Art. Lebedev studio presents...

...site scripting language Parser3.

Parser technology author:
Konstantin Morshnev | http://www.moko.ru

Parser3 author:
Alexander Petrossian (PAF) | http://paf.design.ru

Parser3 support:
Michael Petrushin (Misha v.3) | http://misha.design.ru

Documentation authors:
Alexey Sorokin | lex_sorokin@mail.ru
Vladimir Murov | lir_vl@mail.ru
Alexander Petrossian (PAF) | http://paf.design.ru

English documentation translation authors:
Roman Mamashev | ramesses@yandex.ru
Alexander Petrossian (PAF) | http://paf.design.ru

Table of contents

How to work with the documentation

The documentation is divided into three parts.

The first deals with practical examples of how to use Parser in handling various tasks. In this part, while creating a model site, you'll learn basic opportunities provided by this language and its most commonly used constructions. It doesn't really matter what text editor you'll choose to write code in Parser. The only thing we do recommend, however, is that the editor you choose support auto brace matching and syntax coloring. The simple reason for it is that, as your code grows bigger and more complicated, you'll find it more difficult to understand what each bracket relates to. Auto brace matching will therefore make your work a lot easier. Syntax coloring is also useful, as it makes reading and editing code easier, too.

The practical examples part is divided into lessons. Each lesson starts with working code, which may be simply copied and pasted into certain files. The whole example is then analyzed, and its logic is explained. Each lesson ends with brief enumeration of all key points and recommendations on what you should consider in the future. Close study of provided lessons will give you all knowledge you need to implement your own projects in Parser.

The second part is basic syntax reference providing rules of how to write different constructions.

The third part is the reference on operators and basic classes intended to provide descriptions of methods and brief examples of how to use them.

Information on how to install and configure Parser can be found in Appendices.

Agreed notations

ABCDEFGH - Parser code in examples (colored to be distinguished from pure HTML (**Courier New, 10**) and to make understanding easier).

ABCDEFGH – Files and directories.

ABCDEFGH – Additional/reference information.

[3.1.4] – Parser version number, when this feature or option was introduced.

Symbol "|" in the reference is equal to conjunction "OR".

Introduction

> And the LORD said, I have surely seen the affliction of my people who are in Egypt, and have heard their cry by reason of their taskmasters; for I know their sorrows;
> And I am come down to deliver them out of the hand of the Egyptians, and to bring them up out of that land unto a good land flowing with milk and honey...
> (Exodus, 3, 7-8)

Parser?

Our dearest reader is now most likely wondering what it may mean. You'll get the answer quite soon, but to begin with, we'd like to make several assumptions:

The first and foremost (which is undoubtedly a prerequisite) is that you know what HTML is. If this abbreviation is unclear to you, you will surely find further reading boring and useless, since—being a programming language—Parser is designed to simplify and systematize HTML-programming.

The second and essential is that we encourage you to practice a lot (since only practice makes perfect), which presupposes that you have Parser3 handy (that is, installed). A comprehensive guide on how to install and configure Parser may be found in a relevant Appendix.

And the third (just the third) is that you have some time to spare, you're patient enough, your IQ is 50 or higher, and you're eager to make your HTML-programming easy, logical, and elegant. We, in our turn, promise that learning this language is worth spending time, since it will provide you with new valuable opportunities.

As you can see, that's not really much. All the rest is OUR concern.

Parser...

Parser was born in 1997 in Art. Lebedev Studio (www.design.ru). It was designed for those who have been creating best sites in Runet (Russian Internet) to facilitate their work and let them spend less time on routine work and more—on creative. Why should we drive nails with a microscope if there's a hammer?

That is why the most of Internet projects of the Studio are written in Parser. This technology is much simpler than anything designed for the purpose. But simplicity doesn't imply primitiveness here, as Parser can be used by both professional programmers and beginners. Now we provide this opportunity to you.

Parser's concept is quite simple. One embeds special constructions into HTML pages to be processed by Parser before a visitor can see the result. Parser handles the task of final arrangement and layout, too. This reminds a meccano, where one simply has to assemble all ready parts into different combinations. If you fall short of parts most commonly used in this meccano, you can design your own "to-measure" modules which will fulfill your personal needs. It's feasible and, indeed, quite fast to do!

You will see it for yourself when you get down to work.

Parser!

Let's sum it up. What opportunities does Parser provide? You get variables, loops, conditions, and so on—in short, everything lacked by HTML alone. Unless you use Parser, your every document will be much bigger and still many problems will remain unsolved. Parser will deliver you from repeating same instructions all the time and let you form dynamic pages reacting to user's needs, work with databases, XML, and external HTTP-servers, and quickly change pages' layout. This all can be done without complicated programming commonly needed.

Your pages will be assembled from separate ready pieces and you will only let Parser know what to take, how many to take, where to place and what succession to keep. If you need to rearrange or add something, you will just need to specify it and the rest will be done automatically. Besides, the project will become more logical and simpler to understand by means of structuring.

Very soon you will be enjoying the long-expected privilege of those who used sophisticated programming languages, which needed months or sometimes even years of learning and practice.

There is one more evident advantage: separate modules can be developed by different people, who will then be able to support and update them independently. This will ensure comfortable division of labor and possibility for many people to work over one project at the same time in most comfortable conditions.

All in all, we can count advantages of using Parser for ages. Anyway, we hope we said enough for you to get down and try. After all, doesn't our experience prove our case? Moreover, we don't charge any money. We just want Internet to become better! And we have a ready and safe solution—Parser. We are sure you'll love it the way we do.

Let's go and get it!

Lesson 1. Navigation menu

Let's begin at the beginning, as they say. Let's assume you want to build a site. The first thing you'll need to figure out is how the information on the site should be organized, how many categories, sections, etc. should be there. All these questions arise at the very first stage, which is "The site's organization".

And what should the navigation be like? A good navigation system must meet many demands. It must be simple, easily recognizable, uniform, usable, quickly loadable, and it must indicate precisely where the user is at the moment. Moreover, the site shouldn't give out "Error 404" message, that is, none of the links must be "dead". If you have previously made sites, you have probably faced the problem of proper navigation.

Is there anyone who doesn't want to have a handy solution, which could automate the whole process—a solution, which would enable you to write a code once and for ever, leaving just one place to edit further on, and add as many sections as you wish?

Creating a menu which can guide a user safely through the site is the task we want to begin this manual with. Why this? Simply because a great amount of tags like:

```
<a href="some_page.html">
```

is hard to control. What if you have to add one more section? You will have a tough time changing every page with your own hands. And, keeping in mind that "to err is human," can you be sure that after such an update your visitors won't get "Error 404" messages? Here is the problem which can be easily solved by Parser.

The solution is simple: we create a function in Parser that will generate a necessary fragment of HTML. In Parser's terminology, functions are called methods. Wherever we need such a code, we will simply command to insert the navigation menu and the page containing the menu will be created. This needs just a few simple steurips:

1. All information about our links will be stored in one file, which will further allow us to make necessary changes in just one place. In the root directory of our future site we'll thus create file sections.cfg with the following content:

section_id	*name*	*uri*
1	Mainpage	/
2	News	/news/
3	Contacts	/contacts/
4	Prices	/price/
5	Your opinion	/gbook/

Here we use a so-called tab-delimited format, where table's columns are delimited with tab character and rows—with newline character. If you copy this table into a text editor, tab and new line characters will be just pasted by it automatically. However, if you are going to create and edit such tables manually, what you should keep in mind is that when dealing with tables, we ALWAYS use tab-delimited format.

2. In the same directory (root directory) we create file auto.p, where we'll store all the parts, which Parser will use further on to construct the site. AUTO means that these parts will always be available to Parser at any time and extension ".p"—as you have probably guessed already—means... yeah, right—in the flesh!

3. File auto.p will contain the following code:

```
@navigation[]
$sections[^table::load[sections.cfg]]
<table width="100%" border="1">
      <tr>
      ^sections.menu{
            <td align="center">
                <a href="$sections.uri"><nobr>$sections.name</nobr></a>
```

```
            </td>
        }
        </tr>
</table>
```

Data stored in this file is what our navigation menu will be based on further.

All preliminary work is now complete. Now we should create the file where it all will appear (e.g. index.html) and tell Parser to insert the navigation menu. In Parser we use the term "to call method" and write it like this:

```
^navigation[]
```

Now we just open the HTML file in browser and see ready-to-serve navigation menu. From now on, we can put this magic **^navigation[]** in any page and Parser will insert our menu there. The page will be generated "on-the-fly." Gotcha!

If you can see it in your browser—congratulations! You have just entered the world of dynamic sites. Very soon you will be able to use databases to generate your pages and do many other things.

Still, between the cup and the lip a morsel may slip, as they say. Let's now analyze what we've done to succeed. Look at the code in auto.p. Don't be scared if it still seems unclear. In just a few moments we'll clear up the matter. Look at the first line, which is

```
@navigation[]
```

It looks almost exactly like ^navigation[], which we put into our page (index.html) to get the menu. The only difference is the first character (@ instead of ^), but it is this character that makes all difference—by using it, we define a method to be called later. Starting a line in Parser with character @ we imply that we now define some block to be used later. The word following character **@ (navigation)** will be the name of the new method. It is up to us to pick up a name for a method. We may call this method **let_us_place_the_menu_here**, but such a name will be harder to operate with. Still, if you wish, you may call it so.

It is vitally important to give simple and clear names. They must indicate clearly what the object will store and do. Don't fray nerves and don't waste time of yours or those, who may have to analyze your code later. Your names may be in any language, but you should keep uniformity—don't mix languages naming one object in German and another one—in Swahili...

Let's take the next line:

```
$sections[^table::load[sections.cfg]]
```

Here is the key line of our code. It is quite big, so let's examine it part by part. The line starts with **$** (dollar sign) and word **'sections'** after it. This is the way we indicate a variable in Parser. It's easy, yet worth remembering: if you see **$var** in the text, that means you deal with a variable **'var'**. A variable may contain any type of data: numbers, strings, tables, files, images, or even a piece of code. If we want to assign '**www.parser.ru**' to variable **$parser_home_url** we should use structure like this: **$parser_home_url[www.parser.ru]**. Later on, we can access the variable's value by referring to it, that is, writing **$parser_home_url** wherever we need, and then the value, which is **www.parser.ru**, will be output.

In short:

$var[…]	—assign variable
$var	—retrieve value

A detailed explanation can be found in section "Variables".

In our case, variable **$sections** will store the table taken from `sections.cfg`.

Any table in Parser is regarded as an independent object, with which only certain actions can be performed. For example, if object is a table, we can add or delete rows in it. As long as a variable can store any data, we should indicate that the value we assign is nothing else but table.

A lyrical digression:
Let's take a real world example. All vehicles can be roughly divided into certain major classes such as cars, trucks, vans, caterpillars, motorcycles, etc. Any vehicle is inevitably an object of a class. You can tell vehicles of one class from those of another, because all the vehicles belonging to a class have common characteristics, such as vehicle weight, maximum load weight, etc. Any vehicle can perform some actions like move, stand still, or break. Any vehicle has its own distinctive properties. And, what is most important, every vehicle must be CREATED, it cannot just appear by itself. When someone invents a new vehicle model, one knows what class the vehicle will belong to, what properties it will have, and what it will be able to do. It's just the same in Parser: every object belongs to a certain class. Every object of a class can be created by the constructor of this class and will inherit properties (fields) and methods (actions) common to all such objects.

Let's sum it up

Any **object** in Parser belongs to a certain **class** and has the **fields** and **methods** of this very class. To use this object, you must first create it with the class **constructor**. Learn this terminology by heart—it is what your work will be totally based on.

Let's get back to our code. We assigned the following value to variable **$sections**:

```
^table::load[sections.cfg]
```

By this, we have created an object of class **table** with constructor **load**. Common rule we use to create an object looks like the following:

```
^class::constructor[parameters]
```

A detailed description may be found in section "Passing parameters".

As a parameter here, we passed the path to our file with the table.

Variable **$sections** now contains the table with sections of our site. Parser regards it as an object of class **table** and knows precisely what actions can be performed with it. So far, we need only one method of the class—**menu**, which iterates through the table. We also need values from fields of the table itself. The syntax used to call a method of an object is:

```
^object.class_method[parameters]
```

To retrieve a value from object fields (as we deal with a definite table with the fields defined by ourselves) we use a construction:

```
$object.field
```

Now, that we know it all, we can easily see the meaning of the last part of our code:

```
<table width="100%" border="1">
      <tr>
      ^sections.menu{
            <td align="center">
                  <a href="$sections.uri"><nobr>$sections.name</nobr></a>
            </td>
      }
      </tr>
</table>
```

We generate an HTML table, where each column will contain values taken from the fields of our table **$sections**: **uri** (section's uri) and **name** (section's name). We use method **menu** to iterate through the table and retrieve data stored in it. Thus, it doesn't actually matter how many sections we have—none of them will

be lost or skipped. We are free to add or remove sections, or even change their order. All changes will be made only to file `sections.cfg` and the logic of the work will remain intact—simple but nice!

Let's summarize:

What have we done?

We have written our first piece of code in Parser and learnt how to create a navigation menu for any page of our site using data stored in a separate file.

What have we learnt?

We got a glimpse of conceptual definitions of the language (class, object, property, and method) as well as certain basic constructions of Parser.

What should we remember?

Parser is an object-oriented language. Every object belongs to a certain class, has its own properties and can use methods of the class it belongs to. To create an object one must use a constructor of the class.

Syntax of working with objects:

`$variable[value]`	Assigning a variable
`$variable`	Retrieving a variable's value
`$variable[^class_name::constructor[parameters]]`	Creating an object of class **`class_name`** and assigning it to variable
`$variable.field_name`	Retrieving the value of an object's field stored in variable
`^variable.method[]`	Calling an action (method of the class, which the object stored in the variable belongs to)

What's next?

We are going to improve our menu, because it has certain imperfections so far: it places a useless link (which leads to the page we see at the moment), has columns of different width. In our second lesson, we are going to solve these problems and add some useful extras.

Lesson 2. Navigation menu and page structure

We finished the previous lesson with pointing out imperfections in the way our menu worked. Let's now fix them. So far, our menu has a spare link to the current page, which makes our site look rather clumsy. To avoid it, we should check if a section in the menu is the current page. If so, we shouldn't place a link. To indicate current section, we should change background color in current section cell.

Open file `auto.p` and replace its content with:

```
@navigation[]
$sections[^table::load[/sections.cfg]]
<table width="100%" border="0" bgcolor="#000000" cellspacing="1">
      <tr  bgcolor="#FFFFFF">
      ^sections.menu{
            ^navigation_cell[]
      }
      </tr>
</table>
<br />

@navigation_cell[]
$cell_width[^eval(100\$sections)%]
^if($sections.uri eq $request:uri){
      <td width="$cell_width" align="center" bgcolor="#A2D0F2">
      <nobr>$sections.name</nobr>
      </td>
```

```
}{
        <td width="$cell_width" align="center">
        <a href="$sections.uri"><nobr>$sections.name</nobr></a>
        </td>
}
```

What have we changed? Not much, seemingly. But our module has been significantly improved. We have added a new method—**navigation_cell**—which is called from **navigation** method. As you have probably noticed, we introduce here a new structure:

```
^if(condition){code to execute if true}{code to execute if false}
```

What this piece does is not really hard to understand. The round brackets contain some condition and—depending on the value returned (TRUE or FALSE)—the code will follow different branches. If condition contains an expression which equals zero, the resulting value will be 'FALSE,' otherwise—'TRUE'. We use operator **if** to check whether we need to place a link on the section or not. Let's now see how the whole piece of code with condition works. We will compare two strings, where the first is the URI-string contained in column **uri** in table **sections** and the other is the current URI **($request:uri** returns string equal to the current URI). Here you may ask—and what strings can be equal? Of course those which are fully equal in length and characters contained in them.

To compare two strings, in Parser, we use the following operators:

eq – strings are equal (***equal***): **parser eq parser**
ne – strings are not equal (***not equal***): **parser ne parser3**
lt – number of characters in the first string is less than that in the second (***less than***): **parser lt parser3**
gt – number of characters in the first string is greater than that in the second (***greater than***): **parser3 gt parser**
le – number of characters in the first string is greater than or equal to that in the second (***less or equal***)
ge – number of characters in the first string is less than or equal to that in the second. (***greater or equal***)

Here is how it works: if **$sections.uri** equals **$request:uri** a link shouldn't be placed (and the table cell will have different background color—we should always try to make surfing through our site as comfortable as possible), if not—place the link, then!

Another imperfection is that we have columns of varied width. That will do if you don't really care about the way your page looks, but is, frankly speaking, rather clumsy. The problem is quite easy to solve, though: we'll just take the width of the whole menu as 100% and divide it by the number of available sections (the amount of rows in table sections). In this case, we use operator **^eval()** and the number of rows in our table (we can use object of class **table** in mathematical expressions—the numerical value of the table will then be the number of its rows). You should also remember that by using backslash instead of forward slash we use integer division.

Now, we should stop for a while to pay operator **^eval()** more attention. This operator allows us to evaluate a mathematical expression without additional variables. We simply write:

```
^eval(expression)[format]
```

By using [format] we can specify in what format we expect the result of evaluation. By specifying format as [%d] we get our number without fractional part; [%.2f] returns number with two-figure long fractional part, while [%04d] returns number without fractional part, four-figure long, and—as we put zero in front of "4" while specifying format—the absent figures in the front will be padded with zeros on the left. Sometimes we do need formatted number (For example, 12.44 $ looks more sensible than 12.44373434501 $...).

We are through with our menu—it's now ready.

The first building block of our future site is now ready. Let's now proceed to page structure. Each page may be divided into three parts, which are **header** (upper part of a page), **body** (main information including our navigation menu) and **footer** (the lower part of a page). This is a kind of general pattern for most sites.

Footer will be the same for all pages, **header** will remain the same in style but with varying content (at least, page titles will vary) and **body** will always be different but of the same style, common for all pages (for example, it may consist of two information blocks—30% and 70% wide respectively). The menu will be included in **body** block.

Every page will have the following structure:

<table>
<tr><td colspan="2">header</td></tr>
<tr><td>navigation

body_additional
(30%)</td><td>

body_main
(70%)</td></tr>
<tr><td colspan="2">footer</td></tr>
</table>

Each section will be stored in a separate method (function). Let's see how we do it:

To create our **footer** we add the following piece of code to file `auto.p`:

```
@footer[]
<table width="100%" border="0" bgcolor="#000000" cellspacing="0">
      <tr>
            <td></td>
      </tr>
</table>
$now[^date::now[]]
<font size="-3">
<center>Powered by Parser3<br />1997-$now.year</center>
</font>
</body>
</html>
```

There is nothing new here, except the piece where we use class **date**. We create it with constructor **now** to get the current date and then take the value of field **year**. If you find it unclear, please get back to our first lesson where we described working with objects by the Example of objects of class **table**. In the present case, the process is just the same, except that we use another class, which is **date**.

Module **header** is a little harder to make. On one hand, we must supply each page with unique title. On the other hand, we must stick to the same layout while generating unique content. What should we do? We are going to create, in `auto.p` file, a new function—**header**, from within which we will call another function—**greeting**. Function **greeting**, in its turn, will be defined in every page to provide unique greeting for it.

Let's add the following code to file `auto.p`:

```
@header[]
<html>
<head>
<title>Test site in Parser3</title>
</head>
<body bgcolor="#FAEBD7">
<table width="100%" border="0" bgcolor="#000000" cellspacing="1">
      <tr  bgcolor="#FFFFFF" height="60">
            <td align="center">
                  <font size="+2"> <b>^greeting[]</b></font>
            </td>
      </tr>
</table>
<br />
```

And now, the sweetest part: Parser allows us to play an amazing trick—we can once and for ever define uniform structure for all pages in `auto.p` and then—by using functions like **greeting** contained in pages themselves—get unique content for all pages (still sticking to the same layout). How does it work?

To the very beginning of file `auto.p`, we will place function **@main[]**, which will always be automatically executed by Parser in the first place. From within it, we will call functions generating pages' parts.

In the beginning of `auto.p` we thus write:

```
@main[]
^header[]
^body[]
^footer[]
```

...and provide unique title for a page by defining function **greeting**, which will be called from function **header**:

...for the main page:

```
@greeting[]
Welcome!
```

...and for the guestbook:

```
@greeting[]
Leave your mark on history...
```

and so on:

Now, as a page is loading, Parser will do the following:

1. Function **main** defined in `auto.p` will automatically run first;
2. It will call function **header**, which, in its turn, will call function **greeting**;
3. As function **greeting** is defined in the page itself, function **header** will call this very **greeting** and not **greeting** defined in any other page or even in `auto.p` itself (function overriding takes place);
4. After finishing with **greeting** and **header**, the Parser will trigger functions **body** and **footer**.

As a result, we will get a page having all necessary elements and unique greeting in its upper part. Overridable functions are also referred to as **virtual**. From within `auto.p`, we call function which may be overridden and may thus vary from page to page. At the same time, we stick to the same structure and our pages remain intact in both logic and style.

It remains only to define **body**. As we have decided, it will consist of two parts to be generated by two separate functions, for instance, **body_main** and **body_additional**. Since our navigation menu is logically related to the main part of the page, we call **navigation** from within **body** function. In this case, we should also use the mechanism of virtual functions. Thus, we should add to file `auto.p`:

```
@body[]
^navigation[]
<table width="100%" height="65%" border="0" bgcolor="#000000" cellspacing="1">
      <tr  bgcolor="#ffffff" height="100%">
            <td width="30%" valign="top" bgcolor="#EFEFEF">
                  <b>^body_additional[]</b>
            </td>
            <td width="70%" valign="top">
                  ^body_main[]
            </td>
      </tr>
</table>
<br />
```

Functions **body_main** and **body_additional** should be defined in our page the same way we did with **greeting**:

```
@body_additional[]
This is main page

@body_main[]
This is main content
```

This text can be placed in `index.html`. Well done! Structure is now ready. We have defined all necessary modules in file `auto.p`, made up uniform structure, and prepared everything to generate pages. We no longer need to write the same HTML code for every page. A common page will now look like the following (the example is given for `index.html`, the main page):

```
@greeting[]
Welcome!

@body_additional[]
This is main page

@body_main[]
This is main content
```

Simple and clear, isn't it? Everything is in its place and ready to use. After processing a code like this, Parser will create HTML code with unique title, menu, main information block (sticking to the uniform layout and style), and footer, which will be the same for all pages. In fact, we have made up a site ready to be filled with information. This is how you can make up mini business site in a couple of minutes. This is by no means the only solution, but it perfectly puts everything in its place. Some mental workload put in structuring our site will give back easy support and enhancement. All common features are stored in `auto.p` and the rest—which must be unique for every page—will be stored in pages themselves.

You are free to improvise now. If you have to change the layout of your header you will just need to open `auto.p` and change function **`header`** once. As you have done it, your every page will have new header design. If we dealt with pure HTML, we would have to rewrite every HTML page manually. This is just the same for all other modules. If you want to change the general layout (for example, to add some block) just add it as a new function and call it from within **`main`** in auto.p.

Such structure has yet another great advantage: imagine, one of your pages needs footer different from what you usually use (remember—in the beginning, we assumed that footer should be the same for all pages). All you should do is override existing **`footer`** by placing new function **`footer`** in the page. For example, put this code into `/contacts/index.html`:

```
@greeting[]
Contact us

@body_additional[]
Here are our addresses

@body_main[]
:Page's content:

@footer[]
Here are our contacts
```

...and you will change **`footer`** on this page for the one we have just given. That means, if Parser finds some function in the page, it will use it as a substitute for the function with the same name given in `auto.p`. If we don't specify footer in the page itself, Parser will use **`footer`** declared in `auto.p`.

To end with, let us give you some food for thought. We hope it will let you understand Parser better.

In our code, we used **`$request:uri`**. *It looks different from all we have dealt with so far. What is it, then? It resembles* **`$object.property`** *(value of an object's field, which we dealt with in (Lesson 1) , but instead of a dot, we use a colon. Actually, this is also a field's value, but this is not an object's field. This is a field of a class* **`request`**. *Parser doesn't provide any constructors to create objects of this class. Fields of the class are*

*generated by Parser itself and we can directly access them. In technical terms, it is called **static variable**. There are also static methods, which we will get to know as soon as in the next lesson. Such methods can also be called directly, without first creating an object of the class with the help of a constructor. Remember: static fields and methods always need a colon to be used with them. Thus, when writing* **$class:field**, *we access a field of the class itself, and as we write* **^class:method**, *we call a static method of the class. For example, we can look at class* **math** *which is designed for working with mathematical functions. It has only static methods and variables:*

$math:PI—*returns π. This is a static variable of class* **math**.

^math:random(100)—*returns a pseudorandom number from the range of 0-100. This is a static method of class* **math**.

Ways of accessing methods and fields differ only in using dot/colon.

Let's sum it up:

What have we done?

We have fixed some problems in our navigation menu, which we started building in the previous lesson, and added new blocks: **header**, **footer** and **body** to determine the way our pages will look. Now we have an elegant technique, which can help us make a site to start with in a wink.

What have we learnt?

We have learnt code branching, putting results of mathematical calculations into our pages, comparing strings, and getting present URI. We have also learnt new methods of classes **table** and **date** and a powerful tool of Parser's virtual functions.

What should we remember?

We can place function **main** in `auto.p`, and it will be run automatically. We can call any function from within another function. All functions to be called from within function **main** must be declared either in `auto.p` or inside the page. If there are two functions with the same name, the latter overrides the former, which is, in this case, ignored (we call it virtual function).

What's next?

There is always a room for perfection. We start with simple things and go further to more complex ones, such as working with forms and databases, which we'll need to make our site genuinely interactive. At the same time, we're going to learn new opportunities provided by Parser for web-developers' easy living.

Lesson 3. First step—news section

During the previous two lessons we have made up the general structure of our site. It is now nothing but an empty box and we should fill it. Nearly every site has a news section and so will ours. As we start a new section, we should first make up a menu for it. Here, we will do just the same. Our menu for news section will look like a calendar—the thing all people are well accustomed to.

To create a calendar with pure HTML is not an easy task and the code will be rather huge, but as you will see, Parser will do it quite easy. Let's go.

All files related to news section we will locate in directory `/news/`—as we have previously indicated in `sections.cfg`. First, we will create there file `auto.p`. Surprised? Yes, we can create `auto.p` files in any directory. Still, we should remember that the functions we place into these files will be accessible only to the directory they are in (including subdirectories). The reason is that we shouldn't overload one `auto.p` file with ALL functions, so, we should relieve our main `auto.p` of section-dependent functions and keep there only those functions, which we will need for ALL directories (such functions as, probably, **footer** or **header**). The directory-dependent stuff we will place in `auto.p` files in relevant directories.

One more thing: *if we redefine a function in a section's auto.p—thus overriding the function with the same name defined in root auto.p—root function will be ignored in favor of the section's function*. The virtual functions mechanism described in the previous lesson will be triggered.

Let's get back to our codes. To `/news/auto.p` we add the code:

```
@calendar[]
$calendar_locale[
	$.month_names[
		$.1[January]
		$.2[February]
		$.3[March]
		$.4[April]
		$.5[May]
		$.6[June]
		$.7[July]
		$.8[August]
		$.9[September]
		$.10[October]
		$.11[November]
		$.12[December]
	]
	$.day_names[
		$.0[Sun]
		$.1[Mon]
		$.2[Tu]
		$.3[Wed]
		$.4[Thurs]
		$.5[Fri]
		$.6[Sat]
	]
	$.day_colors[
		$.0[#000000]
		$.1[#000000]
		$.2[#000000]
		$.3[#000000]
		$.4[#000000]
		$.5[#800000]
		$.6[#800000]
	]
]
$now[^date::now[]]
$days[^date:calendar[eng]($now.year;$now.month)]
<center>
<table bgcolor="#000000" cellspacing="1">
	<tr>
		<td bgcolor="#FFFFFF" colspan="7" align="center">
			<b>$calendar_locale.month_names.[$now.month]</b>
		</td>
	</tr>
	<tr>
		^for[week_day](0;6){
			<td width="14%" align="center" bgcolor="#A2D0F2">
				<font color="$calendar_locale.day_colors.$week_day">
					$calendar_locale.day_names.$week_day
				</font>
			</td>
		}
	</tr>
^days.menu{
	<tr>
		^for[week_day](0;6){
			^if($days.$week_day){
				^if($days.$week_day==$now.day){
				<td align="center" bgcolor="#FFFF00">
					<font
```

```
color="$calendar_locale.day_colors.$week_day">
                                <b>$days.$week_day</b>
                                </font>
                        </td>
                        }{
                        <td align="center" bgcolor="#FFFFFF">
                                <font
color="$calendar_locale.day_colors.$week_day">
                                $days.$week_day
                                </font>
                        </td>
                        }
                }{
                        <td bgcolor="#DFDFDF"> </td>
                }
        }
    </tr>
}
</table>
</center>
```

We have just defined function **calendar**, which creates HTML-code of our calendar. At first, the code may seem unexpectedly big, but it's only because we are trying to handle a more complicated task. Let's now see what we've done:

The biggest piece of our code—that, which starts with **$calendar_locale**—appears strange. Look at it closely: we seem to define some data for our calendar, and it resembles a table. The piece we define as **$calendar_locale** is called 'hash' or 'associative array.' Why do we need it? As you can see, we link the ordinal numbers of months and days of the week with their names in English, and link hexadecimal color values with certain numbers. Uh-huh! Now it's getting clearer. We need hash to link (associate) a name with an object. In our case, we link numerical values of months and days of the week with their names (strings). Parser uses object model, so a string is also an object. It's quite easy to get ordinal number of a month, but a calendar for "May" seems more sensible for a human eye than a calendar for "5," and names of the days given in calendar like 0, 1, or 2 instead of Sunday, Monday, or Tuesday will look completely crazy. That's why we create an associative array.

A general way of assigning variables-hashes is like this:

```
$name[
    $.key[value]
]
```

Such a construction allows further referring to a hash key by writing **$name.key** and getting associated value. As you have probably noticed, fields of our hash are three other hashes.

After defining hash we see variable **now**, which we use to store present date, but further, there is a completely strange construction:

```
$days[^date:calendar[eng]($date.year;$date.month)]
```

Its logic is similar to that of constructor, since variable **days** now contains a table with a calendar for the current month of this year. Still, we see only one semicolon, whereas for constructor we use two. That shows us that calendar is one of static methods of class **date**. Static methods, like constructors, which we already know, can return objects. That is why we should assign created object to a variable. We have already touched upon static fields and methods in the end of the previous lesson. Such methods exist due to the fact that certain objects or their properties, such as page's URI or calendar for the current month, are unique. That is why such objects and fields comprise a separate group and can be accessed directly, without using constructors. If we call a static field, we get the value of the field of the class itself (but NOT of the object).

Class math, which is designed for working with mathematical functions, can serve as an example. As π is unique, we refer to it by writing **$math:PI** and get the value of static field of class **math** itself.

As a result, variable **days** will contain such table:
Table 1 (result of this code implemented on May 30, 2003)

0	*1*	*2*	*3*	*4*	*5*	*6*	**week**	**year**
			01	02	03	04	18	2003
05	06	07	08	09	10	11	19	2003
12	13	14	15	16	17	18	20	2003
19	20	21	22	23	24	25	21	2003
26	27	28	29	30			22	2003

This is the table we will work with further on. We cannot retrieve the whole content stored in variable **days** by simply writing

```
$days
```

If we do so, Parser won't understand what exactly we need—is it some row, the whole table or a value stored in some column? We also need to elaborate the content of the table so it could be understandable for a human being. For this purpose we have created a hash with names of days and months. Further, we use HTML to create a table where the first row will contain the name of the current month. To get the name of the month we use data stored in our hash, where ordinal number of a month is associated with its name.

```
$calendar_locale.month_names.[$now.month]
```

Let's see how it works: we retrieve the value of field **month_names** of hash **calendar_locale** with ordinal of the current month identified as **$now.month**. This construction will result in name of the month in English (in our case) or any other language (it depends on what language you use when specifying associated strings).

In calendar's next row, we will output names of days of the week using hash data. Let's see precisely what we're after: we will go through the numbers of days of the week (from 0 to 6) step by step and output strings which we have previously associated with these numbers (that is retrieve them from the field **day_names** of hash **calendar_locale**). The best way to do it is to use a loop, i.e. a succession of actions to be executed a certain number of times. We will use loop **for**. The syntax for this loop is:

```
^for[counter](counter values' range, for example 0;6){succession of counter' values}
```

One of the best things provided by loops is that we can use values of the counter within the loop, referring to it as to a variable. That's what we'll do:

```
^for[week_day](0;6){
      <td width="14%" align="center" bgcolor="#A2D0F2">
            <font color="$calendar_locale.day_colors.$week_day">
                  $calendar_locale.day_names.$week_day
            </font>
      </td>
}
```

It is simple if you know what a loop is: sequentially changing the value of **week_day**, starting with 0 and ending with 6 (where **week_day** is loop counter), we get seven values:

$calendar_locale.day_colors.$week_day —font color
$calendar_locale.day_names.$week_day —name of day of the week.

The idea behind this is just the same as that we used to get months' names, but here we use different hash keys.

You would most probably ask, 'why is there **day_colors** key?' The answer is 'fine feathers make fine birds.' If we want our calendar to look very much like real, we should make weekdays different from rest-days.

The next block needs special attention. Let's examine the task carefully. We will:

1. go through the rows of table days (Table 1) step by step;
2. in each row, go through the columns step by step, outputting the values (which are days of the month);
3. correctly indicate empty cells in our calendar (that is to check if there are blanks in the first and the last weeks of the month)
4. Indicate current date (output it in different color and make it bold).

How will we do it? The first step is easy to make with the help of familiar method **menu** of class **table**:

```
^days.menu{...}
```

To go through the columns in each row, we'll use the loop **for**, which we have recently learnt:

```
^for[week_day](0;6){...}
```

To check whether we should output empty cells we will use operator **if**. In fact, any check can be done with this operator:

```
^if($days.$week_day){
        ...
}{
        <td bgcolor="#DFDFDF"> </td>
}
```

Note: in our condition we do not compare **$days.$week_day** with anything. Thus we perform zero-check. Parser understands this construction like the following:

"If **$days.$week_day** exists (not empty), then do {...} otherwise output an empty grey table cell."

Most of the work is complete. Now we should only highlight current date. We can do it with another if, where we shall compare present value in table days with current date (**$days.$week_day==$now.day**):

```
^if($days.$week_day==$now.day){
        <td align="center" bgcolor="#FFFF00">
                <font color="$calendar_locale.day_colors.$week_day">
                        <b>$days.$week_day</b>
                </font>
        </td>
}{
        <td align="center" bgcolor="#FFFFFF">
                <font color="$calendar_locale.day_colors.$week_day">
                $days.$week_day
                </font>
        </td>
}
```

Note: to compare numerical values we use **==** operator, while to compare strings we use **eq** operator.

Let's look back at the general structure we use to form the calendar:

```
# going through the table with calendar
^days.menu{

# going through columns in each row
        ^for[week_day](0;6){
                ^if($days.week_day){
                        ^if($month.$week_day==$date.day){
                                different cell background, boldface font
                        }{
                                simple cell
                        }
```

```
            }{
                empty grey cell
            }
        }
}
```

This is not a simple block—here we use nested constructions. Still, it helps us understand the possibility to combine different means to handle a specific task. A construction could be more graceful if we united checking and coloring into a separate function to be called from within the loop. A similar solution was used in Lesson 2. By doing so, we could make our code simpler and easier to read, but our main purpose here is to show you how to combine several logical structures. You can improve the code on your own. Let it be your home assignment.

If you want to be sure that the code works, you can create file `test.html` in directory `/news/` and put there a single line:

```
^calendar[]
```

Now open this page in your browser and enjoy yourself!

Let's sum it up,

What have we done?

We defined function making up calendar for the current month.

What have we learnt?

- file auto.p may be placed not only in root directory, but also in any other directory (but in this case, functions contained in it will be accessible only within the directory it is in);
- variable-hash is an array used to associate some objects with other ones. In our case, objects were strings;
- static method **`calendar`** creates a table with calendar for the current month;
- loop **`for`** allows repeating some actions specified number of times.

What should we remember?

- along with methods of objects created with constructors of a class, there exist static methods. You can access these methods directly without first creating an object;
- within loop **`for`**, we can refer to the counter as to a variable with specified name and get its current value.

As our code grows bigger, we should place comments for the code to be clearly understood. In Parser, every line starting with **#** is regarded as a comment. So far, we didn't use comments, but as we step further and further, placing comments becomes nearly vital. The following line is an example of a comment:

```
# all this text will be ignored—this is a comment !!!
```

We urge that you place comments in your code! Ideally, your code should be self-descriptive for anyone who reads it to understand the logic of the code and succession of actions. If you neglect it, you yourself may be unable to read it after a while. Remember it!

What's next?

In the next lesson, we will teach our calendar to place links on dates and—what is most important—we will learn how to work with forms and databases to create a full-blown news section.

Lesson 4. Second step—working with databases

First of all, you shouldn't be scared of the title, even if you have never dealt with databases (further referred to as DB). You cannot do without them if you want to build a flexible, easy-to-tune-up site. By refusing to work with databases you don't make your life easier but limit yourself, since databases provide many useful opportunities. Trying to build a professional site without DB is like fishing without a fishing-rod: you surely can catch a fish with your own hands, but why complicate your life? In short, if you have never dealt with DB,

you'd better start it as soon as possible and use it in all your projects. OK, let's get off this little propaganda and presume you now fully realize the necessity of working with DB.

Working with DB in Parser is easy. Parser has a good system of interacting with various DBMS (Database Management Systems), such as MySQL, Oracle, PgSQL or any ODBC-based DBMS (that is MS SQL, MS Access, etc.). Since Parser is an open-source project, one can add support for any DBMS by creating appropriate driver). To work with DB, you don't have to possess any additional skills in Parser. All you need to do is connect to a DBMS and use SQL queries that this DBMS supports. Parser may only replace apostrophes for a relevant construction (that depends on DB type) as a "fool-proof," while the rest will be transferred as-is.

There is also a special construction used for long string literals. Oracle, PgSQL and, perhaps, some servers, drivers to which may be created in the future, cannot handle long strings properly. If a string input, which is transferred, for example, from form to database, is more than 2000 [Oracle 7.x] or 4000 [Oracle 8.x] characters long, the server will report an error like "literal is too long." If you try to cheat by combining "2000 characters" + "2000 characters" there will be another error like "sum is too great." To store such constructions, we usually use data type CLOB [Oracle] and OID [PgSQL] and, to make SQL commands simplest, we should add a control comment which will be properly interpreted by a driver of SQL server:

```
insert into news text values (/**text**/'$form:text')
```

Word ***text*** *in construction* ***/**text**/*** *is the name of a column to which we input the string that follows. There must be NO spaces inside it!*

Of course, we will not try to cover in one lesson each and every opportunity Parser provides for working with various types of DBMS. We will choose MySQL as the most widely used and, therefore, included as a usual service by most of the hosting providers. Besides, it is free of charge and easy to master.

