# The functional approach to the development of parsers in C #

**For whom this article**

This article is aimed at programmers who know the language C #, who do not have immediate need in the study of purely functional programming language, but want to learn how to apply the functional approach in practice, ie solutions for very specific tasks. It is very good that, since version 3.0, C # provides syntax that allows you to express thoughts in a functional style, unless of course adhere to strict rules.

As probably you can guess from the title, the functional approach we will use to solve a particular problem - parsing text. For problems like these functional languages​​, in my personal opinion, ideal and very deeply reveal the full power of its functional paradigm. And, be honest, it was difficult to come up with some other subject areas to an imperative language C # to reveal the beauty of the functional approach.

Let's first define what is a fundamental difference in these approaches, or, as they say, paradigms?

## Function against algorithms

Functional approach contrasted imperative. This is understandable. Most of you who program in languages ​​such as C + + and / or Java, uses mandatory approach, which mainly lies in the fact that you are operating variables, structures, arrays, state computer memory by the algorithm, ie, by sequence-specific commands.

In the functional approach is different:

* No explicit action sequences , ie any algorithm reduces to call the objective function , which takes some data and returns some result ;
* no global variables that you can freely change in any function ;
* No arguments passed by reference;
* function is not the address for a block of code , and a first class object ;
* function can operate only in their arguments, and read-only ;
* functions are not allowed to call any function inside that can change the external objects (arrays, classes, structures , global variables, objects, kernel );
* in general , like, no pointers and pointer arithmetic , or rather in the computer have it all , but you can not use it directly .

And so, the functional approach, in simple terms, operates mainly functions as objects, as well as function arguments. Well, as arguments can also be other functions, because the function - is the data object. It turns out that almost all in the functional approach is to express some functions through the other. Very similar to what it looks like in mathematics. And even cycles in many cases are built using recursion.

It all sounds pretty sad - too many don'ts . But this has a huge advantage. The main advantage is that the function as such restrictions are transformed into deterministic essentially self-contained , that is not dependent on the state of the environment. In other words, there is no such likelihood and / or internal and external circumstances , for some function F, ever changed the truth of the expression F (A) == B, where A - a specific value of the argument ; B - the result of a specific value . This quality is called **lack of side effects.**

And it means that we can operate functions as anything:

- Call in any order (parallelization);   
- Cache the results of calculations (memorizatsiya);   
- Somehow combine functions, ie dynamically generate some of the other functions.

Last opportunity just opens up new approaches to the construction of clear, flexible and relatively efficient parsers.

## definitions

Before you start diving into a sea domain, it is imperative, and briefly popular, understand the terminology, which the theory has acquired many languages ​​to a "neperevarivaniya."

**Parsing** - the process of mapping a sequence of tokens (words, tokens) of a language with its formal grammar.

And now try to explain the meaning of this popular definition.

Take, for example , the language C #. This language strict syntax , it is understood . It is defined over a finite alphabet consisting of letters, numbers and other keyboard characters. His grammar is fully formalized , ie she is clothed in a concrete form with strict parsing (ie algorithm ); in general, this is the case a formal grammar . Tokens should consider all language keywords and operators . Tokens should be considered as the names of variables , classes, structures and functions .

The process of compiling the source code , such as language C #, and indeed in many other languages ​​, usually consists of several stages : lexical analysis ; check grammar and syntax tree construction ; transform tree code ( code generation ) .

Lexical analysis - this is the simplest case of a primitive pre- parsing (linear top-down algorithm ), where the text is broken into a sequence of strings ( tokens ) . The set of tokens is partitioned into disjoint subsets ( lexical classes ) . One class we refer , for example, all identifiers to another - all string literals . On each keyword (if, goto, while, ...), as a rule, create a separate class . And each class is marked with a certain identifier , called a token . Incidentally, this is why keywords are often called tokens - they symbolize something specific and they are finite , unlike hypothetically infinite set of options , such as variable names . Come on ...

