Студия Лебедева представляет...

...язык скриптования сайтов Parser3.

Автор технологии Parser:
Константин Моршнев | http://www.moko.ru

Автор Parser3:
Александр Петросян (PAF) | http://paf.design.ru

Поддержка Parser3:
Михаил Петрушин (Misha v.3) | http://misha.design.ru

Авторы документации:
Алексей Сорокин | lex_sorokin@mail.ru
Владимир Муров | lir_vl@mail.ru
Александр Петросян (PAF) | http://paf.design.ru

Оглавление

Индекс 193

Как работать с документацией

Данное руководство состоит из трех частей.

В первой, учебной части рассматриваются практические задачи, решаемые с помощью Parser. На примере создания учебного сайта показаны базовые возможности языка и основные конструкции. Для создания кода можно пользоваться любым текстовым редактором. Желательно, чтобы в нем был предусмотрен контроль над парностью скобок с параллельной подсветкой, поскольку при увеличении объема кода и его усложнении становится сложно следить за тем, к чему относится та или иная скобка, и эта возможность существенно облегчит ваш труд. Также поможет в работе и выделение цветом конструкций языка. Читать и редактировать код станет намного проще.

Учебная часть построена в виде уроков, в начале которых предлагается рабочий код, который можно просто скопировать в нужные файлы. Далее подробно разбирается весь пример с объяснением логики его работы. В конце каждого урока тезисно перечислены все основные моменты, а также даны рекомендации, на что надо обратить особое внимание. Внимательное изучение представленных уроков обеспечит вас необходимым запасом знаний для последующей самостоятельной работы над проектами на Parser.

Во второй части представлен справочник по синтаксису языка и подробно рассмотрены правила описания различных конструкций.

Третья часть представляет собой справочник операторов и базовых классов языка с описанием методов и краткими примерами их использования.

В приложениях к документации рассмотрены вопросы установки и конфигурирования Parser.

Принятые обозначения

ABCDEFGH — Код Parser в примерах для визуального отличия от HTML (**Courier New, 10**). Для удобства работы с электронной документацией дополнительно выделен цветом.

ABCDEFGH — Файлы и каталоги, рассматриваемые в рамках урока.

ABCDEFGH — Дополнительная и справочная информация.

[3.1] — Номер версии Parser, начиная с которой доступна данная функция или опция.

В справочнике символ "|" равнозначен союзу ИЛИ.

Введение

"И сказал Господь: Я увидел страдания народа Моего в Египте, и услышал вопль его от приставников его; Я знаю скорби его, и иду избавить его от руки египтян и вывести его из земли сей в землю хорошую и пространную, где течет молоко и мед..." (Исход, 3, 7–8)

Parser?

Самый логичный вопрос, который может возникнуть у вас, уважаемый читатель, это, несомненно: «А что это вообще такое?». Итак, если вы хотите узнать ответ – добро пожаловать! Для начала позвольте сделать несколько предположений.

Первое, очень важное. Вы уже имеете представление о том, что такое HTML. Если данное сочетание букв вам незнакомо, дальнейшее чтение вряд ли будет увлекательным и полезным, поскольку Parser является языком программирования, существенно упрощающим и систематизирующим разработку именно HTML документов.

Второе, существенное. Мы предлагаем вам познакомиться с новой версией Parser на практических примерах, поэтому будем считать, что у вас под руками есть установленный Parser3. Теория, как известно, без практики мертва. Как установить и настроить программу, подробно рассказано в приложении.

Третье, просто третье. У вас есть немного свободного времени, терпения, IQ не ниже 50, а также желание сделать свою работу по разработке HTML документов проще, логичнее и изящнее. Со своей стороны обещаем вам, что время, потраченное на изучение этого языка с лихвой окупится теми преимуществами и возможностями, которые он дает.

Вроде бы не очень много, не так ли? Все остальное – это уже наша забота!

Parser...

Parser появился на свет в 1997 году в Студии Артемия Лебедева (www.design.ru). Целью его создания было облегчить труд тех, кто по сегодняшний день успешно и в кратчайшие сроки создает лучшие сайты рунета, избавить их от рутинной работы и позволить отдавать свое время непосредственно творчеству. Зачем забивать гвозди микроскопом, если его настоящее предназначение совсем не в этом?

Именно поэтому большинство интернет-проектов студии делаются на Parser. Он проще в использовании, чем что-либо, созданное для подобных целей. Но эта простота не означает примитивность. Она позволяет использовать Parser не только опытным программистам, но и тем людям, которые далеки от программирования. Позволяет создавать красивые, полноценные сайты быстро, эффективно и профессионально. Мы хотим дать такую возможность и вам.

Идея Parser довольно проста. В HTML-страницы внедряются специальные конструкции, обрабатываемые нашей программой перед тем, как страницы увидит пользователь. Программа сама доделывает за вас работу по окончательному формированию и оформлению сложного документа. Это похоже на собирание из конструктора, в котором есть готовые модули для всех обычных целей. Если же вы мыслите нестандартно, просто создайте свои модули, которые будут делать то, что необходимо именно вам. Ничего невозможного нет, при этом все делается просто и быстро.

Вы сами в этом убедитесь, как только начнете работать с Parser.

Parser!