What are we going to store in our DB? Most obvious answer is news. The table with news must have the following fields: a unique number of a news article in DB (to be generated automatically by DBMS), date indicating when the news was added to DB (this we need to retrieve news related to a certain date), news header and the text (news itself). Such a structure will be simple but effective.

We also need to decide how the news will get into DB. We can use DBMS command line for this purpose, but it is not at all comfortable. If you are going to build a site for an intranet, you can use popular and simple DBMS Microsoft Access. In this case, familiar interface and copy+paste will suit the purpose well and make you a star among your colleagues for many years. We, however, propose a solution for Internet which is to create a section for administration purposes which will include a page with HTML form to input news articles right in your browser.

That's the task which is to be handled. Let's now see how we'll do it. For this lesson, you must have MySQL DBMS installed (without it the whole lot of code will simply not work).

First of all, we should create a DB **p3test**, containing sole table **news** with fields (columns) **id**, **date**, **header** and **body**:

id	**int not null auto_increment primary key**
date	**date**
header	**varchar(255)**
body	**text**

Now we create administration section, which will allow us to fill this DB with news articles. We will create directory **/admin/** and, inside it, file **index.html**, in which we put the following:

```
@greeting[]
News DB management


@body_additional[]
Adding news
```

```
@body_main[]
$now[^date::now[]]

<center>
<form method="POST">
<p>
Date: <input name="date" value="${now.year}-${now.month}-${now.day}">
     
Header: <input name="header">
</p>
<p>Body:<br />
<textarea cols="50" name="body" rows="5"></textarea>
</p>
<p>
<input type="submit" value="Add New" name="posted">   
<input type="reset" value="Cancel">
</p>
</form>

#start processing
^if(def $form:date && def $form:header && def $form:body){
      ^connect[$connect_string]{
            ^void:sql{insert into news
                  (date, header, body)
            values
                  ('$form:date', '$form:header', '$form:body')
            }
            …news added
      }
}{
   …cannot add the news ^;&#151^;  all form fields must be filled
}
</center>
```

You also need to add method **auto** before method **main** in root-directory `auto.p`. This method is used to initialize global variables, i.e. the variables which will be accessible everywhere on the site. Within this method we will set DB connect string, which we'll return to a bit later:

```
@auto[]
$connect_string[mysql://root@localhost/p3test]
```

As you see, the structure of this page is totally compliant with the general structure of all our pages. Elements **greeting**, **body** (both parts of it), **footer** and **header** are all there. By the way, do you remember how we make **header** and **footer** appear on this page? Yes, we call them from within function **main** located in root `auto.p`.

We find unfamiliar constructions only in the main part. Let's examine it. In the beginning we see a usual HTML form with current date included as a default value for field **date**. This we do to make it comfortable for users. However, the line

```
${now.year}-${now.month}-${now.day}
```

seems strange. We use curly brackets here to get a string like "2001-11-06" (this is the format we'll use to store dates in DB). If we don't place curly brackets here, Parser will report an error because it will not understand what to do with this code. In such a construction (without curly brackets), i.e.:

```
$now.year-$now.month-$now.day
```

a hyphen will be regarded as a part of the name. Remember that you should separate the name of a variable from a character that follows (dot, hyphen, semicolon, any letter or number, etc., except space character). Thus, if you need a hyphen to immediately follow the variable value, you should write:

```
${variable_name}-
```

and you will get:

```
variable_value-
```

Please, read the page with name-building rules carefully

*We would best solve the problem with date by using **here** construction **^date.sql-string[]**. You can try to do it by yourself using Parser language reference. If you still can't cope with it—don't worry, we'll show you how to do it in the next lesson.*

Let's go on. If you have already dealt with HTML forms you know that forms send the data filled in by a visitor to some scripts for further processing. In our case the script for processing data will be the page with the form itself. We will need no additional scripts.

After closing tag **</form>**, we have data processing block. First, with the help of **if**, we check whether the form fields are not blank. We might do without it, but we want to make something that will not be a mere exhibit—we want our form to work perfectly in real-world conditions. In order to check, we have to get the values of form fields. In Parser, we do it by simply referring to form fields as to static fields:

```
$form:field_name
```

The values thus retrieved we will check (whether they are blank or not) with the help of operator **def** and logical "AND" (**&&**). We have also performed such a check in Lesson 3, but we didn't use **def**, as we checked whether a table was empty or not. As you remember, a table has a numerical value, which is the number of its rows, so any non-empty table is considered definite. Here, however, we must use **def** the same way we do to check any other object. If a field of our form remained empty when submitted, the value of **$form:field_name** will be considered undefined. Now, that we are sure that all the fields are filled in, we must store them in DB. We do it by first connecting to DB and then sending an SQL query that will put the data into table. Here is how we do it:

```
^connect[$connect_string]{
      ^void:sql{insert into news
            (date, header, body)
      values
            ('$form:date', '$form:header', '$form:body')
      }
      …news added
}
```

The most comfortable thing in Parser is that, except in some rare cases, you don't have to learn any constructions to work with DB except those required by DBMS itself. Database session is contained within operator **connect** which has the syntax:

```
^connect[protocol://connect string]{methods working with SQL queries}
```

For MySQL it will look like:

```
^connect[mysql://username:password@host/data_base]{…}
```

where curly brackets contain methods working with SQL queries. A query may return some data or nothing (in our case, for example, we just add a new entry to DB and don't request any data). In Parser we use different constructions for these two types of queries. In our case, the query is written like this:

```
^void:sql{insert into news
            (date, header, body)
      values
            ('$form:date', '$form:header', '$form:body')
}
```

By the way, this is a static method of class **void** (remember the semicolon?).

The uncolored part of this construction is SQL commands. Everything is easy here. If you know SQL, you will need nothing else but if you don't, we would again strongly recommend you to study it, as the benefits of using SQL are numerous.

Do appreciate how simply and gracefully Parser interacts with DB! It provides a comprehensible access to DBMS and (except in some rare cases) requires no additional knowledge. As you see, we also can add data from our form to SQL queries using Parser constructions. The opportunities provided by this symbiosis are unlimited. DBMS handles the problems connected with data processing (as it is designed for this very purpose and suits it quite well), and we just use the results. The situation is just the same with any DBMS that you may deal with.

Now we have a form allowing us to add records to our DB. Add several records to it. Now we're going to retrieve them. Before we do it, we need to complete function **calendar**, which we created in previous lesson. We should place links on dates so that the date could be passed to our script as a form field. Such a link will then direct a user to news archive and retrieve news for the chosen date. Such an enhancement is not a hard task; we'll just have to add some HTML to **/news/auto.p**. Within operator **if** we will surround **$days.$week_day** with the anchor tags like this:

```
<a href="/news/?day=$days.$week_day">$days.$week_day</a>
```

As a result, visitors will be able to use our calendar as a menu and select news related to a certain date.

Let's now deal with **/news/index.html**. We add to it the code:

```
@greeting[]
News page, Keep up to date!

@body_additional[]
<center>News Archive for Current Month:</center>
<br />
^calendar[]

@body_main[]
$now[^date::now[]]
<b><h1>NEWS</h1></b>
$day(^if(def $form:day){
      $form:day
}{
      $now.day
})
^connect[$connect_string]{
      $news[^table::sql{select
            date, header, body
      from
            news
      where
            date='${now.year}-${now.month}-$day'
      }]
      ^if($news){
            ^news.menu{
                  <b>$news.date—$news.header</b><br />
                  ^untaint{$news.body}<br />
            }[<br />]
      }{
            Sorry, no news for selected period.
      }
}
```

The structure is usual. In additional part of **body** we place calendar by calling **^calendar[]** (remember: this function is defined in **/news/auto.p**). Information part of the page is based on data retrieved from news

database and related to the date user selected by clicking on respective link in our calendar (**where**-part of SQL query). This is a second type of SQL query, which we use to *retrieve* data. Note that our query will result in table which we'll use further on. We therefore need to create an object of class **table**.

Let's get to know another constructor of class **table**, which is based on SQL query. Its logic is similar to that of **^table::load[]**. The difference is that the source of data here is not a text file (such as we used to create navigation menu) but SQL query result, i.e. data retrieved from DB:

```
$variable[^table::sql{SQL query}]
```

You can use this constructor only within operator **^connect[]**, that is when you have connection with DB open, because SQL queries processing is handled by DBMS itself. The returned result will be a table, where column names will be the same as the headers returned by SQL server as answer to the query.

*A short digression: We recommend that you avoid constructions like **select * from ...** because an outsider, who doesn't know the structure of the table addressed, will not understand what data will be returned by DB. Such a construction can be used only when you test the script, but in final version, instead of **select ***, you should always indicate exact names of table's fields which you want to be returned.*

The rest of the code must be clear now: **if** checks whether the form field **day** (i.e. **$form:day**—the day user selected from calendar generated by function **calendar**) is defined (**def**). We do it to figure out whether the user has already chosen a day from calendar or has just come to news section following a link in navigation menu on some other page. If **$form:day** is defined we just make it the value of variable **day**. Otherwise, the value of variable **day** will be today. Then we connect to DB the same way we did when adding new records, create table **news** and fill it with the news related to requested day (SQL-query result). After that, we use method **menu** to go through the table row by row and output the news by referring to the content of its fields. Everything is now clear except one additional operator used for a specific way of outputting the text of the news:

```
^untaint{$news.body}
```

*Here, you would better put aside the lesson for awhile and read the section on operators **taint and untaint** to study the work of these operators closely. These are very important operators and you will most probably need to use them quite often. Besides, a great deal of data processing is handled by Parser itself, behind the curtain. This work isn't seen, but it's important that you understand its logic.*

Have you read it? Let's go further, then. Why do we need **untaint** here? We have a form to manage news records and we want to allow using HTML tags in our articles. It is prohibited by default, because some malicious user can put some JavaScript on your page (which could, for example, redirect user's browser to some other page). How will we do it? We will just mark this text as trustworthy by using operator **untaint**:

```
^untaint{text of news article}
```

In our case, as we don't specify the **first** parameter, the text will be untainted [as-is] (by default). That means the data will be output as it is in DB.

At last we can relax a little: news section is now complete. We can add news and retrieve news related to the date specified by user. Of course, we can improve some little things in our calendar. For example, we can make it leave the days-to-come without links (since we can view only the news for past and present, not for the future), to indicate chosen date in page header, or provide the opportunity to retrieve news of past months (presently, we have only the current month available). This, however, you can do by yourself. The knowledge you got in the previous lessons is quite enough to put these and other ideas, which you may have, into practice. Use your creativity!

Let's sum it up,

What have we done?

We have built administration section to add news articles, enhanced the function responsible for making up a calendar for the current month, filled news section with data retrieved from DB either based on user's date selection or the current date.

What have we learnt?

- the way Parser interacts with MySQL DBMS;
- two different ways of sending SQL queries (static method **sql** of class **void** and constructor **sql** of class **table**);
- operator **untaint**.

What should we remember?

To work with DB in Parser is easy and clear, all you need to know is the constructions used by DBMS itself. Don't deprive yourself of using databases in your work.

What's next?

Now, as the news section is complete, we are going to make a guestbook to keep track of our site's rating and see whether the site needs certain enhancements.

Lesson 5. User-defined classes in Parser

In all previous lessons we manipulated classes and objects predefined in Parser, such as class **table**. This class has its own methods, which we have widely used. The list of all its methods can be found in the reference. Still, if a language doesn't extend beyond basic classes, it may finally become a serious limitation. To satisfy all users' needs we allow them to create their own (user-defined) classes with methods and fields. In this lesson we will create a new class of objects.

Actually, anything may be an object: forum, guestbook, different sections or even entire site. Here we have approached the next stage of structuring—structuring at the level of objects, not methods. What did we do in previous lessons? We just divided separate code pieces into methods and called them when necessary. However, our script could be greatly improved if we included our own objects. For instance, we could create a class **forum** and use its methods: "delete message," or "show all messages" and fields, such as "number of messages". By this we provide a modular approach, which is significantly better than just using multiple scattered and unrelated functions: all code and data (methods and fields) are assembled into one whole and used with one certain object, which is "forum". In terminology used by document-oriented programming such an approach is called 'encapsulation.' Moreover, having once created class **forum** for one project, we can use it for different projects without changing anything in it.

Before we start explaining user-defined classes by the example of guestbook, which we are going to create during this lesson, we would like to remind you of the logic of working with objects. First, we must create an object of a certain class with the help of constructor and then call methods of an object of the class or the fields of the object we have created. When working with user-defined classes, we do just the same, keeping to the same sequence.

Let's again start with determining what we're going to do, since, as we'd say, clearly indicated target is half the battle. Thus, before creating a class we must understand exactly what an object of the class will do (in other words, what *methods* it will have). Let's assume our methods will: a) display messages in guestbook; b) output a form, which a visitor will need to fill to add a new message; and c) process new message and add it to guestbook. We will store our messages in DB—the same way we did with our news.

While it seems quite clear with methods of a class is quite clear, the essence of constructor remains rather vague. As we know from our previous lessons, to start working with an object we must first create it. Let's use a constructor to create a table with messages which will be further used by the method responsible for showing them.

The task is now clear. Let's now implement it. The first thing we need to do is create table **gbook** in DB **p3test**:

id	int not null auto_increment primary key
author	varchar(255)
email	varchar(255)
date	date
body	text

Now we should get the idea behind such things in Parser as class **MAIN** and inheritance. As it has been already said, a class is a unity containing all objects, their methods and fields. Class **MAIN** combines methods and fields given in **auto.p** and the requested document (for example, **index.html**). Each level in directory tree inherits methods given in **auto.p** files located in parent directories. All these methods, including those given in requested HTML document become static functions of class **MAIN** while all variables in **auto.p** files and the requested HTML document become static fields of class **MAIN**.

```
/
|__ auto.p
|__ news/
|      |___auto.p
|      |___index.html
|      |___details/
|              |________ auto.p
|              |________index.html
|__contacts/ |
               |________auto.p
               |________index.html
```

As a user loads **/news/details/index.html**, class **MAIN** will be dynamically combined from of methods given in root directory's **auto.p**, as well as **auto.p** files located in **/news/** and **/news/details/**. Methods given in **/contacts/auto.p** will not be accessible for pages in **/news/** and its subdirectories.

It is now clear with **MAIN**, but, prior to creating a user-defined class, we should first learn how we can call methods and refer to variables contained in class **MAIN** from within a user-defined class. Methods of class **MAIN** are called as static functions:

```
^MAIN:method[]
```

while variables, which are fields of class **MAIN**, are referred to as static fields:

```
$MAIN:field
```

Let's get to practice now. We add to root directory's **auto.p** another method which we can use to connect to DB and send an SQL query.

```
@dbconnect[code]
^connect[$connect_string]{$code}
# connect_string is defined in method @auto[] and is#
$connect_string[mysql://root@localhost/p3test]
```

We put this method to root auto.p so that the DB server could be easily accessible from any page—methods located in root auto.p will always be inherited. Note: we reserve place for an argument. In our case the argument is one—code, with which we will submit SQL-queries. We can declare more arguments for a method. In this case, we will separate them with semicolon.

Further, we create directory—for instance, **classes**—in which we will store our user-defined classes. In this directory we create file `gbook.p` (we advise you to store user-defined classes in files with name extension `.p`) and put into it to it the following code:

```
@CLASS
```

```
gbook

@load[]
^MAIN:dbconnect{
	$messages[^table::sql{select author, email, date, body from gbook}]
}

@show_messages[]
^if($messages){
	^messages.menu{
		<table width="100%">
			<tr>
			<td align="left"><b>$messages.author
				^if(def $messages.email){
					$messages.email
				}{
					No e-mail address
				}</b>
			</td>
			<td align="right">$messages.date</td>
			</tr>
		</table>
		<table width="100%">
		<tr>
			<td>$messages.body</td>
		</tr>
		</table>
	}[<table width="100%" border="0" bgcolor="000000" cellspacing="0">
		<tr><td> ^;</td></tr>
	</table>]
}{
	Guestbook is empty.
}

@show_form[]
<hr />
<br />

$date[^date::now[]]
<center>
<form method="POST">
<p>
Author<sup>*</sup><input name="author"><br />
E-mail  <input name="email"><br />
Text<br />
<textarea cols="50" name="text" rows="5"></textarea>
</p>
<p>
<input type="submit" value="Send" name="post" />   
<input type="reset" value="Cancel" />
</p>
</form>
</center>

@test_and_post_message[]
^if(def $form:post){
	^if(def $form:author){
		^MAIN:dbconnect{
			^void:sql{insert into gbook
				(author, email, date, body)
			values (
				'$form:author',
				'$form:email',
```

```
                        '${date.year}-${date.month}-${date.day}',
                        '$form:text'
                    )}
            }
            $response:location[$request:uri]
        }{
            <center>Field 'author' must be filled in</center>
        }
}
```

Look at the code. In first line we indicate that this is a file with user-defined class:

```
@CLASS
```

If you need to use some other user-defined class as a parent class, you should connect to it and declare it as a base class. In this case, you will have such a construction:

```
@CLASS
name of the class

@USE
file of parent class

@BASE
name of parent class
```

In the line following **@CLASS** we write the name of our class—**gbook**. You should remember that Parser is case-sensitive, so **gbook** and **Gbook** are different names. The name of the class doesn't have to be the same as the name of the file it is stored in. Moreover, you can use any non-Latin characters for your names (for example, Cyrillic).

Further in the code, we define methods of the class. We do it the same way we defined usual methods in previous lessons.

The first method, **load**, will be constructor of our class. We should remember that the purpose of a constructor is to create an object. Moreover, it can also declare variables and assign values to them. These variables are fields of an object of user-defined class. In our case, by using constructor **sql** of class **table**, we create a table. Note: in the methods of the new class we freely use methods of system classes and method **dbconnect** of class MAIN:

```
@load[]
^MAIN:dbconnect{
      $messages[^table::sql{select author, email, date, body from gbook}]
}
```

As it has already been mentioned, if we want to use methods of a class beyond it, we should specify what class we use:

```
^class_name:method[properties]
$class_name:variable
```

and if the class we use is yet another user-defined class, we should add the following construction to the beginning of the code:

```
@USE
file of parent class
```

Such a construction allows us to use module stored in another file. The description of how Parser works with paths can be found in Attachment 1.

So, our new constructor will create table with messages connecting to a specified DB. Now that it is clear with the constructor, we will need to define methods of the new class. Method **show_messages** outputs

messages contained in table **gb** created in method **load**. We go through the table, line by line, with the help of method **menu** of class **table**, which we have already used previously. There is nothing new in other methods, either:

show_form—outputs form to add a new message

test_and_post_message—checks if button **post** was clicked, if field **author** was filled in and, if all conditions were met, adds a new entry to DB using method **dbconnect** defined in class **MAIN**.

By this we finish creating user-defined class **gbook**. All we need to do now is tell Parser on what page we are going to use it. We do it by writing in the first line of **/gbook/index.html**:

```
@USE
/classes/gbook.p
```

Now we can create object of class **gbook** and use its methods within this page. We will do it in the main information part:

```
@body_main[]
Parser3 Example: Guestbook<br />
<hr />

$gb[^gbook::load[]]
^gb.show_messages[]
^gb.show_form[]
^gb.test_and_post_message[]

# and, of course, we shouldn't forget about other parts
@greeting[]
Leave your mark on history...

@body_additional[]
Chronicles...
```

In this piece we use an object of newly created user class the same way we use any other object: we create it by using constructor of the class and then call methods defined in the new class. See how gracious the solution turned out to be: our code is clearly readable and, looking at this piece, we instantly understand what it does. Everything related to our guestbook is located in a separate file where we list all of its opportunities. If we need a new method to use with our guestbook, we will just need to add it to `/classes/guestbook.p`. Everything can be easily enhanced and it doesn't take much to understand what to change and where, if we need to.

In conclusion, it should be noted that we would better place methods like **dbconnect** somewhere beyond class **MAIN** (so that **MAIN** wouldn't be overloaded with methods). Such a solution would also make the whole project easier to read and understand. We can make methods of this class available by adding construction

```
@USE
...
```

wherever we'd need to use it.

Let's sum it up,

What have we done?

We have created a user-defined class and guestbook for our site based on the class we have made.

What have we learnt?

- Class **MAIN**;

- how to create a user-defined class;
- how to pass arguments to a method.

What should we remember?

Classes are the "top level" of structuring. That is why we should always aim at dividing our code into classes. By this, you can make the logic of our projects' work most comprehensible and our further work—most comfortable.

What's next?

By this, we have finished our exemplary site. Of course, it is not perfect and shouldn't be used as it is now. Before we place it in the Internet, you still have a couple of things to do: enhance our calendar in news section, teach our guestbook to check whether messages posted by visitors are correct, etc., but we didn't target at making up a full-scale site. We just wanted to show that Parser is an easy tool to increase your productivity. Now, that you have acquired all basic skills required for full-range work, you just need to reinforce them. Now you have all necessary knowledge to do the whole rest of work by yourself. Remember, "practice makes perfect."

Good luck!

Lesson 6. Working with XML

```
<?xml version="1.0" encoding="windows-1251" ?>
<article>
    <author id="1" />
    <title>Lesson 6. Working with XML</title>
    <body>
        <para>Imagine, you are allowed to invent any tags
          with any attributes. That means, you can define
          by yourself what a tag or attribute that you
          invent means.</para>
        <para>Such a code will contain data, …</para>
    </body>
    <links>
        <link href="http://www.parser.ru/docs/lang/xdocclass.htm">Class
xdoc</link>
        <link href="http://www.parser.ru/docs/lang/xnodeclass.htm">Class
xnode</link>
    </links>
</article>
```

…but not the formatting. One person can handle preparing data and another—formatting. What they need to do is just agree on the tags they are going to use and get down to work over the project... simultaneously.

This idea is no news. There were many template-processing libraries, and many developers created yet more libraries of their own. Libraries were incompatible and totally dependent on scripting languages used, which caused dissociation among developers and made an outsider spend lots of time and efforts on learning yet another library.

Life goes on though, and now we have standards **XML** and **XSLT,** which do not depend on scripting language chosen and allow us to fully implement the idea we have shaped in the beginning. We also have standards **DOM** and **XPath**, which reveal yet more opportunities. All these standards are fully supported in Parser.

While working over this lesson, open a book describing XML and XSLT (the one you bought in the nearest bookstore yesterday) and use it as a reference.

Let's see how we could transform the above XML-coded article to HTML. First, we place the above given code into file `article.xml` and then create file `article.xsl`, where we define the tags we have invented:

```
<?xml version="1.0" encoding="windows-1251" ?>
```

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="article">
    <html>
        <head><title><xsl:value-of select="title" /></title></head>
        <body><xsl:apply-templates select="body | links" /></body>
    </html>
</xsl:template>

<xsl:template match="body">
    <xsl:apply-templates select="para" />
</xsl:template>

<xsl:template match="links">
    Related links:
    <ul>
        <xsl:for-each select="link">
            <li><xsl:apply-templates select="." /></li>
        </xsl:for-each>
    </ul>
</xsl:template>

<xsl:template match="para">
    <p><xsl:value-of select="." /></p>
</xsl:template>

<xsl:template match="link">
    <a href="{@href}"><xsl:value-of select="." /></a>
</xsl:template>

</xsl:stylesheet>
```

The data and the transformation template are ready. Now we should create **article.html**, in which we write:

```
# input xdoc document
$sourceDoc[^xdoc::load[article.xml]]

# transform xdoc document using template article.xsl
$transformedDoc[^sourceDoc.transform[article.xsl]]

# output the result as HTML
^transformedDoc.string[
    $.method[html]
]
```

The code in the first line loads XML-file and gets its DOM-interpretation in **sourceDoc**. The construction is like that loading a table—remember **^table::load[...]**? Yet, this time we do load NOT a table (thus getting an object of class **table**) but XML-document (and get an object of class **xdoc**).

The code in the second line makes input document subject to transformation according to the template defined in **article.xsl**.

The code in the third line outputs the resulted document as HTML (parameter **method** with value **html**).

In this method, we can specify all parameters allowed for **<xsl:output ... />**.
We also recommend that you specify "no indents" (parameter **indent** *with value* **no**: **$.indent[no]** *) to avoid widely-known problem with empty space in front of* **</td>**.

Now, when accessing this page, a visitor will get the result of the transformation:

```
<html>
```

```
<head><title>Lesson 6. Working with XML</title></head>
<body>
<p>Imagine, you are allowed to invent any tags
          with any attributes. That means, you can define
          by yourself what a tag or attribute that you
          invent means.</p>
<p>Such a code will contain data, ...</p>
Related links:
<ul>
<li><a href="http://www.parser.ru/docs/xdocclass.htm">xdoc class</a></li>
<li><a href="http://www.parser.ru/docs/xnodeclass.htm">xnode class</a></li>
</ul>
</body>
</html>
```

As you have probably noticed, tag **<author ... />** was not defined, so the information on the author of the article is not present in resulting HTML. Later, when you decide where you will store and output the data on authors and how you will do it, you will just need to complete the template without changing articles' content.

Note: if you don't want Net surfers to view your **.xml** *and* **.xsl** *files, you should either store these files beyond web-space (*`^xdoc::create[/../directory_outside_of_web_space/article.xml]`*) or disallow access to these files by your web-server directives (an example of how to disallow access to* **.p** *files can be found in "Appendix dedicated to installing Parser on Apache web-server").*

Let's sum it up,

What have we done?

We have created a "building block" to be further used for retrieving information stored in XML, applying XSLT-transformation, and outputting objects in HTML format.

What have we learnt?

- how to use class **xdoc**;
- how to load XML, create XSLT, and use it to transform XML and output objects of class **xdoc** as HTML.

What should we remember?

You should buy a book on XML and XSLT.

What's next?

You should read the book we've mentioned, experiment with examples it gives, and enjoy good standards. You should also read about method **postprocess** and find a way to tune it up so that every access to XML-file would output it as HTML.

Syntax

Variables

Variables can store the following types of data:

- string;
- number (int/double);
- true/false;
- hash (associative array);
- class of objects;
- object of a class (user-defined class as well);
- code;
- expression.

To use a variable, you don't have to declare it in advance.

Different types of data demand different brackets to be used while a variable is assigned:

$variable_name[string]	assigns a string (an object of class **string**) or an object of some class;
$variable_name(expression)	assigns a number or result of some mathematical expression
$variable_name{code}	assigns some code to be executed when the variable is referred to

To retrieve value stored in a variable, you should refer to the variable by name:

$variable_name—retrieves value stored in variable

Examples

Code	Result
$string[2+2] **$string**	**2+2**
$number(2*2) **$number**	**4**
$i(0) **$code{$i}** **$i(1)** **$code**	**1**
$i(0) **$string[$i]** **$i(1)** **$string**	**0**

As a part of variable's name you may use:

...value stored in another variable:
$superman[value of superman variable]
$part[man]
$super$part
will return: **value of superman variable**

$name[picture]
${name}.gif
will return string **picture.gif** (but NOT field **gif** of object **picture**)

...result of some code:

$field.[b^eval(2+3)]
will return field **b5** of object **field**.

Hash (associative array)

Hash, or associative array, allows storing associations between string keys and some values. Hash is created automatically—when a variable is assigned or a method is called—the following way:

```
$hash_name[
    $.key1[value]
    $.key2[value]
    . . .
    $.keyN[value]
```

```
]

or

^method[
    $.key1[value]
    $.key2[value]
    . . .
    $.keyN[value]
]
```

You can also create an empty hash or a copy of another hash. See "Creating an empty hash or copying existing hash".

To retrieve a value stored in a hash key use construction:

```
$hash_name.key
```

Hash allows building multi-level structures, for example, **hash of hash** where key values would be other hashes. Example:

```
$name[
    $.key1_of_level1[$.key1_of_level2[value]]
    . . .
    $.keyN_of_level1[$.keyN_of_level2[value]]
]
```

Object of a class

Creating object

```
^class::constructor[parameters]
```

Constructor of a class creates an object of this class and allows further using common fields and methods of the class. For detailed description of constructors' parameters, please refer to respective chapter.

Note: the result of constructor's work is created object, common result of a method's work is ignored (doesn't get anywhere).

Calling object

```
^class.method[parameters]
```

Calls method of the class the object belongs to. For detailed description of constructors' parameters, please refer to respective chapter.
If object is not specified, this construction calls a method of the current class (if current class lacks the method called, a method of base class will be called) or an operator. In case of identical names, operator will be preferred.

Methods can be static and dynamic.

Dynamic method–code is executed within the scope of the object

Static method code is executed within the scope of the class itself, that is, deals with not a certain object but the entire class (for example, classes **MAIN**, **math**, **mail**)

Value of an object's field

```
$object.field
```

Retrieves the value stored in object's field.

Retreaves object's fields as a hash *[3.4.0]*

```
$h[^hash::create[$object]]
```

Creates a hash with object fields as keys.

Object's system field: CLASS

`$object.CLASS`–contains reference to the object's class

You may need this value to specify the scope of code's compilation (cf. "Process. Compiling and processing string").

Object's class name: CLASS_NAME *[3.2.2]*

`$object.CLASS_NAME`–contains object's class name

Example:

```
$var[123]
$var.CLASS_NAME
```

This example print 'string'.

Static fields and methods

Calling a static method

```
^class:method[parameters]
```

Calls a static method of the class.

Note: dynamic methods of a parent class are called the same way (Cf. Creating User-defined Class).

Value of static field

```
$class:field
```

Retrieves the value stored in static field of the class.

Assigning static field

```
$class:field[value]
```

Assigns value to static field of the class.

User-defined classes and operators

A user class is defined in a file of such a format:

```
@CLASS
name_of_class

# optional
@OPTIONS    [3.3.0]
locals
partial

# optional
@USE
file_with_parent_class

# optional
# user-defined class can't be based on system classes [3.4.0]
@BASE
name_of_parent_class

# recommended way to name method-constructor of the class
@create[parameters]

# other methods are defined
@method1[parameters]
...
```

Module can be linked (see "Linking modules") to any file, which will then be able to use the class defined here. If unknown class was specified, the method **autouse** of class **MAIN** will be called and specified class name will be passed to it as the only parameter. ***[3.4.0]***

If **@CLASS** is not specified, the file will define a number of additional operators.

If method
@auto[]

is defined, it will be automatically called as a static method (so-called static constructor) each time the class is loaded. It is used to initialize static fields of the class.

Note: result of the method's work is ignored, i.e. doesn't get anywhere.

If method is defined to receive the parameter:
@auto[filespec]
In that parameter Parser will pass full name of file containing the method.

Created classes inherit methods of parent classes. Inherited methods can be redefined.

In case one user class must use another one as parent, the file with the parent class should be linked to it, and parent class – declared as base (**@BASE**).

To use methods and fields of parent classes, the following constructions should be used:

^class:method[parameters] –to call a method of parent class (*note: although the syntax of calling such a method looks like the syntax of calling a static method, in fact, in case of dynamic method, the method of parent class will be called dynamically*). To refer to the nearest parent class (base class) you may use constructions **^BASE::constructor[parameters]** and **^BASE:method[parameters]**.

Using **@OPTIONS** you can set additional class behaviour. ***[3.3.0]***
Thus option **locals** automatically declare all variables in all methods of this class as local. If this option specified you must use system variable self for accessing to class or object's field.

With **partial** option you can allow future class modifications. If specified and you load new file with use operator which contain the same class name and the same option, the methods from this file will be added to previously loaded class instead of creating new one with the same name. This option can be useful for loading huge and seldom used methods to the class by demand.

Working with variables in static methods

Value of the variable is searched for in:

- the list of local variables;
- the current class or its parent classes.

The value will be assigned to already existing variable (see the search area given above), if it does exist. Otherwise, a new variable (field) will be created within the current class.

Working with variables in dynamic methods

Value of the variable is searched for in:

- the list of local variables;
- the current object and its class;
- parent objects and their classes.

The value will be assigned to already existing variable (see the search area given above), if it does exist. Otherwise, a new variable (field) will be created within the current class.

Note: try to avoid using fields of class beyond the methods of the class except simplest cases. We should try to communicate with an object through its methods only.

Object's system field: CLASS

$class:CLASS—contains reference to the object's class

You may need this value to specify the scope of code's compilation (cf. "Process. Compiling and processing string").

This reference can also be used to retrieve static fields of the class, for example:

```
@main[]
^method[$cookie:CLASS]

@method[storage]
$storage.field
```

As a result, value of **$cookie:field** be output.

Object's class name: CLASS_NAME *[3.2.2]*

$object.CLASS_NAME—contains object's class name

Example:

```
$var[123]
$var.CLASS_NAME
```

This example print 'string'.

Methods and user-defined operators

```
@name[parameters]
body

@name[parameters][local;variables]
body
```

Method is a code block, which has name, accepts parameters, and returns result. Names of a method's parameters are separated by semicolon. Method can also have local variables, which should be declared in method's header after declaration of parameters. Names of local variables are also separated by semicolon.

Local variables are visible only within the operator or method they belong to and from within the operators or methods they refer to (cf. **$caller** described further in the text).

While defining a method, you can use not only parameters and local variables but also any other names, thus working with fields of a class or object. This will depend on how you called the method statically, or dynamically.

In Parser, you can extend core set of operators, since methods of class MAIN are considered operators. *Important notice: operators are methods of class MAIN, but in contrast to other classes' methods, you can call them from any other class by using their name only, i.e. instead of using sophisticated* **^MAIN:include[…]**, *you can use just* **^include[…]**.

System variable: self

All methods and operators have a local variable **self**. It contains reference to the current object; in static methods, its content is the same is that of **$CLASS**.

Example:

```
@main[]
$a[Static field ^$a of class MAIN]
^test[Method's parameter]

@test[a]
^$a - $a  <br />
^$self.a - $self.a
```

The code will output:

```
$a - Method's parameter
$self.a - Static field $a of class MAIN
```

System variable: result

All methods and operators have a local variable **result**. If any value is assigned to it, it will be considered the result of the method's work. The value of **result** can be read and used in calculations.

Example:

```
@main[]
$a(2)
$b(3)
$summa[^sum[$a;$b]]
$summa

@sum[a;b]
^eval($a+$b)
$result[I won't say anything!]
```

In this case, the client will receive a string **I won't say anything!**, but not the result of addition of the two numbers.

System variable: result, explicit declaration *[3.1.5]*

If **result** variable is explicitly declared, this means to Parser that it should ignore all whitespace characters in method code and perceive as error any non-whitespace characrers, if those characters are not explicitly assigned to **result** variable.

Example:

```
@lookup[table;findcol;resultcol;findvalue;notfound][result]
^if(^table.locate[$findcol;$findvalue]){
	$table.$resultcol
}{
	$notfound
}
```

In this case, the client will receive either a value from **$resultcol** column or **$notfound** value.
What important is there would be **no** whitespace characters returned (no line breaks, tabs or spaces).

System variable: caller

All methods and operators have local variable **caller**, which stores the method's or operator's "scope of the call".

You can use it:

- to find out who called the method or operator. In this case you will need to use **$caller.self**;
- refer to—**$caller.variable_name_to_refer**—or assign **$caller.variable_name_to_assign[value]**—a variable as if you were in the place where the defined method or operator was called from.

For example, you need an operator which would be like system **for**, yet somewhat different from it. You can create it by yourself, using an opportunity to change local variable with name sent to you within **the scope of the call of your operator:**

```
@steppedfor[name;from;to;step;code]
$caller.$name($from)
^while($caller.$name<=$to){
	$code
	^caller.$name.inc($step)
}
```

Now the call...

```
@somewhere[][i]
^steppedfor[i](1;10;2){$i }
```

...will output "**1 3 5 7 9** ". Note: **it is the local variable of method somewhere that is changed.**

Notice: You may need the opportunity to find out the scope of the call to specify the scope of code's compilation (cf. "Process. Compiling and processing string".

System variable: locals, explicit declaration *[3.3.0]*

If **locals** variable is explicitly declared, this means to Parser that all variables used in the method are declarated locally.
To access object or class variables you should use self or CLASS prefixes.

Passing parameters

Parameters can be passed within different brackets and will then be processed different ways:

(expression)	–value of parameter is calculated every time it is referred to from within the method
[code]	–value of parameter is processed only once–before the method is called
{code}	–value of parameter is processed every time it is referred to from within the method

An example to demonstrate difference between brackets:

```
@main[]
$a(20)
$b(10)
^sum[^eval($a+$b)]
<hr />
^sum{^eval($a+$b)}

@sum[c]
^for[b](100;110){
      $c
}[<br />]
```

As you can see, in the first case the code was calculated only once–before method **sum** was called–and the method received the result of this calculation–number **30**. In the second case the code was executed every time the parameter was referred to–that is why the result was different each time, depending on the counter's value.

There can be many parameters or none. If you place many parameters inside single-type brackets, they can be separated by semicolon. Any combination of different types of parameters is allowed.

For example, the construction...

```
^if(condition){when true;when false}
```

...is equal to...