A formal grammar - is the basic rules that govern mainly the order of sequence analysis ( chains) , inference rules of some language other chains . They say that the grammar is defined over the language. Language is defined over the alphabet , as language in general - is a finite set of finite words (strings , chains, tokens ) over a finite alphabet .

With a particular grammar , usually associated a specific type of algorithm - analyzer capable to recognize the language's grammar : finite state machine ; one-or two-way pushdown automaton ; Turing machine , etc.

Delve further into the grammar will not. Especially since they are closely linked with the mandatory algorithms ( all sorts of machine guns ), and our goal is to escape from it all in the functional aspect .

Instead of academic term "parsing" I will also use more approximate algorithms to concepts such as "text parsing", "parsing", "search for a match," "pattern matching". In this article, much of a difference in these terms I do not envisage.

## combinators

In functional programming, one of the main approaches to parsing is based on the so-called combinators.

Combinatorial - a function that takes as arguments to other functions and returns a function that is a combination of arguments on a particular rule. Here is a simple combiner that combines the functions of a and b by the rule of logical "AND":

Func<L, L, L> and = (a, b) => (p) => a(p) && b(p);

Combinators - should be no side effects. All they allowed is to combine the arguments and return the result.Что же можно делать с такими комбинаторами?

If you submit any parser as a function that takes as input the text , and the output returns the results of analysis in a unified form , it will be possible , using combinators do simple parsers more complex (complex) .

How many of these schemers need to analyze the text as you wish ? Well, if you like anything , you do not know ... The analysis of text , anyway , it's necessary for some rules , ie need some restrictions . For example, you read suggestions from left to right from top to bottom , by words, syllables , and that's the rules of grammar ! The order is important . Here we will analyze the text , iterating through the characters and, if necessary , we will fall back to use other options parsers. It's like life : two steps forward - one step back.

In this formulation of the problem to begin with, it will be enough two schemers who , in fact, form the parsing rules and grammar of the language in part :

* Combinator consistent application (**A & B)**. Takes as input parsers A and B, returns a parser that consistently runs parser A, then the parser B, passing the remainder of the text in the returned parser A.
* Combinator alternative use (A | B). Takes two parser (A and B), returns a parser that runs both the parser with the same input text, and combines the results.

**interesting**

In fact, we would like to invent algebra parsers. If the parser was a number, then we would most likely come up to him the operations of addition / subtraction, multiplication / division. Moreover, where there is addition, and multiplication should be as a special case of repeated addition.

In our case, we are much closer to the Boolean algebra, so our combinators operators like logical "AND" and "OR".

Also combinators, we still need some elementary parsers. What we need is something to combine with something!?

To begin with we define the signature of a simple parser

public delegate LpResult **LpsParser**(LpText text);

Then we define the input data:

// Structure describes a block of text.

public struct **LpText**

{

public string Source;

public int Index;

public int Length;

public **LpText**(string source, int index, int length)

{

Source = source; Index = index; Length = length;

}

public **LpText**(string source)

{

Source = source; Index = 0; Length = source.Length;

}

public char **this**[int relativeIndex]

{

get { return Source[Index + relativeIndex]; }

}

public static implicit operator string(LpText text)

{

return text.ToString();

}

public static implicit operator LpText(string text)

{

return new LpText(text);

}

public override string **ToString**()

{

return Length > 0 ? Source.Substring(Index, Length) : "";

}

}

And the result is a match:

// The result returned by the parser LpsParser.

public sealed class **LpResult**

{

public LpText Match;

public LpText Rest;

public **LpResult**(LpText match, LpText rest)

{

Match = match; Rest = rest;

}

public **LpResult**(LpResult r)

{

Match = r.Match; Rest = r.Rest;

}

public static LpResult **Fail**(LpText rest)

{

return new LpResult(new LpText(rest.Source, rest.Index, -1), rest);