Подведем итоги. Что дает вам Parser? Вы получаете в свое распоряжение переменные, циклы, условия и т.д., все то, чего так не хватает привычному HTML. Без использования Parser аналогичный по внешнему виду документ будет гораздо больше по объему, а некоторые задачи останутся неразрешенными. С Parser у вас пропадет необходимость повторять одни и те же инструкции по несколько раз, но появится возможность формирования динамических страниц в зависимости от действий пользователя, работать с базами данных и XML, внешними HTTP-серверами, в считанные минуты менять дизайн страниц. И все это без обычного в подобных случаях сложного программирования.

Ваши страницы будут формироваться из отдельных законченных объектов, а вы просто скажете Parser какие из них, сколько, куда и в какой последовательности поставить. Если нужно что-то поменять местами или добавить, вы просто указываете это – и все. Остальное будет сделано автоматически. При этом сам проект станет логичным и понятным за счет структуризации.

Очень скоро вы сможете делать все то, что раньше могли позволить себе лишь те, кто использовал достаточно сложные языки программирования, требующие месяцы, если не годы, изучения и практики .

Еще один очевидный плюс. Отдельные модули могут быть разработаны различными людьми, которые будут их поддерживать и обновлять самостоятельно и независимо от остальных. Это обеспечит удобное разделение труда и возможность комфортной параллельной работы нескольких людей над одним проектом.

Впрочем, перечислять преимущества можно долго, но может быть, мы и так убедили вас попробовать? Разве наш опыт не является доказательством правоты? К тому же, мы не просим за использование Parser денег, мы просто хотим, чтобы рунет стал лучшим! И у нас есть для этого готовое, проверенное

решение – Parser. Вы полюбите его так же, как и мы.

Приступаем? Вперед!

Урок 1. Меню навигации

Давайте начнем с самого начала. Итак, вы хотите сделать сайт (узел, сервер). Первым делом, необходимо уяснить, каким образом на сайте будет упорядочена та или иная информация. Сколько будет категорий, подразделов т.д. Все эти вопросы возникают на первом этапе – "Организация сайта".

А какой должна быть навигация сайта? Требований к хорошей навигации много. Она должна быть понятна, легко узнаваема, единообразна, удобна в использовании, быстро загружаться, давать четкое понятие о текущем местоположении. При этом на сайте не должно возникать 404-й ошибки, т.е. все ссылки должны работать. Если у вас есть опыт создания сайтов, то вам, скорее всего, приходилось сталкиваться с проблемой создания грамотной навигации.

Не правда ли, хочется иметь какое-то решение, которое всегда будет под рукой и позволит автоматизировать весь этот процесс? Что-то такое, что даст возможность единственный раз написать код и потом, в одном месте, дописывать столько разделов, сколько нужно?

Создание меню, которое ориентирует пользователя на сайте, не дает ему заблудиться – вот задача, с которой нам хочется начать повествование о Parser. Почему именно это? Прежде всего потому, что большое количество тегов:

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

трудно контролировать. А если вам понадобится добавить еще один раздел? Придется в каждую страницу вносить изменения, а человеку свойственно делать ошибки. При этом отнюдь не исключено, что после такой «модернизации» ваш ресурс в ответ на запросы пользователей сообщит о том что «данная страница не найдена». Вот где проблема, которую с помощью Parser можно решить очень легко.

Решение следующее. Мы создаем некую функцию на Parser, которая будет генерировать нужный нам фрагмент HTML-кода. В терминологии Parser функции называются методами. В тех местах, где этот код понадобится, будем просто давать указание «Вставить меню навигации» и сразу же будет создана страница, содержащая меню. Для этого сделаем несколько простых шагов:

1. Всю информацию о наших ссылках будем хранить в одном файле, что позволит впоследствии вносить необходимые изменения только в нем. В корневом каталоге будущего сайта создаем файл sections.cfg, в который помещаем следующую информацию:

section_id	*name*	*uri*
1	Главная	/
2	Новости	/news/
3	Контакты	/contacts/
4	Цены	/price/
5	Ваше мнение	/gbook/

Здесь используется так называемый формат tab-delimited. Столбцы разделяются знаком табуляции, а строки – переводом каретки. При копировании этой таблицы в текстовый редактор данное форматирование будет создано автоматически, но если вы будете создавать таблицу вручную, необходимо это учитывать. Для таблиц ВСЕГДА применяется формат tab-delimited.

2. В том же каталоге, где и `sections.cfg`, создаем файл `auto.p`
В нем мы будем хранить все те кирпичики, из которых впоследствии Parser соберет наш сайт. AUTO означает, что все эти кирпичики будут всегда доступны для Parser в нужный момент, а расширение ".p", как вы, наверное, догадались, это... правильно! Он самый!

3. В файл auto.p вставим следующий код:

```
@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>
```

Данные из этого файла и будут служить основой для нашего навигационного меню.

Вот и все, подготовительные работы закончены. Теперь открываем код страницы, где все это должно появиться (например, **index.html**), и говорим: «Вставить меню навигации». На Parser это называется «вызов метода» и пишется так:

```
^navigation[]
```

Осталось только открыть в браузере файл, в который мы вставили вызов метода и посмотреть на готовое меню навигации. Теперь в любом месте на любой странице мы можем написать заветное **^navigation[]**, и Parser вставит туда наше меню. Страница будет сформирована «на лету». Что хотели, то и получили.