```
^if(condition){when true}{when false}
```

Properties

```
@GET_name[]
code, returns value

@SET_name[value]
code, accepts new $value

@GET_DEFAULT[]    [3.3.0]
@GET_DEFAULT[name]      [3.3.0]
code, executed when non-existing field is accessed

@GET[]      [3.3.0]
@GET[access type] [3.4.0]
code, executed when class/object is used in different calling contexts
```

You can define default getter (**@GET_DEFAULT[]**)–special getter, which will be executed when non-existing field is accessed. The field name, which was accessed, will be available in method only one param.
Important: it is forbidden to work with default getter as with ordinary getter: if you try to write $DEFAULT you will receive an error message.

User-defined classes may have special getter **@GET[]**, which will be executed when class/object is used in different calling contexts such as scalar context, expression, etc. The access type, which was used, will be available in the method only param. The access type values are: **def**, **expression**, **bool**, **double**, **hash**,

table or **file.**

Note: in construction $a[$b] method @GET[] is not executed.

Methods named like that define "**property**", which one can use as an ordinary variable:

we write	*Parser executes*
$name	**^GET_name[]**
$name[value]	**^SET_name[value]**

Note: if writing or reading property is not needed, corresponding method may be ommited.
Important: it is forbidden to have both properties and variables with same name.

Example: age and e-mail

Take a person. It is convenient to store it's birthday, but we often need to output the age. Person needs e-mail, but one can forget to check its validity.

Let class a handle persons, its properties "age" and "e-mail" allow us to hide unnecessary details:

```
@USE
/person.p

@main[]
$person[^person::create[
    $.name[John Dow]
    $.birthday[^date::create(2000;6;3)]
]]
# can change, but they check us
$person.email[john@dow.com]
$person.name ($person.email), age: $person.age<br />
```

Outputs:
**John Dow (john@dow.com), age: 5
** (will be older with time)

It is now allowed to change person's age:

```
# this will cause error!
$person.age(99)
```

It is not allowed to assign invalid e-mail values:

```
# this will cause error!
$person.email[john#dow.com]
```

Definition of person class

Above example works with **person** class, one must define it and it's properties.
In web-space root create `person.p` file, put this code inside it:

```
@CLASS
person

@create[p]
$name[$p.name]
$birthday[$p.birthday]

# "age" property
@GET_age[][now;today;celebday]
$now[^date::now[]]
$today[^date::create($now.year;$now.month;$now.day)]
$celebday[^date::create($now.year;$birthday.month;$birthday.day)]
# numeric value of boolean expression: true=1; false=0
$result(^if($birthday>$today)(0)($today.year - $birthday.year -
($today<$celebday)))

# "e-mail" property
```

```
@SET_email[value]
^if(!^Lib:isEmail[$value]){
    ^throw[email.invalid;Incorrect e-mail: '$value']
}
# variable name must differ from property name!
$private_email[$value]

@GET_email[]
$private_email
```

Note: class **Lib** *with method* **isEmail** *and other useful methods and operators: http://www.parser.ru/offline/examples/lib/Lib.zip.*
Note: it is better to store classes in a separate folder and not to specify path when using them. See **$CLASS_PATH**.

Example of class which is similar to table class and has additional functionality

```
@main[]
$t[^MyTable::create{a   b
0a    0b
1a    1b
2a    2b
3a    3b}]

Object value in expression: ^eval($t)<br />
^^t.count: ^t.count[]<br />
Print content of the object: ^print[$t]<br />


<br />
Copy object and print ^^c.count[]:
$c[^MyTable::create[$t]]
^c.count[]<br />
Remove 2 lines starting with offset=1 and print content of the object:
^c.remove(1;2)
^print[$c]<br />


<br />
Create new table-object based on MyTable and print ^^z.count[]:
$z[^table::create[$t]]
^z.count[]<br />


@print[t]
^t.menu{$t.a=$t.b}[<br />]
```

Definition of MyTable class

```
@CLASS
MyTable

@create[uParam]
^switch[$uParam.CLASS_NAME]{
      ^case[string;void]{$t[^table::create{$uParam}]}
      ^case[table;MyTable]{$t[^table::create[$uParam]]}
      ^case[DEFAULT]{^throw[MyTable;Unsupported type $uParam.CLASS_NAME]}
}

# method will return value in different calling contexts
@GET[sMode]
```

```
^switch[$sMode]{
      ^case[table]{$result[$t]}
      ^case[bool]{$result($t!=0)}
      ^case[def]{$result(true)}
      ^case[expression;double]{$result($t)}
      ^case[DEFAULT]{^throw[MyTable;Unsupported mode '$sMode']}
}


# method will handle access to the "columns"
@GET_DEFAULT[sName]
$result[$t.$sName]


# wrappers for all existing methods are required
@count[]
^t.count[]

@menu[jCode;sSeparator]
^t.menu{$jCode}[$sSeparator]


# new functionality
@remove[iOffset;iLimit]
$iLimit(^iLimit.int(0))
$t[^t.select(^t.offset[]<$iOffset || ^t.offset[]>=$iOffset+$iLimit)]
```

Literals

String literals

In Parser, we can use any characters. The following characters have special meaning:

```
^      $      ;      @
(      )
[      ]
{      }
"      :      #
```

To cancel special meaning of these characters you must precede them with character **^**. For example, to get **$** in the output, you will need to use **^$** in the code.

Besides, you can use character codes:
^#20 – equals to space character
^#XX – **XX** hex code of the character

Numeric literals

Numeric literals can have the following possible forms:

```
1
-8
```
(integer)

```
1.23
-4.56
```
(fractional)

1E3 equals to **1000**
-2E-6 equals to **-0.000002**
(so-called scientific notation, format: stagnat**E**exponent)

0xA8 equals to **168**
(integer in hexadecimal code)

Note: case-insensitive.

Logical literals

In Parser expressions we can use logical literals
true
false

Example

```
$exception.handled(true)
```

Literals in expressions

If a string contains spaces
or starts with digit, ***[3.1.5]***
it must be put within quotation marks or apostrophes:

Example:

```
^if($name eq John){...}
```

John is a string without spaces, so you don't have to put it within quotation marks or apostrophes.

```
^if($name eq "John Smith"){...}
```

This string contains spaces, and is therefore put within quotation marks or apostrophes.

Operators

Operators in expressions and their precedence

Operator	Value	Precedence	Comment
()	Grouping parts of expression	1 (Utmost)	
!	Logic operation NOT	2	
~	Bitwise inversion (NOT)	3	
+	Single plus	4	
-	Single minus	4	
*	Multiplication	5	
/	Division	5	Note,
\	Integer division	5	when dividing by zero
%	Modulus operator	5	you get error number.zerodivision.
+	Addition	6	
-	Subtraction	6	
<<	Bitwise left shift	7	
>>	Bitwise right shift	7	
&	Bitwise operation AND	8	
|	Bitwise operation OR	9	
!|	Bitwise operation XOR	10	
is	Check type	11	
def	Is object defined?	11	
in	Is the current document in directory?	11	
-f	Does file exist?	11	
-d	Does directory exist?	11	
==	Equal	12	
!=	Not equal	12	
eq	Strings are equal	12	
ne	Strings are not equal	12	
<	Number less than	13	
>	Number greater than	13	
<=	Number less than or equal	13	
>=	Number greater than or equal	13	
lt	String is less than	13	
gt	String is greater than	13	
le	String is less than or equal	13	
ge	String is greater than or equal	13	
&&	Logical operation AND	14	second operand is not evaluated if first is FALSE
||	Logical operation OR	15	second operand is not evaluated if first is TRUE
!||	Logical operation XOR	16 (Lowest)	

def. Checking if object is defined

The operator checks if the object is defined and returns Boolean value (true/false). The check can be performed on any object in Parser: table, string, file, object of user-defined class, etc.

def object
As undefined (not **def**) Parser regards: empty string, empty table, empty hash and code.

Example

```
$str[This is a defined string]
^if(def $str){
    String is defined
}{
    String is not defined
}
```

*Important notice: To check if **code** or **method** is defined, use operator* **is**, *not* **def**. *Thus,*
^if(def $hash.delete){-}{hash doesn't contain element delete}.

in. Checking if document is in directory

in "/directory/"

The operator checks if document is in directory and returns Boolean value (true/false).

Example

```
^if(in "/news/"){
   We're in news section
}{
   <a href="/news/">News section</a>
}
```

-f and -d. Checking if a file or directory exists

-f filename—checks if specified file exists on disk
-d dirname—checks if specified directory exists on disk

The operators check if the file/directory exist in specified location and return Boolean value (true/false).

Example

```
^if(-f "/index.html"){
   there is a mainpage
}{
   there is no mainpage
}
```

is. Checking type

object is type

The operator checks if left operand is an object of specified type and returns Boolean value (true/false). It is handy to use the operator in cases when a variable may contain a single value or a set of values (hash), as well as to check if a method is defined.

type—name of type. It may be a system name (**hash**, **junction**, ...), or name of user-defined class.

Simple type check

```
@main[]
$date[1999-10-10]
#$date[^date::now[]]
```

```
^if($date is string){
    ^parse[$date]
}{
    ^print_date[$date.year;$date.month;$date.day]
}

@parse[date_string][date_parts]
$date_parts[^date_string.match[(\d{4})-(\d{2})-(\d{2})][]]
^print_date[$date_parts.1;$date_parts.2;$date_parts.3]

@print_date[year;month;day]
Working with date:<br />
Day:   $day<br />
Month: $month<br />
Year:  $year<br />
```

This example will check the type of variable **$date** and will either perform syntactical analysis or pass to method **print_date** the fields of **$date** (if type is **object** of class **date**).

Checking if method is defined

The value of **$method_name** is also **junction**, that is why we should also use **is** and not **def** in this case.

```
@body[]
body

@main[]
Start
^if($body is junction){
   ^body[]
}{
   Method "body" is not defined!
}
Finish
```

Note: using operator **is** you can't check variables which contains **code** because of any address to such variables execute the code.

Adding comments to parts of expressions

It is possible to add comments to parts of mathematical expressions. In this case, the comments must start with # and extend until the end of the line of the file or the expression.

Example

```
^if(
    $age>=$MINIMUM_AGE  # not too young
    && $age<=$MAXIMUM_AGE  # and not too old
){
    Suitable age
}
```

Important notice: we do recommend you to add comments to parts of complex mathematical expressions. You yourself may find it difficult to understand in a while.

eval. Evaluating mathematical expressions

```
^eval(expression)
^eval(expression)[format string]
```

Operator **eval** evaluates a mathematical expression and outputs the result in the format you specify—with appropriate format string (see "Format strings").

Example

```
^eval(100/6)[%.2f]
```

will return: **16.67**.

Important notice: we do recommend that you add comments to parts of complex mathematical expressions (See "Adding comments to parts of expressions").

Branch operators

Branch operators allow choosing which of two or more tasks is to be accomplished depending on the situation.

There are two branch operators in Parser:

if—checks condition and follows one of the two branches;
switch—searches for a branch to satisfy specified string or value of specified expression.

if. Choose one of the two branches

```
^if(logical expression){code to implement if condition is "true"}

^if(logical expression){
    code to implement if condition is true
}{
    code to implement if condition is false

}
```

First, operator evaluates specified expression. Then, it decides whether to implement the code or not (first example), or whether to follow the first or the second branch (second example). There are no limitations imposed on the code. For example, it may also contain one or more **if**-statements.

Important notice: we do recommend that you add comments to parts of complex mathematical expressions (see "Adding comments to parts of expressions").

switch. Choosing one of multiple branches

```
^switch[string to compare]{
   ^case[case1]{action for 1}
   ^case[case2]{action for 2}
   ^case[case3;case4]{action for 3 or 4}
         ...
   ^case[DEFAULT]{default action}
}

^switch(mathematical expression){
   ^case(case1){action for 1}
   ^case(case2){action for 2}
   ^case(case3;case4){action for 3 or 4}
         ...
   ^case[DEFAULT]{default action}
}
```

Operator **switch** compares string or result of mathematical expression with values provided in **case** list. If compared values match it implements matching option's code. If values don't match, it implements code provided by option **DEFAULT** (must be always in uppercase).

If option **DEFAULT** is not provided and none of the **case**-options matches the value, no code will be implemented.

Example

```
^switch[$color]{
	^case[red]{Stop and think of Eternity...}
	^case[yellow]{You'd better get ready!}
	^case[green]{Show them who's the Highway King!}
	^case[DEFAULT]{You'd better not drive if you're color-blind...}
}
```

Loop-operators

Loop is a process when a certain succession of actions is executed multiple times.

There are two loop-operators in Parser:

for—number of repetitions is limited by specified counter's values

and

while—number of repetitions depends on condition.

To avoid endless loops, Parser uses built-in endless loop detection mechanism. Any loop whose body is implemented more than 20'000 times is regarded as endless.

for. Loop with specified number of repetitions

```
^for[counter](from;to){body}
^for[counter](from;to){body}[delimiter]
^for[counter](from;to){body}{delimiter}
```

Operator **for** repeatedly implements the body of the loop, going through counter's values **from** initial **to** final. Every repetition automatically increments the counter's value by one.

Counter is the name of variable used as the loop's counter.

From and **to**, respectively, are initial and final values of the counter—mathematical expressions specifying the scope of values assigned to the counter. If final value is less than the initial, the body of the loop will not be implemented at all.

Delimiter is string or code to be implemented before every non-empty body, except the first.

Important notice: since the names of the counters can be repeated, we recommend declaring them as local variables of the method which uses **for** *loop.*

You can force finish the loop using **break** operator or finish current step and go to next one using **continue** operator. ***[3.2.2]***

Example

```
^for[week](1;4){
	<a href="/news/archive.html?week=$week">News for week $week</a>
}[<br />]
```

The example outputs references to weeks 1-4. After every string, it puts the newline tag.

while. Loop with condition

```
^while(condition){body}
^while(condition){body}[delimiter]   [3.1.5]
^while(condition){body}{delimiter}   [3.1.5]
```

Operator **while** repeats the body while condition is true. If provided condition is initially false, the body will not be executed at all.

Delimiter is string or code to be implemented before every non-empty body, except the first.

You can force finish the loop using **break** operator or finish current step and go to next one using **continue** operator. ***[3.2.2]***

Example

```
<h2>TEN FAT SAUSAGES</h2>
$sausages(10)
^while($sausages > 0){
    <p>$sausages fat sausages sizzling in a pan<br />
    $sausages fat sausages sizzling in a pan<br />
    One went pop!<br />
    and the other went bang!<br />
    ^sausages.dec(2)
    There were $sausages fat sausages sizzling in a pan</p>
}[<br />]
```

break. Force finishing loop

```
^break[]
```

Operator **break** can be used inside of loop (**for**, **while**, **menu**, **foreach**) for its force finishing.
You can't use this operator outside of loop.

continue. Finishing current loops` step

```
^continue[]
```

Operator **continue** can be used inside of loop (**for**, **while**, **menu**, **foreach**) for force finishing current loops` step and going to next one.
You can't use this operator outside of loop.

connect. Connecting to a database

```
^connect[connect string]{code}
```

Operator **connect** establishes connection to DB server. The code of the operator is processed by Parser within current connection.

When used as a module to Apache or IIS) Parser caches connections to SQL-servers. In case a script attempts a connection with the same connect string twice, Parser doesn't connect to SQL-server but takes result of SQL-connection from cache, if connection is still valid.

CGI-version also caches connections, but for a single http request only. That is why you can certainly use such constructions as:

```
^connect[connect string]{…first SQL query…}
^connect[connect string]{…second SQL query…}
```

There will not be two SQL connections. It is especially useful when a connection is needed only sometimes and you cannot be sure you will always need it. In this case you may avoid doing it beforehand by doing it visually multiple times and be sure that the connection will not be broken.

We use the following methods and constructors to perform SQL-query:

```
table::sql
string:sql
void:sql
hash::sql
int:sql
double:sql
file::sql
```

Note: to work with operator **connect**, you need to have an appropriately configured driver (see Configuration).

Formats of connect string to be used with supported DB servers are described in appendix.

Example

```
^connect[mysql://admin:pwd@localhost/p3test]{
      $news[^table::sql{select * from news}]
}
```

use. Linking modules

```
^use[file]
```

Operator **use** allows using a module from specified file. If path begins with symbol "/", it will be regarded as path from Web-space root. In any other case, Parser will look for the module in directories specified in variable **$CLASS_PATH** in Configuration method.

The following construction can be used to link modules, too:

```
@USE
filename 1
filename 2
```

...

The difference between these constructions lies in that **@USE** loads a module before a code is executed, while operator **use** can be called right from the script's body. For example:

```
^if(condition){
   ^use[module1]
}{
   ^use[module2]
}
```

Note: attempts to use a module which were already used would not cause reread of that module. But it must be kept in mind, that for that full file name of the module being used must fully match with the one that were used before. In case **^use[module.p]** *and* **^use[sub/../module.p]** *were issued, Parser would consider those modules different.*

We do recommend that you save the results of code's work by linking necessary modules with operator **use** within the code of operator **cache**.

cache. Caching results of code's work

```
^cache[file](number of seconds){code}
^cache[file](number of seconds){code}{error handler}
^cache[file][expiration date]{code}
^cache[file][expiration date]{code}{error handler}
^cache[] = expiration date    [3.1.5]
```

Operator **cache** caches the string resulted from **code**'s work. Subsequent calls then do not re-execute the code, but only output cached result. It saves time and servers' resources during request processing.

We do recommend you to link modules (**^use[...]**) from within the **code** of operator **cache** instead of doing it statically (**@USE**).
We also strongly recommend that you work also with DB (**^connect[...]**) within **cache** when possible, to save your SQL-server's resources and increase your sites' productivity.

File is a name of cache-file. If this file exists and is not expired, its content will be sent to the client. If it doesn't exist, the code will be executed and result will be saved in the file with specified name.

Number of seconds is time to store result of the code's work, given in seconds. If the number is zero, the result is not saved and the file with previously cached result is deleted.

Expiration date is time, until which result of the code's work is considered valid. If the date is in the past, the result is not saved and the file with previously cached result is deleted.

Code is the code, whose result is to be cached.

Error handler—here the error in **code** can be handled. In this respect the operator resembles **try**, see section "Error handling". Unlike **try**, **$exception.handled[cache]** can be specified, which gives Parser the command to handle the error in a special way: to get from **file** the expired content, earlier saved result of **code**'s work, ignoring the fact that the content has expired.

It is possible to use within the **code** commands to change the time for the result of the code's work to be stored:

```
^cache(number of seconds)
^cache[expiration date]
```

Minimum time for the code to be stored is used.
Current expiration date can be learned by
$expire_date[^cache[]]

Example

```
^cache[/data/cache/test1](5){
      Press 'reload', changes every 5 seconds:^math:random(100)
}
```

Changing expiration time

```
^cache[/data/cache/test2](5){
      Within cache code you found out
      that the page shouldn't be cached: ^cache(0)
}
```

process. Compiling and processing string

```
^process{string}
^process[scope]{string}
^process[scope]{string}[options]
```

String will be compiled and executed as code in Parser, within specified **scope** or current scope. Specified **scope** can be an object or a class, but **not method** (this meaning if you process something inside your method, the method`s local varuables will not available inside processed code).

This operator is useful when you need to store fragments of code or your own methods in files with extension other than `.html`—and which therefore will not be processed by Parser by default—or in a DB.

Several **options** (hash) may be specified:
- **$.main[**a new name for **main** method, declared in code in **string]**
- **$.file[**a name of file, from which this **string** comes from**]**
- **$.lineno(**a line number in file, where this **string** cames from**.** *may be negative***)**

Simple examples

```
^process{@extra[]
      PS: you look really good…
}
```

Method **extra** will be added to the current class and you will be able to call it later on.

```
^process[$engine:CLASS]{@start[]
      5… 4… 3… 2… 1… GO!
}
```

Method **start** will be added to user class **engine**.

```
$running_man[^man::create[Jack]]
^process[$running_man]{
    Name: $name<br />
}
```

As the code is executed within the scope of object **$running_man**, it is able to use the object's field **name** and output "Jack".

Include operator

```
@include[filename][file]
$file[^file::load[text;$filename]]
^process[$caller.self]{^taint[as-is][$file.text]}[
      $.file[$filename]
]
```

The code loads specified file and executes it within the scope of the current object/class when **include** was called. **File** option allows us to specify the name of file, where this code were loaded from. In case there would be some error, you would see this "file name".
Note: "scope of current call" does not include any local variables or parameters!

Complex example

It is often convinient to compile a code to some method, which name evaluated dynamically:

```
# this is source code, note ^^
$source_code[2*2=^^eval(2*2)]
# it is evaluated dynamically, that we need to create the "method1" method
$method_name[method1]
# compiling source code, storing it to new method
^process{$source_code}[
      $.main[$method_name]
]
…
# later in code it can be called
^method1[]
```

This example would continue to work even if in **$source_code** there would be declared several methods, because **main option** sets the name of **main** method.

rem. Adding comments

```
^rem{comment}
```

All code contained in the operator will not be executed. Operator is used to comment code blocks.

External and internal data

While creating a script in Parser, we deal with two main types of data. One of them is part of code. The other is incoming data received from HTML-forms, environment variables, files, and SQL-servers. Part of code is not to be proofed. Yet, when the data is received from a form filled in by a visitor, for example, it is potentially dangerous to output it **as-is**. Thus, we need to transform such data according to certain rules. The lion's share of such transformations is performed by Parser automatically, on its own. For example, if Parser must output data received from an HTML-form field, characters **<** and **>** contained in the input will be automatically

substituted by **<** and **>** respectively. Yet, sometimes we will need to allow outputting this type of data to be output **as-is**, without any transformation.

The code created personally by the coder is regarded **clean**. All incoming data is considered **tainted**.

Parser code—code is created personally by the coder and is therefore not to be proofed;

$form:field—outputs data sent by user through HTML-form;

$my_table[^table::sql{sql-query}]—data is retrieved from DB.

As for **$form:field**, **tainted** data received from a form field will be automatically transformed and some characters will be substituted according to the built-in table of replacements. After this, they will be regarded as **clean**, not **tainted**. In other words, they will implicitly undergo operation **untaint**. Automatic transformation will be employed at the moment the data is output. Thus, a data retrieved from an DB and assigned to **$my_table** will be transformed when this data is output (sent to browser, saved to file or DB).

Besides, there may be a situation when the data should be either not transformed at all or transformed according to rules different from those used by default. For example, we allow a visitor to use HTML tags in the input, for example, for additional text formatting. Yet, since it is potentially dangerous (for example, a JavaScript submitted by user to guestbook may redirect other visitors' browsers to another site), Parser will by itself make replacement of "undesirable" characters according to predefined rules. This problem can be solved by using operator **untaint**.

untaint, taint. Transforming data

```
^untaint{code}
^untaint[transformation type]{code}
^taint[text]
^taint[transformation type][text]
```

Parser enables automatic data transformations to protect your system against intrusion and the "default" security level is high. It works even if your code contains no operators **taint/untaint**. If you interfere by using these operators (especially for **as-is** transformations), you may increase the risk of security vulnerability. Therefore, study the mechanism carefully before writing code.

Operator **taint** marks the **text** received as "needing transformation of a certain type". If transformation type is **unspecified**, taint marks it as "tainted" (needing undefined transformation). Text marked "tainted" is subject to the type of transformation applied to external text (coming from from field, database, file, cookies, etc.).

Operator **untaint** executes the **code** received and marks "needing transformation of a certain type" the tainted parts of the execution result (i.e. pieces that did not constitute part of the Parser code within the document body, either external or marked "tainted" by the taint operator). It does not concern parts subject to transformation of a certain type. If transformation type is **unspecified**, untaint marks the tainted pieces of the execution result as **as-is**.

Text is marked for transformation to be **performed later**, when the document is outputted to browser, sent to SQL server, saved into a file, sent out through e-mail, etc.

For simplicity you can think about it as if Parser interprets external characters as **^taint[external text]**, and text within the body as **^taint[optimized-as-is][typed text]**.

In some cases **^taint[transformation type][text]** and **^untaint[transformation type]{text}** produce the same result. It happens when the whole text is tainted (for example, **$form:field**). However, keep in mind that these operators have different default parameters, and applying both without transformation types to a tainted text will create absolutely different results.

When outputting to browser, Parser automatically applies type **optimized-html**, and the code looks like this:

```
^untaint[optimized-html]{typed code}
```

It means that if you write `$form:field` (not using **taint/untaint**) within the body, then even if "**?field=</html>**" is called, the page shall not be "crippled" due to the closing tag **</html>** appearing too early, because the content of **$form:field** is tainted and will be subjected to automatic **optimized-html** transformation that replaces greater-than and less-than signs ('<' and '>') with entity references '<' and '>'.
Other automatic transformations are performed in the same way. For instance, an SQL query containing `^string:sql{SELECT name FROM table WHERE uid = '$form:uid'}` (again, not using **taint/untaint**) cannot be subverted by SQL injection using parameter "**?uid=' OR 1=1 OR '**", because Parser shields the single quotes in the **$form:uid** received before sending the query to server.

Text within the body is also automatically transformed. Parser optimizes whitespace symbols: space, tabulation characters and line breaks. If these symbols appear in a row, they are replaced with the first one of them. In other words, if you type several spaces, they become only one before viewing. If you need to disable this optimization (for example, when using **<pre/>**), do it explicitly by writing, for instance, the following:

```
<pre>
^taint[as-is][
   I strode off the
     high cathedral
       top-most step like a
         miracle worker, or a
           Blessed
             passing the final exam for
               Saint. The
                 city expanded at my
                   feet. For one
                     pico-second, I
                       flew.
]
</pre>
```

In this case, you must use **taint,** as the typed characters are untainted and **untaint** would not produce any effect.

Example

```
$clean[<br />]
# the above expression is equivalent to this: $clean[^taint[optimized-as-
is][<br />]]

$tainted[^taint[<br />]]

Strings: ^if($clean eq $tainted){match}{do not match}<br />

Tainted data—'$tainted'<br />
Untainted data—'$clean'<br />
```

This example shows that although comparison show that strings are equal, a browser will display different results—the untainted string is not transformed, whereas '<' and '>' in the tainted one are replaced with '<' and '>'.

Example

```
Example using ^^untaint.<br />
<form>
<input type="text" name="field" />
<input type="submit" />
</form>
```

```
$tainted[$form:field]
Tainted data—'$tainted'<br />
Untainted data—'^untaint{$tainted}'
```

Transformation type for **untaint** is specified inside square brackets. Here it is omitted, which means using the default parameter **as-is**. Note that while **untaint** with unspecified transformation type is equivalent to **untaint** with **as-is** transformation, **taint** has no transformation equivalent to **taint** with unspecified type.

Example

```
Example ^^taint.<br />
$city[New York]
<a href="city.html?city=^taint[uri][$city]">$city</a>
```

As a result, contents of variable city are transformed into URI type. Cyrillic characters, white spaces and other characters which must be encoded, would be replaced with hex entities and represented as %XX.

Example

```
Ouputting and saving user submitted data and generating XML<br />
You specify: '$form:field'

^connect[$SQL.connect-string]{
      ^void:sql{INSERT INTO news SET (body) VALUES ('$form:field')}
}

$doc[^xdoc::create{<?xml version="1.0" encoding="UTF-8"?>
<root>
      <data>$form:field</data>
</root>
}]
```

In this case, you need neither **taint** nor **untaint**, as all the necessary transformations will occur automatically with transformation type **optimized-html** for output to browser, **sql** for sending data to server and **xml** for generating xdoc object.
Note that you also do not need to write **taint/untaint** in SQL queries when saving data to a database using administrative interface.

Example

```
Outputting user submitted data or data coming from a database (may contain
tags) to an edit form<br />
^if(def $form:body){
   $body[$form:body]
}{
   ^connect[$SQL.connect-string]{
      $body[^string:sql{SELECT body FROM news WHERE news_id = $id}]
   }
}
<textarea>$body</textarea>
```

In this example **optimized-html** transformation will be performed automatically, because the data submitted by the user or coming from a database are tainted. If the data contains any tags, they will not affect the page. Remember that sequences of white spaces in **$body** will be optimized during output.

Example

```
Outputting data coming from a database containing administrator written tags<br
/>
^connect[$SQL.connect-string]{
  $body[^string:sql{SELECT body FROM news WHERE news_id = $id}]
```

```
}
^taint[as-is][$body]
```

Here you should use **taint** specifying transformation type **as-is** (or **untaint** specifying this type), for the tags included in the news code by the administrator need not undergo any transformation. This method must not be used for the data submitted by visitors to the website such as guest book information, forum entries, etc.

Example

```
Outputting user submitted data or data coming from a database (may contain
tags) to an edit form keeping spacing symbols<br />
^if(def $form:body){
  $body[$form:body]
}{
  ^connect[$SQL.connect-string]{
     $body[^string:sql{SELECT body FROM news WHERE news_id = $id}]
  }
}
<textarea>^taint[html][$body]</textarea>
```

In this case, use **taint** specifying transformation type **html** (or **untaint** with this type) to avoid crippling the page and to disable optimization of space characters.

In the above examples operator **taint** was used only three times: for displaying administrator added tags in database-derived text, for disabling optimization of spacing symbols, and for outputting query string containing encoded characters (for example, white spaces and Cyrillic letters).
Otherwise, there was no need for **taint/untaint**, and Parser managed everything on its own.

Remember that it is better not to use these operators unless necessary.

You might have noticed that none of the examples used **untaint**. This raises the question of its usefulness. Here are a couple of practical examples.

Firstly, it sometimes helps to reduce the number of the **taint** operators in the code. For example, when outputting data to a multi-field form with spacing optimization disabled. In this case, you can apply `^untaint[html]{…}` to the whole form instead of writing `^taint[html][…]` for each textarea value.

Example

```
Outputting user submitted data or data coming from a database (may contain
tags) to a large edit for keeping spacing symbols<br />
^if(def $form:title){
   $data[$form:fields]
}{
  ^connect[$SQL.connect-string]{
      $data[^table::sql{SELECT title, lead, body FROM news WHERE news_id =
$id}]
   }
}

^untaint[html]{
   <p>
      <b>Heading</b><br />
      <textarea name="title">$data.title</textarea>
   </p>
   <p>
      <b>Announcement:</b><br />
      <textarea name="lead">$data.lead</textarea>
   </p>
```

```
<p>
      <b>News</b><br />
      <textarea name="body">$data.body</textarea>
   </p>
}
```

Secondly, you can use it to output xml to browser (for instance, for ajax, RSS, SOAP, etc.). In this situation **optimized-html** is not appropriate, and you must enclose the code in **^untaint[optimized-xml]{…}** to ensure correct output.

The transformation is replacement of some characters by others, according to built-in transformation tables. The following types of transformation are available:

as-is
file-spec
http-header
mail-header
uri
sql
js
regex ***[3.1.5]***
xml
html

optimized-as-is
optimized-xml
optimized-html

Transformation table

as-is	no transformation
file-spec	characters * ? ' " < > | are replaced with "_XX", where XX is character's hex-code
uri	characters other than numbers or lower/uppercase Latin letters as well as characters _ - . " are replaced with %XX, where XX is a character's hex-code
http-header	the same as URI
mail-header	if charset is known (if not, upper/lowercase will not work), the fragment starting with the eighth-bit first letter and until the end of the string will be represented in such a way: Subject: Re: parser3: =?koi8-r?Q?=D3=C5=CD=C9=CE=C1=D2?=
sql	depending on SQL-server for Oracle, ODBC and SQLite ' is replaced with '' for PgSQL is transformed by means of PgSQL itself for MySQL is transformed by means of MySQL itself for transformation needed that code which made a transformation are located inside **^connect[]{}** operator.
js	" is replaced with \" ' is replaced with \' \ is replaced with \\ newline character is replaced with \n character with code 0xFF is preceded by \
regex	Characters \ ^ $. [ ] | ( ) ? * + { } - are prefixed by \ ***[3.1.5]***
parser-code	Special characters are prefixed by ^
xml	& is replaced with & > is replaced with > < is replaced with < " is replaced with " ' is replaced with '
html	& is replaced with & > is replaced with > < is replaced with < " is replaced with "
optimized-as-is **optimized-xml** **optimized-html**	in addition to replacements, optimizes "white spaces" (space, tab, newline characters). multiple repetition of above-mentioned characters in a row is replaced with a single one—that which goes first in the row

A number of **taint** transformations are made automatically. Thus, names of files and paths are always automatically transformed with **file-spec** and when you write...

```
^file::load[filename]
```

...Parser executes...

```
^file::load[^taint[file-spec][filename]]
```

Similarly, when HTTP-headers and mail headers are defined, Parser executes **http-header** and **mail-header** transformations respectively. During DOM-operations, text parameters of all methods are automatically **xml**-transformed.

Parser also performs a number of automatic **untaint** transformations:

type	*what is transformed*
sql	body of SQL-query
xml	XML-code—while an object of class **xdoc** is created
optimized-html	page output to browser
regex	REGEX-patterns
parser-code	body of operator process

Error handling

To err is human. You should be ready for error messages to pop up unexpectedly from time to time. Unfortunately, this is nearly inevitable. In the beginning, error messages will crop up rather often. At first, the main reason for it will most probably be unbalanced brackets (remember—we mentioned text editors, which support auto brace matching) or mistyping Parsers constructions.

If an error occurs, page processing will stop, all currently active SQL connections will be rolled back, and method **unhandled_exception**, will be called. This method will receive information on the error as well as the stack of calls that caused it. The method's work will result in a custom message to be output to a visitor. The result of the page's code with error will not be output at all. The error will also be recorded in web-server's error log.

Still, it is often desirable to intercept an error and do something useful with it. Let's assume you want to check if XML code from an untrustworthy source is correct. In this case, you do not want processing to stop, quite the contrary, you do expect an error of a certain type and want to handle it. Parser is glad to meet your wishes and gives you a powerful tool: operator **try**.

During a complex data processing, an error may appear in a method which is called from another one, which is, in its turn, is called from a third, and so on... How can we simply report and handle the error in this case? Use operator **throw**, to report the error—and handle the error on the top level. In this case you will not have to check it on all nesting levels of the method calls.

It is also very often that Parser itself or its system classes report errors. See "System errors".

try. Intercepting and handling errors

```
^try{the code whose errors get...}{...into this handler as $exception}
^try{the code whose errors get...}{...into this handler as $exception}{the code
which will be executed anyway}      [3.3.0]
```

If an error occurred during processing the **code**, a variable **$exception** will be created and control over processing will be handed over to **handler**.
If third parameter was specified, that code will be executed anyway regardless of unhandled exception.

$exception is such a **hash**:

$exception.type	string, error type. There is a number of system error types; a type can also be defined in operator **throw**.
$exception.source	string, error source (wrong filename, method's name, ...)
$exception.file **$exception.lineno** **$exception.colno**	file containing **source**, line and column numbers in it
$exception.comment	error comment, in English
$exception.handled	true or false, flag "if error has been handled" you will need to set the flag in the **handler** if you have handled the received error

Handler must report Parser if the error has been handled. For this purpose, it must set the flag but **only** for the needed error types:
$exception.handled(true)

If **handler** has not set the flag, the error is considered unhandled and will be handed over to another handler, if it exists.

If the error remains unhandled, method **unhandled_exception** is called. This method will receive information on the error as well as the stack of calls that caused it. The method's work will result in a custom message to be output to a visitor. The error will also be recorded in server's error log.

Example

```
^try{
	$srcDoc[^xdoc::create{$untrustedXML}]
}{
	^if($exception.type eq xml){
		$exception.handled(true)
		Invalid XML,
		<pre>$exception.comment</pre>
	}
}
```

throw. Reporting an error

```
^throw[type]	[3.3.0]
^throw[type;source]
^throw[type;source;comment]
^throw[hash]
```

Operator **throw** reports error of **type**, which was caused by **source**, and provides **comment**.

This error can be intercepted and handled by using operator **try**.

Do not intercept errors only to provide a good-looking output. Let method **unhandled_exception**, do it all instead, if no handler can be found. Besides, the method will add entries to server's error log, which you can regularly look through to find problems that might crop up.

Example

```
@method[command]
^switch[$command]{
	^case[add]{
		adding...
	}
	^case[delete]{
		deleting...
	}
	^case[DEFAULT]{
```

```
            ^throw[bad.command;$command;Wrong command $command, good are
add&delete]
            ^rem{
                  the next format also acceptable:
                  ^throw[
                        $.type[bad.command]
                        $.source[$command]
                        $.comment[Wrong command $command, good are add&delete]
                  ]
            }
      }
}

@main[]
$action[format c:]
^try{
      ^method[$action]
}{
      ^if($exception.type eq bad.command){
            $exception.handled(true)
            Wrong command '$exception.source', in file $exception.file, in line
$exception.lineno.
      }
}
```

The result of this code's work will be:

```
Wrong command 'format c:', in file c:/parser3tests/www/htdocs/throw.html, in
line 15.
```

We would like to remind you that visitors should not see errors' technical details, especially if such details contain paths to files—it is both ugly and unsafe.
Outputting **`$exception.file`** *is nothing but an example that you can use while debugging the site at server, but by no means in production mode.*

@unhandled_exception. Outputting unhandled errors

If an error has not been handled by any of the handlers (see operator **`try`**), Parser calls method **`unhandled_exception`**, This method receives information on the error as well as the stack of calls that caused it. The method's work results in a custom message to be output to a visitor. The error is also recorded in server's error log.

The **`unhandled_exception`** message would look best if framed within usual layout of your site. It would be also good if you check technical details and **hide** them from your visitors.

We recommend placing this method in your site's configuration file.

There is a way to prevent recording an error into error log. **Only** for particular errors set this flag on:
`$exception.handled(true)` ***[3.1.4]***

Example

```
@unhandled_exception[exception;stack]
$response:content-type[
        $.value[text/html]
        $.charset[$response:charset]
]

<title>UNHANDLED EXCEPTION (root)</title>
<body bgcolor=white>
<font color=black>
<pre>^untaint[html]{$exception.comment}</pre>
^if(def $exception.source){
```

```
        <b>$exception.source</b><br />
        <pre>^untaint[html]{$exception.file^($exception.lineno^)}</pre>
}
^if(def $exception.type){exception.type=$exception.type}
^if($stack){
        <hr />
        ^stack.menu{
                <tt>$stack.name</tt> $stack.file^($stack.lineno^)<br />
        }
}
```

System errors

type	Possible reason	Description
parser.compile	^test[}	Error in code compilation. Unbalanced bracket, etc.
parser.runtime	^if(0).	Method passed wrong number of parameters or parameters are of wrong type
parser.interrupted		Page loading has been interrupted (visitor cancelled page loading or browser download timed out)
number.zerodivision	^eval(1/0), ^eval(1\0) or ^eval(1%0)	Division by zero or Modulus by zero
number.format	^eval(abc*5)	Attempt of converting nonnumeric data into number
file.missing	^file:delete[skdfjs.delme]	Specified file is missing
file.access	^table::load[.]	Access to file is denied
file.read		Problems while reading file
file.execute		Error while executing external program
date.range	^date::create(1950;1;1)	Date out of valid range
pcre.execute	^string.match[((\w)]	Error while compile or execute PCRE pattern
image.format	^image::measure[index.html]	Image file is of wrong format (possibly, extension does not match the content or a file is empty)
sql.connect	^connect[mysql://baduser:pass@host/db]{}	DB server cannot be found or is temporarily unavailable
sql.execute	^void:sql{bad select}	Error in SQL-query
xml	^xdoc::create{<forgot?>}	XML code or operation with it contains error
smtp.connect		SMTP server cannot be found or is temporarily unavailable
smtp.execute		Error in sending message via SMTP protocol
email.format		Error in email address: address is absent or contains unacceptable characters
email.send		Error in executing mail-sending application
http.host	^file::load[http://notfound/there]	Server cannot be found
http.connect	^file::load[http://not_accepting/there]	Server has been found but does not accept the connection
http.response	^file::load[http://ok/there]	Server has been found and connection accepted, but generated incorrect response status
http.status	^file::load[http://ok/there]	Server returned response not equal to 200 (unsuccessful request processing)
http.timeout		Loading a document from HTTP-server was not completed in due time

User-defined operators

Sometimes it will seem to you that Parser lacks some operators. Parser allows you to define your own operators which could be later used along with system operators.

Operators in parser are methods of class MAIN, By adding new methods into this class you extend built-in set

of operators.

Important notice: while defining an operator you may use not only local variables, but also global ones. By doing so, you will assign and refer to fields of class MAIN.

User-defined operators may be defined in separate files without header **@CLASS** and be linked to relevant sections of a site. If you define an operator (e.g. **@include[]**) in such a file, every call **^include[]** will be addressed to the user-defined operator.

CAUTION: If the name of the operator you define is same as a system operator's, user-defined operator will be called. Using of system operator will then be impossible. We advise you to use as few user-defined operators as possible. Consider using static methods of user-defined classes instead.

Creating classes and using their methods is far more comfortable than employing user-defined operators for the same purpose. For example: there are several sections of the site and each one needs a help section. By creating several files defining different classes, we can get methods of different classes bearing the same name. While calling these methods as static ones, we can clearly see the relation between methods and sections:

```
^news:help[]
^forum:help[]
^search:help[]
```

Examples

Place the code...