}

public static LpResult **Empty**(LpText rest)

{

return new LpResult(new LpText(rest.Source, rest.Index, 0), rest);

}

public static LpResult **Concat**(LpResult prev, LpResult next)

{

return new LpResult(new LpText(prev.Match.Source,

prev.Match.Index, next.Rest.Index - prev.Match.Index),

next.Rest);

}

public static LpResult **Take**(LpText text, int length)

{

return new LpResult(new LpText(text.Source, text.Index, length),

new LpText(text.Source,

text.Index + length, text.Length - length));

}

public bool **Success** { get { return Match.Length >= 0; } }

}

Listing shows that the parser returns an instance of class LpResult, in which there are two fields Match and Rest. Match - is that the parser found, ie match is found. Rest - this is what remains to examine.

LpText structure describes a certain block of text. Listing shows all auxiliary functions and constructors that we need more. While there is only need to delve into the fields of classes and structures.

  Now let's write a couple of simple parsers, which analyze only one symbol:

public static LpsParser **Digit**()

{

return (text) => text.Length > 0 && char.**IsDigit**(text[0]) ?

LpResult.Take(text, 1) : LpResult.Fail(text);

}

public static LpsParser **Letter**()

{

return (text) => text.Length > 0 && char.**IsLetter**(text[0]) ?

LpResult.Take(text, 1) : LpResult.Fail(text);

}

It can be seen that the algorithm parsers and Letter Digit same and begs for generalization:

public static LpsParser **Char**(Func<char, bool> **predicate**)

{

return (p) => p.Length > 0 && **predicate**(p[0]) ? LpResult.Take(p, 1) : LpResult.Fail(p);

}

Char function is not a schemer, she is the designer of elementary parsers!

Mathematically and / or functionally we could say this: Char function takes a predicate lambda, which is a function of displaying a set of alphabet letters on a binary set of belonging to something, for example, you can display the alphabet (or set of characters) into a plurality of accessories to the numbers, t . e (3,4, B, C, 9) -> (1,1,0,0,1).

Now, it's time to write the first combiner for our simplest of parsers:

public static LpsParser **And**(this LpsParser leftParser, LpsParser rightParser)

{

return (text) =>

{

var left = leftParser(text);

if (!left.Success)

return LpResult.Fail(text);

var right = rightParser(left.Rest);

if (!right.Success)

return LpResult.Fail(text);

return LpResult.**Concat**(left, right);

};

}

Red marked auxiliary function Concat, which combines the results of the left and right one matching result.

For convenience, we will conclude our static class functions in Lp (Lambda Parsers).

public static partial class **Lp**

{

...

public static LpsParser **And**(this LpsParser leftParser, LpsParser rightParser) { ... }

public static LpsParser **Char**(Func<char, bool> predicate) { ... }

public static LpsParser **Digit**();

public static LpsParser **Letter**();

...

}

And so, we have a combinatorial and elementary parsers And, then one can attempt to parse some text:

var text = new LpText("12A 34B 56C");

var digit = Lp.Char((c) => char.IsDigit(c));

var letter = Lp.Char((c) => char.IsLetter(c));

var space = Lp.Char((c) => c == ' ' || c == '\t');

// Compose a parser that is almost the same regular expression:“\d\d[a-zA-Z]\s”

**var parser = digit.And(digit).And(letter).And(space);**

var result = parser(text);

Assert.IsTrue(result.Match.Length == 4);

Assert.IsTrue(result.Match == "12A ");

## Well, everything seems to work out. Now you can see that even with a schemer and two primitive parsers already possible to make relatively complex structures.