Если у вас дела обстоят именно так, то поздравляем – вы вступили в мир динамических сайтов. Очень скоро вы также запросто будете использовать базы данных для формирования страниц и делать многое другое.

Однако не будем радоваться раньше времени. Давайте разберемся, что же мы сделали, чтобы добиться такого результата. Взгляните на код в auto.p. Если кажется, что все непонятно, не надо бежать прочь. Уверяем, через несколько минут все встанет на свои места. Итак, посмотрим на первую строчку:

```
@navigation[]
```

Она аналогична строке **^navigation[]**, которую мы вставили в текст страницы для создания меню. Различие только в первом символе: ^ и @. Однако логический смысл этого выражения совершенно иной – здесь мы определяем метод, который вызовем позже. Символ @ (собака) в первой колонке строки в Parser означает, что мы хотим описать некоторый блок, которым воспользуемся в дальнейшем. Следующее слово определяет имя нашего метода: **navigation**. Это только наше решение, как ее назвать. Вполне допустимы имена: **a_ну_ка_вставь_меню_быстро**. Но читаться такая программа будет хуже, впрочем, кому как понятнее, можете назвать и так.

Жизненно необходимо давать простые, понятные имена. Они должны точно соответствовать тому, что именуемый объект будет хранить и делать. Сохраните нервы и время себе и всем тем, кому придется разбираться в ваших текстах, отнеситесь, пожалуйста, к именам внимательно. Имена могут быть русские или латинские, главное соблюдать единообразие: или все по-русски, или по-английски.

Идем дальше.

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

Это ключевая строка нашего кода. Она достаточно большая, поэтому давайте разберем ее по частям.

Строка начинается символом **$** (рубль) и следующим сразу за ним именем **sections**. Так в Parser обозначаются переменные. Это надо запомнить. Все просто: видим в тексте $var – имеем дело с переменной var. Переменная может содержать любые данные: числа, строки, таблицы, файлы, рисунки и даже часть кода. Присвоение переменной **$parser_home_url** значения **www.parser3.ru**

на Parser выглядит так: **$parser_home_url[www.parser3.ru]**. После этого мы можем обратиться к переменной по имени, т.е. написать **$parser_home_url** и получить значение **www.parser3.ru**.

Еще раз тоже самое:

$var[…] – присваиваем

$var – получаем

Подробности в разделе «Переменные».

В нашем случае переменная **$sections** будет хранить таблицу из файла **sections.cfg**.

Любую таблицу Parser рассматривает как самостоятельный объект, с которым он умеет производить только вполне определенные действия, например, добавить или удалить из нее строку. Поскольку переменная может хранить любые данные, необходимо указать, что присвоенное нами переменной значение является именно таблицей.

Лирическое отступление.
Пример из жизни. Всю автомобильную технику можно грубо разделить на несколько классов: легковые автомашины, грузовики, трактора и гусеничная техника. Любой автомобиль является объектом одного из этих классов. Вы легко можете определить, к какому классу относится автомобиль, поскольку их всех объединяют общие характеристики, такие как вес, масса перевозимого груза и т.д. Любой автомобиль может совершать действия: двигаться, стоять или ломаться. Каждый из автомобилей обладает своими собственными свойствами. И самое главное, автомобиль не может появиться сам собой, его нужно создать. Когда конструктор придумывает новую модель автомобиля, он точно знает, автомобиль какого класса он создает, какими свойствами будет наделено его творение и что оно сможет делать. Также и в Parser: каждый объект относится к определенному классу, объект класса создается конструктором этого класса и наделен характеристиками (полями) и методами (действиями), общими для всех подобных объектов.

Итог

Любой **объект** в Parser принадлежит конкретному **классу**, характеризуется **полями** и **методами** именно этого класса. Чтобы он появился, его нужно создать. Делает это **конструктор** данного класса. Разберитесь с этой терминологией, это основа.

Отвлеклись? Продолжим. Переменной **$sections** мы присвоили вот что:

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

Буквально это означает следующее: мы создали объект класса **table** при помощи конструктора `load`. Общее правило для создания объекта записывается так:

```
^имя_класса::конструктор[параметры_конструктора]
```

Подробности в разделе «Передача параметров».

В качестве параметра конструктору мы передали имя файла с таблицей и путь к нему.

Теперь переменная **$sections** содержит таблицу с разделами нашего сайта. Parser считает ее объектом класса **table** и точно знает, какие действия с ней можно выполнить. Пока нам понадобится только один метод этого класса – **menu**, который последовательно перебирает все строки таблицы. Также нам потребуются значения из полей самой таблицы. Синтаксис вызова методов объекта:

```
^объект.метод_класса[параметры]
```

Получение значений полей объекта (мы ведь имеем дело с вполне определенной таблицей с заданными нами же полями):

```
$объект.имя_поля
```

Знания, полученные выше, теперь позволяют без труда разобраться в последней части нашего кода:

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

Мы формируем HTML-таблицу, в каждый столбец которой помещаем значения, содержащиеся в полях нашей таблицы **$sections**: **uri** – адрес и **name** – имя. При помощи метода **menu** мы автоматически перебираем все строки таблицы. Таким образом, даже если у нас будет несколько десятков разделов, ни один из них не будет потерян или пропущен. Мы можем свободно добавлять разделы, удалять их и даже менять местами. Изменения вносятся только в файл `sections.cfg`. Логика работы не нарушится. Все просто и красиво.