```
@default[a;b]
^if(def $a){$a}{$b}
```

...into file `operators.p` in root directory of your website.

After you have done it, you can link this module whenever you need additional operators. For example, write such a construction in your root `auto.p`:

```
@USE
/operators.p
```

...and you will be able to use construction of type **^default[$form:name;Anonymous]** not only on any page, but also in any user-defined class.

Details can be found in section Defining methods and user operators.

Charsets

We are sure: existence of various charsets gives you as much pleasure as it does to us.

Parser has a built-in capability of transcoding documents from charset used on server into that used by visitor and back. Parser transcodes:

- form data;
- strings (before transformation of type uri);
- text resulting from page processing.

You specify charset used in documents on server in field **$request:charset**.
You specify charset to be used in output in field **$response:charset**.
You should do it in one of **auto** methods.

We recommend you to specify result charset in HTTP-header **content-type**, so that a browser knew about it and a visitor did not have to select charsets manually.

```
$response:content-type[
        $.value[text/html]
        $.charset[$response:charset]
```

]

Charsets to be used in email messages can be specified as different from that of the output, see **^mail:send[…]**.

While working with databases, you should specify connection settings in such a way that SQL query and response data were in charset given in **$request:charset**, see Format of connect string.

A list of allowable charsets is defined in Configuration file.
Default charset for all documents is **UTF-8**.

*Note: when transcoding **from** UTF-8 if some character is not specified in transcode table, a sequence* **&#DDDD;** *is inserted instead.* **DDDD** *is decimal Unicode of that character.*

*Note: when transcoding **to** UTF-8 if some character is not specified in transcode table, a sequence* **%HH** *is inserted instead.* **HH** *is hexadecimal code of that character.* ***[3.1.4]***

Note: charset's name is case insensitive.

Class MAIN. Processing request

Parser processes requested document in the following way:

1.
It reads, compiles, and initializes:
a) Configuration file;
b) all files named `auto.p`, which are searched for in root directory and down—through directories tree until the directory where requested document belongs;
c) requested document itself.
Taken all together, they are what is defined as class **MAIN**.
Initialization is done by calling method **auto** in each of the loaded files. If method's definition contains a parameter, the loaded file's name will be passed.
Note: result of method's work will not be output to a visitor.

2.
Then, method **main** of class **MAIN** is called without parameters.
This means that each of the mentioned files can define method **main**. The one which was defined last will be called. This method's definition will override all other possible definitions.
The result of this method's work will be output to the visitor unless method **postprocess** is defined.
If file has not a single method defined, its whole content will be regarded as definition of method **main**.
Note: specifying **$response:body[*of non-standard response*]** *redefines text received by a visitor.*

3.
If class **MAIN** has method **postprocess** defined, result of method **main**'s work is passed to it as the only parameter and it is the result of **postprocess** that a visitor will get.
Thus, you get an opportunity of "extra polishing" the result of your code's work.

Simple example

If we add this definition into file `auto.p` located in your root directory...

```
@postprocess[body]
^if($body is string){
    ^body.match[Jack][g]{Jill}
}{
    $body
}
```

...it will result in replacing **Jack** with **Jill** in every page.
Do not forget to check the type, there can be some file.

Bool class

Objects of classes **bool** are logical values **true** and **false.**

Console class

This class is designed for creating simple interactive services, which work in text line-by-line mode.

These services can work with help of standard UNIX `inetd` program.

For example, it is possible to implement news-server (NNTP) in Parser.
Add a line like this to your `/etc/inetd.conf` file and restart `inetd`:
```
nntp stream tcp nowait unix_user /path/to/parser3 /path/to/parser3
/path/to/nntp.p
```

In `nntp.p` script code your NNTP server.
This would give people an ability to use it—`nntp://your_server`.

Static field

Reading a line

$console:line

This construction reads a line from console.

Writing a line

$console:line[text]

This construction writes a line to console.

Cookie class

The class is designed for working with HTTP **cookies**.

Static fields

Accessing

$cookie:name_of_cookie

Returns value of cookie with specified name.
Example:
$cookie:my_cookie

Retrieves and outputs value of cookie named **my_cookie**.

*Note: cookies' values are accessible for reading **immediately** after they have been assigned.*

Storing

$cookie:name[value]
Saves cookie with specified **name** and specified **value** to be stored for 90 days.

Example

```
$cookie:user[Peter]
```

...will create cookie named **user** and assign value **Peter** to it. The cookie thus created will be stored on user's disk for 90 days.

Note: cookies' values are accessible for reading immediately after they have been assigned.

```
$cookie:name[
   $.value[value]
   $.expires(number of days)
]
$cookie:name[
   $.value[value]
   $.expires[session]
]
$cookie:name[
   $.value[value]
   $.domain[domain name]
]
$cookie:name[
   $.value[value]
   $.path[subsection]
]
$cookie:name[
      $.value[value]
      $.httponly(true)     [3.2.2]
      $.secure(true)       [3.2.2]
]
```

...assigns data to cookie with specified **name**.

Optional modifiers:
$.expires(number of days)—specifies how many days a cookie may be accessible (number of days may be fractional, i.e. 1.5 will mean "one day and half").
$.expires[session]—creates session cookie (cookie will be deleted when visitor closes all browser windows);
$.domain[doman name]—specifies domain from which the cookie may be accessed;
$.path[subsection]—specifies subsection of the site from which the cookie may be accessed.

During storing **cookie** you can add any bool option. In this case the http header will contains this option without value. You can use it for set httponly option for example.

Example

```
$cookie:login_name[
   $.value[guest]
   $.expires(14)
]
```

...will create a cookie named **login_name** with value **guest** and store it a fortnight.

fields. All cookies

```
$cookie:fields
```

Such a construction returns hash with all cookies.

Example

```
^cookie:fields.foreach[name;value]{
   $name - ^if($value is "hash"){$value.value}{$value}
}[<br />]
```

...will output all cookies' names and their values.

Date class

Class **date** is designed for working with dates. Possible variants of using it include: calendars, various checks based on dates, etc.

Values may range from 01/01/1970 to 01/01/2038.

Do not forget that we have different gaps and overlaps: many countries have so-called Daylight Saving Time, when clock is set ahead (in spring) or back (in autumn) one hour.
For example, in Moscow, there cannot be "02:00, 31 March 2002," while "02:00, 27 October 2002" can be twice.

Numeric value of object of class **date** equals to the number of days from EPOCH (00:00:00, 1 January 1970, UTC) to the date specified in the object. This feature is useful when you want to get a relative date, e.g.:

```
# checking if the file was updated more than a week ago
^if($last_update > $now-7){
   new
}{
   old
}
```

The number of days can be fractional, e.g. a day and half is equal to **1.5**.

The class usually operates local date and time. Still, you can get date and time in arbitrary time zone (see **^date.roll[TZ;...]**.

To communicate between computers that are in different time zones it is convinient to exchange values of date/time which do not depend on timezone—UNIX format, which is number of seconds passed since EPOCH, is very convinient here.

Unix format can be used in JavaScript and several other scripting languages that work in browser.

Parser fully supports work with UNIX date format.

Constructors

create. Relative date

```
^date::create(number of days since EPOCH)
```

Constructor with only one parameter is designed for specifying **relative** date values. Having object of class **date**, one can make up a new object of the same type, whose value will be shifted with respect to the initial.

Example

```
$now[^date:now[]]
$date_after_week[^date::create($now+7)]
```

The example creates a date to come a week after the current.

Parameter of the constructor does not have to be an integer number.

```
$date_after_three_hours[^date::create($now+3/24)]
```

create. Arbitrary date

```
^date::create(year;month)
^date::create(year;month;day)
^date::create(year;month;day;hour;minute;second)
```

The constructor creates an object of class **date** containing value of an arbitrary date accurate to a second. **Year** and **month** are obligatory parameters, while **day**, **hour**, **minute**, and **second** are optional parameters. If these are not specified, the **day** value will be set to 1, while **hours**, **minutes**, and **seconds**—to 0.

Example

```
$year_2000_start[^date::create(2001;12;31;23;55)]
```

As a result, the code will create an object of class date, whose fields' values will contain time for year 2000 to begin.

create. Date and time in standard DBMS format

```
^date::create[year]
^date::create[year-month]
^date::create[year-month-day]
^date::create[year-month-day hour]
^date::create[year-month-day hour:minute]
^date::create[year-month-day hour:minute:second]
^date::create[year-month-day hour:minute:second.millisecond]
^date::create[hour:minute]
^date::create[hour:minute:second]
```

Creates an object of class **date**, containing value of an arbitrary date and/or time accurate to a second. Obligatory parameters are **year** or **hour** and **minute**, while **month**, **day**, **hour**, **minute**, **second** and **millisecond** are optional. If these are not specified, **day** value will be assigned 1 or current day's value, while **hour**, **minute**, and **second** will be assigned 0.
Note: **millisecond** *value is ignored.*

This feature is useful if you retrieve a date from DB, since the query will return you values of fields with date or time, or both date and time as strings.

Example

```
# articles created/updated 3 days ago and later are "new"
$new_after[^date::now(-3)]
$articles[^table::sql{select id, title, last_update from articles where …}]
^articles.menu{
    $last_update[^date::create[$articles.last_update]]
    <a href=${articles.id}.html>$articles.title</a>
    ^if($last_update > $new_after){new}
    <br />
}
```

Note for Oracle users: to get date and time in convenient format, specify the format of date and time in server connection string, as recommended in Appendix 3.

create. Copying existing date

```
^date::create[date object]
```

The constructor copy an existing object of class **date**.

Example

```
$now[^date::now[]]
$dt[^date::create[$now]]
^dt.roll[month](-1)
```

The example creates a date to come a month before the current.

now. Current date

```
^date::now[]
^date::now(shift in days)
```

Constructor creates object of class **date**, containing value of the current date accurate to a second, using server's system time. If **shift in days** is specified, the date will be shifted a specified number of days.

The constructor uses local time of the server where Parser works. To find the time in another time zone, use **^date.roll[TZ;...]**.

Example

```
$date_now[^date::now[]]
$date_now.month
```

As a result, the code will create an object of class **date** containing current date's value and output the number of current month.

unix-timestamp. Date and time in UNIX format

```
^date::unix-timestamp(date_time_in_UNIX_format)
```

Constructor creates object of class **date**, containing value, corresponding to passed numerical value in UNIX format (see also brief description).

Fields

By referring to the fields of objects of class **date**, you can retrieve the following values:

$date.month month
$date.year year
$date.day day
$date.hour hours
$date.minute minutes
$date.second seconds
$date.weekday weekday, i.e. number of day in a week (0 = Sunday, 1 = Monday, etc.)
$date.week week number in year (according to ISO 8601 standard) ***[3.1.5]***
$date.weekyear year for this week (according to ISO 8601 standard) ***[3.2.2]***
$date.yearday year day (0 = January 1, 1 = January 2, etc.)
$date.daylightsaving 1 - Daylight Saving Time, 0 - standard time
$date.TZ time zone; contains the value, if the date was **^date.roll[TZ;...]**

Example

```
$date_now[^date::now[]]
$date_now.year<br />
$date_now.month<br />
```

```
$date_now.day<br />
$date_now.hour<br />
$date_now.minute<br />
$date_now.second<br />
$date_now.weekday
```

As a result, an object of class **date** will be created, containing current date, and the value of:

```
year
month
day
hour
minute
second
weekday
```

...will be output.

Methods

roll. Shifting date

```
^date.roll[year](shift)
^date.roll[month](shift)
^date.roll[day](shift)
^date.roll[TZ][new time zone]
```

This method increases/decreases values of fields **year**, **month**, and **day** of objects of class **date**.

You can also get date/time stored in an object of class **date** in another time zone by specifying system name of a new time zone. For the list of these names, please see your system documentation (search for: "Environment variable TZ").

Example of shifting a month

```
$today[^date::now[]]
^today.roll[month](-1)
$today.month
```

In this example, we assign variable **$today** the value of current day and then decrease the number of the current month by one. As a result, we get the value of the previous month.

Example of shifting time zone

```
@main[]
$now[^date::now[]]
^show[]
^show[Moscow;MSK-3MSD]
^show[Amsterdam;MET-1DST]
^show[London;GMT0BST]
^show[New York;EST5EDT]
^show[Chicago;CST6CDT]
^show[Denver;MST7MDT]
^show[Los Angeles;PST8PDT]

@show[town;TZ]
^if(def $town){
    $town
    ^now.roll[TZ;$TZ]
}{
    Server local time
}
```

```
<br />
$now.year/$now.month/$now.day, $now.hour hrs $now.minute mins<hr />
```

sql-string. Getting date in DBMS-style format

```
^date.sql-string[]
```

Method transforms the date into **YYYY-MM-DD HH:MM:SS** format, used by DBMS for storing dates. Using this method you can add date values to DB without any additional transformations.

Example

```
$date_now[^date::now[]]
^connect[connect string]{
	^void:sql{insert into access_log
		(access_date)
	values
		('^date_now.sql-string[]')
	}
}
```

We get string of format '**2001-11-30 13:09:56**' with current date and time and at once place it into a DB field. Without this method at hand, we would have to put together the needed strings manually. Note: the method doesn't form the apostrophes—you should add them by yourself.

unix-timestamp. Converting date and time to UNIX format

```
^date.unix-timestamp[]
```

Converts date and time to value in UNIX format (see also brief description).

last-day. Getting last day of month

```
^date.last-day[]
```

The method return last day of month.

Example

```
$date[^date::create(2008;02;01)]
^date.last-day[]
```

Will return **29**

gmt-string. Converting date to string in RFC 822 format

```
^date.gmt-string[]
```

This method convert date to string in RFC 822 format (**Fri, 23 Mar 2001 09:32:23 GMT**).
Usually you don't need to do anything and Parser convert date to such string automatically (for example when you set HTTP-response header: **$response:expires[^date::now(+1)]**). But somethime (when you generate RSS feed for example) this method can be usable.

Static methods

calendar. Creating calendar for specified week

^date:calendar[rus|eng;year;month;day]

The method makes up a table with calendar for a week of specified **month** of the **year**. Parameter **day** is used to specify the week. Parameter **rus|eng** is used to specify calendar's format. In format **rus**, the week starts with Monday, whereas in **eng**—with Sunday.

Example

```
$week_of_month[^date:calendar[eng](2001;11;30)]
```

As a result, variable **$week_of_month** will be assigned a table with calendar for the week containing 30 October 2001. The table's format will be:

year	*month*	*day*	*weekday*
2001	11	25	00
2001	11	26	01
2001	11	27	02
2001	11	28	03
2001	11	29	04
2001	12	30	05
2001	12	01	06

calendar. Creating calendar for specified month

^date:calendar[rus|eng](year;month)

The method makes up a table with calendar for specified **month** of the **year**. Parameter **rus|eng** is used to specify calendar's format. In format **rus**, the week starts with Monday, whereas in **eng**—with Sunday.

Example

```
$calendar_month[^date:calendar[eng](2005;1)]
```

As a result, variable **$calendar_month** will be assigned a table with calendar for January 2005:

0	*1*	*2*	*3*	*4*	*5*	*6*	*week*	*year*
						01	53	2004
02	03	04	05	06	07	08	01	2005
09	10	11	12	13	14	15	02	2005
16	17	18	19	20	21	22	03	2005
23	24	25	26	27	28	29	04	2005
30	31						05	2005

Method's work results in a new object of class **table** with columns 0...6 plus columns **week** and **year** containing, respectively, number of week according to standard ISO 8601 and year it belongs to.

last-day. Getting last day of month

```
^date:last-day(year;month)
```

The method return last day of month.

Example

```
^date:last-day(2008;2)
```

Will return **29**

Double, Int classes

Objects of classes **double** and **int** are real and integer numbers. These may result from calculations or transformations, or be specified by user. Numbers falling within the range of class **double** are those with the floating point. The scope of values depends on platform, yet, as a rule, the scopes are

for **double**	from **1.7E-308**	to **1.7E+308**
for **int**	from **-2147483648**	to **2147483647**

Class **double** usually has 15 significant digits and doesn't guarantee preservation of numbers in the last orders. Precise number of significant digits depends on the platform you use.

Methods

int, double, bool. Transforming objects into numbers or bool

^name.int[]	or	**^name.int(default)**
^name.double[]	or	**^name.double(default)**
^name.bool[]	or	**^name.bool(default)**

The method transforms value of variable **$name** into either integer or real number or bool respectively and returns it. If real number is transformed into integer, it will be truncated.

One may also specify default value, which will be returned if transformation is impossible. Default value may be used when you process data received from visitors interactively. It will prevent text values from appearing in mathematical expressions, when a user inputs, for example, a string instead of a number initially expected.

Important notice: empty string or a string containing "white spaces" only (space characters, tab characters, or new line characters) is regarded as zero.

Example

Code...

```
$str[Item]
^str.int(1024)
```

...will output number **1024**, as object **str** cannot be transformed into object of class **int**.

Code...

```
$double(1.5)
^double.int[]
```

...will output number **1**, as the number was truncated.

Code...

```
^if(^form:search_in_text.bool(false){
      ...searching in text...
}
```

inc, dec, mul, div, mod. Simple operations on numbers

^name.inc[] – increases variable's value by 1 or **number**
^name.inc(number)
^name.dec[] – decreases variable's value by 1 or **number**
^name.dec(number)
^name.mul(number) – multiplies variable's value by **number**
^name.div(number) – divides variable's value by **number**
^name.mod(number) – puts into variable the modulus of its value division by **number**

Example

Code...

```
$var(5)
^var.inc(7)
^var.dec(3)
^var.div(4)
^var.mul(2)
$var
```

...will return **4.5** and is equal to construction: **$var((5+7-3)/4*2)**.

format. Outputting number in specified format

^name.format[format string]

The method outputs variable's value in specified format (see Format Strings).

When you output number without **format**, simply:
$name
Parser for numbers with zero fraction part does this:
^name.format[%.0f] ***[3.1.5]***
for others that:
^имя.format[%g]

Examples

Code...

```
$var(15.67678678)
^var.format[%.2f]
```

...will return: **15.68**

Code...

```
$var(0x123)
^var.format[0x%04X]
```

...will return: **0x0123**

Static methods

sql. Retrieving number from database

```
^int:sql{query}
^int:sql{query}[$.limit(1) $.offset(o) $.default(expression)]
^double:sql{query}
^double:sql{query}[$.limit(1) $.offset(o) $.default(expression)]
```

The method returns number resulted from SQL-query to a database server. The query must return value of single column of single row.

query – query to a DB, written in SQL language;
$.offset(o) – ignore first **o** query records;
if SQL-server response was empty (0 records), ...
$.default(expression) ...the given **expression** will be evaluated returned;
$.default{code} ...the given **code** will be executed and string result returned.

This method demands connection with database server (see operator **connect**).

Example
Code...

```
^connect[connect string]{
      ^int:sql{select count(*) from news}
}
```

...will return number of records in table **news**.

Env class

The class is designed for retrieving values of environment variables. The list of standard environment variables is available at http://www.w3c.org/cgi. Apache web server assigns a number of additional variables.

Static fields

```
$env:environment_variable
```

Construction returns the value of specified environment variable.

Example
```
$env:REMOTE_ADDR
```

Will return IP-address of computer which has requested the document.

Retrieving values of HTTP-header fields

```
$env:HTTP_HEADER_FIELD
```

Such a construction will return the value of HTTP-header field, sent by browser to web-server (by HTTP protocol).

Example
```
^if(^env:HTTP_USER_AGENT.pos[MSIE] >= 0){
    User is probably using MicroSoft Internet Explorer<br />
}
```

Names of HTTP-header fields are all uppercase and begin with **HTTP_**. All hyphens ('-') in these names are

substituted by underscores ('_'). For additional information, please read your web-server documentation.

Retrieving Parser version

```
$env:PARSER_VERSION
```

Such a construction will return the full Parser version and platform.

Something like...
```
3.2.0 (compiled on i386-pc-win32)
```

File class

Class **file** is designed for working with files. Objects of this class can be created by different means:

1. by means of method POST through form field
   <form method="post" enctype="multipart/form-data">...
   <input name="photo" type="file">.
2. by one of constructors of class **file**.

While sending files to client (for example, by method **mail:send** or through field **response:body**) one should define HTTP-header `content-type`. Parser determines file type by its extension with the help of table **MIME-TYPES**, defined in Configuration method (see also Chapter Installing and configuring Parser). Parser uses it to automatically determine, by file extension, the type of content to be sent in header `content-type`. If type of content cannot be determined, `content-type` will be `application/octet-stream`.

Constructors

load. Loading file from disk or HTTP-server

```
^file::load[format;filename]
^file::load[format;filename;download options]
^file::load[format;filename;new filename]
^file::load[format;filename;new filename;download options]
```

Loads file from disk or HTTP-server.

Format defines format of loaded file and can be either **text** or **binary**. The two types differ in newline characters. For PC these characters are **0D 0A**. If file is being loaded in format **text**, **0D** will be deemed unnecessary and truncated. These characters will be added back to the file by method **save**.

Filename—name of file with path or file's URL on HTTP-server.

It should be kept in mind that if argument **new filename** is specified, its value will be assigned to field **name**. This argument is especially useful in dealing with method **mail:send** to send a file with needed name.

Download options—see "Working with HTTP-servers".

If a file was loaded from an HTTP-server, fields of HTTP-response headers can be accessed as fields of object of class **file**.
Also there would be field **tables**, hash, keys of which are HTTP-response headers in upper case, and values are tables with sole column **value**, containing all values of HTTP-response fields of same name.

Example of downloading file from disk

```
$f[^file::load[binary;article.txt]]
File $f.name is $f.size in size and has the text:<br />
$f.text
```

Example of downloading file from HTTP-server

```
$file[^file::load[text;http://www.parser.ru/;
      $.timeout(5)
]]
Server software: $file.SERVER
<hr />
<pre>$file.text</pre>
```

sql. Loading file from SQL-server

```
^file::sql{query}
^file::sql{query}[$.name[name] $.content-type[content-type] $.limit(1)
$.offset(o)]
```

Loads file from SQL-server. Result of query execution must be one record (use limit option in needed). Parser considers its

- first column contains file body;
- second column contains file name;
- third column contains `content-type` of the file (if not specified, it will be determined by **$MIME-TYPES** table).

Optional parameters:
$.limit(1) - limit response to one row only; ***[3.3.0]***
$.offset(o) —ignore first **o** retrieved entries; ***[3.3.0]***
File **name** and **content-type** may be also specified as a parameters. ***[3.1.4]***

File name and its `content-type` will be passed to visitor if **$response:download** is used.

Note: for now only MySQL server is supported.

stat. Retrieving information about a file

```
^file::stat[filename]
```
Object created by this constructor has additional fields (objects of class **date**).

$some_file.size—size of file in bytes;
$some_file.cdate—creation date;
$some_file.mdate—modification date;
$some_file.adate—last access date.

filename—path and name of file.

Example

```
$f[^file::stat[some.zip]]
Size in bytes: $f.size<br />
Created in: $f.cdate.year<br />
$new_after[^date::now(-3)]
Status: ^if($f.mdate >= $new_after){new;old}
```

cgi and exec. Executing a program

```
^file::cgi[filename]
^file::cgi[filename;env_hash]
^file::cgi[filename;env_hash;argument1;argument2;...]
^file::cgi[format;filename;env_hash;argument1;argument2;...]  [3.2.2]
^file::exec[filename]
^file::exec[filename;env_hash]
^file::exec[filename;env_hash;argument1;argument2;...]
^file::exec[format;filename;env_hash;argument1;argument2;...] [3.2.2]
```

Constructor **cgi** creates an object of class **file**, containing results returned by a program according to CGI standard .

Note: when a program/script is executed, directory with it will be its working directory.

Headers, returned by the CGI-script, will then become fields of class **file** converted into UPPERCASE. E.g., if a CGI-script `script.pl` returns a header **some_field:some_value**, then, on having processed
$f[^file::cgi[script.pl]]
we can address to **$f.SOME_FIELD** and get value **some_value**.

Constructor **exec** is similar to **cgi** but doesn't separate HTTP-headers from the text returned by the script.

format—defines format of loaded file and can be either **text** (default) or **binary**. While using **binary** format the result will not transcoded to **$request:charset** and will not truncated on first zero char.

filename—path and name of file.

Object, created by these constructors, has additional fields:
status—information on the status of program's termination (usually **0** (zero) means successful termination, while non-zero status means error);
stderr—standard errors stream.

Example
```
$cgi_file[^file::cgi[new.cgi]]
$cgi_file.text
```

Outputs text resulting from execution of `new.cgi`.

Optional arguments of constructors:

env_hash—hash, which can include
- additional environment variables to be later accessed from within the script,
- key **stdin**, containing text sent to the script in standard input stream,
- key **charset**, which indicates charset in which script operates (data to and from script will be transcoded accordingly). ***[3.1.3]***

*Note: you can specify only standard CGI environment variables, or variables, whose names start with **CGI_** or **HTTP_** (restricted to: UPPERCASE Latin characters, numbers, underscore and hyphen).*
Note: while processing `HTTP POST` *request, you can use construction* **$.stdin[$request:body]** *to send received POST data to the script's standard input stream.*
Note: variables which were set by http server while executes Parser also will be availabe for scripts.

Example of how to use an external cgi-script
```
$search[^file::cgi[search.cgi;$.QUERY_STRING[text=$form:q&page=$form:p]]]
```

Example of how to use an external script
```
$script[^file::exec[script.pl;$.CGI_INFORMATION[I have had it enough]]]
```

Information being sent can be accessed and used within script `script.pl`:
```
print "Additional information: $ENV{CGI_INFORMATION}\n";
```

Example of receiving binary data from an external script
```
$response:body[^file::exec[binary;getfile.pl;$.CGI_FILENAME[$form:filename]]]
```

Example of passing several arguments

You can also send a number of arguments to the program by specifying them—separated by semicolon—after **env_hash**:

```
$script[^file::exec[script.pl;;height;width]]
```

...or specify arguments as a table with one column: ***[3.2.2]***

```
$args[^table::create{arg
height
width}]
$script[^file::exec[script.pl;;$args]]
```

Note: we insist that you store scripts to be run by constructors **cgi** *and* **exec** *beyond web-space, since executing a script with arbitrary arguments may case unexpected consequences.*

base64. Decoding from Base64

^file::base64[encoded]

Decodes a file from Base64 representation. To encode a file use
^file.base64[]

Detailed information on Base64 is available here http://www.ietf.org/rfc/rfc2045.txt and here http://en.wikipedia.org/wiki/Base64.

Example

$encoded[

R0lGODdhyAAyANUAAP////j88fLz8O/v7+v21uns4uTyyd/f397vu9vf1NfsrtDpoM/Pz8rmk8Tj
hr+/v73geK+vr6nWUJ+fn52jkJzQNZXNJ4+Pj39/f3BwcGBgYFBQUEBAQDAwMCAgIBAQEAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACwAAAAyAAyAAAG/0CAcEgsChkRjHIZ
ORif0Kh0Sq1ar9isdkudaD6gsHj80US46LR6zW5zBxjweE73YAbuvH7P71/kdIFzHxN9hoeIiUQH
HIKOgRx4ipOUlVgPgHQcGUsYGY2CHwyWpKWlE4IbZ1ADE6CDo6ays3wRkE5VD69iorS+v2gMmSAf
q1h/g7jAy8ysHnMdylnC0M3W1wAZ0JJvz2MY2OG+DIPcwcPS4uqUuyAPbbZjGuv0kwdzGXkbc+n1
/nnaeJkzQqCBwQYIDASQcm/MhSmdIkrE8HDKgAcYM2rE2M8Ig40gNw68GLJkxwEXJqoE96SkSwDe
wrCEQqCChZs4JSyMomFMh/8pjwLNizKgQ1BisaCgOjrm3ZCiTMXMfGo0apgnPa2CIDemo5CaOMNa
0BmFqxivQrSGGepxmKNeT5ZqdQqAQcyoUwFAVWskq9YLPqOAFRuWLJS7haKoXdtWK1wicucecXt0
6l6+RPxq1ZzvyWDChXf2/SZlcZjOk00bAxBZ8gHKlYlUXSzbNIhdq4d8xglBAWHDRQCL4SCFg/Hj
yJPfJT4Ew5zkyTNN3eUBunXjox48vw49cVp5KyUSgcbdelOCNsVCEOJbLPAh2oezcS6/+ZzipIfs
ykslvhguc9CFRYBXEEjEbjetN0R7oRXh323zjcGcEPT9F8V+RGB4yX1bGJj/hYdUZCIgghYoSASD
OYkGwIMTplEhhPZ9s9Jd0+V3xYMghIdBbkOQVx5y451nxS50kagAFCjeBByLEdZHoW01SpXFa5s9
YRsIQYoh4BS4fZVeWEdGkeRYOwkXRotovNjii1pFKZMW1FjFVo+2ZRnGllJoBo6RVIxZAQEA6Nnk
mUSwaZWbOW4RZ1RzAnClne5cYSYIPy1AWJh9EuYATGN456KEhUKZoY1ZXMZURd+ZBimeZc1RAAAo
YloFiuslwM+gMD4po0o0jirlGy5hNFuidIpxQbAbrYpFJhSwd5OsVrSnYEBheNCGmqGOgd+vQmjI
h7eOCvmhuFVo9oEA7EF7/wUBm+qVCWpqYBujhVCAC64e4IJYILlUmPWmG9SGgRaFOi6xy5oc1kvq
vf3+iJx0kHbg8HHKYtFOUmrg2KiVpsHL5rb/dktqLqY9Fmidxd6ZBY4eDLTFAOhQYVqjH1+48Mj9
LWayEJpZVTEWPXfgMhamggCvYmptXLPC3ALA8BQ4HrXzED0z9fMVMG/DxQHDUjq0EUk/sfQT9uIM
tWMYGxGw1SlHCicdH7A6RQTDTA3FHBqEh2oRByRblkbSfJTRwE+QhOzg/R6uEREicdHaWoQPccA+
dPCItJb/ZJ7F42ulXUQEVYfhqcz8am56FBPA9sEGGEyA0QQYbKC65aWVflL67UUw0LVpHXhOOua4
B1+4oVZ9wJ8V+gqvPAAHhP7IBxl8XUXyyyuPUjuBbDCB9FY0Xv33Dqa0gXF5Hwv++einr/767Lfv
/vvwxy///PTXj0YQADs=

```
]
$original[^file::base64[$encoded]]
$filespec[/parser3logo.gif]
^original.save[binary;$filespec]
<img src="$filespec" />
```

Outputs...

create. Text file creation

```
^file::create[format;name;content]
^file::create[format;name;content;options]        [3.4.0]
```

Constructs an object of class **file**, with specified **name** and **content**.

Format defines format of created file. For now it can be only **text**.
Options—hash, in which you can specify **$.charset[**charset**]** of creating text file

*Note: if there is a need to save file to **server** disk, there is simple way:* **^string.save[**…**]**.

Example of export data in XML format

```
#export.html
^connect[connection string]{
$products[^table::sql{select product_id, name from products}]
$file[^file::create[text;export.xml;^untaint[xml]{<?xml version="1.0"
encoding="$request:charset"?>
<products>
    ^products.menu{<product id="$products.product_id" name="$products.name"/>}
</products>
}]]
$response:download[$file]
}
```

When this document is opened then file `export.xml` is created and browser suggests visitor to save it. Sample result:

```
<?xml version="1.0" encoding="UTF-8"?>
<products>
      <product id="1" name="Can be &amp;quot;Quoted&amp;quot;"/><product id="2"
name="Johnson&amp;amp;Johnson"/>
</products>
```

Fields

name. Name of file

```
$some_file.name
```

The field contains the name of file. Object of class **file** has field **name** if a visitor has uploaded the file through form field **<input type=file>**. Constructor **file::load** may also provide an alternative name of file.

size. Size of file

```
$some_file.size
```

The field contains size of file in bytes.

text. Text of file

```
$some_file.text
```

The field contains text of file. By using this field, one can output the content of text files or text resulted from **file::cgi** and **file::exec**.

Information about file

$some_file.size—size of file in bytes;
$some_file.cdate—creation date;
$some_file.mdate—modification date;
$some_file.adate—last access date.

These fields available if object was created within constructor **file::stat** or **file::load** by loading local file *[3.3.0]*.

stderr. Standard error text of program execution

$some_file.stderr

After **file::cgi** and **file::exec** here goes text from standard error program stream.

status. Status of getting this file

$some_file.status

After **file::cgi** and **file::exec** in **status** field one can find status of program execution (success=0). After **file::load** from HTTP-server here is status of HTTP request (success=200).

content-type. MIME-type of file

$some_file.content-type

The field may contain file's MIME-type. If a cgi-script is executed (see **file::cgi**) MIME-type may be specified by the script—in header "**content-type.**" If a file is loaded (see **file::load**) or its status is retrieved (see **file::stat**) MIME-type will be defined with the help of table **$MAIN:MIME-TYPES** (see "Configuration method"), If file extension cannot be located in the table, MIME-type will be defined as "**application/octet-stream**."

HTTP response headers

$some_file.HTTP_RESPONSE_HEADER

If a file was loaded from an HTTP-server, HTTP response headers will be accessible in UPPERCASE as fields of object of class **file**.

$some_file.HTTP_RESPONSE_FIELD (in UPPERCASE)

For example: **$some_file.SERVER**.

If one response header occurs in a response several times, all its values are accessible in **tables** field:

```
$.tables[
   $.HTTP_RESPONSE_FIELD[table of values with sole column value]
]
```

Example:

```
$f[^file::load[binary;http://www.parser.ru/en/]]
^f.tables.foreach[key;value]{
    $key=^value.menu{$value.value}[|]<br />
}
```

Methods

save. Saving file to disk

^some_file.save[format;filename]
^some_file.save[format;filename;options] *[3.4.0]*

Method saves object to file in specified format and with specified name.

format—format of saving file (**text** or **binary**)
filename—name of file and path for the file to be saved
options—hash, in which you can specify **$.charset[**charset**]** of saving text file

Note: if file with specified filename is exist, it will be overwrited.
Note: to append some text to a file, use ^string.save[append;...].

Example

```
^archive.save[text;/arch/archive.txt]
```

This code will save object of class **file** as `archive.txt` in text format to directory `/arch/`.

sql-string. Saving file to SQL-server

```
^file.sql-string[]
```

Returns the string, which can be used in SQL-query. Allows saving file in database.

Attention: currenty only MySQL-server is supported.

Example

```
$name[image.gif]
$file[^file::load[$name]]
^connect[connect string]{
      ^void:sql{insert into images (name, bytes) values ('$name', '^file.sql-
string[]')}
}
```

base64. Encoding to Base64

```
^file.base64[]
```

Method encodes file to Base64 representation.
To decode a file from Base64 to it's original, use
^file::base64[encoded]

Detailed information on Base64 is available here http://www.ietf.org/rfc/rfc2045.txt and here http://en.wikipedia.org/wiki/Base64.

Example

```
$original[^file::load[binary;http://www.parser.ru/i/artlebedev.gif]]
<pre>^original.base64[]</pre>
```

Outputs...