## Квантификаторы (quantifiers)

Above all, we restricted ourselves one schemer And, because we are more and more of a challenge to imagine that the parser can return more than one match, but several. In other words, the parser can return many options parsed. How is this possible and why it is necessary to try to explain with an example:

Suppose we need a parser that looks for a point, and then a number of any characters up until it encounters an ellipsis, for example, may return: «. A.. B ...». It would seem that the task is divided into three phases and three simple parser

1. Identify the point.
2. Miss one or more of any characters.
3. Identify ellipsis.

But you probably already guessed that the second stage is a logical error: missing any number of any characters, we will miss the ellipsis. That's where we need the analysis of all the options for parsing the second part, namely:

. stage 1) looking for a point

A stage 2) 1 found any character, then check the ellipsis

A. stage 2) 2 found any symbols, then check the ellipsis

A.. stage 2) 3 found any symbols, then check the ellipsis

A..B stage 2) 4 found any symbols, then check the ellipsis

... stage 3) found ellipsis

This so-called not **greedy search**. It is also called **lazy search**, but I personally do not see a lazy here, rather the contrary. A non-greedy search - a search that does not try to capture maximum sequence of characters at a time, and that captures the characters one by one and for each option parsing algorithm tries to continue the search using the search expression residue. In our example, multiple check the ellipsis - it is the very attempt to continue the search algorithm using the search expression residue.

Hereinafter we will keep this example in context (like in the mind), and to develop our combinators to implement a parser for it.

And so, we need a parser that does not return a result, and a few:

public delegate IEnumerable<LpResult> **LpmParser**(LpText text);

In this example, the second stage must find one or more characters. Let us write this parser and, for simplicity, we'll do it on the forehead:

private static IEnumerable<LpResult> **OneOrMoreChars**(LpText text)

{

LpsParser **any** = Lp.Char(c => true); // Parser any 1st character.

LpResult left = **any**(text); // Take the first any character

while (left.Success) // Truth is, if not reached the end

{

yield return left; // Return result for each

                                           compliance

LpResult right = **any**(left.Rest); // Take the next character

if (!right.Success)

break;

// combine the results

left = LpResult.Concat(left, right);

}

}

Then generalize to the case of function OneOrMoreChars any parsers, ie do combinatiorial:

public static LpmParser **OneOrMore**(this LpsParser parser)

{

return (p) => OneOrMore(parser, p);

}

private static IEnumerable<LpResult> **OneOrMore**(LpsParser **parser**, LpText text)

{

var left = **parser**(text);

while (left.Success)

{

yield return left;

var right = **parser**(left.Rest);

if (!right.Success)

break;

left = LpResult.Concat(left, right);

}

}

Combinators such as OneOrMore called **quantifiers**. And if generally, we can say: "quantifier called combinators such that repeatedly cause the same search function, passing as arguments the results of a previous search, and so as long as a specified condition is true."

In practice, most commonly used to quantify the following options:

1. 1. One or more times until the true satisfaction.
2. 2. Zero or more times until the true satisfaction.
3. 3. Zero or one.
4. 4. From N, but not more than K times, where 0 ≤ N <K.

Many of you know what a regular expression, so for clarity, give examples of expressions with quantifiers:

|  |  |
| --- | --- |
| \d{3,4} | Three-digit or four-digit search integer. |
| \s\* | Search space zero or more times. |
| \s{0,} | Same as \s\* |
| \.(.|\n)+?\.\.\. | Expression to search for a string in our example «. A.. B ...».  Plus with the question "+?" - Is not greedy quantifier, which looks for a match for the expression (.|\N) one or more times.  If instead of "?" Put "+", then get sverhzhadnuyu (jealous) quantification, in which the title match will be found. |

To combine parsers type LpmParser, you must add a little differently parsed results. Here is a schemer And for such parsers:

private static IEnumerable<LpResult> **Next**(this IEnumerable<LpResult> prevResults,   
 LpmParser nextParser)

{

foreach (var l in prevResults)

foreach (var r in nextParser(l.Rest))

yield return LpResult.Concat(l, r);

}

public static LpmParser **And**(this LpmParser left, LpmParser right)

{

return (p) => left(p).Next(right);

