code that is a total maintenance nightmare and cannot ever be reused because it is designed to do the very specific thing that's bound to change at any moment.

This article presents a solution that eliminates these problems.

Terminology

First, a little bit of terminology.

In this article, we will use the term **regex tree**. It means 'tree of regexes' (that's a construction of superposed regexes), and so differs from the 'tree of the regex components' that is explained in other writings.

Regex tree file is an XML file of predefined format where the regex tree is stored.

Parsed data tree is a tree-like structure returned by Cliver.Parser as a result of parsing by a regex tree.

The solution essence

The general objectives of the solution are:

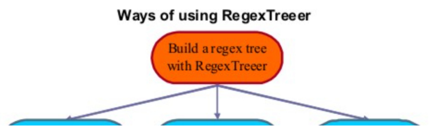
1. Simplify building and debugging regex trees with a GUI utility. It is done by a tool called RegexTreeer. It provides the following advantages:
   - modification and debugging of regex trees is visual and easy;
   - use of regexes in a tree like manner allows simplifying the regex syntax and allows complex parsing to novices in regexes;
   - regex tree is stored in an XML file of predefined format so that the regex tree can be opened and updated with RegexTreeer anytime after;
2. Keep regexes and the parsing process separated from the code where the parsed data is used. It is done with the Cliver.Parser component exposed by RegexTreeer. It provides the following advantages:
   - parsing results are formed as a tree-like structure where data can be addressed by an obvious path;
   - regexes being stored in a regex tree file do not obscure your code so that it appears simple and not-depending on regexes;
   - the same regex tree file can be used by many parsers/applications.

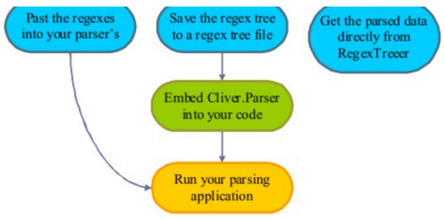
RegexTreeer

Development of a parser with RegexTreeer implies the following general steps:

1. building a regex tree with RegexTreeer;
2. storing a regex tree in a regex tree file;
3. embedding Cliver.Parser in your code;
4. getting parsed data as a tree-like structure;

Generally, after you have built a regex tree with RegexTreeer, there are three ways to use the regexes as it is displayed in the diagram:


Ways of using RegexTreeer
Build a regex tree with RegexTreeer


Past the regexes into your parser's
Save the regex tree to a regex tree file
Get the parsed data directly from RegexTreeer
Embed Cliver.Parser into your code
Run your parsing application

We'll consider how to use Cliver.Parser since it is meant as a main way of RegexTreeer use.

Cliver.Parser

Cliver.Parser is a .NET library that performs parsing text by regex trees that were built with RegexTreeer. It can be linked to your code by adding a reference to *RegexTreeer.exe* in your project.

The general objective of Cliver.Parser is to keep regexes and the parsing process separate from the code where the parsed data is used. That means the code does not depend on regexes anymore, and thus they can be changed without requiring changes in the code, and vice versa. That's why although RegexTreeer remains yet a helpful tool without Cliver.Parser, the last one makes the parsing solution perfect.

Example

To bettee understand how it works, let's consider the following example. We want to parse from the text below company names, addresses, and all information for each staff person: name, phones, mobile, email, etc., as separate fields.

Plain Text Shrink ▲

```
Company:
Orange Hotel Inc.
5823 Orange Beach, Honolulu, HI 54365

Staff:
Maria Bronte
Phone 808.373.4559
Mobile 808.306.5183
maria@orangebeach.zzz

John Thompson
Phone 888.343.4259, 888.343.4258
Mobile 888.292.5180
john@orangebeach.zzz

Company:
New Technologies Co.
43 Light River, San Francisco , CA 33456
www.newtechnologies.zzz

Staff:
Benjamin J. Jonson
Mobile 777.233.6367
johnson@newtechnologies.zzz

Victoria Gramm
Phone 777.546.1353
Mobile 777.754.9645

<...>
```

As we can depicture in advance, the parsed data that we want to obtain will be a tree-like structure that can be represented in JSON format as it is displayed below:

JavaScript Shrink ▲

```
{
    Company:{
        CompanyName:"Orange Hotel Inc.",
        CompanyAddress:"5823 Orange Beach, Honolulu, HI 54365",
        Employee:{
            EmployeeName:"Maria Bronte",
            EmployeePhone:"808.373.4559",
            EmployeeMobile:"808.306.5183",
            EmployeeEmail:"maria@orangebeach.zzz"
        }
        Employee:{
            EmployeeName:"John Thompson",
            EmployeePhone:["888.343.4259","888.343.4258"],
            EmployeeMobile:"888.292.5180",
            EmployeeEmail:"john@orangebeach.zzz"
        }
    }
    Company:{
        CompanyName:"New Technologies Co.",
        CompanyAddress:"43 Light River, San Francisco , CA 33456",
        CompanySite:"www.newtechnologies.zzz",
        Employee:{
            EmployeeName:"Benjamin J. Jonson",
            EmployeeMobile:"777.233.6367",
            EmployeeEmail:"johnson@newtechnologies.zzz"
        }
        Employee:{
            EmployeeName:"Victoria Julius Gramm",
            EmployeePhone:"777.546.1353",
            EmployeeMobile:"777.754.9645"
        }
    }
    <...>
}
```

In order to obtain structured data, we'll have to apply to the text several superposed regexes (e.g., a regex tree).

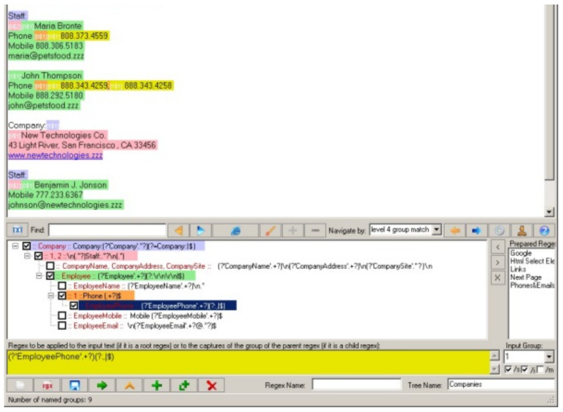
Regex tree

The needed regex tree can be built using RegexTreeer. We'll not review the RegexTreeer interface here because it is simple enough. Having taken a brief look at RegexTreeer Help, you can quickly learn how to build regex trees there. Let's imagine the regex tree is already created and saved in a regex tree file named *Companies.rgx*.

You can see the RegexTreeer screenshot with the regex tree that was built for our example:

C:\Program Files\CliverSoft\RegexTreeer\Samples\Companies.txt - SOURCE - RegexTreeer2.4.7

```
Company:
Orange Hotel Inc.
5823 Orange Beach, Honolulu, HI 54365
```



```
Staff:
Maria Bronte
Phone 808.373.4559
Mobile 808.306.5183
maria@petsfood.zzz

John Thompson
Phone 888.343.4259, 888.343.4258
Mobile 888.292.5180
john@petsfood.zzz

Company:
New Technologies Co.
43 Light River, San Francisco., CA 33456
www.newtechnologies.zzz

Staff:
Benjamin J. Jonson
Mobile 777.233.6367
johnson@newtechnologies.zzz
```

```
:: Company :: Company:(?'Company'.*?)(?=Company:|$)
  :: 1, 2 :: \n(.*?)Staff:.*?\n(.*)
    :: CompanyName, CompanyAddress, CompanySite ::   (?'CompanyName'.+?)\n(?'CompanyAddress'.+?)\n(?'CompanySite'.*?)\n
    :: Employee :: (?'Employee'.+?)(?:\r\n\r\n|$)
      :: EmployeeName :: (?'EmployeeName'.+?)\n.*
      :: 1 :: Phone (.+?)$
        EmployeePhone    (?'EmployeePhone'.+?)(?:,|$)
      :: EmployeeMobile ::  Mobile (?'EmployeeMobile'.+?)$
      :: EmployeeEmail ::  \n(?'EmployeeEmail'.+?@.*?)$
```