Давайте подведем итоги первого урока.

Что мы сделали: написали свой первый код на Parser, а именно, научились создавать меню навигации на любой странице сайта, опираясь на данные, хранящиеся в отдельном файле.

Что узнали: познакомились с концептуальными понятиями языка (класс, объект, свойство, метод), а также некоторыми базовыми конструкциями Parser.

Что надо запомнить: Parser использует объектную модель. Любой объект языка принадлежит какому-то классу, имеет собственные свойства и наделен методами своего класса. Для того чтобы создать объект, необходимо воспользоваться конструктором класса.

Синтаксис работы с объектами:

$переменная[значение]	– задаем значение
$переменная	– получаем значение
$переменная[^имя_класса::конструктор[параметры]]	– создаем объекта класса **имя_класса** и присваиваем его переменной
$переменная.имя_поля	– получаем поле самого объекта, хранящегося в переменной
^переменная.метод[]	– вызываем действие (метод класса, к которому принадлежит объект, хранящийся в переменной)

Что будем делать дальше: заниматься модернизацией меню. Ведь пока оно не отвечает многим требованиям: ставит лишнюю ссылку на текущий раздел, выдает столбцы разной ширины. На втором уроке мы исправим все эти недостатки и сделаем еще кое-что.

Урок 2. Меню навигации и структура страниц

Предыдущий урок мы закончили тем, что определили недостатки в реализации меню. Давайте займемся их устранением. Наше меню выводит лишнюю ссылку на текущую страницу, что нисколько не украшает будущий сайт. Чтобы этого избежать, необходимо проверить, не является ли раздел, на который мы выводим ссылку, текущим. Иными словами, нам нужно сравнить URI раздела, на который собираемся ставить ссылку, с текущим URI. В случае если они совпадают, ссылку на раздел ставить не надо. Дополнительно для удобства пользователей мы изменим в меню навигации цвет столбца текущего раздела.

Открываем файл `auto.p` и меняем его содержимое на:

```
@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>
}
```

Что изменилось? На первый взгляд не так уж и много, но функциональность нашего модуля существенно возросла. Мы описали еще один метод – **navigation_cell**, который вызывается из метода **navigation**. В нем появилась новая структура:

```
^if(условие){код если условие "истина"}{код если условие "ложь"}
```

Что она делает, понять не сложно. В круглых скобках задается условие, в зависимости от того, какое значение возвращает условие, "ложь" или "истина", можно получить разный результат. Также, если в условии записано выражение, значение которого равно нулю, то результат – "ложь", иначе – "истина". Мы используем оператор **if** для того, чтобы в одном случае поставить ссылку на раздел, а другом нет. Осталось только разобраться с условием. Будем сравнивать на равенство две текстовых строки, в одной из которых – значение URI раздела из таблицы **sections**, в другой – текущий URI (**$request:uri** возвращает строку, содержащую URI текущей страницы). Тут возникает вопрос о том, какие же строки равны между собой? Несомненно, только те, которые полностью совпадают и по длине, и по символьному содержанию.

Для сравнения двух строк в Parser предусмотрены следующие операторы:

eq – строки равны (***equal***): **parser eq parser**
ne – строки не равны (***not equal***): **parser ne parser3**
lt – первая строка меньше второй (***less than***): **parser lt parser3**
gt – первая строка больше второй (***greater than***): **parser3 gt parser**
le – первая строка меньше или равна второй (***less or equal***)
ge – первая строка больше или равна второй (***greater or equal***)

С условием разобрались: если **$sections.uri** и **$request:uri** совпадают, ссылку не ставим (а заодно красим столбец в другой цвет – подумаем о наших пользователях, так им будет удобнее), если нет – ставим.

Идем дальше. Меню из первого урока выводило столбцы разной ширины. Ничего страшного, но некрасиво. Проблема решается очень просто: всю ширину меню (100%) делим на количество разделов, которое равно количеству строк в таблице **sections**. Для этого воспользуемся оператором **^eval()** и тем, что можно использовать объекты класса **table** в математических выражениях. При этом их числовое значение равно числу записей в таблице. Обратите внимание также на то, что мы пользуемся целочисленным делением, используя обратный слеш вместо прямого.

На **^eval()** остановимся чуть подробнее. Он позволяет получить результат математического выражения без введения дополнительных переменных, иными словами, хотим что-то посчитать – пишем:

^eval(выражение)[формат]

Использование [формат] дает возможность вывода результата выражения в том виде, который нужен. Форматная строка [%d] отбрасывает дробную часть, [%.2f] дает два знака после запятой, а [%04d] отводит 4 знака под целую часть, дополняя недостающие символы нулями слева. Форматированный вывод нужен, когда необходимо представить число в определенном виде (скажем, 12.44 $ смотрится куда лучше 12.44373434501 $).

Вот, собственно, и все, что касается меню. Теперь оно функционально и готово к использованию.