}

There had to use a helper function Next, because directly into the body of the lambda expression can not use an enumerator type yield. Although, it is not so critical, in any case, the code is not worse, and even better, since we were able to isolate functionality for reuse. In other matters, there is a realization of academic schemer And at LinQ, but it is much slower:

public static LpmParser **And**(this LpmParser left, LpmParser right)

{

Return (text) =>

(

from l in left(text) // Call the parser left, get a list of options

// apply for each option right parser.

let rs = right(l.Rest) // Apply for each balance right parser

where rs.Any() // Select only those results for which there is

// parsing options right parser

select rs.Select(r => LpResult.Concat(l, r)) // Combine the results of the   
 previous

// Compliance with the following

).

SelectMany(sec => sec); // Combine the results section.

}

**Interesting**Certainly at this point some of you have already thought about the fact that we already have to deal with two types of parsers one of which - LpsParser, is just a special case of the parser LpmParser. So is it possible to leave only one option LpmParser? You can ! But I would like you to explain why a particular case easy to leave (the first type parser ) :

- Private parser implementation with a result ( option parsing ) is almost always faster ;

- The analysis of the text is always important to understand what algorithm you use, and what options are now indiscriminately ;

- By combining parsers , and then you will receive one type of parser output , which will significantly improve the control and understanding of the principles of combination, as well as simplify debugging.

The following combinators (for different combinations of Lps and Lpm parsers) propose to implement their own, so that we can finally implement the parser for our example (well, who points):

public static LpmParser And(this LpsParser left, LpmParser right) { ... }

public static LpmParser And(this LpmParser left, LpsParser right) { ... }

But, in fact, the test case of the string matching non-greedy search form «. A.. B ...»:

[TestMethod]

public void Test\_Lapa\_NonGreedy\_OneOrMore()

{

LpsParser point = Lp.Char(c => c == '.'); // item

**LpmParser** any = Lp.Char(c => true).OneOrMore(); // Any one or more symbol times

LpsParser point3 = point.And(point).And(point); // Троеточие

**LpmParser** parser = point.And(any).And(point3); // The resulting parser

Assert.IsTrue(parser(".A..B....").**Single**().Match == .A..B...");

}

Pay attention to the notes made bold. Now you can see directly in the code where it is used non-greedy parsers that can return multiple options parsed .

And so, the quantifiers are capable of generating parsers with non-greedy algorithm. However , the multiplicity of options analysis is needed not only quantifiers . For example , the usual logical operator "OR" returns true if either operand - the truth , and it does not matter what. But parsing that is not enough - we need options for both operands , as we do not know beforehand which of the options will be successful in combination with other parsers , dismantle the remaining text .

That is why the implementation combinatorial OR, as an analogue of the logical operator , we have postponed until now , we needed LpmParser:

public static LpmParser **Or**(this LpsParser p1, LpsParser p2)

{

return (text) =>

{

var r1 = p1(text);

var r2 = p2(text);

if (r1.Success)

{

return r2.Success ? new[] { r1, r2 } : new[] { r1 };

}

else if (r2.Success)

{

return new[] { r2 };

}

return new LpNode[0];

};

}

public static LpmParser Or(this LpmParser p1, LpmParser p2)

{

return (p) => p1(p).Union(p2(p));

}

public static LpmParser Or(this LpsParser p1, LpmParser p2)

{

return (text) =>

{

var r1 = p1(text);

return r1.Success ? new[] { r1 }.Union(p2(text)) : p2(text);

};

}

**Trees**

Parsing text is divided , usually into two main tasks:

1) The problem of determining compliance with the specified text grammar.  
2) The problem of constructing a parse tree and / or parse tree .

The first task we decided surfactant , ie we have created a simple parsers and combinators that can be used , for example, to build a parser to validate a string by (-1.2E +10).

To solve the second problem , it is necessary to return the parser is not just a list of matching options , and a list of all varieties of trees indiscriminately. But first , we confine ourselves to a single tree , and then go to options.