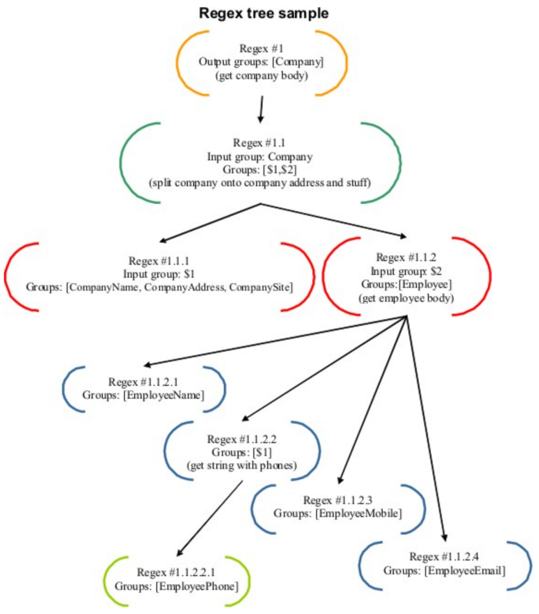
Regex to be applied to the input text (if it is a root regex) or to the captures of the group of the parent regex (if it is a child regex):

```
(?'EmployeePhone'.+?)(?:,|$)
```

Navigate by: level 4 group match
Input Group: 1
Tree Name: Companies
Number of named groups: 9

The regex tree is seen in the `TreeView` control in RegexTreeer's window. The used regex engine is .NET, so refer MSDN for the regex syntax. Please notice an important thing: those regex groups whose captures are the end data are named. You'll see below that using these group names, we reference the parsed data in our code.

The regex tree for our example has the following structure (you can see the regexes in the screenshot, or find them in the regex tree file in the code attached to this article):



As we can see from the regex tree diagram, the parsed data will be a tree of named values. This observation directs us to the next section.

Parsed data tree

The view of the regex tree suggests that it would be fine to obtain the parsing results formed as a tree-like structure – then we can manage the parsed data in our code in a clear and vivid manner. Thus, while iterating through the array of companies, we would get the company's name like `Company[i].CompanyName`, or employee's phone like `Company[i].Employee[j].EmployeePhone[k]`.

Cliver.Parser does something like this - it returns the parsed data as a tree of `GroupCapture` objects that correlates to the regex tree that was applied to text. Each string captured by a named group in the regex tree is represented by its `GroupCapture` object. Each `GroupCapture` contains its captured string and also keeps references to `GroupCaptures` originated by the next level (i.e., child) regex.

To clarify this better, let's consider the parsed data tree being a result of parsing of our example text. Below it is represented in JSON form:

Plain Text   Shrink ▲

```
{
    Company:[
        {
            value:<capture #1 of Company group>,
            CompanyName:[
                {
                    value:<capture #1 of CompanyName group>
                }
                <…GroupCaptures for the rest captures of CompanyName group…>
            ],
            CompanyAddress:[
                {
                    value:<capture #1 of CompanyAddress group>
                }
                <…GroupCaptures for the rest captures of CompanyAddress group…>
            ],
            CompanySite:[
                {
                    value:<capture #1 of CompanySite group>
                }
                <…GroupCaptures for the rest captures of CompanySite group…>
            ],
            Employee:[
                {
                    value:<capture #1 of Employee group>,
                    EmployeeName:[
                        {
                            value:<capture #1 of EmployeeName group>,
                        }
                        <…GroupCaptures for the rest captures of EmployeeName group…>
                    ]
                    EmployeePhone:[
                        {
                            value:<capture #1 of EmployeePhone group>,
                        }
                        <…GroupCaptures for the rest captures of EmployeePhone group…>
                    ],
                    EmployeeMobile:[
                        {
                            value:<capture #1 of EmployeeMobile group>,
                        }
                        <…GroupCaptures for the rest captures of EmployeeMobile group…>
                    ],
                    EmployeeEmail:[
                        {
                            value:<capture #1 of EmployeeEmail group>,
                        }
                        <…GroupCaptures for the rest captures of EmployeeEmail group…>
                    ]
                }
                <…GroupCaptures for the rest captures of Employee group…>
            ]
        }
        <…GroupCaptures for the rest captures of Company group…>
    ]
}
```

In this JSON structure, any element denoted as {…} is a `GroupCapture` object. Thus `GroupCapture`s together form a parsed data tree.