```
R0lGODlhWgAlAMQAAP///4CAgOX0yb/jeKXXQtnurvn88uz318Xmhszok/L55KzaUJ/VNbnga9Lr
od/xvM/qmeXzx+PzxaXXQbLdXdHrndzvtbzhccvokaPWPMHje+Hyv8PlgcTlgpnSKAAAACH5BAEA
AB4ALAAAAABaACUAAAX/ICCOZGmeaKqubOu+cCzPdG3feK7jXi/6HoDvNOwFhcEh8ohsLn9Hpc4I
FTpNxeyVWs1GvzsoOMksebtMbvPsl0a4wKv5O04rgfSq/K0lo9h4I2qBT21lfGN7gnlrdnaGYIo3
cEmSkGiLj4CRO5RWbotimKKjnpY1nn5EjISFjaRaYamWm1t3qqagk320rKquhLmHPLy6l6+kyMJh
hXHGtbeavnqdxcN6qY6htac0s8/T0dvT3TOpGz0CAlh0BkgH0e7vXgYKsTkFRh4ISAP6oD789WBg
4F8CAAI9EMynD6ERAhCKENihQICBBXgWIOjQQ8GqBz0cYEyQDqOHCiEJ83jA0IMCSA8MHaiM6OGA
PX46DjRo2OMAtTkk9e0MooCBh4Q9hnrsEdRIgp0IENTcoZJBUFsj5EEhOUCdHqP+uq5j6JPpUQE+
AeQjQGCBrKNSf/Uo8ICDBw2L1goo0DWdPwJ6+VrooXGuBwJ7uxYMGSbu0Cv5rE7wkCFU3MMCEhLw
eRkxQ5MAOq8T6DGMAq0nSke4IOGEAbRZYcdOC+B1ide0menezbu379/Amf0bTry48ePIkytfzry5
8+fQPQQIYGR6deo9rEffzr04durfpWcXT767efPhxacvf749dOzkwY9n777+8unatUv/jt++/+Qh
AAA7
```

md5. MD5 hash of file

```
^file.md5[]
```

The method gets 16-byte hash of file and outputs it as a string—bytes are output in hexadecimal code without delimiters, lowercase.

It is believed that:

- it is practically impossible for two strings to have the same MD5-hash;
- it is practically impossible to restore original string from its MD5-hash.

Detailed information on MD5 is available at http://www.ietf.org/rfc/rfc1321.txt

crc32. File checksum calculation

```
^file.crc32[]
```

The method gets CRC32 checksum for the file and outputs it as an integer.

Static methods

delete. Deleting file from disk

```
^file:delete[path]
```

Deletes specified file.
path—path and name of file

If the directory is found empty after a file is deleted, the directory will also be deleted (if possible).

Example

```
^file:delete[story.txt]
```

find. Finding file on disk

```
^file:find[file]
^file:find[file]{code to be executed if file is not found}
```

The method returns a string (object of class **string**) containing name of file and full path if it exists in specified directory or in any of the parent directories. Otherwise, if a code is specified, it will be executed.

Example without path

```
<img src="^file:find[header.gif]{/i/header.gif}">
```

Assume, the code is located in `/news/sport/index.html`. Then, the file **header.gif** will be searched within **/news/sport/** intended solely for sports news. In case it cannot be found, and **/news/sports/header.gif** doesn't exist, a standard headline image for news section will be used.

Example with path specified

```
<img src="^file:find[/i/$section/$subsection/header.gif]">
```

File **header.gif** will be searched within **/i/section/subsection/**. If it still cannot be found, it will be looked for in (in the order as follows):

- **/i/section/**
- **/i/**
- **/**

list. Getting directory listing

```
^file:list[path]
^file:list[path;filter]
```

Makes up a table (object of class **table**) containing one column—**name**—containing files and directories—within specified directory—matching pattern, if specified.

filter—a regular expression (see also method **match** of class **string**) used to specify a pattern for names of file to match. It could be specified as a string or **regex-object** ***[3.4.0]***. If **filter** is not specified, all files located in specified directory will be listed.

Example

```
$list[^file:list[/;\.zip^$]]
^list.menu{
      $list.name<br />
}
```

Will output the names of all archives with extension **.zip**, located in web-server's root directory.

copy. Copying file

```
^file:copy[source filename;filename for new file]
```

The method copy file.

Note: you should be very careful with everything that involves writing within web-space, as this feature (writing something to somewhere) is now widely used by malicious users.

Example

```
^file:copy[/path/source.txt;/path/destination.txt]
```

Will copy **source.txt file**.

move. Moving or renaming a file

```
^file:move[old_filename;new_filename]
```

The method renames/moves file or directory (under Win32, objects cannot be moved to another disk). New directories are created with file permissions `755`. The directory of the old file is deleted if it is found empty after the file is deleted.

Note: you should be very careful with everything that involves writing within web-space, as this feature (writing something to somewhere) is now widely used by malicious users.

Example

```
^file:move[/path/file1;/file1]
```

Will move file1 into root directory.

lock. Exclusive use of code

```
^file:lock[file_to_be_locked]{code}
```

Code is not simultaneously executed by multiple visitors. **File_to_be_locked** is used to ensure exclusive use.

Example

```
^file:lock[/counter.lock]{
    $file[^file::load[text;/counter.txt]]
    $string[^eval($file.text+1)]
    ^string.save[/counter.txt]
}
Number of visitors: $string<br />
```

If locking is not used, two simultaneous requests can increase the counter's value... by 1, not by 2:

- first visitor comes;

- second visitor comes;
- first visitor reads counter's value—value equals 0;
- second visitor reads counter's value—value equals 0;
- first visitor increases counter's value—value now equals 1;
- second visitor increases counter's value—value now equals 1;
- first visitor writes new value—1;
- second visitor writes new value **immediately after the first visitor**, the value is **1, not 2**.

Note: you should always keep in mind simultaneous requests. If you work with databases, SQL-servers usually have built-in means that provide correct processing for simultaneously incoming requests.

Note: when there are more then one lock, always analize there mutual relations to avoid "A waits B, B waits A", so called deadlock situation.

dirname. Path to file

```
^file:dirname[filespec]
```

Retrieves path from path and filename to file/directory (**filespec**).

Example

```
#filename
^file:dirname[/a/some.tar.gz]
#directory's name…
^file:dirname[/a/b/]
```

In both cases the result will be:
```
/a
```

basename. Name of file without path

```
^file:basename[filespec]
```

Retrieves name of file with extension but without path from full path (**filespec**).

Example

```
^file:basename[/a/some.tar.gz]
```

...will return...

```
some.tar.gz
```

justname. Name of file without extension

```
^file:justname[filespec]
```

Retrieves name of file without path and extension from full path (**filespec**).

Example

```
^file:basename[/a/some.tar.gz]
```

...will return...

```
some.tar.gz
```

justext. File's extension

^file:justext[filespec]

Retrieves extension without dot, from full path (**filespec**).

Example

```
^file:justext[/a/some.tar.gz]
```

...will return...

gz

fullpath. Full name of file from server's root directory

^file:fullpath[filename]

Retrieves full name of file from server's root directory (see also "**Appendix 1: Paths to files and directories**").

Example: page `/document.html` contains a link to some image. True path to the requested document, however, may be different (e.g. if you use module `mod_rewrite` on Apache web-server). In this case, if you place a relative link to the image, the image will not be displayed by browser, since browser has no idea about `mod_rewrite` and will regard all relative paths as relative to the requested document.

That is why it is better to replace relative path with absolute:

```
$image[^image::measure[^file:fullpath[image.gif]]]
^image.html[]
```

Construction...

```
<img src="/image.gif" width="..." height="..." />
```

...will result in code containing absolute path.

base64. Encoding to Base64

^file:base64[filename]

Method encodes file with specified **filename** to Base64 representation.
To decode a file from Base64 to it's original, use
^file::base64[encoded]

md5. MD5 hash of file

^file:md5[filename]

The method gets 16-byte hash of file with specified **filename** and outputs it as a string—bytes are output in hexadecimal code without delimiters, lowercase.

It is believed that:

- it is practically impossible for two strings to have the same MD5-hash;
- it is practically impossible to restore original string from its MD5-hash.

Detailed information on MD5 is available at http://www.ietf.org/rfc/rfc1321.txt

crc32. File checksum calculation

```
^file:crc32[filename]
```

The method gets CRC32 checksum for file with specified **filename** and outputs it as an integer.

Form class

Class form is designed for working with form fields. The class has static fields available for reading only.

It is useful to check form fields for being empty and edit available database records with such an approach:

```
^if($edit){
# record from database
    $record[^table::sql{… where id=…}]
}{
# new record, error (some fields are empty) output
# form fields
    $record[$form:fields]
}
<input name="age" value="$record.age" />
```

Static fields

Getting form field value

```
$form:field_name
```

Such a construction returns value of form field. Returned object may belong either to class **file**, if field type is **file**, or class **string**.Further actions with object can be performed only by methods prescribed for relevant classes.

Field bearing no name is referred to as **nameless**.
Coordinates sent by browser when a visitor clicks image with attribute **ISMAP** can be accessed through **$form:imap**.

You should remember that if <input type="image" name="fieldname" /> is used in html and visitor click on this image, the browser will send coordinates of this action in **fieldname.x** и **fieldname.y** fields.

Example: text field, image field and file uploading

```
^if(def $form:photo){
      ^form:photo.save[binary;/upload/photos/beauty.^file:justext[$form:photo.n
ame]]
      Image $form:photo.name was uploaded.
}
^if(def $form:user){
      User: $form:user<br />
}
^if(def $form:[action.x]){
      Coordinates:<br />
      X: $form:[action.x]<br />
      Y: $form:[action.y]<br />
}
<form method="post" enctype="multipart/form-data">
<input type="file" name="photo">
<input type="text" name="user">
<input type="image" name="action" src="/i/button.gif" width="75" height="25" />
</form>
```

...will store picture uploaded to server by a visitor through form field in specified file.

Example: nameless field

```
<img src="/show.html?123&a=b" />
```

Within `show.html` string **123** can be accessed as **$form:nameless**.

imap. Getting mouse click coordinates

$form:imap

If a visitor clicked on an image with attribute ISMAP, such a construction returns **hash** with fields **x** and **y** containing mouse click coordinates.

Example

In file `/go.html` you write:

```
$clicked[$form:imap]
^if(def $clicked){
    Visitor clicked on ISMAP link:<br />
    x=$clicked.x<br />
    y=$clicked.y<br />
}
```

In file `/test.html` you write:

```
<a href="/go.html?a=b"><img src="map.png" ismap /></a>
```

Open `/test.html` in your browser and click on the picture. You will go to...
`/go.html?a=b?10,30`

...and you will see...

```
    Visitor clicked on ISMAP link:
    x=10
    y=30
```

qtail. Getting query string remainder

$form:qtail

Returns the part of **$request:query** after the second question sign (**?**).

Example

Assume, requested page is...
`http://www.mysite.ru/news/article.html?year=2000&month=05&day=27?thisText`

Then,

```
$form:qtail
```

...will return...

```
thisText
```

fields. All form fields

```
$form:fields
```

Such a construction returns hash with all form fields. Hash keys' names are the same as the names of form fields. Hash keys' values are the form fields' values.

Example

```
^form:fields.foreach[field;value]{
   $field - $value
}[<br />]
```

...will output all form fields' names and their values.
Assume, requested URI is...
www.mysite.com/testing/index.html?name=worst&grade=F

Then the example will output:

```
name - worst
grade - F
```

tables. Getting multiple field values

```
$form:tables.field_name
```

If a form field has at least one value, such a construction returns a table (object of class **table**) with single column **field**, containing all field values. It is used for getting multiple field values.

Important notice: before performing operations with a table, you should first check if it exists.

Example

```
Choose what you like doing in your free time:
<form method="POST">
   <p><input type=checkbox name=hobby value="wsurf">windsurfing</p>
   <p><input type=checkbox name=hobby value="tv">watching TV</p>
   <p><input type=checkbox name=hobby value="books">reading books</p>
   <p><input type=submit value="OK"></p>
</form>
$hobby[$form:tables.hobby]
^if($hobby){
    Your hobbies are:<br />
    ^hobby.menu{
        $hobby.field
    }[<br />]
}{
    None selected}
```

The example will output either selected variants or a message informing that nothing has been selected.

files. Getting multiple files

```
$form:files
```

Such a construction return hash with all form files. Hash keys' names are the same as the names of form fields. See below.

```
$form:files.field_name
```

If a form field has at least one file-value, such a construction returns a hash (object of class **hash**) with keys 0, 1, 2..., containing all files with specified field name. It is used for getting multiple files with the same field

name.

Important notice: before performing operations with a hash, you should first check if it exists.

Example

```
^if(def $form:picture){
      <p>Loaded pics (^form:files.picture._count[]):
      ^form:files.picture.foreach[sNum;fValue]{
            $fValue.name
            ^fValue.save[binary;/upload/pictures/${sNum}.^file:justext[$fValue.
name]]
      }[,]
      </p>
}
<form method="post" enctype="multipart/form-data">
   <p>Choose some pictures for uploading:<br />
      <input type="file" name="picture" /><br />
      <input type="file" name="picture" /><br />
      <input type="file" name="picture" /><br />
      <input type="submit" value="Upload" />
   </p>
</form>
```

Note: uploaded images names will be shown in random order.

Hash class

The class is designed for working with hashes, or associative arrays. A hash is considered defined (**def**), if it isn't empty. Numerical value of a hash is the number of its keys (value returned by method `^hash_name._count[]`).

Constructors

Usually, we don't use constructors to create a hash. Instead, we do it the way described in "Parser's Constructions".

create. Creating an empty hash or copying existing hash

```
^hash::create[]
^hash::create[existing hash or hashfile]
```

If **existing hash** or **hashfile** is not specified, an empty hash will be created. Otherwise, the constructor will make a copy of it.

An empty hash is needed when we are going to dynamically fill it with data, e.g.:

```
$dyn[^hash::create[]]
^for[i](1;10){
   $dyn.$i[$value]
}
```

Before performing loop **for**, we have defined what exactly we are going to fill with data.

If we are planning to change a hash's content intensively, but still want to preserve initial values, we had better create a copy of the hash. In this case, only the hash's copy will be changed, while initial values will remain intact. For example:

```
$pets[
```

```
$.pet[Dog]
   $.food[Bone]
   $.good[Collar]
]
$pets_copy[^hash::create[$pets]]
```

Note: field **_default** *is also copied.* ***[3.1.4]***

sql. Getting SQL-query result as a hash

```
^hash::sql{query}
^hash::sql{query}[$.limit(n) $.offset(o) $.distinct(true/false)
$.bind[variables hash] $.type[hash/string/table]]
```

This constructor creates hash, in which keys' names are the values of fields in the first column of SQL-query's result. Other columns' names become nested keys' names, and their values become respective keys' values. When the result contains only one column, constructor creates the hash, where values of the column become keys of hash associated with logical value **truth** .

Optional parameters:

$.limit(n)	get only **n** records.
$.offset(o)	skip first **o** records of the query result.
$.bind[hash] ***[3.1.4]***	variables to bind, see «Queries with bound variables»
$.distinct(true/false)	**false** or **0**=consider duplicate an error (default); **true** or **1**=get records with unique keys.
$.type[hash/string/table] ***[3.3.0]***	**hash**=each hash item contain hash (default); **string**=each hash item contain string. You must specify **exactly two columns** in your SQL query; **table**=each hash item containing table.

By default, duplicate of a value in key column is considered an error, but if you want the method to get the records with unique keys, set flag **$.distinct(true)** .
Note: such use results in spare data interchange between client and server. You had better change the query so that the desired uniqueness should be the server's responsibility. If you need data as both table and hash, consider using ***table::sql*** *and* ***table.hash*** *together.*

Example: hash of hash

With database containing `hash_table`...

pet	*food*	*aggressive*
cat	milk	very
dog	bone	never

...the code...

```
^connect[connect string]{
      $hash_of_hash[^hash::sql{
            select
                  pet,
                  food,
                  aggressive
            from
                  hash_table
      }]
}
```

...will result in hash of the following structure:

```
$hash_of_hash[
      $.cat[
            $.food[milk]
            $.aggressive[very]
```

```
	]
	$.dog[
		$.food[bone]
		$.aggressive[never]
	]
]
```

...from which we can effectively retrieve information, e.g. in such a way:

```
$animal[cat]
$animal likes eating $multi_level_hash.$animal.food
```

Example: hash of bool

With database containing `participants` table...

```
name
Konstantin
Alexander
```

...the code...

```
^connect[connect string]{
	$participants[^hash::sql{select name from participants}]
}
```

...will result in hash of the following structure:

```
$participants[
	$.Konstantin(true)
	$.Alexander(true)
]
```

...from which we can effectively retrieve information, e.g. in such a way:

```
$name[Ivan]
$name ^if($participants.$name){participates}{do not participate} in the project
```

Fields

Fields of hash are the keys, the value of which we get by referring to it:

```
$hash.key
```

Such a construction will return value associated with the key. If non-existing key is referred to, the value of key **`_default`** will be returned, if specified.

Assigning something to hash key actually adds or updates a pair key/value in the hash:

```
$my_hash.key[value]
```

For better interchangeability of hashes and tables, field **`fields`** contains reference to hash itself, see "Using hash instead of table".

Using hash instead of table

```
$hash.fields
```

Hash itself.

For better interchangeability of hashes and tables, field **`fields`** contains reference to hash itself, see **`table.fields`**.

Methods

_keys. List of hash keys

```
^hash._keys[]
^hash._keys[column name] [3.2.2]
```

The method returns table (object of class **table**), containing single column with all hash keys listed (since version ***3.4.0*** the keys in the table are listed in order of putting the elements into the hash, before–the order is not defined).
The name of column–"**key**" or the **column name** passed as a parameter.

Example

```
$man[
      $.name[Jack]
      $.age[22]
      $.sex[m]
]
$tab_keys[^man._keys[]]
^tab_keys.save[keys.txt]
```

...will create file **keys.txt** with such a table:

key
sex
age
name

_count. Number of hash keys

```
^hash.count[]
```

The method returns number of hash keys.

Example

Code...

```
$man[
   $.name[Jack]
   $.age[22]
   $.sex[m]
]
^man._count[]
```

...will return: **3**

When used in mathematical expressions, numerical value of hash is equal to its keys' number:

```
^if($man > 2){greater}
```

foreach. Going through hash keys

```
^hash.foreach[key;value]{body}
^hash.foreach[key;value]{body}[delimiter]
^hash.foreach[key;value]{body}{delimiter}
```

The method works the same way as the method **menu** of class **table**. It goes through all hash keys and relevant values (since version ***3.4.0*** the method goes through elements in order of putting the elements into the hash, before–order is not defined).

key–name of variable to return keys' names

value—name of variable to return keys' values
body—code to be executed for each key-value
delimiter—code to be executed before each non-empty non-first body

You can force finish the loop using **break** operator or finish current step and go to next one using **continue** operator. ***[3.2.2]***

Example

Code...

```
$man[
    $.name[Jack]
    $.age[22]
    $.sex[m]
]
^man.foreach[key;value]{
        $key=$value
}[<br />]
```

...will return...

```
name=Jack
age=22
sex=m
```

delete. Deleting key/value pair

```
^hash.delete[key]
```

The method deletes specified **key**/**value** pair from **hash**.

Example

Code...

```
^man.delete[name]
```

...will delete key **name** and related value from hash **$man**.

contains. Check for existance key in hash

```
^hash.contains[key]
```

The method checks if hash contains specified key. It returns Boolean **TRUE** if hash contain record with specified key, or **FALSE** if they don't.

Example

Code...

```
^if(^man.contains[birthday]){
      Birthday specified for visitor.
}
```

Working with sets

sub. Subtracting hashes

```
^hash.sub[hash_to_be_subtracted]
```

The method subtracts **hash_to_be_subtracted** from hash, deleting keys common for both hashes.

Example

```
$man[
    $.name[Jack]
    $.age[22]
    $.sex[m]
]
$woman[
    $.name[Mary]
    $.age[20]
]
^man.sub[$woman]
```

As a result, hash **$man** will remain with single key **$man.sex** containing value **m**.

add. Adding hashes

```
^hash.add[hash_to_be_added]
```

Adds **hash_to_be_added** to **$hash**. Keys bearing the same name are overwritten by those in **hash_to_be_added**.

Example

```
$man[
    $.name[Jack]
    $.age(22)
    $.sex[m]
]
$woman[
    $.name[Mary]
    $.age(20)
    $.smile[yes]
]
^man.add[$woman]
```

New content of hash **$man** will be:

```
$man[
    $.name[Mary]
    $.age(20)
    $.sex[m]
    $.smile[yes]
]
```

Note: field **_default** *is also added, if it existed, it is overwritten with new value.* ***[3.1.4]***

union. Joining hashes

```
^hash_a.union[hash_b]
```

The method joins two hashes. It returns hash containing all keys from **$hash_a** and those keys from **$hash_b**, which are absent in **$hash_a**. The result has to be assigned to a new hash.

Example

Code...

```
$man[
    $.name[Jack]
    $.age[22]
    $.sex[m]
]
$woman[
    $.name[Mary]
    $.age[20]
    $.weight[50]
]
$union_hash[^man.union[$woman]]
```

...will result in hash **$union_hash**:

```
$union_hash[
    $.name[Jack]
    $.age[22]
    $.sex[m]
    $.weight[50]
]
```

intersection. Intersecting hashes

```
^hash_a.intersection[hash_b]
```

The method intersects two hashes. It results in a hash containing keys that belong to both **$hash_a** and **$hash_b**. The result has to be assigned to a new hash.

Example

Code...

```
$man[
    $.name[Jack]
    $.age[22]
    $.sex[m]
]
$woman[
    $.name[Mary]
    $.age[20]
    $.weight[50]
]
$int_hash[^man.intersection[$woman]]
```

...will return hash **$int_hash**:

```
$int_hash[
    $.name[Jack]
    $.age[22]
]
```

intersects. Checking if hashes intersect

```
^hash_a.intersects[hash_b]
```

The method checks if hashes intersect (i.e. have common keys). It returns Boolean **TRUE** if hashes intersect, or **FALSE** if they don't.

Example

```
^if(^man.intersects[$woman]){
   Intersection found
}{
   Intesection not found
}
```

Hashfile class

The class is designed for working with hashes kept on disk. Unlike **hash** class, objects of this class are considered to be always defined (**def**) and have no numeric value.

While **hash** class keeps its values in memory, **hashfile** keeps them on disk and it is possible to separately specify time to keep each key-value pair.
Note: currently to keep one **hashfile** *two files are used: .dir and .pag.*
Note: there is a limit on key and value strings, together they must not exceed 8000 bytes.

Reading and writing of data performed very quickly—Parser works only with necessary data files fragments. On simple tasks **hashfile** performs considerably faster then databases.
Note: file can be changed only by one script at a time, others are waiting for it to complete processing of request.

Example

Say, it's desirable to get some information from visitor on one page of site and to be able to show it on other page. And it is necessary to prevent visitor from seeing or faking it in the middle.

It is possible to store information to **hashfile**, associated with some random string—session identifier. That identifier can be stored to **cookie**, data are now kept on server, are not reachable and cannot be faked by visitor.

```
# opening/creating file with information
$sessions[^hashfile::open[/sessions]]
^if(!def $cookie:sid){
    $cookie:sid[^math:uuid[]]
}
# after that…

$information_string[arbitrary value]
# …storing arbitrary $information_string under sid key for 2 days
$sid[$cookie:sid]
$sessions.$sid[$.value[$information_string] $.expires(2)]

# …like this can read the value stored earlier
# if since the moment we stored it passed less then 2 days
$sid[$cookie:sid]
$information_string[$sessions.$sid]
```

Constructor

open. Opening or creating

```
^hashfile::open[file name]
```

Opens existing disk file or creates a new one.
Currently to keep data two files are used, with extensions .dir and .pag.

Note: file can be changed only by one script at a time, others are waiting for it to complete processing of request. Before changing script waits all others scripts to stop reading.

Note: it is not allowed to open same file twice.

Reading

$hashfile.key

Returns the string, associated with a **key**, provided that association is not expired yet.

Writing

$hashfile.key[string]

Stores to disk the association between **key** and **string**.

```
$hashfile.key[
      $.value[string]
      $.expires(number of days)
]
$hashfile.key[
      $.value[string]
      $.expires[date]
]
```

Such a construction allows to specify the date for association to expire. There can be specified **number of days** or some specific **date**.

Optional modifiers:
$.expires(number of days) – specifies number of days (can have fractional part, 1.5=day and a half), during which to keep **key**/**string** pair, 0 days=forever;
$.expires[$date] – specifies date and time until which the association will be kept, here **$date**—the variable of **date** type.

Note: there is a limit on key and value strings, together they must not exceed 8000 bytes.

Methods

hash. Converting to usual hash

^hashfile.hash[]

Converts the **hashfile** to usual hash.

foreach. Going through hash keys

```
^hash.foreach[key;value]{body}
^hash.foreach[key;value]{body}[delimiter]
^hash.foreach[key;value]{body}{delimiter}
```

The method goes through all keys and relevant values of **hashfile** (order is not defined). Method is analogous to **foreach** of **hash** class.

You can force finish the loop using **break** operator or finish current step and go to next one using **continue** operator. ***[3.2.2]***

delete. Deleting key/value pair

^hashfile.delete[key]

The method deletes the **key/value** pair from file.

Note: nothing deleted from files. Pair wirh specified key just marked as deleted so following writing to hashfile can use freed space.

delete. Deleting files from disk

^hashfile.delete[]

The method deletes from disk files, in which data of hash file are stored.

cleanup. Delete expired pairs

^hashfile.cleanup[]

The method goes through all pairs and delete expired.

Note: nothing deleted from files. Expired pairs just marked as deleted so following writing to hashfile can use freed space.

release. Save data on disk and unlock files

^hashfile.release[]

Save all changes to disk and remove all locks.
After this operation hashfile will be available for concurrent processes. Any access to hashfile will authomatically reopen file.

Image class

The class is designed for dealing with images. There may be two types of objects of class **image**. First type includes objects based on existing images in supported formats, whereas the second—objects created by Parser itself.

One can also retrieve EXIF information from JPEG files (http://www.exif.org).

For color presentation, Parser uses RGB system, where each shade of color consists of three components (R-Red, G-Green, B-Blue). Each component has value starting with 0x00 and ending with 0xFF (0-255 in decimal system). Final color represents an integer number of format 0xRRGGBB, where each component is allocated two digits in the given sequence. The formula, according to which the color is calculated, is:

(R*0x100+G)*0x100+B

Thus, white color, which has maximum values (FF) for all components, is made up by the formula:

(0xFF*0x100+0xFF)*0x100+0xFF = 0xFFFFFF

Constructors

measure. Creating an object based on existing graphics file

^image::measure[file]
^image::measure[file_name]

Creates an object of class **image**, measuring dimensions of an existing graphics file or an object of class **file** in supported format (Parser presently supports GIF, JPEG and PNG formats).

Constructor **measure** also reads EXIF information stored in JPEG files (http://www.exif.org), if it is present.

While creating JPEG file, most of today cameras also add information about the image, exposure parameters and other information in EXIF format.

The picture itself is not used—the constructor only keeps in memory the dimensions and the name of the file. Basic function of the constructor is to later call method **html** for the created object.

Parameters:
file—object of class **file**
filename—filename with path

Note: supports EXIF 1.0 and reads tags IFD0 and SubIFD, if any.

Example of creating tag IMG with width and height attributes

```
$photo[^image::measure[myphoto.png]]
^photo.html[]
```

will create object **photo** of class **image**, based on existing graphics in PNG format. Tag `IMG` will be created with reference to the file and `width` and `height` specified.

Example of working with EXIF information

```
$image[^image::measure[jpg/DSC00003.JPG]]
$exif[$image.exif]
^if($exif){
    Camera manufacturer, model: $exif.Make $exif.Model<br />
    Shooting time: ^exif.DateTimeOriginal.sql-string[]<br />
    Exposure time: $exif.ExposureTime seconds<br />
    Aperture: F$exif.FNumber<br />
    Flash used: ^if(def $exif.Flash){^if($exif.Flash){yes;no};not known}<br />
}{
    No EXIF information<br />
}
```

create. Creating an object with specified dimensions

```
^image::create(dimension X; dimension Y)
^image::create(dimension X; dimension Y; background color)
```

Creates an object of class **image** with dimensions X (width) and Y (height). As an optional parameter, you can specify background color. If this parameter is omitted, created image will have white color for background.

Example

```
$square[^image::create(100;100;0x000000)]
```

An object **square** of class **image** with dimensions 100x100 and black background will be created.

load. Creating an object based on graphics file in GIF format

```
^image::load[file_name.gif]
```

Creates an object of class **image** based on ready background. This allows using existing GIF images as a mat on which other graphic elements can be drawn—it can be used to draw graphs, graphic counters, etc.

Example

```
$background[^image::load[counter_background.gif]]
```

An object of class **image** will be created, based on existing image in GIF format. This object can later be used as a mat for drawing.

Fields

$image.src	—filename
$image.width	—width
$image.height	—height
$image.exif	—hash with EXIF information

Keys of **$image.exif** are names of EXIF-tags, see specification (http://www.exif.org/specifications.html). Values may be of type **string**, **int**, **double**, **date**. When a tag has several values, they are turned into hash, with numbers as keys (0...number_of_values-1).

Frequently used EXIF-tags are (for detailed description see specification):

Tag	*Type*	*Description*
Make	**string**	Camera manufacturer
Model	**string**	Camera model
DateTimeOriginal	**date**	Shooting date and time
ExposureTime	**double**	Exposure time (in seconds)
FNumber	**double**	Aperture number F
Flash	**int**	0=was not used, other values=was used

Note: Keys of non-standard EXIF-tags are their values in decimal numbers.

Example

```
$photo[^image::measure[photo.jpg]]
Filename: $photo.src<br />
Image width in pixels: $photo.width<br />
Image height in pixels: $photo.height<br />
$date_time_original[$photo.exif.DateTimeOriginal]
^if(def $date_time_original){
    Picture taken on ^date_time_original.sql-string[]<br />
}
```

As a result, filename, as well as width and height of the image stored in this file will be output. If picture was taken with a digital camera, shooting date and time will most likely be output.

Methods

html. Displaying an image

```
^image.html[]
^image.html[hash]

<img src="$image.src" width="$image.width" height="$image.height" />
```

As a parameter, a hash may be specified in the method, containing optional image attributes, such as **alt** and **border**, determining pop-up text—which appears when cursor is placed over the image—and border width.

Note: image attributes can be re-defined.

Example

```
$myphoto[^image::measure[myphoto.jpg]]
^photo.html[
      $.border[0]
      $.alt[That's me at school…]
]
```

The browser will display an image stored in variable **$myphoto**. Each time, cursor is placed over the image, a pop-up text **That's me at school**... will appear.

gif. Encoding objects of class image in GIF format

```
^image.gif[]
^image.gif[file name]
```

Used to encode objects of class image created by Parser in GIF format.
File name will be passed to visitor if **$response:download** is used.

Important notice: as a result, this method creates an object of class **file**, *not image*!

Besides, it is important to keep in mind that colors are taken from the palette, and when there are no colors left in the palette, nearest shades will be picked up. If you create a complex graphics, especially with background loaded in advance, you must keep in mind the sequence of colors captured.

Example

```
$square[^image::create(100;100;0x000000)]
$response:body[^square.gif[]]
```

Browser will display a black 100x100 pixels square.

Drawing methods

These methods can be applied only to the objects of class **image**, created by constructors **create** and **load**. You can use these methods to draw lines and various geometrical figures on images, fill areas of graphics with various colors. The methods provide an opportunity to create dynamically changed images used as graphs, graphic counters, etc.

Coordinates are calculated starting with left upper corner—the point referred to as (0:0).

Line style and width

```
^image.line-style[line style]
^image.line-width(line width)
```

Before calling any drawing method, you can specify line style and width to be used.
Line style is specified with a string where spaces imply absence of dots in the line, while all other characters imply dots.

Example

```
$image.line-style[***   ]
$image.line-width(2)
```

Drawing methods will use 2-pixel-wide line of type:

```
***   ***   ***   ***   ***
```

line. Drawing a line on an image

```
^image.line(x0;y0;x1;y1;color)
```

The method draws on the image a line of specified color from (x0:y0) to (x1:y1).

Example

```
$square[^image::create(100;100;0x000000)]
^square.line(0;0;100;100;0xFFFFFF)
$response:body[^square.gif[]]
```

Browser will display a black 100x100 square diagonally crossed by a white line.

pixel. Work with image pixels

```
^image.pixel(x;y)
```

Returns the color of the image pixel specified. If coordinates are out of image bounds, returns -1.

```
^image.pixel(x;y;color)
```

Sets **color** of the image pixel specified.

fill. Filling one-color areas of an image

```
^image.fill(x;y;color)
```

The method is used to fill single-color areas of an image with a new color. The area to fill is defined by a dot with coordinates (x;y), which is located within it.

Example

```
$square[^image::create(100;100;0x000000)]
^square.line(0;0;100;100;0xFFFFFF)
^square.fill(10;0;0xFFFF00)
$response:body[^square.gif[]]
```

Browser will display a black 100x100 square diagonally crossed by a white line. The lower section will be black and the upper—yellow.

rectangle. Drawing rectangles

```
^image.rectangle(x0;y0;x1;y1;line color)
```

The method draws on an image an unfilled rectangle with specified coordinates and specified line color.

Example

```
$square[^image::create(100;100;0x000000)]
^square.rectangle(5;40;95;60;0xFFFFFF)
$response:body[^square.gif[]]
```

Browser will display a black 100x100 square with an unfilled 90x20 rectangle with white outline according to the given coordinates.

bar. Drawing filled rectangles

```
^image.bar(x0;y0;x1;y1;rectangle color)
```

The method draws on an image a rectangle with specified coordinates and filled with specified color.

Example

```
$square[^image::create(100;100;0x000000)]
^square.bar(5;40;95;60;0xFFFFFF)
$response:body[^square.gif[]]
```

Browser will display a black 100x100 square with a white 90x20 rectangle, drawn according to the given coordinates.

polyline. Drawing broken lines through joints coordinates

```
^image.polyline(color)[table with junctions coordinates]
```

The method draws a line according to joints coordinates specified in the table. It is used to create broken lines.

Example

```
$table_with_coordinates[^table::create{x  y
10    0
10    100
20    100
20    50
50    50
50    40
20    40
20    10
60    10
65    15
65    0
10    0
}]
$square[^image::create(100;100;0xFFFFFF)]

$square.line-style[***   ]
$square.line-width(2)

^square.polyline(0xFF00FF)[$table_with_coordinates]

$file_withgif[^square.gif[]]
^file_withgif.save[binary;letter_F.gif]

$letter_F[^image::load[letter_F.gif]]
^letter_F.html[]
```

Browser will display letter F drawn by a dotted line against white background. In current directory, a file `letter.gif` will be created. This example uses objects of class **image** of two different types. The table specifies coordinates of broken line. Then, against the background created by constructor **create** a line is drawn through specified coordinates. Created object of class **image** is encoded into GIF format. Resulted object of class **file** is saved to disk. Afterwards, a new object of class **image**, based on saved file, is created. Method **html** will output this object to browser window.

polygon. Drawing polygons through joints coordinates

```
^image.polygon(line color)[table with junctions coordinates]
```

The method draws an unfilled polygon with the coordinates given in table outlined by specified color. The last joint is automatically connected to the first one with a line.

Example

```
$rectangle_coordinates[^table::create{x   y
0     0
50    100
```

```
100    0
}]

$square[^image::create(100;100;0x000000)]
^square.polygon(0x00FF00)[$rectangle_coordinates]
$response:body[^square.gif[]]
```

Browser will display isosceles triangle outlined with green against black background. The table gives coordinates of triangle's vertices.

polybar. Drawing filled polygons through joints coordinates

```
^image.polybar(polygon's color)[table with joints coordinates]
```

The method draws a polygon of specified color through joints coordinates given in the table. The last joint is automatically connected to the first one.

Example

```
$rectangle_coordinates[^table::create{x    y
0      0
50     100
100    0
}]

$square[^image::create(100;100;0x000000)]
^square.polybar(0x00FF00)[$rectangle_coordinates]
$response:body[^square.gif[]]
```

Browser will display a green isosceles triangle against black background. The table gives coordinates of triangle's vertices.

replace. Replacing color in the area specified by coordinates table

```
^image.replace(old color;new color)[coordinates table]
```

The method is used to replace one color with another within the area restricted by coordinates table.

Example

```
$paint_nodes[^table::create{x y
10     20
90     20
90     80
10     80
}]

$square[^image::create(100;100;0x000000)]
^square.line(0;0;100;100;0xFFFFFF)
^square.line(100;0;0;100;0xFFFFFF)

^square.replace(0x000000;0xFF00FF)[$paint_nodes]
$response:body[^square.gif[]]
```

Browser will display a black square, crossed diagonally with white lines, with a pink rectangle within it. Since method replace tells to replace with pink only black color, the lines will remain white.

circle. Drawing an unfilled circle

```
^image.circle(center x;center y;radius;line color)
```

The method draws an unfilled circle of a specified radius, outlined with given color, relative to the center with coordinates X and Y.

Example

```
$square[^image::create(100;100;0x000000)]
^square.circle(50;50;10;0xFFFFFF)
$response:body[^square.gif[]]
```

Browser will display a black square with a circle of 10 pixels radius drawn by a line with (50;50) as a center.

arc. Drawing an arc

```
^image.arc(center x;center y;width;height;start in degrees;end in
degrees;color)
```

The method draws an arc with specified parameters. The arc represents a part of an ellipse (as a special case of circle) and is defined by center coordinates (X;Y) and width and height as well as initial and final angles given in degrees.

Example

```
$square[^image::create(100;100;0x000000)]
^square.arc(50;50;40;40;0;90;0xFFFFFF)
$response:body[^square.gif[]]
```

Browser will display a black square with an arc equal to a quarter (0-90 degrees) of a circle with a 40-pixel radius.

sector. Drawing a sector

```
^image.sector(center x;center y;width;height;start in degrees;end in
degrees;color)
```

The method draws a sector with specified parameters, outlined with given color. Parameters of the method are the same as in method **arc**.

Example

```
$square[^image::create(100;100;0x000000)]
^square.sector(50;50;40;40;0;90;0xFFFFFF)
$response:body[^square.gif[]]
```

Browser will display a black square with a sector equal to a quarter (0-90 degrees) of a circle with 40-pixel radius. The sector is outlined with white.

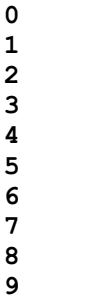
font. Loading font file to make an inscription on an image

```
^image.font[set_of_characters;font_file_name.gif](space_character_width)
^image.font[set_of_characters;font_file_name.gif](space_character_width;charact
er_width)
```

Besides drawing, Parser provides for a possibility of making inscription on an image. To realize this opportunity, it is necessary to have special files with font images. You can either use existing font files or create those of your own, with a needed set of characters.

Having loaded such a file with the help of method **font**, set of characters specified in method parameters is associated with fragments of image stored in a file. This must be an image in GIF format with unfilled background, containing image of necessary set of characters looking like the following:

Example of file `digits.gif` with image of numbers:



Height of each character is defined as the ratio of image height to the number of characters in the set.
The method has the following parameters:

`Set of characters`—list of characters included in the font file
`Name and path`—of and to the font file
`Space character width`—in pixels
`Character width`—optional parameter

By default, when the file is loaded the width of each of its character is measured, and when outputting the text, proportional font is used. If you specify character width, monospaced font will be used.
All characters must be left aligned to start right from left edge of image.

Example

```
$square[^image::create(100;100;0x00FF00)]
^square.font[0123456789;digits.gif](0)
```

In this case, the file will be loaded, containing images of characters from 0 to 9, and the set of characters will be associated with their graphic equivalents. After the font for the inscription is defined, one can use method **`text`** to make the inscription itself.

text. Making an inscription on an image

```
^image.text(x;y)[inscription text]
```

The method outputs specified text according to the given coordinates (X;Y) using font file loaded by method **`font`** in advance.

Example

```
$square[^image::create(100;100;0x00FF00)]
^square.font[0123456789;digits.gif](0)

^square.text(5;5)[128500]
$response:body[^square.gif[]]
```

Browser will display a green square with "128500" inscribed on it, left top point of the text located at (5;5).

length. Getting inscription's length in pixels

```
^image.length[inscription text]
```

The method calculates inscription's full length in pixels.

Example

```
$square[^image::create(100;100;0x00FF00)]
^square.font[0123456789;digits.gif](0)
^square.length[128500]
```

As a result, full length of the inscription "128500" will be calculated in pixels, paying attention to spaces.

copy. Copying image fragments

```
^image.copy[source](x1;y1;width1;height1;x2;y2)
^image.copy[source](x1;y1;width1;height1;x2;y2;width2;height2;color_precision)
```

The method copies a fragment of one image to another. It is very useful in such tasks as placing signs on a map. The method gets the following parameters:

1. **Source** image
2. Coordinates (**X1**;**Y1**) of the left top corner of copied fragment
3. **Width** and **height** of copied fragment
4. Coordinates (**X2**;**Y2**) to which copied fragment will be pasted
5. As optional parameters you can specify new width and height of pasted fragment (in this case the fragment will undergo scaling), and value characterizing precision of color reproduction, The less this value is, the more precise the color reproduction will be, but number of reproduced colors is decreased in this case—and vice versa (default number of colors is 150)

```
$mygif[^image::load[test.gif]]

$resample_width($mygif.width*2)
$resample_height($mygif.height*2)

$mygif_new[^image::create($resample_width;$resample_height)]
^mygif_new.copy[$mygif](0;0;20;30;0;0;$mygif_new.width;$mygif_new.height)

$response:body[^mygif_new.gif[]]
```

In this example, we create two objects of class **image**. The first is based on existing GIF file; the second, which is twice as big, is generated by Parser itself. After that, we copy into it the fragment of the first file scaled up to the entire width and height of the second image. The last line of the code outputs the scaled fragment. It is advisable to use this approach only to the images, which do not demand high quality.

Inet class

Class inet does not have constructors and therefore cannot create objects. It contains only static methods.

Static methods

aton. Convert string with IP address to number

```
^inet:aton[IP address]
```

IP address will be converted to number. This methos is similar to inet_aton perl and MySQL server functions.

Example

```
^inet:aton[10.0.0.2]
```

...returns number 167772162.

ntoa. Convert number to a string with IP address

```
^inet:ntoa(number)
```

Number will be converted to a string with IP address. This method is similar to inet_ntoa perl and MySQL server functions.

Example

```
^inet:ntoa(167772162)
```

...returns string '10.0.0.2'.

Junction class

This class is designed for storing **code** and **scope** of its execution.
While referring to variables containing **junction**, Parser executes **code** within the stored **scope**.

Value of type **junction** appears in variable:

...when it is assigned a code:

```
$junction{Code to be assigned to variable: ^do_something[]}
```

...when passing code as parameter:

```
@somewhere[]
^method{Code passed as parameter: ^do_something_else[]}
...
@method[parameter]
#in this case junction will be passed into $parameter
```

...while referring to the name of a class method:

```
$action[$user:edit]
#$action[$user:delete]
^action[parameter]
```

In this case, **$action** contains reference to the method and its class. Calling **action** is then identical to calling **^edit[parameter]**.

...when referring to the name of an object method:

```
$action[$person.show_info]
^action[full]
```

In this case, **$action** contains reference to the method and its object. Calling action is then identical to calling **^person.show_info[parameters]**.