Now define some terminology that we need to understand some of the subtleties .

**Definition (special case)**

Abstract syntax tree ( or just syntax tree ) - a finite , labeled , directed tree , in which nodes are labeled by operators , and leaves with the corresponding operands .

Parse tree - it is a syntax tree , which is built for a particular specified text without losing information about the original text. Of the parse tree can get back to the original text , from a purely syntactic tree - no. In particular, the parse tree - this tree decomposition of text , where parts also divided into parts , if necessary .

To distinguish a parse tree from the syntax tree, we give examples of trees for some mathematical expression (a + b) / (3-100), recorded text:

|  |  |
| --- | --- |
| parse tree  **(a+b)/(3–100)**  **(a+b)**  **/**  **(3–100)**  **(**  **a+b**  **)**  **a**  **+**  **b**  **(**  **3–100**  **)**  **3**  **–**  **100** | syntax tree  **/**  **+**  **–**  **a**  **b**  **3**  **100** |

The example shows that in the syntax tree nodes are the operators (/, +, -), and leaves the operands (a, b, 3, 100), and it is typical for the majority of languages. In the parse tree nodes and leaves and are blocks of text. Thus, we can conclude that the parse tree in some cases is an intermediate state between the source code and the syntax tree.

In the syntax tree is missing all too much: spaces, tabs, and other formatting characters, which do not affect the semantics of the language.

**definition**

Semantics - endowed with a system of rules for constructing meaning of language constructs.

In the syntax tree as no grouping parentheses; Though they are part of the grammar and form a recursive complexity of the language, but in the syntax tree they are not needed because tree nodes themselves are grouped.

Now it is clear that the construction of a syntax tree is necessary to know the language semantics . Well, at least you need to know , for example, that an operator that is an operand ( variable , constant , etc.). To construct a parse tree such deep knowledge is needed, so just start to disassemble the text parsers .

Returning to the heart of the combinatorial approach , repeat and stating the following : Sophisticated parsers consist of simple , simple parsers consist of elementary . Complex ( Compound ) parsers are formed by combinations of simpler parsers and / or elementary .

**interesting**

At this stage, get under a composite complex parser and a parser Compound complex. So far, there are no differences.

Get under a basic parser parser that does not consist of other parsers.

Suppose we took out some of the container parser (both from the black box). How do we know that this parser is a composite, because they all look the same and have the exact same signature!? The answer is simple - no way. And forget about it. And do not do it at all. And all because functional programming any function that without the side effects seen as whole, indivisible, atomic, monadic.

**Self determination**

Monadic - one-component, unary. Monad - (from the Greek. Moás, genus. § moádos unit, single).

How do we analyze what they are doing in other parsers parsers, and what interim analysis options are selected and which rejected? Perhaps the only option - is to remember the results of simple parsers included in the complex, and return the results. Overall, we came again to the fact that the parser should return not just a list of matching options, and a list of all varieties of trees indiscriminately.

And now show how easily and gracefully it can be done in a few lines of code. Modify class LpResult:

public sealed class **LpResult**

{

...

// The results of nested parsers

public IEnumerable<LpResult> **Children**;

// Add auxiliary constructor

public LpResult(LpText match, LpText rest, params LpResult[] children)

{

Match = match; Rest = rest; Children = children;

}

...

// modify the function

public static LpResult **Concat**(LpResult prev, LpResult next)

{

return new LpResult(new LpText(prev.Match.Source, prev.Match.Index,

next.Rest.Index - prev.Match.Index),   
 next.Rest, **prev**, **next**);

}

}

Everything is now class LpResult - this tree node. It contains the results of subsidiaries parsers or results of subsidiaries parsers. It also contains a modified function Concat, which stores the results of prev and next as child objects . Concat function used combinators And unchanged , so we more than anything in the code should not be changed .