Наш первый кирпичик для будущего сайта готов. Теперь займемся структурой страниц. Давайте разобьем их на следующие блоки: **header** – верхняя часть страницы, **body** – основной информационный блок, включающий также наше меню и **footer** – нижняя часть страницы. Многие сайты имеют похожую структуру.

Footer будет для всех страниц одинаковым, **header** – для всех страниц одинаковый по стилю, но с разными текстовыми строками – заголовками страницы, а **body** будет разный у всех страниц, сохраняя только общую структуру (предположим, два вертикальных информационных блока, соотносящихся по ширине как 3:7). К **body** отнесем и наше меню.

Каждая из страниц будет иметь следующую структуру:

<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>

Также, как в случае с меню, опишем каждый из этих блоков методом (функцией) на Parser. Давайте подробно разберемся с каждым блоком.

С `footer` все очень просто – в `auto.p` добавляем код:

```
@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>
```

Никаких новых идей здесь нет, разве что мы впервые использовали класс **date** с конструктором **now** для получения текущей даты, а затем из объекта класса **date** взяли поле **year** (год). Если это кажется вам непонятным, обязательно вернитесь к первому уроку, где рассказано о работе с объектами на примере класса **table**. Все идентично, только теперь мы имеем дело с объектом другого класса.

Немного сложнее с модулем **header**. С одной стороны, нам нужно формировать уникальный заголовок-приветствие для каждой страницы. В то же время он будет одинаковым с точки зрения внешнего вида, различие только в тексте, который будет выводиться. Как же быть? Мы предлагаем сделать следующее: определить в нашем `auto.p` новую функцию **header**, внутри которой будет вызываться другая функция – **greeting**. А функция **greeting**, в свою очередь, будет определяться на самих страницах сайта и содержать только то, чем отличаются заголовки страниц (в нашем случае

строку-приветствие).

Дополняем `auto.p` следующим кодом:

```
@header[]
<html>
<head>
<title>Тестовый сайт 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 />
```

Теперь внимание, кульминация. Parser позволяет сделать очень интересный финт: определить один раз общую структуру страниц в файле `auto.p`, создать каркас, а затем, используя функции, подобные **greeting**, в тексте самих страниц, получать разные по содержанию страницы одинаковой структуры. Как это работает?

В самом начале файла `auto.p` мы определим функцию **@main[]**, которая всегда, причем автоматически, исполняется первой. В нее включим вызовы функций, формирующих части страниц.

В начале `auto.p` пишем:

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

А для получения уникального заголовка страниц в каждой из них определим функцию **greeting**, которая вызывается из **header**:

для главной страницы:

```
@greeting[]
Добро пожаловать!
```

для гостевой книги:

```
@greeting[]
Оставьте свой след...
```

и т.д.

Теперь при загрузке, например, главной страницы произойдет следующее:

1. Из файла `auto.p` автоматически начнет выполняться **main**.
2. Первой вызывается функция **header**, из которой вызывается функция **greeting**.
3. Поскольку функция **greeting** определена в коде самой страницы, будет выполнена именно она, вне зависимости от того, определяется она в `auto.p` или нет (происходит переопределение функции).
4. Затем выполняются функции **body** и **footer** из **main**.

В результате мы получаем страницу, у которой будут все необходимые элементы, а в верхней части дополнительно появится наше уникальное приветствие. Переопределяемые функции носят название виртуальных. Мы из файла `auto.p` вызываем функцию, которая может быть переопределена на любой из страниц и для каждой из них выполнит свой код. При этом общая структура страниц будет абсолютно одинаковой, и сохранится стилистическое и логическое единство.

Осталось описать только основной блок — **body**. Как мы договорились, он будет состоять из двух

частей, каждую из которых будем создавать своей функцией, например, **body_main** и **body_additional**, а поскольку навигационное меню, по логике, относится к основной части страниц, вызовем **navigation** также из **body**. Снова воспользуемся механизмом виртуальных функций. Редактируем auto.p – дополняем:

```
@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 />
```

Определение функций **body_main** и **body_additional**, также как и в случае с **greeting** вставим в страницы:

```
@body_additional[]
Главная страница сайта

@body_main[]
Основное содержание
```

Этот текст приводится как образец для index.html. Отлично! Структура окончательно сформирована. Мы описали все необходимые модули в файле auto.p, сформировали общую структуру и теперь можем запросто генерировать страницы. Больше не нужно помногу писать одни и те же куски HTML кода. Привычные HTML-страницы трансформируются примерно в следующее (примерное содержание index.html файла для главной страницы):

```
@greeting[]
Добро пожаловать!

@body_additional[]
Главная страница сайта

@body_main[]
Основное содержание
```

Просто и понятно, не правда ли? Все разложено по полочкам и легко доступно. При этом после обработки подобного кода Parser создаст HTML-код страницы, у которой будет уникальный заголовок, меню, основной информационный блок заданной структуры и **footer**, одинаковый для каждой страницы. Фактически, мы уже создали готовый сайт, который осталось только наполнить информацией. Это готовое решение для изящного сайта-визитки, который можно создать прямо на глазах. Естественно, это не единственное решение, но такой подход дает отличную структуризацию нашего сайта. Некоторые умственные усилия при разработке структуры с лихвой окупятся легкостью последующей поддержки и модернизации. Каркас хранится в auto.p, а все, что относится непосредственно к странице, – в ней самой.