```
@check_if_old_enough[age;order_alcohol]
^myif($age<21 && !$order_alcohol){
    Sorry, but we cannot sell strong drinks to ${age}-year-olds.
}
```

Example of using junction of expressions and code

```
@myif[condition;action][age]
$age(11)
^if($condition){
    $action
}
```

Note: *parameter* with expression is code calculating the expression. It is executed—i.e. expression is calculated—every time the parameter is referred to within the call.

In this case, operator **myif** receives code which—along with everything else—outputs **$age**. Operator performs check and executes code within the stored **scope** (**$condition** and **$action**). Therefore, what is to be checked and what is to be output will not depend on whether local variable **age** exists or what its value

is.

Example of checking if method exists

```
^if($some_method is junction){
    ^some_method[parameter]
}{
    no method
}
```

Method **some_method**, will be called only if it is defined.

Mail class

The class is designed to deal with electronic mail. Description of how to configure this class can be found in chapter Configuration.

Static methods

send. Sending a message via e-mail

^mail:send[message]

The method sends message to the specified e-mail address. One can specify several addresses separated by comma.

```
^mail:send[
      $.from[Fred <freddy@hotmail.com>]
      $.to[Peter <peter@hotmail.com>]
      $.subject[Hi there!!!]
      $.text[How is it going? Haven't seen you for ages!]
]
```

As a result of this code, a message will be sent to **peter@hotmail.com** containing text: "How is it going? Haven't seen you for ages!"

message—is a hash, where you can specify the following keys:

- **header_field**
- **text**
- **html**
- **file**
- **charset**
- **options**
- **print-debug** ***[3.4.0]***

Note: It is recommended that in `errors-to` *header you specify the address to which delivery report will be sent if the message cannot be delivered to the addressee. Default value is "*`postmaster`*".*

charset—if this key is specified, the headers and text blocks will be transcoded using specified charset. Default charset for all messages is that, specified in **$request:charset** (i.e. is not transcoded).

Example:

```
$.charset[koi8-r]
```

options—these options will be passed to command line of `sendmail` program (only on UNIX).

print-debug—the message text will be printed instead of sending message.

You can also specify all message headers, specifying their values in the following way (short form):

```
$.header_field[value]
```

or with parameters (complete form):

```
$.header_field[
    $.value[string]
    $.parameter[string]
]
```

Examples:

```
$.from[Fred <freddy@hotmail.com>]
$.to[Peter <peter@hotmail.com>]
$.subject[How is it going? Haven't seen you for ages!]
$.x-mailer[Parser 3]
```

Along with the header you can send one or both text blocks (**text**, **html**) as well as any number of blocks **file** and **message** (see below).
If you send both text blocks, section `MULTIPART/ALTERNATIVE` will be formed, and having received this message, modern mail clients will display HTML, whereas obsolete ones will display plain text.

short form:

```
$.text[string]
```

full form:

```
$.text[
    $.value[string]
    $.header_field[value]
]
```

...where **value** is the value of text block and you can specify all header fields the same way we did with hash **message** (see above).
Note: It is not imperative to specify `content-type` *header—it will be generated automatically. This header does not affect transcoding process and is used only to tell mail client what charset it must use to display message.*

Sending HTML. Short form:

```
$.html{string}
```

full form:

```
$.html[
    $.value{string}
    $.header_field[value]
]
```

Curly brackets are necessary to switch default transformation type to HTML.

Attaching a file. Short form:

```
$.file[file]
```

full form:

```
$.file[
    $.value[file]
    $.name[filename]
    $.content-id[XYZ]          [3.2.2]
    $.format[uue|base64]       [3.2.2]
    $.header_field[value]
]
```

File is an object of class **file**, which will be attached to the message. MIME-type of sent data (`content-type` header of a part) is determined according to table **MIME-TYPES** (see also Configuration method).

Filename is the name that the file to be sent will bear.
By default the file will be sent in uuencode form.

Default file encoding **format** is *base64* since version 3.4.0 and *uue* prior version 3.4.0.

Attaching a message:
```
$.message[message_text]
```

The format of the message is the same as that of the whole method's parameter.

There may be several attachments. In this case you must add an integer after the name.
Example:
```
    $.file
    $.file2
    $.message
    $.message2
```

Example of how to use alternative blocks and attachments
```
^mail:send[
    $.from[Fred <freddy@hotmail.com>>]
    $.to[Peter <peter@hotmail.com>]
    $.subject[How is it going?]
    $.text[How is it going? It's really great in here!!!]
    $.html{How is it going? It's really <b>great</b> in here!!!
        <br /><img src="cid:pic2" border="0" width="100" height="50" />
    }
    $.file[^file::load[binary;perfect_life1.jpg]]
    $.file2[
        $.value[^file::load[binary;perfect_life2.jpg]]
        $.name[pf_life2.jpg]
        $.content-id[pic2]
    ]
]
```

As a result, a message will be sent to **peter@hotmail.com** containing text "How is it going? It's really great in here!!!" in plain text and HTML. Two photographs will be attached to the message to support the idea, and on these photos...

Math class

Class math does not have constructors and therefore cannot create objects. It contains only static methods and is used to deal with mathematical expressions. While working with this class it is important to keep in mind values precision of class **double**.

Static fields

Pi number

$math:PI—π value

Static methods

abs, sign. Operations with number sign

The methods perform operations with number sign

^math:abs(number)	returns absolute value of number (module)
^math:sign(number)	returns **1** if number is positive, **–1** if negative and **0** if number equals zero

Example

^math:abs(-15.506)	returns 15.506
^math:sign(-15.506)	returns -1

round, floor, ceiling. Rounding of number

^math:round(number)	– rounding to the closest integer
^math:floor(number)	–rounding towards lesser integer
^math:ceiling(number)	–rounding towards greater integer

The methods return round value of the given number of class **double**.

Example

^math:round(45.50)	–Will equal 46
^math:floor(45.60)	–Will equal 45
^math:ceiling(45.20)	–Will equal 46
^math:round(-4.5)	–Will equal -4
^math:floor(-4.6)	–Will equal -5
^math:ceiling(-4.20)	–Will equal -4

trunc, frac. Operations with integer/fractional part

^math:trunc(number)	– Returns integer part
^math:frac(number)	– Returns fractional part

Example

^math:trunc(85.506)	–Will return 85
^math:frac(85.506)	–Will return 0.506

degrees, radians. Degrees–radians transformation

The methods transform degrees into radians and vice versa.

^math:degrees(number_of_radians)	returns a number of degrees equal to the specified number of radians
^math:radians(number_of_radians)	returns a number of radians equal to the specified number of degrees

Example

`^math:degrees($math:PI/2)`	returns 90 (degrees)
`^math:radians(180)`	returns π

sin, asin, cos, acos, tan, atan. Trigonometric functions

`^math:sin(radians)`	sine
`^math:asin(number)`	arc sine
`^math:cos(radians)`	Cosine
`^math:acos(number)`	arc cosine
`^math:tan(radians)`	Tangent
`^math:atan(number)`	arc tangent

These methods calculate values of trigonometric functions of a specified number.

Example

```
^math:cos(^math:radians(180))
```

will return `-1 (cos π = -1)`.

exp, log, log10. Logarithmic functions

`^math:exp(number)`	the exp function returns the exponential value of parameter
`^math:log(number)`	natural logarithm
`^math:log10(number)`	the base 10 logarithm

These methods calculate values of logarithmic functions with specified number.

Note: (if you have only vague memories of your school years):
the base-B logarithm of V is calculated as log(V)/log(B)

pow. Raising a number to power

```
^math:pow(number;power)
```

This method raises a number to power.
Example

```
^math:pow(2;10)
```

...returns 1024, i.e. ($2^{10} = 1024$)

sqrt. Square root of a number

```
^math:sqrt(number)
```

This method calculates square root of the number.

Example

```
^math:sqrt(16)
```

...returns 4.

Note (if you have completely forgotten what you learnt at school):
n''' root of a number is calculated by raising it to the power 1/n.

random. Random number

```
^math:random(upper_limit)
```

The method returns a random number, which is taken from the range starting with 0 and ending with the number specified in **upper_limit** (the number given as **upper_limit** is not included in the range).
Note: In some systems it outputs a pseudorandom number.

Example

```
^math:random(1000)
```

The code returns a random number from the range starting with 0 and ending with 999.

uuid. Universally unique identifier

```
^math:uuid[]
```

The method outputs random string of format...
22C0983C-E26E-4169-BD07-77ECE9405BA5

Note: in some OSes outputs pseudorandom string.

This method is useful in cases when it is hard or insensible to use through-numbering of objects, e.g. while performing distributed computing.

UUID is also known as GUID.

Example

A company's branches accumulate orders and periodically send them to headquarters. To ensure identifier's uniqueness, we use UUID.

```
# different branches accumulate order's information in tables 'orders' and
'order_details'

# create unique identifier
$order_uuid[^math:uuid[]]

# add record about order
^void:sql{
insert into orders
    (order_uuid, date_ordered, total)
values
    ('$order_uuid', '$date_ordered', $total)
}
# cycle adding records on ordered goods should be here
^void:sql{
insert into order_details
    (order_uuid, item_id, price)
values
    ('$order_uuid', $item_id, $price)
}


# parts of tables 'orders' and 'order_details' are periodically retrieved
# and sent (^mail:send[…]) to headquarters,
# where these parts of tables are added to common tables 'orders' and
'order_details'
# …WITHOUT any problems with multiple instances of 'order_id'
```

Note: Parser generates UUID based on random numbers, not on time. Parameters are:
variant = DCE;

version = DCE Security version, with embedded POSIX UIDs.
...that means that not all of the UUID bits are picked up at random. It is to be so, indeed:
xxxxxxxx-xxxx-4xxx-{8,9,A,B}xxx-xxxxxxxxxxxx

Detailed information on UUID is available at: http://www.opengroup.org/onlinepubs/9629399/apdxa.htm

uuid64. 64-bit unique identifier

^math:uid64[]

The method returns a random string of format:
BA39BAB6340BE370

Note: in some OSes it results in pseudorandom string.

See **^math:uuid[]**.

md5. MD5 hash of a string

^math:md5[string]

The method gets 16-byte hash of specified **string** and outputs it as a string—bytes are output in hexadecimal code without delimiters, lowercase.

It is believed that:
- it is practically impossible for two strings to have the same MD5-hash;
- it is practically impossible to restore original string from its MD5-hash.

Example
As a name of cache-file we will use an MD5-hash of **$request:uri**, It will not only provide univocal match of filename and request string, but also deliver us from necessity of shortening request string and removing special characters from it.

```
^cache[$cache_directory/^math:md5[$request:uri]]($cache_time){
    ...
}
```

Detailed information on MD5 is available at http://www.ietf.org/rfc/rfc1321.txt

crypt. Hashing passwords

^math:crypt[password;salt]

The method hashes **password**. Parameters are **password** to be encrypted and **salt** to base encryption on.

Arguments:
password—initial string;
salt—string determining hashing algorithm and introducing an element of randomness into hashing process—consists of head and body. If body is not specified, Parser will generate a random body.

It is not very sensible to store users' passwords simply storing them in a database or saving to disk—since, having managed to steal a file or DB table with passwords, someone will be able to use them. That is why one should store not passwords themselves but their hashes—that is the result of safe and irreversible transformation of password string. While password typed in by a visitor is checked, the received string is encrypted according to the same algorithm as that of password stored in a file/database (this encrypted password is used as **salt**), and the two strings are then compared .

Table with available algorithms:

Algorithm	*Description*	*salt head*	*salt body*
MD5	built-in in Parser, available on all platforms	**$apr1$**	Up to 8 random letters (in uppercase or lowercase) or numbers
MD5	if supported by UNIX OS	**1**	Up to 8 random letters (in uppercase or lowercase) or numbers
DES	if supported by UNIX OS	(no)	2 random letters (in uppercase or lowercase) or numbers
others	those supported by UNIX OS	read the documentation on your operating system, function crypt	read the documentation on your operating system, function crypt

Note: to use **$** *in Parser, you must precede it with* **^**.

Note: Apache web-server allows using hashed passwords in password files (`.htpasswd`*). In this case you may use hashes of passwords created by any of the algorithms given in the above table, including algorithm built into Parser.*

How to create .htpasswd file:

```
@main[]
$users[^table::create{name    password
alice xxxxxx
bob   yyyyyy
}]

$htpasswd[^table::create[nameless]{}]
^users.menu{
    ^htpasswd.append{$users.name:^math:crypt[$users.password;^$apr1^$]}
}

^htpasswd.save[nameless;.htpasswd-parser-test]
```

How to check password

```
$right[123]
$from_user[123]
$crypted[^math:crypt[$right;^$apr1^$]]
#Note: $crypted will be different every time it is referred to
$crypted<br />
^if(^math:crypt[$from_user;$crypted] eq $crypted){
    Eat, drink, and be merry
}{
    Call 911…
}
```

Detailed information on MD5 is available at *http://www.ietf.org/rfc/rfc1321.txt*

crc32. String checksum calculation

^math:crc32[string]

The method gets CRC32 checksum for specified **string** and outputs it as an integer.

sha1. SHA1 hash of string

^math:sha1[string]

The method gets SHA1 hash for specified **string**.

Memory class

This class is designed for working with Parser's memory.
Using it will help you save memory in your scripts.

Note for inquisitive minds: Parser uses famous and widely respected conservative garbage collector Boehm-Demers-Weiser, see http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

Static method

compact. Collecting garbage

```
^memory:compact[]
```

This method collects so-called "garbage" in memory, cleaning it up for reuse by your code.
Garbage is memory no longer used by your code, i.e. to which there are no references in your code.

For example:

```
$table[^table::sql{query}]
$table[]
# free up memory occupied by SQL-query result
^memory:compact[]
```

Parser does not collect garbage automatically leaving decision-making to coder: call **compact** from place(s) where you expect greatest benefit, for example before XSL-transformation.

$status:memory will help you fix and find places most favorable for collecting garbage.

Important notice: it is necessary to use local variables as intensely as possible and zero out, which you no longer need. All this will help ***compact*** *free up more memory.*

Important notice: total memory cleaning is not guaranteed.

Reflection class

The class is designed for getting information about objects, classes and methodes.

Static methods

create. Create an object

```
^reflection:create[class name;constructor name]
^reflection:create[class name;constructor name;parameters]
```

Creates an object of a class with the specified name by calling a constructor with the specified name.
This method can be useful if you need to create an object of the class which name you have in a variable.

Note: in this method you can not specify more than 100 parameters.

classes. Classes listing

```
^reflection:classes[]
```

Returns the hash with all classes.
The keys of the hash are classes' names, the values are strings **methoded** (for classes with methods) or **void**.

class. Object's class

```
^reflection:class[object]
```

Returns the object's class (similar to **$object.CLASS**).

class_name. Name of object's class

```
^reflection:class_name[object]
```

Returns the object's class name (similar to **$object.CLASS_NAME**).

base. Object's base class

```
^reflection:base[class]
^reflection:base[object]
```

Returns the object's base class (if any) or **void**.

base_name. Name of object's base class

```
^reflection:base_name[class]
^reflection:base_name[object]
```

Returns the object's base class name (if any) or **void**.

methods.Class's methods listing

```
^reflection:methods[class name]
```

Returns the hash with all methods of the class with the specified name.
The keys of the hash are methods' names, the values are strings **native** (for the system classes) or **parser** (for the user-defined classes).

method_info. Getting information about method

```
^reflection:method_info[class name;method name]
```

Returns the hash with information about the specified method of a class with the specified name.

For the system classes returns:

```
$hash[
    $.inherited[class name, where the method was defined]
    $.min_params(minimum required number of method's parameters)
    $.max_params(maximum allowed number of method's parameters)
    $.call_type[method's allowed call type: static, dynamic or any]
]
```

For the user-defined classes returns:

```
$hash[
    $.inherited[class name, where the method was defined]
    $.0[the name of the first method's parameter]
    $.1[the name of the second method's parameter]
    ...
]
```

fields. Object's fields listing

```
^reflection:fields[class]
^reflection:fields[object]
```

For class the method returns the hash with static fields.
For object the method returns the hash with dynamic fields.

dynamical. Getting method's call type

```
^reflection:dynamical[]
^reflection:dynamical[class]
^reflection:dynamical[object]
```

Without the parameter the method returns **true** if the calling method was called dynamically and returns **false** if the calling method was called statically.

With the parameter the method returns **true** if an object was passed and returns **false** if a class was passed.

^reflection:dynamical[] may be useful inside the methods when you need to know if these methods were called–dynamically or statically.

Regex class

The class is designed for working with PCRE—Perl-compatible regular expression.
A regex-object is always defined (**def**). Numerical value of a regex-object is the size of compiled pattern (in bytes).

Constructor

create. Creating an object

```
^regex::create[pattern]
^regex::create[pattern][options]
```

Creates a regex-object from **string-pattern**. Pattern is a PCRE—Perl-compatible regular expression. Some examples of PCRE are given in "Attachment 4: Perl Compatible Regular Expressions".

The following search **options** may be used:

i—case-insensitive;
x—ignore "white space" characters and allow **#comments till the end of the line**;
s—regard **$** as the end of the whole text (default);
m—regard **$** as the end of the line, but not the whole text;
U—inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by **?**; ***[3.3.0]***
g—find not only the first, but all occurrences of the pattern;
n—return number of matches instead of table with search results; ***[3.2.2]***
'—evaluate values for **prematch**, **match**, **postmatch** columns.

Characters **^** and **$** are used in Parser's syntax, that is why if you want to include them in your pattern, they must be given as **^^** and **^$** respectively (see also Literals).

Fields

pattern
$regex_object.pattern

The field contains the **string-pattern**.

options
$regex_object.options

The field contains the **string-options**.

Request class

Class request contains static fields, which allow getting information sent by browser to web-server (via HTTP protocol).

To work with form fields (**<FORM>**) and string after second **?** (**/?a=b?thisText**), use **form**.

A part of information on the request is accessible through environment variables, see "Retrieving values of HTTP-header fields".

Static fields

uri. Getting the URI of the page

$request:uri

Returns document's URI.

Example
Let's assume, a visitor requests the following page:

http://www.mysite.ru/news/index.html?year=2000&month=05&day=27

Then
$request:uri

will return:
/news/index.html?year=2000&month=05&day=27

query. Getting the query string

$request:query

Returns the string coming after **?** in URI (the value of environment variable **QUERY_STRING**).
To work with form fields (**<FORM>**) and string after second **?** (**/?a=b?thisText**), use class **form**.

Example
Let us assume, a visitor requests page at

http://www.mysite.ru/news/index.html?year=2000&month=05&day=27

then

$request:query

will return

```
year=2000&month=05&day=27
```

charset. Specifying server's charset

$request:charset[charset]

Specifies the charset of documents processed at server.
While processing users' requests the server regards all documents as having the same charset.

The default charset is UTF-8.

The list of possible charsets is specified in Configuration method.
It is recommended to specify the documents' charset in Configuration file.

The charset, in which the result of the Parser's code will be output may be specified by $response:charset.

post-charset. Getting the character set specified in incoming POST request

$request:post-charset

If content-type HTTP header for incoming POST request contains character set information, this character set name will be available in this field.
During building form fields from such request the incoming data will be transcoded from this charset instead of charset specified in $response:charset.

Note: if character set which specified in content-type HTTP header for incoming POST request wasn't pluged in (at configuration method for example) you will receive an error message.

body. Getting query's text

$request:body

The field contains text of HTTP POST-query.

Example: one can create one's own **XML-RPC** server (see http://www.xmlrpc.com).

document-root. Root of web-space

$request:document-root[/disk/path/to/the/root/of/your/web-space]

By default **$request:document-root** equals the value, which is configured in web-server. But sometimes it is convenient to change it.

See also "Paths to files and directories".

argv. Command line parameters

$request:argv

The field contains hash with command line parameters (keys: 0, 1, 2 etc) which can be usable while using parser as a standalone interpreter (in cron for example).

$request:argv.0 contains the name of processing file.

Response class

Class response allows complementing standard HTTP-responses of the server. The class doesn't have constructors and, therefore, cannot create objects.

Static fields

HTTP-response headers

```
$response:field[value]
$response:field
```

The field corresponds with HTTP-response header generated by Parser. It can be both assigned and referred to. The value may be a date, a string or a hash with obligatory key **value**.

Note: during output to the browser all HTTP-response headers' names are capitalized (for example: ***content-type*** *are transformed to* ***Content-Type****).* ***[3.4.0]***

Example of redirecting a visitor to site's mainpage

```
#works if web-site administrator correctly configured SERVER_NAME environment
variable
#usually he/she did
$response:location[http://$env:SERVER_NAME/]
```

Another example of redirecting a visitor to site's mainpage

```
#works regardless of SERVER_NAME
$response:refresh[
      $.value[0]
      $.url[/]
]
```

Example of assigning header "expires" a value "tomorrow"

```
$response:expires[^date::now(+1)]
```

headers. HTTP-response headers

```
$form:headers
```

Such a construction returns hash with all HTTP-response headers set so far.

Example

```
$response:expires[^date::now(+1)]
^response:headers.foreach[header;value]{
      $header — ^if($value is "string" || $value is "int" || $value is
"double"){$value}{not printable}
}[<br />]
```

...will output all HTTP-response headers that were set up to that moment.

body. Specifying a new response body

$response:body[DATA]

Here, **DATA** substitutes for the whole response body.

DATA may be a string, file or hash of parameters.

Keys of hash of parameters: ***[3.1.4]***
file — name of file on disk (in this case Parser supports continuing of broken downloads. ***[3.1.4]***);
name — name of file to pass to visitor;
mdate — date and time of file last modification to pass to visitor.

If **content-type** of sent file is known, Content-Type header is also output to the browser (see "**Fields of object of class file**").

See also **$response:download**.

Example of how to replace the whole body of the response with the results of the script's work

```
$response:body[^file::cgi[script.cgi]]
```

...will replace the body of the response with the data returned by the program **script.cgi**.

Example of how to create and output an image

```
$square[^image::create(100;100;0x000000)]
^square.circle(50;50;10;0xFFFFFF)
$response:body[^square.gif[]]
```

As a result, the browser will output a black square with a white circle. Besides, a necessary type of file (content-type) according to table **MIME-TYPES** will be reported to the browser.

download. Specifying a new response body

$response:download[DATA]

This field is identical to **$response:body**, but it sets flag that browser interprets as "Suggest that visitor save file to disk."

Browsers are able to display certain file types right within their windows (for example: `.doc`, `.pdf` files). Still, sometimes we should enable a visitor to download the file by simply clicking a relevant link.

Example: outputting a PDF file

A visitor is at page with such HTML:

```
<a href="/download_documentation.html">Download documentation</a>
```

`download_documentation.html`:

```
$response:download[^file::load[binary;documentation.pdf]]
```

...visitor clicks the link and browser suggests Open/Download.

charset. Specifying response charset

$response:charset[charset]

Specifies charset of the response.
The data resulted from the request's processing will be transcoded into specified charset.

The default encoding is **UTF-8**.

The list of possible charsets is specified in Configuration method.
It is recommended to specify the documents' charset in Configuration file.

See also "Specifying server's charset".

Static methods

clear. Cancelling re-definition

```
^response:clear[]
```

The method will cancel all actions on redefining response fields.

Status class

This class is designed for analyzing current status of a Parser script.
Using it will help you find bottle necks in your scripts.

If you use parser as Apache module you will receive error message **class not found** while using this class until you don't add to httpd.conf lines:

<Location />
allow to use status class
ParserStatusAllowed
</Location>

and don't restart Apache server.

Fields

rusage. Information on resources used

This field is a hash containing information on server's resources currently used by system for processing your Parser-script.

Some systems cannot return complete range of values listed here (WinNT/Win2000/WinXP can return all values, while Win98 can return only **`tv_sec`** and **`tv_usec`**).

Key	*Unit*	*Value description*	*How to reduce?*
utime	second	Pure time, i.e. that used by current process (does not include time used by other tasks)	Simplify data manipulation within Parser (improve algorithm, hand some actions over to SQL-server)
stime	second	Time used by system to read your files, directories, and libraries	Decrease number and size of files needed for script's work; do not use modules which are not needed to process current document
maxrss	block	Memory used by process	Decrease number of loaded useless data. Find and fix all "select *" by specifying only the fields you will really need. Do not load unnecessary data from SQL-server, filter out as much as you can by means of SQL-server itself.
		Exact system time. Allows evaluating time used for awaiting response from SQL-, HTTP-, SMTP-servers.	Simplify SQL queries. If you use MySQL, use EXPLAIN; for Oracle: EXPLAIN PLAN (see your server documentation); for other SQL-servers: see relevant documentation.
		How much time passed since Epoch...	
tv_sec	second	...whole seconds;	
tv_usec	millisecond (10E-6)	...milliseconds passed (millionths of seconds in addition to whole seconds)	

Recommended way of analysing

At the end of your script place construction...

```
^rusage[total]
```

...to call this method:

```
@rusage[comment][v;now;prefix;message;line;usec]
$v[$status:rusage]
$now[^date::now[]]
$usec(^v.tv_usec.double[])
$prefix[[^now.sql-string[].^usec.format[%06.0f]] $env:REMOTE_ADDR: $comment]
$message[$v.utime $v.stime $request:uri]
$line[$prefix $message ^#0A]
^line.save[append;/rusage.log]
$result[]
```

…and analyze the log.

For a more precise analysis, surround the block to be checked with calls…

```
^rusage[before XXX]
^rusage[after XXX]
```

Note: it is not recommended to store log file within web-space.

WinNT/2K/XP

Under these OSes, certain extra values are available:

Key	*Unit*	*Value description*	*How to reduce?*
ReadOperationCount **ReadTransferCount**	items bytes	Number of operations on reading from disk and total number of bytes read	Decrease number and size of file needed for the process; do not use modules not needed for processing current document. Use SQL-server rather than files.
WriteOperationCount **WriteTransferCount**	items bytes	Number of operations on writing to disk and total number of bytes written	
OtherOperationCount **OtherTransferCount**	items bytes	Number of other operations with disk (apart from read/write) and total number of bytes transferred	
PeakPagefileUsage QuotaPeakNonPagedPoolUsage QuotaPeakPagedPoolUsage	bytes	Memory-paging file size limit	see above comment to **maxrss**

memory. Information on memory—controlled by garbage collector

This field is a hash containing information on memory controlled by garbage collector.

Field	*Value (in kilobytes)*	*Details*
used	memory used	This number does not include size of housekeeping data used by garbage collector itself.
free	free memory	Free memory is most probably fragmented.
ever_allocated_since_compact	How much memory was allocated since last garbage cleaning, see **memory:compact**.	This number constantly increases between garbage cleaning procedures. Freeing memory procedures alone do not affect it. It is affected only by garbage cleaning procedures.
ever_allocated_since_start	How much memory was allocated during the whole request processing	This number constantly increases. It is affected by neither garbage cleaning procedures nor freeing memory procedures between them.

Recommended way of analysing

Surround the block to be checked with constructions...

```
^musage[before XXX]
^musage[after XXX]
```

...to call this method:

```
@musage[comment][v;now;prefix;message;line]
$v[$status:memory]
$now[^date::now[]]
$prefix[[^now.sql-string[]] $env:REMOTE_ADDR: $comment]
$message[$v.used $v.free $v.ever_allocated_since_compact
$v.ever_allocated_since_start $request:uri]
```

```
$line[$prefix $message ^#0A]
^line.save[append;/musage.log]
$result[]
```

...and analyze the log.

Important notice: while working, Parser takes additional memory blocks from system as required. That is why it is normal when both ***used*** *and* ***free*** *increase from time to time.*

Note: it is not recommended to store log file within web-space.

pid. Process identifier

Identifier of the OS process in which Parser is running.

tid. Thread identifier

Identifier of the OS thread in which Parser is running.

String class

The class is designed for working with strings. String is considered defined (**def**), if it isn't empty. If string contains a number, the content of the string will be automatically converted to **double**. When used in a mathematical expression. If the string is empty, its numerical "value" in mathematical expressions will be regarded as zero.

Creating object of class **string**:

```
$str[content of the string]
```

Static methods

sql. Retrieving string from a database

```
^string:sql{SQL-query}
^string:sql{SQL-query}[$.limit(1) $.offset(o) $.default{code} $.bind[variables hash]]
```

Note: this is method, not a constructor!

Returns **string** retrieved from database through SQL-query. The query must result in only one column of only one row. For this operator to work, you must have connection with database server established (see operator **connect**).

Optional parameters:
$.limit(1) - limit response to one row only;
$.offset(o) - ignore first **o** records in the query results;
$.bind[hash] - variables to bind, see «Queries with bound variables» ***[3.1.4]***.
if SQL-server response was empty (0 records), ...
$.default{code} ...the given code will be executed and string result returned;
$.default(expression) ...the given expression will be evaluated returned;
$.default[string] ...the given string returned;

Example

```
^string:sql{select name from company where company_id=$company_id}
```

While using this method, it is recommendable to construct SQL-query in such a way as to limit response to one column in one row only.

base64. Decoding from Base64

```
^string:base64[encoded]
```

Note: this is method, not a constructor!

Decodes a string from Base64 representation. To encode a string use
^string.base64[]

Detailed information on MD5 is available here http://www.ietf.org/rfc/rfc2045.txt and here http://en.wikipedia.org/wiki/Base64.

Example

```
$encoded[
pyAxOTczLiBUaGVyZSBhcmUgcnVtb3VycyB0aGF0IJNHcmVlbiBzbGVldmVzlCB3ZXJlIHdyaXR0
ZW4gYnmF
]
$original[^string:base64[$encoded]]
$original
```

Outputs...
§ 1973. There are rumours that "Green sleeves" were written by...

js-unescape. Decoding similar to unescape function in JavaScript

```
^string:js-unescape[escaped]
```

Note: this is method, not a constructor!

Unescapes a string. This method do the transformation similar to **unescape** function described in ECMA-262. Using this method you can decode strings which were escaped in browser by JavaScript function **escape**.

To escape a string use
^string.js-escape[]

Detailed information on ECMA-262 is available here:
http://www.ecma-international.org/publications/standards/Ecma-262.htm (B.2.2)

Example

```
$escaped[abcd%20%60+-
%3D%7E%21@%23%25%26*%28%29_%20%5B%5D%7B%7D%3C%3E%3A%27%22%2C./%3F%u0430%u0431%u
0432%u0433%u0434]
$original[^string:js-unescape[$escaped]]
$original
```

Outputs...
abcd `+-=~!@#%&*()_ []{}<>:'",./?абвгд

Methods

int, double, bool. Converting string into number or bool

```
^string.int[]
^string.int(default value)
^string.double[]
^string.double(default value)
^string.bool[]
^string.bool(default value)
```

Converts value of variable **$string** into integer or real number or bool respectively and returns the result.

One can specify default value to be returned if conversion is impossible, a string is empty or consists of white space characters (tabs, spaces, newlines).

Default value can be used while processing data, which is received from users interactively. It will help avoiding text values in mathematical expressions—in cases of incorrect input (e.g. when a string is received instead of expected number).

Note: method **.bool** can convert to **bool** not only strings with numbers (0–**false**, not 0–**true**) but strings containing values 'true'/'false' as well (case insensitive). It can be usable for reading data from external source (xml for example).

Note: using empty string in mathematical expressions expressions is not considered error. Its value is then regarded as zero.

Note: attempt of converting non-integer string into integer is considered error (e.g. string "1.5" is not an integer).

Example

```
$str[123]
^str.int[]
```

...outputs number **123**, since object **str** can be converted into number.

```
$str[much]
^str.double(-1)
```

...outputs number **-1**, since conversion is impossible.

```
$str[1]
^if(^str.bool[]){true}

$str[True]
^if(^str.bool[]){true}
```

...outputs strings "**true**".

format. Outputting a number in specified format

```
^string.format[format_string]
```

The method outputs variable's value in **specified format** (see format strings).
The **string** is automatically converted into a number.

Example

```
$var[15.67678678]
^var.format[%.2f]
```

The code will return **15.68**

split. Splitting a string

```
^string.split[delimiter]
^string.split[delimiter;splitting options]
^string.split[delimiter;splitting options;column name]        [3.2.2]
```

The method splits **string** into substrings using **delimiter** substring and creates an object of class **table**,

containing:

- either a table with single column, where it places the resulted parts,
- or a nameless table where resulted parts are columns of single row.

Splitting options include:
l—split from left to right (default);
r—split from right to left;
h—form nameless table with resulted parts placed horizontally;
v—form table with single column, where resulted parts are placed vertically (default).

The name of column for vertical split—"**piece**" or the **column name** passed as a parameter.

Example of using vertical split

```
$str[Strangers in the night…]
$parts[^str.split[the]]
^parts.save[parts.txt]
```

The code in the example will create file `parts.txt` containing...

```
piece
Strangers in
 night…
```

Example of using horizontal split

```
$str[/a/b/c/d]
$parts[^str.split[/;lh]]
$parts.0, $parts.1, $parts.2
```

...outputs:

```
, a, b
```

upper, lower. Changing case of the string

```
^string.upper[]
^string.lower[]
```

These methods convert **string** to uppercase or lowercase. For this method to work, we need **$request:charset** to be specified.

Example

```
$str[Keep off the grass!]
^str.upper[]
```

The code will return: **KEEP OFF THE GRASS!**

length. Getting string's length

```
^string.length[]
```

The method returns string's length.

Example

```
$str[Strangers in the night…]
^str.length[]
```

The code will return: **23**

mid. Getting substring from a specified position

```
^string.mid(P;N)
^string.mid(P)
```

The method returns substring which starts from position **P** and has length specified as **N** (if **N** is not given, the method will return the substring from position **P** and to the end of the string). **P** is counted from zero position. If value of **P**+**N** equals more that the length of the string the method will return all characters of the string after **P**.

Example

```
$str[Strangers in the night…]
^str.mid(2;19)
```

The code will output: **rangers in the nigh**

left, right. Getting substring on the left and on the right

These methods return **N** first or last characters of the string respectively. If value of **N** is more than the string's length, the method will output the whole string.

Example

```
$str[Strangers in the night…]
^str.left(7) ^str.right(9)
```

The code will output: **Strange night**...

pos. Getting substring's position

```
^string.pos[substring]
^string.pos[substring](offset)        [3.3.0]
```

The method returns a number **int**, that is the position of the first character of the substring (beginning with zero), or **-1** if substring cannot be found.
If **offset** was specified the substring will be searching from specified position.

Examples

```
$str[Strangers in the night…]
^str.pos[range]
```

The code will return: **2**

```
$str[Strangers in the night…]
^str.pos[t](2)
```

The code will return: **13**

replace. Replacing substrings in the string

```
^string.replace[table_with_substitution_settings]
```

Replaces substrings in the **string** using **substitution settings**.

Table_with_substitution_settings is an object of class **table**, containing two columns:

The first contains the substring to be replaced.
The second contains the substring to replace the first one.

It is not necessary to specify column names – you may call it 'from' and 'to' or simply skip naming by using **nameless** table.

Example

```
$s[An ugly moment I'll remember!]
Original: $s<br />
$rep[^table::create{from      to
An    A
ugly  magic}]
After replace: ^s.replace[$rep]
```

The code will output:

```
Original: An ugly moment I'll remember!
After replace: A magic moment I'll remember!
```

save. Saving string to a file

```
^string.save[path_and_filename]
^string.save[append;path_and_filename]
^string.save[path_and_filename;options]    [3.4.0]
```

Saves or appends **string** to a file in specified directory.
While being saved, string fragments undergo necessary transformation, see "Transforming data".

The **options** are hash, with such keys as:
$.charset[charset]
$.append(true)

Example

Task: retrieve data from SQL-server A and store them to SQL-server B.

If both servers are accessible from some computer, it can be done this way:

```
^connect[A]{
    $data[
#       code to fill 'data' with data from SQL-server A
    ]
    ^connect[B]{
         ^void:sql{insert into table x (x) values ('$data')}
    }
}
```

In this case, **$data** in SQL-query insert will be correctly adapted to SQL-dialect used by server B.

Yet, if one CANNOT access both servers from one computer, the task may be accomplished the following way:

```
^connect[A]{
    $data[
#       code to fill 'data' with data from SQL-server A
    ]
    $string[^untaint[sql]{insert into table x (x) values ('$data')}]
    ^connect[local fictitious B]{
#       this connection is needed
#       only to specify rules to conform SQL-syntax used by SQL-server B
        ^string.save[B-inserts.sql]
    }
```

```
}
```

In this case file `B-inserts.sql` will contain correctly transformed SQL-query.

match. Matching a pattern

```
^string.match[pattern]
^string.match[pattern][options]
```

The operator searches for a match of a **pattern** in a **string**. Pattern could be a string with PCRE—Perl-compatible regular expression—or **regex-object** ***[3.4.0]***.
Some examples of PCRE are given in "Attachment 4: Perl Compatible Regular Expressions".

The following search **options** may be used:

i—case-insensitive;
x—ignore "white space" characters and allow **#comments till the end of the line**;
s—regard **$** as the end of the whole text (default);
m—regard **$** as the end of the line, but not the whole text;
U—inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by **?**; ***[3.3.0]***
g—find not only the first, but all occurrences of the pattern;
n—return number of matches instead of table with search results; ***[3.2.2]***
'—evaluate values for **prematch**, **match**, **postmatch** columns.

Characters **^** and **$** are used in Parser's syntax, that is why if you want to include them in your pattern, they must be given as **^^** and **^$** respectively (see also Literals).

If option **g** is specified, a table with the results of the match will be created with one row per each occurrence. If option **g** is **not** specified, a table with the results will contain only one record with first occurrence. If substring is not found, the result of operation will be **empty** table. If option **n** is specified, a number of matches will be returned instead of table.

A matches' table (object of class table) contain the next columns **1**, **2**, ..., **n**, **prematch**, **match**, **postmatch**, where
prematch is the column with substring coming from the beginning of the string to the place where the pattern-matching substring was found
match is the column with the pattern-matching substring
postmatch is the column with the substring that comes after pattern-matching substring and up to the end of the entire string
1, **2**, ..., **n** are the columns with pattern-matching substrings enclosed in round brackets, where **n** is number of the left bracket

Note 1: values for **prematch**, **match**, **postmatch** *columns are evaluated only if option* **'** *is specified.*
Note 2: values for **1**, **2**, ..., **n** *are evaluated only if round brackets used in pattern.*
Note 3: you can use (?:...) instead of (...) in pattern if you don't need some parts of matches in table with results

Examples

```
$str[www.parser.ru?user=admin]
^if(^str.match[\?.+]){match found}{match not found}
```

The code will output: **match found**

```
$str[www.parser.ru?user=admin]
$mtc[^str.match[(\?.+)]['']]
^mtc.save[match.txt]
```

The example will create a file `match.txt`, with the following table:

prematch	match	postmatch	1
www.parser.ru	?user=admin		?user=admin

match. Replacing pattern-matching substring

```
^string.match[pattern][search options]{replacer}
^string.match[pattern][search options][replacer] [3.4.0]
```

The method searches the string for a match and replaces the pattern-matching substring with a substring given in curly brackets. The search mechanism is the same as in the previously given method. Automatically created matches' table **match**, described in the previous method, is available within the code.

Example

```
$str[2002.01.01]
^str.match[(\d+)\.(\d+)\.(\d+)][g]{Year: $match.1, month: $match.2, day:
$match.3}
```

The code will output: **Year: 2002, month: 01, day: 01**

trim. Trimming letters

```
^string.trim[]
^string.trim[from]
^string.trim[from;set]
```

The methods trims any letters from **set** from ends of string. By default it trims white space characters from the beginning and end of then string.
It can be specified, where to trim letters **from**, by passing one of values:

- **both**—trim from either the beginning or the end;
- **left** or **start**—trim from the beginning;
- **right** or **end**—trim from the end.

It can also be specified, which **letters** to trim.

Example: white space trimming

```
$name[ Bob ]
"$name"
"^name.trim[]"
```

Will output...