Now, if you go back to the definition , we can see an additional requirement : the syntax tree must be labeled ! Indeed, while there is little sense from the parser , which returns a tree with a nameless nameless nodes and leaves. Nodes from each other must be somehow different, need some information that will tell us exactly who created the site and what it stands for.

Well, again this is very easy to do :

public sealed class **LpResult**

{

...

// ID node to metadata.

public string **Id**;

...

public static IEnumerable<LpNode> **Select**(IEnumerable<LpNode> nodes, string id)

{

foreach (var cn in nodes)

{

if (cn.Id == id)

yield return cn;

if (cn.Children != null)

foreach (var cnn in Select(cn.Children, id))

yield return cnn;

}

}

public IEnumerable<LpNode> **Select**(string id)

{

return LpNode.Select(new[] { this }, id);

}

}

We've added just one Id field and that's all we ever need. There was also added a helper function to search for Select node identifier.

Now we need some universal mechanism that will mark the nodes of the tree. Perfectly suitable for this specific combinator:

public static partial class **Lp**

{

...

public static LpsParser **Id**(this LpsParser parser, string id)

{

return (text) => { var res = parser(text); res.Id = id; return res;};

}

public static LpmParser **Id**(this LpmParser parser, string id)

{

return (text) => Id(parser, text, id);

}

private static IEnumerable<LpNode> Id(LpmParser parser, LpText text, string id)

{

foreach (var r in parser(text))

{

r.Id = id; yield return r;

}

}

...

}

Id combinator parser wraps a call to assign an ID result and that's all.

Next, try to take advantage of all this more than life example parse numbers written in scientific notation:

// parser numbers

public LpmParser **ScientificNumberParser**()

{

var sign = Lp.Char(c => c == '+' || c == '-').Maybe();

var digits = Lp.Digit().OneOrMore();

var exp = Lp.Char(c => c == 'E' || c == 'e').And(sign.Maybe()).And(digits);

var frac = Lp.Char(c => c == '.' || c == ',').And(digits).And(exp.Maybe());

var number = sign.And(digits).Id(“**Integer**”).Or(sign.And(digits).And(frac));

return number;

}

[TestMethod]

public void **Test\_ScientificNumber**()

{

var node = ScientificNumberParser()("-123456789000000,0042E+10");

Assert.IsTrue(node.Success);

Assert.IsTrue(node.Select(“Integer”).FirstOrDefault() == null);

var intNum = "-123456789000000";

node = ScientificNumberParser()(intNum);

Assert.IsTrue(node.Select(“**Integer**”).First().Match == intNum); // Нашли целое!

}

Now you see how easy it is to mark a parse tree, and most importantly do it selectively and intelligently!

Function Code ScientificNumberParser essentially describes the syntax tree. Analysis of this code gives us a good idea of ​​what will be the structure of the parse tree, the main thing to know what to do combinators.

## monad

It is good that we had said nothing about monads because imperative languages ​​monads, as if not needed. Rather, something like that is not called monads, and more prosaic.

Every monad defines some way to pass the result from one calculation to another

**interesting**

With regard to parsers we have not noticed how steel handle such concepts as complex / simple , a composite , complex / simple .

It's time to invest in these concepts specific and deeper meaning.

After all, we wrote a simple combination of parsers and know what it consists of combinators .

Because , in fact , the parser , which we define some transformations by - this monad .

In functional programming, use the concept monad .

It came from the mathematical theory of categories , so few people understand what it means.

Bredyatina !

for the purposes of this article we will call this and that so and so, although the strict definition is broader

**A formal grammar** - a set of rules (formulas, algorithms, etc.) to describe the formal language.

Language - is a finite set of finite words (strings, chains, tokens) over a finite alphabet.

Token - literally interpreted as a sign, symbol. But in practice, the token is also identified with a certain sequence of characters or code message key.

Token - the word (including all word forms) as an independent unit of language. In other words - is the set of sequences that have something in common with each other.