Дальше открываются безграничные просторы для фантазии. Допустим, вам понадобилось поменять внешний вид заголовка страниц на сайте. Мы открываем auto.p, редактируем один единственный раз функцию **@header[]** и на каждой из страниц получаем новый заголовок, по стилю идентичный всем остальным. Для обычного HTML нам пришлось бы вручную переписывать код для каждой страницы. Та же самая ситуация и с остальными модулями. Если возникло желание или необходимость изменить общую структуру страниц, например, добавить какой-то блок, достаточно определить его новой функцией и дополнить функцию **main** в auto.p ее вызовом.

Подобная организация страниц сайта дополняет проект еще одним мощным средством. Предположим,

на одной из страниц нам понадобилось получить **footer**, отличный от других страниц (напомним, изначально мы предполагали, что **footer** везде одинаковый). Единственное, что нужно сделать, это переопределить функцию **footer** на нужной странице. Например, такое наполнение `/contacts/index.html`:

```
@greeting[]
Наша контактная информация

@body_additional[]
Главная страница тестового сайта

@body_main[]
Основное содержание

@footer[]
Здесь у нас контакты
```

изменит привычный **footer** на обозначенный выше, т.е. если Parser находит в тексте страницы код для функции, вызываемой из `auto.p`, он выполнит именно его, даже если функция определена в самом `auto.p`. Если же функция не переопределена на странице, то будет использован код из `auto.p`.

В заключение немного теории для любознательных. Мы будем давать подобную информацию для тех, кто хочет глубже понимать логику работы Parser.

Помните, мы использовали в нашем коде конструкцию **$request:uri***? Она отличается по синтаксису от всего того, с чем мы имели дело раньше. Что же это такое? Внешне похоже на* **$объект.свойство** *(урок 1) – значение полей объекта, только вместо точки использовано двоеточие. На самом деле, это тоже значение поля, только не объекта, а самого класса* **request***. В Parser не предусматриваются конструкторы для создания объектов этого класса. Поля подобных классов формируются самим Parser, а мы можем сразу напрямую обращаться к ним. Техническим языком это называется статическая переменная (поле)* **uri** *класса* **request***. Она хранит в себе URI текущей страницы. Также, наряду со статическими переменными, существуют статические методы, с которыми мы столкнемся уже в следующем уроке. При этом можно сразу же вызывать их также без создания каких-либо объектов с помощью конструкторов. Запомните, что в синтаксисе статических полей и методов всегда присутствует двоеточие. Если встречается конструкция вида* **$класс:поле** *– мы получаем значение поля самого класса, а запись* **^класс:метод** *является вызовом статического метода класса. Например, для работы с математическими функциями в Parser существует класс* **math***. В нем используются только статические методы и переменные:*

$math:PI — *возвращает число π. Это статическая переменная класса* **math***.*

^math:random(100) — *возвращает псевдослучайное число из диапазона от 0 до 100. Это статический метод класса* **math***.*

Отличие от записи методов и полей объектов состоит только в двоеточии.

Давайте подведем итоги второго урока.

Что мы сделали: исправили недостатки в меню навигации, созданном на предыдущем уроке, а также описали новые блоки **header**, **footer** и **body**, формирующие внешний вид страниц нашего сайта. Теперь мы имеем готовое решение для быстрого создания сайта начального уровня.

Что узнали: познакомились с ветвлением кода, научились вставлять в текст страниц результаты математических вычислений, сравнивать строки и получать URI текущей страницы. Также мы узнали новые методы объектов класса table и класса date и познакомились с мощным механизмом виртуальных функций Parser.

Что надо запомнить: первым методом в файле `auto.p` можно определить функцию **main**, которая выполняется автоматически. Любая из функций может содержать вызовы других функций. Все вызываемые из **main** функции обязательно должны быть определены или в `auto.p`, или в тексте

страниц. В случае если функция будет определена и там и там, то больший приоритет имеет функция, определенная в тексте страницы. Она переопределяет одноименную функцию из main (т.н. виртуальная функция) и выполняется вместо нее.

Что будем делать дальше: нет предела совершенству! От создания сайта начального уровня мы переходим к более сложным вещам, таким как работа с формами и базами данных для создания по-настоящему интерактивного сайта. Параллельно с этим познакомимся с новыми возможностями, предоставляемыми Parser для облегчения жизни создателям сайтов.

Урок 3. Первый шаг — раздел новостей

На двух предыдущих уроках мы описали общую структуру нашего сайта. Сейчас это всего лишь каркас. Пора приступать к его информационному наполнению. Практически на каждом сайте присутствует раздел новостей, и наш не должен стать исключением. Также как и с разделами сайта, мы предлагаем начать работу над разделом новостей с создания меню к этому разделу. В качестве средства доступа к новостям будем использовать привычный глазу календарь на текущий месяц.

Создать календарь средствами одного HTML — задача достаточно нетривиальная, к тому же код получится очень громоздким. Сейчас Вы увидите, как легко это сделать на Parser. Приступаем.