```
" Bob "
"Bob"
```

Example: trimming custom letters

```
$path[/section/subsection/]
^path.trim[right;/]
```

Will output...

```
/section/subsection
```

base64. Encoding to Base64

```
^string.base64[]
```

Method encodes string to Base64 representation. To decode a string from Base64 to it's original, use **^string:base64[encoded]**

Detailed information on MD5 is available here http://www.ietf.org/rfc/rfc2045.txt and here

http://en.wikipedia.org/wiki/Base64.

Example

```
$original[§ 1973. There are rumours that "Green sleeves" were written by…]
<pre>^original.base64[]</pre>
```

Outputs…

```
pyAxOTczLiBUaGVyZSBhcmUgcnVtb3VycyB0aGF0IJNHcmVlbiBzbGVldmVzlCB3ZXJlIHdyaXR0
ZW4gYnmF
```

js-escape. Encoding similar to escape function in JavaScript

```
^string.js-escape[]
```

Escapes a string. This method do the transformation similar to **escape** function described in ECMA-262. Strings, which were escaped using this method, can be unescaped in browser with JavaScript function **unescape**.

To unescape a string use

```
^string:js-unescape[escaped]
```

Detailed information on ECMA-262 is available here:
http://www.ecma-international.org/publications/standards/Ecma-262.htm *(B.2.1)*

Example

```
$value[abcd `+-=~!@#%&*()_ []{}<>:'",./?абвгд]
<pre>^value.js-escape[]</pre>
```

Outputs…

```
abcd%20%60+-
%3D%7E%21@%23%25%26*%28%29_%20%5B%5D%7B%7D%3C%3E%3A%27%22%2C./%3F%u0430%u0431%u
0432%u0433%u0434
```

Table class

The class is designed for working with tables.

The table is considered defined (**def**) if it isn't empty. The numeric value equals the amount of rows in the table.

Constructors

create. Creating an object based on a specified table

```
^table::create{table_data}
^table::create[nameless]{table_data}
^table::create{table_data}[format options]        [3.2.2]
```

The constructor creates an object of class **table**, using **table_data** defined in the constructor itself.

Table_data—data provided in *tab-delimited* format, that is—the columns are separated by tab symbol, while rows are separated with a new line symbol. At that, the parts of the first row—divided by the tabulation—are regarded as columns' names and thus a named table is created. Blank lines are ignored. If you want to create a table without columns' names (which is actually NOT recommended), you should precede table data with option **nameless**. In this case, the constructor regards the columns of the first row as table data and instead of columns' names their ordinal numbers—starting with zero—will be used.

Only **$.separator** is available in format options yet.

Example

```
$tab[^table::create{name      age
Bob   27
Alex  22}
]
```

A new object of class table—**tab**—will be created, containing two rows with columns' **name** and **age**.

create. Copying existing table

```
^table::create[existing_table]

^table::create[existing_table;options]
```

This constructor creates an object of class **table** by copying data from already existing table. One can also specify a number of options to control copying, "Copying and search options".

Example
Code...

```
$orig[^table::create{name
Jack
Nick
Mary
}]

# sets row with "Nick" as current in $orig
^orig.offset(1)

# copies data starting with current row, taking 10 records at the most
$copy[^table::create[$orig;
    $.offset[cur]
    $.limit(10)
]]

^copy.menu{$copy.name}[, ]
```

...will output:

```
Nick, Mary
```

load. Loading table from a file or HTTP-server

```
^table::load[filename]
^table::load[filename;loading options]
^table::load[nameless;filename]
^table::load[nameless;filename;loading options]
```

The constructor creates an object using the table stored in a file or a document on HTTP-server. The data in the file must be provided in *tab-delimited* format (see also **table::create**).

Filename—name of file with path or document's URL on HTTP-server;

Loading options—for general options read "with HTTP-servers" section, there are additional options, see "Options of file format".

The usage of parameter **nameless** is the same as in constructor **table::create**.

Example of loading table from disk

```
$loaded_table[^table::load[/addresses.cfg]]
```

The code given in example creates an object of class **table**, containing named table stored in file **addresses.cfg** located in the root directory of the website.

Example of loading file from HTTP-server

```
$table[^table::load[nameless;http://www.parser.ru/;
      $.headers[
            $.USER-AGENT[table load example]
      ]
]]
Number of rows: ^table.count[]
<hr />
<pre>$table.0</pre>
```

sql. Querying database

```
^table::sql{SQL-query}
^table::sql{SQL-query}[$.limit(n) $.offset(o) $.bind[variables hash]]
```

The constructor creates an object of class **table** containing table based on the data retrieved from a database. To use this constructor you must have connection with data server established (see operator **connect**).

SQL-query—query sent to a database.

It is possible to use additional parameters with the constructor:
$.limit(n)—retrieve no more than **n** entries;
$.offset(o)—ignore first **o** retrieved entries;
$.bind[hash] – variables to bind, see «Queries with bound variables» ***[3.1.4]***.

Example

```
$sql_table[^table::sql{select * from news}]
```

The code will result in an object containing all data from table **news**.

Important notice: you should always provide exact list of fields you need.
Using "" in queries is strongly NOT recommended, since another developer (or you yourself—in a while) will have no idea which fields are to be retrieved from database. Moreover, by using such a construction one may retrieve unneeded fields (say, those which were added during project's development), which will demand additional resources for retrieving and storing superfluous data.*

Options of file format

When file loaded or saved there can be set column separator and column enclosing characters.

Option	*By default*	*Description*
$.separator[character**]**	tab	Specifies column separator character
$.encloser[character**]**	none	Specifies column encloser character

Example of loading .txt file created by Miscrosoft Excel

Excel can store data into simple tab-delimited text file:
File|Save as... Text (Tab delimited) (`.txt`).
The data is stored in this format:

name	*description*
"New company ""Smith&Co"""	Text

(Values of several columns is quoted and quotation marks in value itself doubled)

To read such a file, one can specify this option:

```
$companies[^table::load[companies.txt;
      $.encloser["]
]]
$companies.name
```

Parser can also work with `.csv` files, just set this option:

```
      $.separator[^;]
```

Copying and search options

While copying records from one table to another, see...

table::create
table.join

and while searching records, see...

table.locate

one can specify a hash of options:

$.offset(number of rows)	omit specified number of rows;
$.offset[cur]	start with current table row;
$.limit(maximum)	maximum rows to be processed;
$.reverse(1/0)	1=in the reverse order.

Retrieving data stored in a column

$table.column_name

...returns data from a specified column in current table row.

Example

```
$tab.name
```

...will return value stored in column **name** of the current table row.

Retrieving data stored in current row as a hash

$table.fields—data stored in the current table row, returned as hash (not available for **nameless** tables).

Returns data stored in the current table row as a hash. The names of the columns then become hash keys, while columns' data—respective values.

It is necessary to use this method if columns' names coincide with names of methods or constructors of class **table**. In this case you cannot retrieve their values directly—Parser will report an error. If it is necessary to work with fields with such names, it is safe to use field **fields** and work with not table but hash.

Example

```
$tab[^table::create{menu      line
yes   first
no    second}
]
$tab_hash[$tab.fields]
$tab_hash.menu
$tab_hash.line
```

As a result, you will get values of fields **menu** and **line** (such names will coincide with methods of class **table**) as keys of hash **tab_hash**.

Methods

save. Saving table to a file

```
^table.save[path]
^table.save[path;options]
^table.save[nameless;path]
^table.save[nameless;path;options]
^table.save[append;path]              [3.3.0]
^table.save[append;path;options]      [3.3.0]
```

Saves a table in a text file in *tab-delimited* format.
Using **nameless** option will save table without columns' names.
With **append** option the table will be saved with columns' names only if file doesn't exist on disk.
One can also specify saving options , see "Options of file format", allowing, for example, to save a file in `.csv` format to be used in programs which understand it (Miscrosoft Excel).

Example

```
^conf.save[/conf/old_conf.txt]
```

Table **$conf** will be saved in text file **old_conf.txt** in **/conf/** directory.

count. Number of rows in table

```
^table.count[]
```

Returns the number of rows in the specified table (**int**).

Example

```
$goods[^table::create{pos     good  price
1     Monitor display   1000
2     System control unit     1500
3     Keyboard    15
}]
Amount: ^goods.count[]
```

The example will output:

```
Amount: 3
```

In expressions, the numeric value of the table equals the amount of its rows:

```
^if($goods > 2){more}
```

menu. Iterating through all table rows

```
^table.menu{code}
^table.menu{code}[separator]
^table.menu{code}{separator}
```

method **menu** executes code for each of the table rows, iterating through all table rows one by one—in the given order.

Separator is string or code to be implemented before every non-empty body, except the first. The **separator** code given in square brackets is processed only once, while that in the curly brackets is processed every time it is inserted.

You can force finish the loop using **break** operator or finish current step and go to next one using **continue** operator. ***[3.2.2]***

Example

```
$goods[^table::create{pos     good        price
1     Monitor display   1000
2     System control unit     1500
3     Keyboard    15
}]
<table border=1>
^goods.menu{
      <tr>
            <td>$goods.pos</td>
            <td>$goods.good</td>
            <td>$goods.price</td>
      </tr>
}
</table>
```

The example outputs the entire content of table **$goods** as HTML-coded table.

append. Appending data to a table

```
^table.append{data}
^table.append[data]      [3.4.0]
```

The method appends data to the end of the table. The **data** must be provided in *tab-delimited* format. The table data must have the same structure as the table to which it is appended.

Example

```
$stuff[^table::create{name    pos
Alexander   boss
Sergey      coder
}]
^stuff.append{Nikolay   designer}
^stuff.save[stuff.txt]
```

The example code will append a new row to the table **$stuff** and save the whole table to the disk. After it, you cannot retrieve the value stored in the third column of the appended row, but it may be retrieved from the saved file `stuff.txt`.

offset. Changing current row offset

```
^table.offset(number)
^table.offset[cur|set](number)
```

Shifts current row specified **number** of times down. If the numeric value of parameter **number** is negative, current row is shifted up. Current row shift is made cyclically—that is, having reached the last row in the table, current row is shifted back up to the first row.

Optional parameters:
cur—shifts the offset with respect to the current row
set—shifts the offset with respect to the first row

Example

```
<table border="1">
^goods.offset(-1)
      <tr>
            <td>$goods.pos</td>
            <td>$goods.good</td>
```

```
            <td>$goods.price</td>
        </tr>
</table>
```

The given example will result in HTML-coded table containing the last row of the table from the previous example (i.e. given in the description of method **menu**).

offset and line. Getting current row offset

Method **offset** with no parameters specified returns current row offset with respect to the beginning of the table.

Example

```
$men[^table::create{name
Jack
Joe
Roger
}]
^men.menu{
    ^men.offset[]—$men.name
}[<br />]
```

The code will return:

```
0—Jack
1—Joe
2—Roger
```

Unlike the computer, human beings tend to count beginning with one, not zero. To make the output of numbered lists more comfortable for human understanding, you may use method line:

```
^table.line[]
```

It allows getting the position number in a more comprehensible manner—when the number of the first row equals one. If **^men.line[]** is used in the above example, the rows enumeration will start with one and end with three.

sort. Sorting table data

```
^table.sort{function_sorting_by_string}
^table.sort{function_sorting_by_string}[sorting_direction]
^table.sort(function_sorting_by_number)
^table.sort(function_sorting_by_number)[sorting_direction]
```

The given method sorts the table according to the specified function.

Sorting_function is the function, whose current value determines the position of the row in the final (sorted) variant of the table. This value may be a string (values are compared in alphabetical order) or a number (values are compared as real numbers).

Sorting_direction determines sorting direction. The parameter may have two values:
desc—descending
asc—ascending
Ascending value is used by default.

Example

```
$men[^table::create{name      age
Sergey      26
Alex  20
Mishka      29
```

```
}]
^men.sort{$men.name}
^men.menu{
    $men.name: $men.age
}[<br />]
```

As the result of this code, the rows of table **$men** will be sorted according to the values given in column **name**:

```
Alex: 20
Mishka: 29
Sergey: 26
```

You may sort the table rows by the values given in column **age** in descending order (**desc**) if you substitute the line of the code which is calling method sort for this one:

```
^men.sort($men.age)[desc]
```

The code will result in:

```
Mishka: 29
Sergey: 26
Alex: 20
```

join. Joining two tables

```
^table1.join[table2]
^table1.join[table2;options]
```

The method joins **table2** data to the end of **table1**. Here, the method will retrieve from **table2** the value placed in the column, whose name coincides with the name of the column in **table1** or a blank line (if such column cannot be found).
One can set several options, controlling the process, see "Copying options".

Example

```
^stuff.join[$just_hired_people]
```

All entries of table **$just_hired_people** will be joined with table **$stuff**.

flip. Transposing a table

```
^table.flip[]
```

The method creates a new table **nameless** with entries resulted from transposing a table. That means, the method turns columns of the given table into rows and rows into columns.

Example

```
$words[^table::create{id      number
Zero  01
One   02
Two   03
Three       04
}]
$fliped[^words.flip[]]
^words.save[words.txt]
```

As the result of the code, the following table will be saved as a file named **flipped.txt**:

0	1	2	3
Zero	One	Two	Three
01	02	03	04

locate. Locating a specified value in a table

```
^table.locate[column_name;value_to_be_located]
^table.locate(logical_expression)
^table.locate[column_name;value_to_be_located;опции]
^table.locate(logical_expression)[опции]
```

The method locates a specified **value** in a specified **column** in the table and returns Boolean value "true/false" depending on whether it found the value or not. In case it locates the specified value, the row where the value is found is set as current. If the value was not located, current row is not shifted.

The second variant of calling method searches first record to conform **logical_expression**.

One can also specify a number of options to control search, see "Copying and search options".

Search is case-sensitive.

Example

```
$stuff[^table::create{name          pos
Ivanov      boss
Petrov      engineer
Lebedev     art-director
}]
^if(^stuff.locate[name;Lebedev]){
      The entry is found in line ^stuff.line[].<br />
      $stuff.name: $stuff.pos<br />
}{
      No such entry
}
```

The code will output:

```
The entry is found in line 3.
Lebedev: art-director
```

select. Selecting entries

```
^table.select(selection_criterion)
```

The method looks through the table row by row, examining each row in respect to the specified **criterion** (a mathematical expression). The rows which satisfy the criterion (returned Boolean value is "true") are collected into the table with the same structure as that of the original table.

Example

```
$men[^table::create{name      age
Stephen     26
Alex  20
Michael     29
}]
$thoseAbove20[^men.select($men.age > 20)]
```

Variable **$thoseAbove20** will contain table made up of rows with **Stephen** and **Michael**.

hash. Transforming a table into hash with specified keys

```
^table.hash[key]
^table.hash[key][options]
^table.hash[key][column_of_values]
^table.hash[key][column_of_values][options]
^table.hash[key][table_with_columns_of_values]
^table.hash[key][table_with_column_of_values][options]
```

Key may be specified as:

- **[string]**—name of column, whose value will be regarded as key;
- **{code}**—code, whose result will be regarded as key;
- **(mathematical expression)**—whose result will be regarded as key.

With default options the method transforms table into hash of the following structure:

```
$hash[
      $.value_of_key[
            $.name_of_column[value_of_column]
            ...
      ]
      ...
]
```

In other words, the method creates hash, where the values from the specified column serve as hash keys. Every key is associated with a hash, where the keys are the names of all table's columns.

If a column of values is specified, every key will be associated with a hash with one key/value pair (the name of the specified column).

Besides, one may specify several columns to serve as keys of hash relevant to the specified column—in this case, as an additional parameter a table must be given with all necessary columns listed.

Options—hash with transformation options.

$.type[hash/string/table] ***[3.2.2]***	**hash**=each hash item contain hash (default); **string**=each hash item contain string. You must specify **one column_of_values**; **table**=each element containing table. Using this option you can't specify **column_of_values** or **table_with_column_of_values.** This made for save memory because of tables in resulting hash just have links to tables' rows which already exist in memory.
$.distinct(0/1)	**0**=identical values in key column are considered error (default); **1**=get identical values from key column.
$.distinct[tables] ***[3.0.8]***	make up **hash** of tables containing rows with key. Deprecated option which do the same as **$.distinct(1)** and **$.type[table]** if they specified together.

Example

We have a list of goods, where each item has a **name** and a unique **id**. We also have a price-list of available goods. Instead of the name of each item, we use relevant ids given in the goods list. This all is stored in two tables, which referred to as "linked". We need to get data in the format "item-price", that is to get data from two tables simultaneously.

Realisation:

```
# this is the table with goods
$product_list[^table::create{id     name
1     bread
2     meat
```

```
3     butter
4     whisky
}]

# this is the table with prices
$price_list[^table::create{id price
1     6.50
2     70.00
3     60.85
}]

#hash of the table with prices by id field
$price_list_hash[^price_list.hash[id]]

#looking through the entries of the table with goods
^product_list.menu{
    $product_price[$price_list_hash.[$product_list.id].price]
#checking if there is a price for the item in our hash
    ^if($product_price){
#printing item's name and price
        $product_list.name—$product_price<br />
    }{
#and this item has no price, i.e. is unavailable
        $product_list.name—unavailable<br />
    }
}
```

The output will be:

```
bread—6.50
meat—70.00
butter—60.85
whisky—unavailable
```

columns. Getting a table's structure

```
^table.columns[]
^table.columns[column name]
```
[3.2.2]

The method creates named table consisting of sole field containing names of the original table's columns. The name of column—"**column**" or the **column name** passed as a parameter.

Example

```
$columns_table[^stuff.columns[]]
```

Void class

This class is designed for working with "void" objects. It does not have any constructors—objects of this class are created automatically, e.g. when you refer to a variable that does not exist.

Methods

int, double, bool

```
^object.int[]
^object.int(default value)
^object.double[]
^object.double(default value)
^object.bool[]
^object.bool(default value)
```

If an object does not exist, these methods return 0 or **default value**. These methods work when you transform undefined objects—e.g. form fields (see class **form**)—to **int/double** or **bool**.

Example

```
^form:number.int[]
```

If field **number** is defined, i.e. has been received, its value will be transformed to class **int**. Otherwise, non-existent value belonging to class **void** will be considered 0 and the code will not stumble.

length. Length of a "string"

```
^object.length[]
```

If an object does not exist, the method will return 0.

Example

```
^if(^form:password.length[]<$MIN_PASSWORD_LENGTH){
    Password length is less than $MIN_PASSWORD_LENGTH
}
```

If form field **password** is defined, i.e. has been received, its length will be calculated and checked. This is a usual way of calling method **^string.length[]**. If the field is not defined, i.e. has not been received, the length of non-existent value belonging to class **void** will be regarded as 0 and the code will not—the length will be successfully checked.

pos. Getting position of substring

```
^object.pos[substring]
^object.pos[substring](offset)        [3.3.0]
```

If an object does not exist, this method returns -1.

Example

```
^if(^form:email.pos[@]>0){
    May be...
}
```

If field **email** is defined, i.e. has been received, this method will work as usual **^string.pos[]**.Otherwise, method will regard non-existent value as belonging to class **void**, and assume it contains no substrings at all. The code will go on and the check will be accomplished.

left, right, mid. Getting substring

```
^string.left(N)
^string.right(N)
^string.mid(P;N)
^string.mid(P)
```

If an object does not exist, these methods return empty substring.

Example

```
^form:email.left(50)
```

If field **email** is defined, i.e. has been received, this method will work as usual **^string.left[]**.Otherwise, method will regard non-existent value as belonging to class **void**, and assume it contains no substrings at all. The code will go on and successfully return empty string.

Static method

sql. SQL-query returning no result

```
^void:sql{SQL-query}
^void:sql{SQL-query}[$.bind[variables hash]] [3.1.4]
```

This method sends SQL-query that returns no result (operations on data management in a database). For this method to work, you must have connection with DB-server established (see operator **connect**).

It is possible to use additional parameter with the constructor:
$.bind[hash] – variables to bind, see «Queries with bound variables» ***[3.1.4]***.

Example

```
^connect[connect string]{
      ^void:sql{create table users(id int,name text,email text)}
}
```

As a result of this code's work, new table `users` will be created. The query will return no result. This example is for MySQL DBMS.

XDoc class

This class is designed for working with tree-structure data, along with **xnode**, It supports reading files in XML format, writing XML (http://www.w3.org/XML) and HTML, and **XSLT** transformation (http://www.w3.org/TR/xslt).

Working with tree is performed in **DOM** model (http://www.w3.org). DOM1 and some opportunities of DOM2 are available.

Class **xdoc** implements DOM-interface Document and is heir to class **xnode**.

Errors in DOM operations DOMException) interface) are converted into exceptions of **xml**-type.

Constructors

create. Creating a document based on specified XML

```
^xdoc::create{XML-code}
^xdoc::create[base_path]{XML-code}
```

This constructor creates object of class **xdoc** based on **XML-code**. One can also specify **base path**.

Example

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
text
</document>}]
$response:body[^document.string[]]
```

create. Creating a new empty document

```
^xdoc::create[tag_name]
^xdoc::create[base_path;tag_name]
```

This constructor creates object of class **xdoc**, containing single tag **tag_name**. One can also specify **base path**.

Example

```
$document[^xdoc::create[document]]
$paraNode[^document.createElement[para]]
$addedNode[^document.documentElement.appendChild[$paraNode]]
$response:body[^document.string[]]
```

create. Creating a document based on specified file

```
^xdoc::create[file]
```

This constructor creates object of class **xdoc** based on **XML-code** in specified file.

Example

```
$file[^file::load[binary;http://server/data.xml;
      $.timeout(10)
]]
$xdoc[^xdoc::create[$file]]
$response:body[^xdoc.string[]]
```

load. Loading XML from disk or HTTP-server or other source

```
^xdoc::load[filename]
```

This constructor loads XML-code from a file or document on HTTP-server and creates a new object of class **xdoc** based on it.
Parser can load XML from arbitrary source, see "Reading XML from arbitrary source".

filename – path and filename or URL of document on HTTP-serve.

Example of loading a file from disk

```
$document[^xdoc::load[article.xml]]
$response:body[^document.string[]]
```

Example of loading an XML document from HTTP-server:

```
$xdoc[^xdoc::load[http://www.cbr.ru/scripts/XML_daily.asp]]
Rate exchange of
    $node[^xdoc.selectSingle[/ValCurs/Valute[CharCode='USD']]]
    "^node.selectString[string(Name)]"
for
    ^xdoc.selectString[string(/ValCurs/@Date)]
is
    ^node.selectString[string(Value)]
<hr />
<pre>^taint[^xdoc.string[]]</pre>
```

parser://method/parameter. Reading XML from arbitrary source

Parser can read XML from arbitrary source.
Everywhere where XML can be read, one may specify the address of the document in this form...
`parser://method/parameter`

Reading a document from address like this is, in fact, reading the result of Parser **^method[/parameter]** call.

Example of keeping XSL templates in database

```
@main[]
...
# at this point $xdoc contains a document we want to transform
^xdoc.transform[parser://xsl_database/main.xsl]
```

```
@xsl_database[name]
^string:sql{select text from xsl where name='$name'}
```

Relative links would be handled exactly same way as if files would be read from disk.
Say, if `parser://xsl_database/main.xsl` template refers to `utils/common.xsl`, the document `parser://xsl_database/utils/common.xsl` would be read, by calling Parser method **`^xsl_database[/utils/common.xsl]`**.

Parameter of creating a new document: Base path

When creating a new document with one of the constructors, one can specify parameter **`base_path`**.

Its purpose is similar to that of attribute...

```
<...
    xmlns:xml="http://www.w3.org/XML/1998/namespace"
    xml:base="base URI" ...
```

...yet, the way of specifying the path is different—it conforms with general approach to paths used in Parser (see "Appendix 1. Paths to files and directories"), This approach is much more comfortable, since you do not have to specify full path including that to web space. By default, path to the currently processed document is used.
Note: character "/" at the end of the path is obligatory.

Example

```
$sheet[^xdoc::create[/xsl/]{<?xml version="1.0" encoding="$request:charset"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:import href="import.xsl"/>
</xsl:stylesheet>
}]
```

File `import.xsl` will be read from directory `/xsl/`.

Methods

DOM

DOM1-interface Document:

```
$Element[^document.createElement[tagName]]
$DocumentFragment[^document.createDocumentFragment[]]
$Text[^document.createTextNode[data]]
$Comment[^document.createComment[data]]
$CDATASection[^document.createCDATASection[data]]
$ProcessingInstruction[^document.createProcessingInstruction[target;data]]
$Attr[^document.createAttribute[name]]
$EntityReference[^document.createEntityReference[name]]
$NodeList[^document.getElementsByTagName[tagname]]
```

DOM2-interface Document:

```
$Node[^document.importNode[importedNode](deep)]
$Element[^document.createElementNS[namespaceURI;qualifiedName]]
$Attr[^document.createAttributeNS[namespaceURI;qualifiedName]]
$NodeList[^document.getElementsByTagNameNS[namespaceURI;localName]]
$Element[^document.getElementById[elementId]]
```

In Parser
- DOM-interfaces Node and Element are implemented in class **xnode**;
- DOM-interface NodeList—is class **hash** with keys 0, 1, ...;
- DOM-type DOMString—is class **string**;
- DOM-type Boolean is Boolean value: 0=FALSE, 1=TRUE.

Detailed specification of DOM1 is available at: http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

Detailed specification of DOM1 is available at: http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

string. Converting document into string

```
^document.string[]
^document.string[Document_to_text_conversion_parameters]
```

This method converts document into text. One can also specify **conversion_parameters**.
By default, method will create XML-representation of the document with header **<?xml ... ?>** (This method converts document into text. One can also specify conversion_parameters).

The result goes to visitor **as-is**. ***[3.1.4]***

Example

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
string1<br />
string2<br />
</document>}]
^document.string[
    $.method[html]
]
```

save. Saving document to file

```
^document.save[path]
^document.save[path;Document_to_text_conversion_parameters]
```

This method saves document to a text file. One can also specify **conversion_parameters**.
By default, method will create XML-representation of the document with header **<?xml ... ?>** (one can also disable this header by specifying relevant parameter).

Path—path to file.

Example

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
string1<br />
string2<br />
</document>}]
^document.save[saved.xml]
```

file. Converting document into object of class file

```
^document.file[Document_to_text_conversion_parameters]
```

This method converts object of class **xdoc** into object of class One can also specify **conversion_parameters**. By default, method will create XML-representation of the document with header **<?xml ... ?>**(one can also disable this header by specifying relevant parameter).

Example

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
string1<br />
string2<br />
</document>}]
$response:body[^document.file[]]
```

transform. XSL transformation

```
^document.transform[template]
^document.transform[template][XSLT-parameters]
```

This method applies XSL transformation to the **document** using specified **template**. One can also specify **XSLT-parameters**.

Template—either **path_to_template_file** or **xdoc** document.
Parser can load XML from arbitrary source, see "Reading XML from arbitrary source".

XSLT-parameters — hash of strings, which can be accessed in templates via **<xsl:param ... />**.

*Note: Parser (as Apache module or IIS) module) caches the result of **template_file** compilation into internal form. Recompilation is not performed. Instead, already compiled template is taken from cache. CGI-version caches the template, too, but for a single request only. The template is recompiled if modification date of any of template files has changed.*

Example (see also "Lesson 6. Working with XML")

```
# source xdoc document
$sourceDoc[^xdoc::load[article.xml]]

# transformation of xdoc document using template
article.xsl$transformedDoc[^sourceDoc.transform[article.xsl]]
```

```
# outputting result as HTML
^transformedDoc.string[
    $.method[html]
]
```

If template is not loaded from disk but created dynamically, it is important to determine where **<xsl:import href="some.xsl"/>** should be taken from, since, in this case, document's base path does not exist and its directory, therefore, cannot be determined. That means, you will need to specify base path in standard "xml:base" attribute.

Document-to-text conversion parameters

Certain methods accept **Document_to_text_conversion_parameters** hash.

These parameters are identical to attributes of **<xsl:output ... />**.
except **doctype-public** and **doctype-system**, which cannot be specified this way.
So far, **cdata-section-elements** are also excluded.

By default text rendered in **$request:charset**, but in XML-header or in **meta** element for HTML-method Parser specifies **$response:charset**. This behaviour can be altered by specifying the charset in **<xsl:output ... />** or corresponding conversion parameter.

While creating object of class **file** one can also specify parameter **media-type**: when new response body **body** is generated, response header **content-type** will be assigned the value of this parameter.

Example

```
# output document as HTML without indents
^document.string[
    $.method[html]
    $.indent[no]
]
```

Outputting XHTML

If you need an XHTML output, you must specify these attributes for **<xsl:stylesheet ... />** element:

```
<xsl:stylesheet version="1.0"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>
```

Note **xmlns** without prefix specification, this should be done for all created in template prefixless elements to get to `xhtml` name space. It is necessary to specify **xmlns** without prefix in each `.xsl` file, because this parameter does not influence included files.

Also these attributes for **<xsl:output ... />** must be set:

```
<xsl:output
      doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
      doctype-system="DTD/xhtml1-strict.dtd"
      />
```

Note: do not specify method attribute. XHTML is an xml with a certain difference in rendering, it switches on when any of these doctypes are used:

```
      -//W3C//DTD XHTML 1.0 Strict//EN
      -//W3C//DTD XHTML 1.0 Frameset//EN
      -//W3C//DTD XHTML 1.0 Transitional//EN
```

Fields

DOM

DOM-1 Document-interface:

```
$DocumentType[$document.doctype]
$Element[$document.documentElement]
```

In Parser, DOM interfaces Node and Element and their derivatives are implemented in class **xnode**.

Detailed specification of DOM1 is available at: http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

search-namespaces. Name spaces hash to search in

```
$document.search-namespaces
```

In order to use prefixes of name spaces in **xnode.select*** methods one must define these prefixes in this hash.

Here
- keys—name space prefixes,
- values—their URIs.

Adding several prefixes

```
$xdoc[^xdoc::create{<?xml version="1.0"?>
<document xmlns:s="urn:special">
      <s:code xmlns:o="urn:other" o:attr="123">let's play hide-and-
seek</s:code>
</document>
}]
^xdoc.search-namespaces.add[
   $.s[urn:special]
   $.o[urn:other]
]
^xdoc.selectString[string(//s:code[@o:attr=123])]
```

Adding one prefix

```
$xdoc.search-namespaces.s[urn:special]
```

XNode class

This class is designed for working with tree-structured data, along with **xdoc**, It supports **XPath** (http://www.w3.org/TR/xpath) queries.

Class **xdoc** implements DOM interfaces Node and Element and their derivatives.
Class is not created directly. Instead relevant methods of class **xdoc** are used.

Instead of DOM-interface NamedNodeMap Parser uses class **hash**.

Methods

DOM1

DOM1-interface Node:

```
$Node[^node.insertBefore[$newChild;$refChild]]
$Node[^node.replaceChild[$newChild;$oldChild]]
$Node[^node.removeChild[$oldChild]]
$Node[^node.appendChild[$newChild]]
^if(^node.hasChildNodes[]){…}
$Node[^node.cloneNode(deep)]
```

DOM1-interface Element:

```
^node.getAttribute[name]
^node.setAttribute[name;value]
^node.removeAttribute[name]
$Attr[^node.getAttributeNode[name]]
$Attr[^node.setAttributeNode[$newAttr]]
$Attr[^node.removeAttributeNode[$oldAttr]]
$NodeList[^node.getElementsByTagName[name]]
^node.normalize[]
```

DOM2-interface Element:

```
$string[^node.getAttributeNS[namespaceURI;localName]]
^node.setAttributeNS[namespaceURI;qualifiedName;value]
^node.removeAttributeNS[namespaceURI;localName]
$Attr[^node.getAttributeNodeNS[namespaceURI;localName]]
$Attr[^node.setAttributeNodeNS[$newAttr]]
$NodeList[^node.getElementsByTagNameNS[namespaceURI;localName]]
^if(^node.hasAttribute[name]){…}
^if(^node.hasAttributeNS[namespaceURI;localName]){…}
^if(^node.hasAttributes[]){…}    [3.2.2]
```

In Parser
- DOM-interface NodeList is class **hash** with keys 0, 1, ...;
- DOM-type DOMString is class **string**;
- DOM-type Boolean is Boolean value: 0=FALSE, 1=TRUE.

Detailed specification of DOM1 is available at:
http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

Detailed specification of DOM1 is available at:
http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

select. XPath search for node

```
$NodeList[^node.select[XPath-query]]
```

This method returns list of nodes found in the scope of specified **node** and satisfying specified **XPath-query**. If no node was found, empty list will be returned.

Before using prefixes of name spaces in the **query** one must define them, see **$xdoc.search-namespaces**.

Example

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
    <t/><t/>
```

```
</document>}]
# result is list of two elements "t"
$list[^d.select[/document/t]]
# iterating through found lists:
# this code will work
# even if query returns no nodes at all
^for[i](0;$list-1){
    $node[$list.$i]
    Name: $node.nodeName<br />
    Type: $node.nodeType<br />
}
```

In Parser, DOM interface NodeList is class **hash** with keys 0, 1, ...;

Detailed specification of XPath is available at: http://www.w3.org/TR/xpath

selectSingle. XPath search for single node

^node.selectSingle[XPath-query]

This method returns node found in the scope of specified node and satisfying specified **XPath-query**. If no node was found, **void** is returned. If more than one node is returned exception is thrown.

Before using prefixes of name spaces in the **query** one must define them, see **$xdoc.search-namespaces**.

Example

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<t attr="hello" n="123"/>}]
# result=1 element "t"
$element[^d.selectSingle[t]]
# result=2 (number of attributes <t>)
Number of attributes: ^element.attributes._count[]<br />
```

Detailed specification of XPath is available at: http://www.w3.org/TR/xpath

selectString. XPath search for a string

^node.selectNumber[XPath-query]

This method returns result of **XPath-query** in the scope of specified **node**, if it is a **number**. If resulted value is not a **number**, method returns **exception** of type **parser.runtime**.

Before using prefixes of name spaces in the **query** one must define them, see **$xdoc.search-namespaces**.

Example

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<t attr="hello" n="123"/>}]
# result=hello
^d.selectString[string(t/@attr)]
```

Detailed specification of XPath is available at: http://www.w3.org/TR/xpath

selectNumber. XPath search for a number

^node.selectNumber[XPath-query]

This method returns result of **XPath-query** in the scope of specified **node**, if it is a number. resulted value is not a **number**, method returns exception of type **parser.runtime**.

Before using prefixes of name spaces in the **query** one must define them, see **$xdoc.search-namespaces**.