Code with Cliver.Parser

Now we only have to see how Cliver.Parser and its output can be used in .NET code. Below is a C# code snippet that prints to the console the data parsed from the example text:

C# Shrink ▲

```
Cliver.Parser company_parser = new Parser("../../_config_files/Companies.rgx");

/// <summary>
/// Process the page by Cliver.Parser
/// </summary>
/// <param name="page">text to be parsed</param>
void process_company_list(string page)
{
    Cliver.GroupCapture gc = company_parser.Parse(page);

    foreach (Cliver.GroupCapture company in gc["Company"])
    {
        Console.WriteLine("\n\n>>>>>>>Company:>>>>>>>");

        Console.WriteLine("Name: " + company.FirstValueOf("CompanyName"));
        Console.WriteLine("Address: " + company.FirstValueOf("CompanyAddress"));
        Console.WriteLine("Site: " + company.FirstValueOf("CompanySite"));

        foreach (Cliver.GroupCapture employee in company["Employee"])
        {
            Console.WriteLine("\n-------Employee:-------");
            Console.WriteLine("Name: " + employee.FirstValueOf("EmployeeName"));

            //employee can have more than one phone in
            //our sample, that's why we enum them in a cycle
            foreach (string phone in employee.ValuesOf("EmployeePhone"))
            {
                Console.WriteLine("Pnone: " + phone);
            }

            Console.WriteLine("Mobile: " + employee.FirstValueOf("EmployeeMobile"));

            Console.WriteLine("Email: " + employee.FirstValueOf("EmployeeEmail"));
        }
    }
}
```

As you can see, any parsed value can be accessed by a name path, like a certain employee's phone: `gc["Company"][0]["Employee"][0]["EmployeePhone"][1].Value`.

It looks much better than if regexes were within the code, doesn't it?

Using no-named groups

Draw your attention, a parsed data tree can contain only captures of named groups, while captures of no-named groups are not taken to the parsed data tree, in spite of the fact that they participate in the parsing process. That means, if you leave certain groups no-named, then captures of the next regex that is applied to the captures of the no-named groups are collected into one array.

In our example, leaving groups of regex #1.1 no-named means that all captures of regex #1.1.1 will be placed into one array with no distinguishing what capture of group $1 of regex #1.1 was parsed. (Of course, the same can be said about regex 1.1.2 too.) We can do so because captures of regex #1.1 are not the end data used in the code, and also, as expected, regex #1.1 has only one match within each company. Thus, leaving its groups no-named, we only made the reference path to the data shorter by one name.

Tip: use simple regexes

Regular Expressions are flexible, and powerful enough to allow in many cases writing one regex instead of two or more. However, such travail usually results in non-readable, non-editable code that is hypersensitive for the parsed text's deviations.

So do not try to use complex regexes, instead, use a tree of regexes which are as simple as possible. This approach provides a clear logic of data hierarchy that will save your development time. In most cases, it also brings the highest performance.

The conclusion

This article is only an outline of using RegexTreeer + Cliver.Parser embedded in .NET code. If you want to use this technology, you have to refer RegexTreeer's help where you can find more detailed information.

The RegexTreeer install package can be downloaded from here. It contains samples including the considered example (find *Companies.txt* and *Companies.rgx* there).

For those who are interested, the sources of RegexTreeer and Cliver.Parser can be found here.

Bug reports and feature requirements are welcome here (Tracker->Bugs and Tracker->Feature).

Using the code

In the attached code, you can find:

- *Cliver.Parser_Test* project with the considered example (see *Test2* there). It references *RegexTreeer.exe* and performs parsing with Cliver.Parser.

The latest Cliver.Parser_Test can be found here.

Issues

- RegexTreeer Help needs to be edited for better English.
- Also, the RegexTreeer GUI has no professional-style icons still.
- The following issue concerns RegexTreeer only while building regexes for HTML pages. It could not be solved as yet: implement the selection coordination between HTML code and its view in web browser the same way as it is implemented in Visual Studio's HTML editor. As a result, the selection coordination for web pages having `div` tags, performed by RegexTreeer, may be incorrect. Does anybody know how to implement it?

If anybody wants to help in any of these issues, please contact me.

Be happy!