Все файлы, относящиеся к разделу новостей, будем размещать в разделе `/news/`, что было указано нами в файле `sections.cfg`. Для начала создадим там (!) файл `auto.p`. Удивлены? Да, файлы `auto.p` можно создавать в любом каталоге сайта. Однако при этом надо иметь в виду, что функции, описанные в `auto.p` разделов, будут явно доступны только внутри этих разделов. Согласитесь, ни к чему загромождать корневой `auto.p` функциями, которые нужны для одного раздела. Логичнее вынести их в отдельный файл, относящийся именно к этому разделу.

Еще одно замечание: если в `auto.p` раздела переопределить функцию, ранее описанную в корневом `auto.p`, то будет исполняться именно эта, переопределенная функция. Сработает механизм виртуальных функций, описанный в предыдущем уроке.

Итак, в `auto.p` раздела news пишем такой код:

```
@calendar[]
$calendar_locale[
	$.month_names[
			$.1[Январь]
			$.2[Февраль]
			$.3[Март]
			$.4[Апрель]
			$.5[Май]
			$.6[Июнь]
			$.7[Июль]
			$.8[Август]
			$.9[Сентябрь]
			$.10[Октябрь]
			$.11[Ноябрь]
			$.12[Декабрь]
	]
	$.day_names[
			$.0[пн]
			$.1[вт]
			$.2[ср]
			$.3[чт]
			$.4[пт]
			$.5[сб]
			$.6[вс]
	]
	$.day_colors[
			$.0[#000000]
			$.1[#000000]
```

```
                    $.2[#000000]
                    $.3[#000000]
                    $.4[#000000]
                    $.5[#800000]
                    $.6[#800000]
      ]
]
$now[^date::now[]]
$days[^date:calendar[rus]($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>
```

Мы определили функцию **calendar**, которая создает HTML-код календаря. Получился довольно громоздкий код, но ведь и задачи, которые мы ставим перед собой, тоже усложнились. Не волнуйтесь, сейчас во всем разберемся.

Самая объемная часть кода, начинающаяся с определения **$calendar_locale**, оказалась незнакомой. Посмотрите на эту структуру. Похоже, в ней мы определяем какие-то данные для календаря, напоминающие таблицу. То, что определено как **$calendar_locale**, в терминологии Parser называется «хеш», или ассоциативный массив. Зачем он нужен можно сказать, просто бегло просмотрев код примера: здесь мы сопоставляем русское написание месяца его номеру в году (3 —

март), название дня недели его номеру, а также связываем шестнадцатиричное значение цвета с некоторым числом. Теперь идея хешей должна проясниться: они нужны для сопоставления (ассоциативной связи) имени с объектом. В нашем случае мы ассоциируем порядковые номера месяцев и дней с их названиями (строками). Parser использует объектную модель, поэтому строка тоже является объектом. Нам несложно получить порядковый номер текущего месяца, но намного нагляднее будет вывести в календаре «Ноябрь» вместо «11» или «пн» вместо «1». Для этого мы и создаем ассоциативный массив.

В общем виде порядок объявления переменных-хешей такой:

```
$имя[
        $.ключ[значение]
]
```

Эта конструкция позволяет обратиться к переменной по имени с ключом **$имя.ключ** и получить сопоставленное значение. Обратите внимание, что в нашем случае мы имеем хеш, полями которого являются три других хеша.

После определения хеша мы видим уже знакомую переменную **now** (текущая дата), а вот дальше идет незнакомая конструкция:

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

По логике работы она напоминает конструктор, поскольку в переменную **days** помещается таблица с календарем на текущий месяц текущего года. Тем не менее, привычного **::** здесь не наблюдается. Это один из статических методов класса **date**. Статические методы наряду с уже знакомыми конструкторами могут возвращать объекты, поэтому в данном случае необходимо присвоить созданный объект переменной. Про статические переменные и методы уже было немного сказано в конце предыдущего урока. Своим появлением они обязаны тому факту, что некоторые объекты или их свойства (поля) существуют в единственном экземпляре, как, например, календарь на заданный месяц или URI страницы. Поэтому подобные объекты и поля выделены в отдельную группу, и к ним можно обращаться напрямую, без использования конструкторов. В случае если мы обращаемся к статическому полю, мы получаем значение поля самого класса. В качестве примера можно привести класс **math**, предназначенный для работы с математическими функциями. Поскольку существует только одно число π, то для того, чтобы получить его значение, используется статическое поле **$math:PI** – это значение поля самого класса **math**.

В результате исполнения этого кода в переменной **days** будет содержаться такая таблица:
Таб.1 (для ноября 2001 года)

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

Это таблица, содержащая порядковый номер дней недели и календарь на 11.2001.

С ней мы и будем дальше работать. Нельзя сразу же выводить содержимое переменной **$days**, просто обратившись к ней по имени. Если мы обратимся к таблице просто как к переменной, будет непонятно, что мы хотим получить – строку, всю таблицу целиком или значение только из одного столбца. Также явно требуется доработка содержимого полученной таблицы. Но ведь не зря же мы создавали наш хеш с названиями дней недели и месяцев. Поэтому далее по коду средствами HTML создается таблица, в первой строке которой мы выводим название текущего месяца, пользуясь данными из хеша, связанными с номером месяца в году:

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

Что здесь что? Мы выводим значение поля **month_names** хеша **calendar_locale** с порядковым номером текущего месяца, полученным как **$now.month**. Результатом выполнения этой конструкции будет название месяца на русском (или любом другом) языке, которое было определено в хеше.