Example

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<t attr="hello" n="123"/>}]
#result=124
^d.selectNumber[number(/t/@n)+1]<br />
#result=4
^d.selectNumber[2*2]<br />
```

Detailed specification of XPath is available at: http://www.w3.org/TR/xpath

selectBool. XPath search for a Boolean value

^node.selectBool[XPath-query]

This method returns result of **XPath-query** in the scope of specified **node**, if it is a Boolean value. If resulted value is not a Boolean, method returns exception of type **parser.runtime**.

Before using prefixes of name spaces in the **query** one must define them, see **$xdoc.search-namespaces**.

Example

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<t attr="hello" n="123"/>}]
^if(^d.selectBool[/t/@n > 10]){
    /t/@n greater than 10
}{
    not greater
}
```

Detailed specification of XPath is available at: http://www.w3.org/TR/xpath

Fields

DOM

DOM1-interface Node:

```
$node.nodeName
$node.nodeValue
$node.nodeValue[new value]
^if($node.nodeType == $xnode:ELEMENT_NODE){…}
$Node[$node.parentNode]
$NodeList[$node.childNodes]
$Node[$node.firstChild]
$Node[$node.lastChild]
$Node[$node.previousSibling]
$Node[$node.nextSibling]
$NodeList[$node_of_type_ELEMENT.attributes]
$Document[$node.ownerDocument]
```

DOM2-interface Node:

```
$node.prefix
$node.namespaceURI
```

DOM1-interface Element:

```
$node_of_type__ELEMENT.tagName
```

DOM1-interface Attr:

```
$node_of_type__ATTRIBUTE.name
^if($node_of_type__ATTRIBUTE.specified){…}
$node_of_type__ATTRIBUTE.value
```

DOM1-interface ProcessingInstruction:

```
$node_of_type__PROCESSING_INSTRUCTION.target
$node_of_type__PROCESSING_INSTRUCTION.data
```

DOM1-interface DocumentType:

```
$node_of_type__DOCUMENT_TYPE.name
$node_of_type__DOCUMENT_TYPE.entities
$node_of_type__DOCUMENT_TYPE.notations
```

DOM1-interface Notation:

```
$node_of_type__NOTATION.publicId
$node_of_type__NOTATION.systemId
```

In Parser

- DOM interface **NodeList** is class **hash** with keys 0, 1, ...;
- DOM-type DOMString is class **string**;
- DOM-type Boolean is Boolean value: 0=FALSE, 1=TRUE.

Detailed specification of DOM1 is available at:
http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

Detailed specification of DOM1 is available at:
http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

Constants

DOM. nodeType

DOM-elements may be of different types, element's type is stored in integer field of **nodeType**.
Class **xdoc** has the following constants, useful to check values of this field:

```
$xdoc:ELEMENT_NODE                  = 1
$xdoc:ATTRIBUTE_NODE                = 2
$xdoc:TEXT_NODE                     = 3
$xdoc:CDATA_SECTION_NODE            = 4
$xdoc:ENTITY_REFERENCE_NODE         = 5
$xdoc:ENTITY_NODE                   = 6
$xdoc:PROCESSING_INSTRUCTION_NODE   = 7
$xdoc:COMMENT_NODE                  = 8
$xdoc:DOCUMENT_NODE                 = 9
$xdoc:DOCUMENT_TYPE_NODE            = 10
$xdoc:DOCUMENT_FRAGMENT_NODE        = 11
$xdoc:NOTATION_NODE                 = 12
```

Example

```
^if($node.nodeType == $xnode:ELEMENT_NODE){
    <$node.tagName />
}
```

Appendix 1. Paths to files and directories, working with HTTP–servers

To access files and directories in Parser, one may use absolute or relative paths.

Absolute path is started with slash. In this case, the file is searched for from web-space root. If a relative path is used, the file will be searched for from directory where requested document is located.

Example of absolute path:
`/news/archive/20020127/sport.html`

Example of relative path:
relative to directory `/news/archive`...
`20020127/sport.html`

While a file is saved, needed directories are created automatically.

Note: the root of web-space, passed by web-server, can be cahnged: see "Root of web-space"

*Note: Parser transforms paths to **file-spec** (see "External and internal data").*

Methods...
- **file::load**
- **table::load**

...can work with external HTTP-servers, provided the name of document to be loaded starts with prefix `http://`

While using these methods, one can also specify extra options to control download behavior. These options are hash, with such keys as:

Option	*Default*	*Value*
$.charset[charset**]**	taken from HTTP response header	Charset used in documents on rem This charset is used to transcode re and response body. This option also allowed while load files. ***[3.2.2]***
$.timeout(seconds**)**	2 seconds	HTTP server's response timeout in s download operation is not finished period, exception will be thrown.
$.method[HTTP-METHOD**]**	GET	The name of HTTP-method should in uppercase only. It's possible to specify it in lowerca ***[3.3.1]***
$.enctype[CONTENT-TYPE**]**	application/x-www-form-urlencoded	Possible values are: application/x-www-form-urlencode or multipart/form-data. Last one with method `POST` should you need to send files to external ***[3.3.1]***

$.form[
$.field[string**]**
$.field[file**]**
$.field[**$table]**
…
]
	none	Request parameters. For `GET`-requ
		be passed in`?query_string`. For
		other **method**, parameters will be
		`Content-type: application`
		`form-urlencoded`
		Parameter value can be string, tabl
		column or file ***[3.3.1]***.
		
		It is preferable to pass parameters
		$.forms, *and not pass it in*`?para`
		hand.
		
		It is allowed to pass parameters in
		simultaneously. ***[3.1.5]***

$.body[string**]**	none	Text body of the query. (do not us **METHOD[**GET**]** when you use **body**
$.cookies[** **$.name[value**]** … **]**	none	Hash with list of cookies to be pass server. ***[3.2.3]***
$.headers[** **$.HTTP-HEADER[value**]** … **]**	`$.User-Agent[parser3]`	Hash with additional HTTP-headers to HTTP-server HTTP-header's value may be a date hash with obligatory key **value**. Date may be used as either field va attribute value. In this case, it will b standard formatting.
$.any-status(true)	false/0	Boolean: is response status not equ allowed? If Boolean is FALSE, and r is not equal to 200, system excepti **http.status** will be thrown.
$.omit-post-charset(true)	false/0	Don't add charset info to HTTP-hea type for outgoing POST request. ***[3***
$.user[user**]**	none	These are request parameters to se
$.password[password**]**	none	which uses standard HTTP-authori

For **^file::load[**…**]** one can also specify additional loading options. These options are hash with such keys

as:

Option	*Default*	*Value*
$.offset(offset**)**	0	While loading data, offset is specified in number of bytes.
$.limit(limit**)**	-1	Load no more than specified number of bytes

Variable CLASS_PATH

One may assign variable **CLASS_PATH** in Configuration file. This variable contains path(s) to classes' directory (directories). If path to a module is relative, the module will be search for in **CLASS_PATH** (if **CLASS_PATH** is a table, rows with paths will be iterated through from bottom to top).

Example of table **CLASS_PATH**:

```
$CLASS_PATH[^table::create{path
/classes/common
/classes/specific
}]
```

In this case, relative path `my/class.p` will be searched for as:

```
/classes/specific/my/class.p
/classes/common/my/class.p
```

Appendix 2. Format strings

Format string determines the format in which a number will be represented. It has the following general structure:
%Length.PrecisionType

Type determines the way of converting number into string.

The following types are available:

d —decimal number with sign

u —decimal number without sign

o —octal integer without sign

x —hexadecimal integer without sign; to output numbers greater than 9 one should use letters a, b, c, d, e, and f

X —hexadecimal integer without sign; to output numbers greater than 9 one should use letters A, B, C, D, E, and F

f —real number

Precision is how precisely fractional part is to be represented, i.e. number of digits after period. If actual number contains more numbers than specified in **Precision**, the value will be rounded. Generally, **Precision** is specified when format type **f** is used. It is not recommended to specify **Precision** in cases of all other types. If **Precision** is not specified, **f** uses default value of **6**. If **Precision** is **0**, number will be output with no fractional part at all.

Length is number of signs to be allotted for value. If actual number contains fewer symbols than specified in **Length** (e.g. Length equals **10** and actual number's value is **123**), space characters will be used to substitute

lacking digits. If it is desirable to use zeros rather than space characters in such cases, one should start **Length** value with **0**, that is, specify value NOT as **10**, but as **010**. In case **Length** is not specified, actual number will use as much (or as few) digits as it requires.

Appendix 3. Format of connect string used by operator connect

Connect string (except that used by ODBC) is processed by Parser's database driver.

For MySQL

```
mysql://user:password@host[:port]|[/unix/socket]/database?
      charset=value& [value must be conversion for MySQL 3.x/4.0 or character set name for
MySQL 4.1+]
      ClientCharset=charset&
      timeout=3&
      compress=0&
      named_pipe=1&
      autocommit=1&
      multi_statements=0 [3.3.0]
```

Optional parameters are:

`Port` is number of port used by database server. One can use:
`user:password@hostname:port_number/database`.

One can also replace `hostname` and `port_number` with path to UNIX socket in square brackets (UNIX socket is a magical set of characters (path), which your administrator may tell you, provided you yourself are not the administrator in the flesh. This socket may be used to communicate with server):

```
user:password@[/unix/socket]/database
```

`charset`—right after connection executes "`SET CHARACTER SET value`";
`ClientCharset`—specifies the charset, in which Parser must communicate with SQL server. Conversion will be done by driver;
`timeout`—specifies value of parameter **Connect timeout** in seconds;
`compress`—mode of compressing traffic between server and client;
`named_pipe`—use named pipes to connect to MySQL-server, working under Windows NT;
`autocommit`—if set to 0 after connection executes "`SET AUTOCOMMIT=0`";
`multi_statements`—if set to 1, the single SQL query can contains more then one SQL statements separated by ";" character (*character ";" must be escaped by character "^"*).

Example: transcoding by SQL server (it works with many character sets but requires MySQL version 4.1 or higher)

MySQL server version 4.1 or higher can transcode data in different ways itself so it is recommended to use these server abilities using `charset` option and don't use `ClientCharset` option at all. With MySQL server version 4.1 or higher you can even store data in different tables using different character sets but we recommend to store it in `UTF-8`.

Assume, data in your database is stored in `UTF-8`, while pages are in `windows-1257`, connect string should look like this:
`mysql://user:password@host/database?charset=cp1257`

In this case right after connection to the server Parser will executes command **"SET CHARACTER SET cp1257"** and server will transcode received data from `cp1257` to character set used for storing data in requested database/table/column and transcode it back while send response.

Note: in this case you should specify character set used for storing pages.
Note: this option executes the MySQL command so you must use MySQL server character set names which

are not equal to Parser character set names, defined in configuration file.

Example: DB in koi8-r, pages are in windows-1251, transcoding by SQL server (MySQL 3.x and 4.0)

MySQL server version 3.x and 4.0 can't transcode data in arbitrary ways but can transcode it in most common for Russian language case when data stored in `koi8-r`, while pages are in `windows-1251`.

In this case you can also transcode data with driver transcode functions using `ClientCharset` option (see below), however it's much better to do it using `charset` option with `cp1251_koi8` value:
`mysql://user:password@host/database?charset=cp1251_koi8`

In this case right after connection to the server Parser will executes command **`"SET CHARACTER SET cp1251_koi8".`**
MySQL server version 3.x and 4.0 understands this option in different way, it will transcode receiving data from `cp1251` to `koi8-r` and transcode sending data from `koi8-r` to `cp1251`.

Note: for MySQL server version 3.x and 4.0 value defines transcode directions, herewith server is not supported other values, for example koi8_cp1251.

Example: DB in windows-1251, pages are in koi8-r, transcoding by Parser driver (for any version of MySQL server)

In some cases it's unable to use MySQL server transcode functions. In this case you can use driver transcode functions with `ClientCharset` option.
Assume, data in your database is stored in `windows-1251`, while pages are in `koi8-r`, connect string should look like this:
`mysql://user:password@host/database?ClientCharset=windows-1251`

For Parser code all received data will be automatically converted from `windows-1251` to **`$request:charset`** (`koi8-r` in this example).

Note: in this case you should specify character set used for storing data in database.
Note: in this option you must use Parser character set names, defined in configuration file.

For SQLite

```
sqlite://path-to-DB-file?
      autocommit=1& [3.3.0]
      multi_statements=0& [3.3.0]
```

Path to file with database specified from document_root.
As path to file with database the driver also accepts special values **`:memory:`** and **`:temporary:`**. First one means that for this session will be created temporary database in memory. Second one—for this session will be created temporary database on disk (you don't need to remove database file manually).

`autocommit`—by default SQLite commits all queries automatically. If this option sets to 0, Parser executes `BEGIN` statement at the beginning and `COMMIT/ROLLBACK` at the end of operator **`connect`**, so all statements in one connect operator will be executed in single transaction;
`multi_statements`—if set to 1, the single SQL query can contains more then one SQL statements separated by ";" character (*character ";" must be escaped by character "^"*).

Examples:

Assume, file **`my.db`** with database located in **`data`** directory which located near your document root directory.

Connect string should look like this:
`sqlite://../data/my.db`

Assume, you need temporary table in memory without autocommit (one connect–one transaction). Connect string should look like this:
`sqlite://:memory:?autocommit=0`

For ODBC

```
odbc://connection_string_see_ODBC_documentation?
      ClientCharset=charset&
      autocommit=1& [3.3.0]
      SQL=MSSQL|Pervasive|FireBird [3.3.0]
```

`ClientCharset`—specifies the charset, in which Parser must communicate with SQL server. Conversion will be done by driver;
`autocommit`—by default Parser executes `COMMIT` after each sucessfull query. If option `autocommit=0` was specified this behaviour will changed and all queries inside one **`connect`** operator will be executed in single transaction;
`SQL`—if specified Parser will modify queries with `limit/offset` and add server specific features. For now driver accepts only next values: `MSSQL`, `Pervasive` и `FireBird`. For `MSSQL` and `Pervasive` it will add to query "`TOP (limit+offset)`", for `FireBird` — "`FIRST (limit) SKIP (offset)`".

We recommend this website with huge collection of connection strings to numerous databases: www.connectionstrings.com.

Note: MS-SQL server converts dates and numbers according to language setting, which is absolutely inconvenient in programmatic processing. We do recommend to switch language setting to us_english, which will enable dates in ANSI SQL92 standard notation in numbers with decimal separator '.':
`^void:sql{SET LANGUAGE us_english}`

Examples

MS-SQL:
`odbc://DRIVER={SQL Server}^;SERVER=server^;DATABASE=db^;UID=user^;PWD=password`

Microsoft Access (`.mdb` file):
`odbc://Driver={Microsoft Access Driver (*.mdb)}^;Dbq=C:\full\path\to\file.mdb`

Link to **system** data source configured in Start|Settings|Control Panel|Data sources(ODBC).
`odbc://DSN=dsn^;UID=user^;PWD=password`

Note: Parser requires character ";" in connect string to be escaped by character "^".

Example

Assume, your data is in MS-SQL database in windows-1257 charset, connect string should look like this:

```
odbc://DRIVER={SQL
Server}^;SERVER=server^;UID=user^;PWD=password?ClientCharset=windows-
1257&SQL=MSSQL
```

For PostgreSQL

```
pgsql://user:password@host[:port]|[local]/database?
      charset=value& [value is pgsql charset name]
      ClientCharset=charset&
      autocommit=1& [3.3.0]
      datestyle=value [valid values are ISO,SQL,Postgres,European,US,German.
ISO by default]
```

Optional parameters:
`port`—port number.

One can also specify:
`user:password@host:port/database`,

or:
`user:password@local/database`

In latter case, Parser will connect to server established at local computer.

`charset`—right after connection executes "`SET CLIENT ENCODING=value`";
`ClientCharset`—specifies the charset, in which Parser must communicate with SQL server. Conversion will be done by driver;
`autocommit`—by default Parser executes `COMMIT` after each sucessfull query. If option `autocommit=0` was specified this behaviour will changed and all queries inside one **`connect`** operator will be executed in single transaction;
`datestyle`—if this parameter is specified, after connection the driver will executes "`SET DATESTYLE=value`"

Example
Assume, data in your database is stored in windows-1257, connect string should look like this:
`mysql://user:password@host/database?ClientCharset=windows-1257`

For Oracle

```
oracle://user:password@service?
      ClientCharset=charset&
      LowerCaseColumnNames=0&
      NLS_LANG=RUSSIAN_AMERICA.CL8MSWIN1251&
      NLS_DATE_FORMAT=YYYY-MM-DD HH24:MI:SS&
      NLS_LANGUAGE=language-dependent conventions&
      NLS_TERRITORY=territory-dependent conventions&
      NLS_DATE_LANGUAGE=language for day and month names&
      NLS_NUMERIC_CHARACTERS=decimal character and group separator&
      NLS_CURRENCY=local currency symbol&
      NLS_ISO_CURRENCY=ISO currency symbol&
      NLS_SORT=sort sequence&
      ORA_ENCRYPT_LOGIN=TRUE
```

`ClientCharset`—specifies the charset, in which Parser must communicate with SQL server. Conversion will be done by driver.

If you do not quote columns' names in **`select`** query, oracle will convert them to UPPERCASE. By default, Parser converts them to lowercase. By specifying `LowerCaseColumnNames=0` one can disable this lowercase conversion.

While execution queries with `limit/offset` the driver modifies statements for cutting off not redundant data using SQL server instructions. But if any problems occurs this behaviour can be switched off with option `DisableQueryModification=1`.

Information on other parameters can be found in Oracle documentation.

Example
Assume, data in your database is stored in windows-1257, connect string should look like this:
`oracle://user:password@service?ClientCharset=windows-1257&NLS_LANG=RUSSIAN_AMERICA.CL8MSWIN1251&NLS_DATE_FORMAT=YYYY-MM-DD HH24:MI:SS`

ClientCharset. Connect parameter—charset of communication with SQL server

Parameter `ClientCharset` specifies charset, in which Parser should communicate with SQL server. If parameter is not defined, Parser communicates with SQL server in **`$request:charset`**.

List of charsets is defined in Configuration file.

Appendix 4. Perl Compatible Regular Expressions

Detailed information on PCRE (Perl Compatible Regular Expressions) can be found in Perl documentation (see http://perldoc.perl.org/perlre.html), in documentation on PCRE 2.09 used by Parser (see http://www.pcre.org/man.txt), as well as in many other sources which also contain many practical examples. Most detailed information on regular expressions is given in *Regular Expressions* by J. Friddle, O'Reilly (ISBN 1-56592-257-3).

A draft description given here is only a short reference.

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern "**The quick brown fox**" matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of meta-characters, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of meta-characters:
1. Those that are recognized anywhere in the pattern except within square brackets;
2. Those that are recognized in square brackets.

Outside square brackets, the meta-characters are as follows:

****	general escape character with several uses, more detailed description is given later
^	assert start of subject (or line, in multiline mode)
$	assert end of subject (or line, in multiline mode)
.	character class containing all characters; match any character except newline
[...]	character class definition. Matches any of bracketed characters
|	meta-character "OR": allows joining several patterns into one set of alternative matches
(...)	delimit subpattern within general match pattern
?	match 1 non-alphanumeric character
*****	match 0 or more of any characters, specified on the left
+	match 1 or more of any characters, specified on the left
{min, max}	minimum/maximum quantifier: require minimum occurrences, allow maximum occurrences.

Part of a pattern that is in square brackets is called a "character class". In a character class the only meta-

characters are:

****	general escape character
^	negate the class, but only if the first character of class definition, any characters but those in class will match
-	indicates character range
[...]	terminates the character class

Backslash usage ("\")

The backslash character has several uses. Firstly, if it is followed by a non-alphameric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes. For example, if you want to match a "*" character, you write "*" in the pattern. This applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphameric with "\" to specify that it stands for itself. In particular, if you want to match a backslash, you write "\\".

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. It is usually easier to use one of the following escape sequences than the binary character it represents:

\a	alarm, that is, the BEL character
\cx	"control-x", where x is any character
\e	escape, the ASCII character
\f	formfeed
\n	newline
\r	carriage return
\t	tab
\xhh	character with hex code hh
\ddd	character with octal code ddd

The third use of backslash is for specifying generic character types:

\d	any decimal digit [0-9]
\s	any white space character
\w	any "word" character
\D \S \W	NOT **\d \s \w**

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. These assertions may not appear in character classes (but note that "**\b**" has a different meaning, namely the backspace character, inside a character class).

\b	word boundary
\B	not a word boundary
\A	start of subject (independent of multiline mode)
\Z	end of subject or newline at end (independent of multiline mode)
\z	end of subject (independent of multiline mode)

Appendix 5. How to name variables, methods, and classes correctly

A name should be clear at least to you, and ideally—to anyone else reading your code. The name may be in any language. The only principle you should stick to is uniformity. We however recommend that you use English language (what if you become world-famous one day?). Words in names had better be in singular. Whenever a need occurs, use compound names like "**column_color**": it is always easy to understand what such a name

implies.

Parser is case-sensitive!

$Parser and **$parser** are different variables!

There are certain characters which should not be used in names. In Parser, name ends before:
space character
tab character
newline character
;] }) " < > # + * / % & | = ! ' , ?
character "-" in expressions.

Code...

```
$var[value_of_variable]
$var>text
```

...outputs...

```
value_of_variable>text
```

..., i.e. Parser regards character ">" as end of name of variable **$var** and outputs its value. That is why the characters listed above had better be avoided in names.

Whenever one needs any of characters not listed above to immediately (i.e. with nothing in between) follow a variable's value, one should use syntax:
${var}.text

In this case, the output will be:
value_of_variable.text

One must NOT use characters ".", ":", "^" in names, since these will be regarded as part of Parser's code, which will inevitably cause errors during code processing.

All other symbols are allowed, theoretically. We however recommend that you use no special characters at all when giving names, except in case you really have to (that is, practically, NEVER). The only character we recommended to use is underscore, which is not reserved by Parser and whose meaning is clear enough.

Appendix 6. How to fight errors and read someone else's code

To begin with, study exception message carefully. It contains name of file and number of line where error cropped up. You should be very attentive when coding and turn to reference from time to time. You should also always remember that Parser operates in object model: never forget what class of objects you use. Certain methods return an object of different class!

For example, certain methods of class **date** return object of class **table**. If you attempt to apply methods of class **date** to such an object, it will cause error. You can NOT apply method of one class to an object of different class. This stage, however, will soon be over. Errors of another type are those, which lie in the code's logic. This kind of problem is not that easy to solve and demands more patience. We insist that you give correct names to variables, methods, and classes and comment your code.

If you still are not able to realize what the mistake is—turn to the reference. "If all else fails, try reading the instructions..." The last stage in trying to fight an error is when you are on the verge of madness: you read your prayers or cast spells, but your code does not work anyway. In this case, you should turn to those who know Parser a little better than you: post your question to forum dedicated to Parser, and they will try to help you. You are not alone! Good luck!

Appendix 7. SQL queries with bound variables

Parser's Oracle SQL driver can work with bound variables. `IN`, `OUT` and `IN/OUT` variables are supported, they are bound to hash you pass to query.

There are known problems with `CALL` and `EXECUTE` constructs in Oracle versions, we recommend using PL/SQL wrapper (`begin …; end;`), do not forget to escape «;» character.

Note: values of **`void`** *type correspond to NULL. In second example below* **`days`** *is initially* `NULL`.

Example of using IN variables

```
#procedure ban_user(user_id in number, days in number)

^void:sql{begin ban_user(:user_id, :days)^; end^;}[
    $.bind[
        $.user_id(7319)
        $.days(10)
    ]
]
```

Example of using IN and OUT variables

```
#procedure read_user_ban_days(user_id in number, days out number)

$variables[
    $.user_id(7319)
#we still must pass something in, though current value will be discarded
    $.days[]
]

^void:sql{begin read_user_ban_days(:user_id, :days)^; end^;}[
    $.bind[$variables]
]

User is banned for $variables.days days!
```

Installing and configuring Parser

Parser is available as:
- CGI-script (and interpreter);
- module of web-server Apache, version 1.3;
- ISAPI extension of Microsoft Internet Information Server, version 4.0 or higher.

One can also install drivers for various SQL-servers (currently supported: MySQL, PgSQL, Oracle, ODBC).

Description of directories and files:
`parser3[.exe]` - CGI-script (and interpreter);
`ApacheModuleParser3.dll` - module of web-server Apache, version 1.3;
`parser3isapi.dll` - ISAPI extension of Microsoft Internet Information Server, version 4.0 or higher.

`auto.p.dist` - example of Configuration file

`parser3.charsets/` – - directory with charset tables:

`koi8-r.cfg` – Cyrillic [KOI8-R]
`windows-1250.cfg` – Central european[windows-1250]
`windows-1251.cfg` – Cyrillic[windows-1251]
`windows-1257.cfg` – Baltic[windows-1257]

As long as Parser is open-source project, you can compile it on your own (see "Compiling Parser from source code") and create your own SQL-driver.
Already compiled versions of Parser and SQL-drivers for certain platforms are available at http://www.parser.ru/en/download/.
Note: for security reasons, these versions were compiled in such a way that they can read and execute only those files which belong to the user or user group under the name of which Parser itself works.

How are configuration files linked?

For CGI-script (`parser3[.exe]`):
configuration file is read from file specified in environment variable `CGI_PARSER_CONFIG`.
If this variable is not defined, Parser will search for the file in directory with CGI-script itself.

For Apache module, path to configuration file is specified by directive:
```
#can be in .htaccess
ParserConfig full_path
```

For ISAPI extension (`parser3isapi.dll`):
configuration file will be searched for in the directory with `.dll` file itself.

Configuration file

Exemplary of file is included in distribution package (see `auto.p.dist`).

This is the cornerstone of class **MAIN**. It may contain configuration method, which will be executed first, before method **auto**, to set vital system parameters.

After configuration method is executed, output charset and code charset may be specified (default charset in both cases is **UTF-8**).

Recommended code:
```
@auto[]
#source/client charsets
$request:charset[windows-1251]
$response:charset[windows-1251]
$response:content-type[
      $.value[text/html]
      $.charset[$response:charset]
]
```

Note: if you want methods **upper** *and* **lower** *(class* **string***) to work with languages other than English correctly, you will need to correctly specify* **$request:charset***.*

It is also recommended that you specify path to classes used at your site here:
```
$CLASS_PATH[/../classes]
```

...as well as connect string for SQL-server you are going to use (example for ODBC):
```
$SQL.connect-string[odbc://DSN=www_mydomain_ru^;UID=user^;PWD=password]
```

Note: it will be used in your code like...
```
^connect[$SQL.connect-string]{...}
```

It is recommended that you also define method **unhandled_exception**, which will output error messages on problems at your site.

Note: configuration file is definitely optional. You can place your configuration method in file `auto.p` *in your web-space root. Configurations, however, will most probably be different for different hosting locations (for example: debug and production servers). That is why it would be more comfortable to keep these differences in a separate file outside web-space.*

Configuration method

If configuration file contains method **conf**, this method will be executed first, before method **auto**, to set vital system parameters:

- Files defining character sets;
- HTTP POST-request size limit;
- Mail-sending server/application;
- SQL-drivers and their parameters;
- Table to associate filename extension with its mime-type.

We recommend that you place this method in in configuration file.

Method is defined like the following:

```
@conf[filespec]
```

...where **filespec** is full name of file containing the method.

Charset **UTF-8**, used in Parser by default, is always available and thus doesn't have to be loaded. To use other charsets, specify files defining them. This should be done in the following way:

```
$CHARSETS[
   $.windows-1251[/full/path/to/windows-1251.cfg]
   ...
]
```

See "Files defining character sets".

Size limit for POST data:

```
$LIMITS[
   $.post_max_size(10*0x400*0x400)
]
```

Parameters for mail-sending program (see **^mail:send[...]**)...

...under Windows and UNIX—SMTP-server address

```
$MAIL[
   $.SMTP[mail.office.design.ru]
]
```

...under UNIX:

in safe mode versions you can configure mail-sending program only if you compile Parser from source code, by yourself. Binary versions, which are available for download directly from `http://parser.ru/en/download/`, configure mail-sending in such a way:

```
   /usr/sbin/sendmail -i -t -f postmaster
```

It is only in unsafe-mode versions that you can specify mail-sending program by yourself:

```
$MAIL[
   $.sendmail[/custom/mail/sending/program params]
]
```

...and by default Parser uses command...

```
   /usr/sbin/sendmail -t -i -f postmaster
```

...or command...

```
   /usr/lib/sendmail -t -i -f postmaster
```

...depending on system you use.

When a message is being sent, Parser will replace "postmaster" with mail-sender's address from obligatory header field "from".

One can also provide a table of SQL-drivers:

```
$SQL[
$.drivers[^table::create{protocol   driver      client
mysql /full/disk/path/parser3mysql.dll    /full/disk/path/libmySQL.dll
odbc  /full/disk/path/parser3odbc.dll
pgsql /full/disk/path/parser3pgsql.dll    /full/disk/path/libpq.dll
sqlite /full/disk/path/parser3sqlite.dll  /full/disk/path/sqlite3.dll
oracle      /path/to/parser3oracle.dll    C:\Oracle\Ora81\BIN\oci.dll?PATH+=^;C
:\Oracle\Ora81\bin
}]
]
```

Column **client** of table **drivers** may contain parameters passed to client's library, delimited from file name with character **?**. The whole construction will look like:

name1=value1&name2=name2&...

...as well as...

name+=value

These variables will replace (=) or be appended to (+=) already existing value in program environment before the library is initialized. Such an approach is particularly useful when you add path to Oracle libraries, if this path has not been already specified in program's system environment.

Table to associate filename extension with its mime-type:

```
# file created with ^file::load[…],
# will specify this $response:content-type when output in $response:body
$MIME-TYPES[^table::create{ext mime-type
zip application/zip
doc application/msword
xls application/vnd.ms-excel
pdf application/pdf
ppt application/powerpoint
rtf application/rtf
gif image/gif
jpg image/jpeg
jpeg image/jpeg
png image/png
tif image/tiff
html text/html
htm text/html
txt text/plain
mts application/metastream
mid audio/midi
midi audio/midi
mp3 audio/mpeg
ram audio/x-pn-realaudio
rpm audio/x-pn-realaudio-plugin
ra audio/x-realaudio
wav audio/x-wav
au audio/basic
mpg video/mpeg
avi video/x-msvideo
mov video/quicktime
swf application/x-shockwave-flash
}]
```

File name extensions in this table should be given in lowercase. Table search is case-insensitive, so, for example, file FACE.GIF will acquire mime-type image/gif.

File defining charset: format description

The data is represented in tab-delimited format. The columns are:

char—character or its code in decimal or hexadecimal (0xHH) representation in charset specified by this file.

white-space, **digit**, **hex-digit**, **letter**, **word**—a set of flags specifying the class that the character belongs to. Empty field means the symbol does not belong to this class, whereas non-empty field (e.g. 'x') means it does.

For more detailed information on character classes see regular expressions description in special literature.

lowercase—if character has a pair in lowercase, the field contains this pair (as either character or code). For example, 'W' pairs with 'w'. This field is used in regular expressions for case-insensitive search, as well as in methods **upper** and **lower** of class **string**.

unicode1—character's main Unicode value. If it coincides with character code, this field can remain empty.

unicode2—character's additional Unicode value, if exists.

Installing Parser on Apache web-server, CGI script

To install Parser, one should make changes to server's main configuration file. If you are not authorized to make such changes, you should be able to use `.htaccess` files.

By default, Apache has usage of `.htaccess` disabled.
You will need to enable it (at least allow specifying `FileInfo`) by adding directives to server's configuration file (usually `httpd.conf`), inside `<virtualhost ...>` section allotted to your site or outside it—for all sites:

```
<Directory /path/to/your/web/space>
AllowOverride FileInfo
</Directory>
```

Place Parser's executable file (in current version, parser3) into your CGI scripts directory (if you upload it using ftp you must do it in binary mode) and set necessary rights (ask your hosting provider for details, but usually it's—755).

For Win32:
If you use version with XML support, unpack XML libraries into the same directory.

Add these blocks to your `.htaccess` file (or `httpd.conf`—inside `<virtualhost ...>` section allotted for your site or outside it—for all sites):

```
# declare Parser as .html files handler
AddHandler parser3-handler html
Action parser3-handler /cgi-bin/parser3

# deny access to .p files, mainly: auto.p
<Files ~ "\.p$">
  Order allow,deny
  Deny from all
</Files>
```

If you would rather change implicit configuration file (see "Installing and configuring Parser") location, you can explicitly specify it :

```
# assign environment variable containing path to auto.p
SetEnv CGI_PARSER_CONFIG /path/to/file/auto.p
```

Note: In this case, you will need Apache module `mod_env`, which is, however, installed by default.

Parser makes records about errors to error log file `parser3.log`, which is implicitly located in the same directory where parser3 CGI script is. If Parser is not allowed to write to that file, errors are reported to standard error stream and are recorded in web-server error log file. If you would rather change implicit location of `parser3.log`, you can explicitly specify it.

```
# assign environment variable containing path to parser3.log
SetEnv CGI_PARSER_LOG /path/to/file/parser3.log
```

Note: In this case, you will need Apache module `mod_env`, which is, however, installed by default.

Installing Parser on Apache web-server, server module

To install Parser, one should make changes to server's main configuration file. If you are not authorized to make such changes, you should be able to use `.htaccess` files.

By default, Apache has usage of `.htaccess` disabled.
You will need to enable it (at least allow specifying `FileInfo`) by adding directives to server's configuration file (usually `httpd.conf`), inside `<virtualhost ...>` section allotted for your site or outside it—for all sites:

```
<Directory /path/to/your/web/space>
AllowOverride FileInfo
</Directory>
```

Parser3 performs necessary transcoding,by itself, so, if you use non-English Apache version, you should add this line to server's main configuration file (usually `httpd.conf`):

```
CharsetDisable On
```

...which will prohibit Apache from transcoding. If you are not authorized to make changes in server's main configuration file, add this line to `.htaccess` file.

Under UNIX:
You will need to compile Parser from source codes by running script `configure` with key `--with-apache13`, then running `make` and following the instructions that will be output.
Note: under some systems, standard `make` does not work with Parser3 make-files. In this case you will need to use GNU variant—`gmake`.

Under Win32:
Place Parser's executable files (in current version, `ApacheModuleParser3.dll`) into an arbitrary directory. If you use version with XML support, unpack XML libraries into directory specified in environment variable PATH (for example, C:\WinNT).
Add these lines to `httpd.conf`, after existing `LoadModule` directives:

```
# load module dynamically
LoadModule parser3_module x:\path\to\ApacheModuleParser3.dll
```

Note: If necessary, place accompanying `.dll` files into the same directory.

And add these lines after existing `AddModule` directives (if no directives exist already, do not add):

```
# adding module to the list of active modules
AddModule mod_parser3.c
```

Add these blocks to your `.htaccess` file (or `httpd.conf`—inside `<virtualhost ...>` section allotted for your site or outside it—for all sites):

```
# declare Parser as .html files handler
AddHandler parser3-handler html

# specify  configuration file
ParserConfig x:\path\to\parser3\config\auto.p
```

```
# deny access to .p files, mainly: auto.p
<Files ~ "\.p$">
    Order allow,deny
    Deny from all
</Files>
```

Installing Parser on web-server IIS, version 5.0 or higher

Place Parser's module executables (`parser3isapi.dll` in current version) into an arbitrary directory. If you use version with XML support, unpack XML libraries into directory specified in environment variable PATH (for example `C:\WinNT`).

Having placed files to needed locations, you need to declare Parser as .html files handler:

1. Run **Management Console**, right-click icon with your web server and choose **Properties**;
2. Go to **Application settings** and under **Home directory** click on the **Configuration button**;
3. Click **Add**;
4. In the **Executable** box, type full path to `parser3.exe` or `parser3isapi.dll`;
5. In the **Extension** box, type **.html**;
6. Check **Check that file exists** box;
7. Click **OK**.

mod_rewrite analogue

For IIS web server there is no built in analogue to Apache module `mod_rewrite`, there are only modules by third parties.

But one can set up any arbitrary page `handler.html` as handler of 404 page (we also recommend set it up as a handler for 403.14 and 405 errors).

Original uri accessible in **$request:uri**.

Regretfully when IIS handles `POST` requests which have no document name (.../), IIS **does not pass** `POST`ed body to CGI-scripts.
Possible walkaround: for such pages set this action:
```
<form action="form.html"...
```
and handle inevitable "`form.html` file not found" error in **@unhandled_exception** and suppress writing it to error log.

Using Parser as standalone interpreter

```
/path/to/parser3 script_file
x:\path\to\parser3 script_file
```

You can use Parser to interpret scripts with no web-server running. In this case you will just need to run Parser in command line with parameter—name of script to be interpreted. In this case, current directory will be considered web-space root.

Errors will get into standard error stream, which can be redirected to needed file:

```
command 2>>error_log
```

Note: do not forget to clean it up between whiles.

When working under UNIX, one can also take standard approach, which is specifying path to interpreter in script's first line:

```
#!/path/to/parser3
```

```
#your code
Check: ^eval(2*2)
```

Note: do not forget to set attributes allowing owner/user group to run the script. It can be done with:
`chmod ug+x file`

Compiling Parser from source code

Download Parser's source code and necessary modules via CVS. This can be done by running command:
`cvs -d :pserver:anonymous@cvs.parser.ru:/parser3project login`
Password string is empty.

`cvs -d :pserver:anonymous@cvs.parser.ru:/parser3project get -r branch_name module_name`

`Branch_name`—having specified no **`-r`**, you will get currently developed version (HEAD).
To get a stable version, get branch "`release_3_X_XXXX`".

`Module_name`:
Name of main module: `parser3`.

Module with SQL drivers: `sql`.
It presently has the following directories:
`sql/mysql`
`sql/pgsql`
`sql/oracle`
`sql/odbc`
`sql/sqlite`

To compile SQL drivers the source codes of Parser3 are required. Because of some .h files included using relative path, the directory structure must be the next:

```
parser3project  <- directory where you decide to put source codes for Parser3
project
  |
  +-parser3     <- Parser3 source codes
  |
  +-sql
     +-mysql    <- mysql driver source codes
     +-...      <- source codes for other SQL drivers
```

To compile under UNIX:

...as Apache 1.3 module, you should first go to directory with Apache source files and, under this directory, execute command:

`./configure`

...and it is only after it has been completed that you can compile Parser.

To compile under Win32:

...you need directory:

`win32/tools`

...you need directories for SQL drivers:

```
win32/sql/mysql
win32/sql/pgsql
win32/sql/oracle
win32/sql/sqlite
```

...to you use version with XML support, you need a directive in `parser3/src/include/pa_config_fixed.h`:

```
#define XML
```

...to you use mail-receiving version, you need a directive in `parser3/src/include/pa_config_fixed.h`:

```
#define WITH_MAILRECEIVE
```

For *UNIX/Cygwin* users:

read instructions on compilation and installation in INSTALL files contained with each module.

To compile under Win32, use *Microsoft Visual Studio.NET (2003 or higher)*, use `.sln` files contained with each module. Unpack all modules to directory `parser3project`, contained in disk's root (important!) directory.

Index

– B –

– C –

– – –

– D –

– E –

– – –

– F –

– G –

– H –

– I –

– J –

– K –

– L –

– M –

– N –

– O –

– P –

– Q –

– R –

– S –

– T –

– U –

– V –

– W –

– X –

– Y –