В следующей строке календаря выведем названия дней недели, пользуясь данными хеша. Давайте чуть подробнее определимся с задачей. Нам надо последовательно перебрать номера дней недели (от 0 до 6) и поставить в соответствие номеру дня его название из поля **day_names** хеша **calendar_locale**. Для этой цели удобнее всего воспользоваться циклом: последовательностью действий, выполняющейся заданное количество раз. В данном случае мы используем цикл **for**. Его синтаксис такой:

```
^for[счетчик](диапазон значений, например 0;6){последовательность действий}
```

Одно из достоинств циклов – возможность использования значения счетчика внутри цикла, обращаясь к нему как к переменной. Этим мы и воспользуемся:

```
^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>
}
```

Все просто и понятно, если знать, что такое цикл: последовательно меняя значение **week_day** от 0 до 6 (здесь **week_day** является счетчиком цикла), мы получаем семь значений:

$calendar_locale.day_colors.$week_day – для цвета шрифта
$calendar_locale.day_names.$week_day – для названия дня недели.

Идея получения данных та же, что и для получения названия месяца, только используются другие ключи хеша.

Возможно, возник вопрос: зачем в хеше ключ **day_colors**? Ответ прост – все должно быть красиво! Если есть возможность максимально приблизить наш календарь к реальному, то так и сделаем – перекрасим выходные дни в красный цвет.

Далее по тексту следует большой красивый блок. Чтобы в нем разобраться, определимся с задачами. Итак, нам нужно:

1. Последовательно перебрать строки таблицы **days** (Таб.1).
2. В каждой строке таблицы **days** последовательно перебрать и вывести значения столбцов (числа месяца).
3. Корректно вывести пустые столбцы (то есть проверить первую и последнюю недели месяца на полноту).
4. Как-то выделить текущее число, например другим цветом и жирным шрифтом.

Приступаем. Первый пункт решается с помощью знакомого метода menu класса **table**:

```
^days.menu{…}
```

Перебор столбцов логичнее всего сделать циклом **for**, с которым мы только что познакомились:

```
^for[week_day](0;6){…}
```

Для проверки столбцов на пустоту для вывода столбцов без чисел используем оператор **if**. Вообще, любые проверки условий всегда можно реализовать с помощью **if**:

```
^if($days.$week_day){
      …
}{
```

```
        <td bgcolor="#DFDFDF"> </td>
}
```

Обратите внимание, что в условии if мы ни с чем не сравниваем **$days.$week_day**. Так осуществляется проверка на неравенство нулю.

Parser это условие понимает так:

«Если существует **$days.$week_day**, то **{…}**, если нет, то вывести пустую ячейку таблицы серого цвета»

Основная часть работы выполнена. Осталось только выделить текущее число. Решается это использованием еще одного **if**, где условием задается сравнение текущего значения таблицы **days** с текущим числом (**$days.$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>
}
```

Обратите внимание на то, что здесь мы проверяем на равенство два числа, поэтому используем оператор **==** вместо **eq**, используемый для проверки равенства строк.

Еще раз посмотрим на общую структуру формирования календаря:

```
#перебираем строки таблицы с календарем
^days.menu{

#перебираем столбцы таблицы с календарем
        ^for[week_day](0;6){
                ^if($days.week_day){
                        ^if($month.$week_day==$date.day){
                                число на другом фоне жирным шрифтом
                        }{
                                число
                        }
                }{
                        пустой серый столбец
                }
        }
}
```

Эту конструкцию простой не назовешь. Здесь используются вложенные друг в друга конструкции. Однако она позволяет понять возможность комбинирования различных средств Parser для решения конкретной задачи. Есть более элегантное решение – вынести проверку текущей даты и ее раскрашивание нужным цветом в отдельную функцию, которую и вызывать внутри цикла. Похожее решение мы использовали во втором уроке. Это позволит разгрузить блок и сделать его более читабельным. Но поскольку в данном примере мы хотели показать вам возможность комбинирования нескольких логических структур, то оставляем эту возможность вам в качестве задания.

Если хотите убедиться в работоспособности этого модуля, создайте в разделе `/news/` файл `test.html` и в нем наберите одну единственную строчку **^calendar[]**. Теперь откройте этот файл из браузера и полюбуйтесь результатом своих трудов.

Подведем итоги третьего урока.

Что мы сделали: описали функцию, формирующую календарь на текущий месяц.

Что узнали:

- файл `auto.p` может содержаться не только в корневом каталоге сайта, но и в любом другом, при этом функции, в нем определенные, явно доступны только внутри этого каталога
- переменная-хеш – это массив, нужный для построения ассоциативной связи одних объектов с другими. В нашем случае объектами выступали строки
- статический метод **`calendar`** создает таблицу с календарем на текущий месяц
- цикл **`for`** позволяет последовательно выполнить определенные действия заданное количество раз

Что надо запомнить:

- наряду с методами объектов, создаваемых с помощью конструкторов класса, существуют статические методы. Вы можете непосредственно обращаться к этим методам без предварительного использования конструктора для создания объекта
- в циклах **`for`** можно обращаться к счетчику как к переменной по имени и получать его текущее значение

Поскольку код становится все объемнее, неплохо бы начать снабжать его комментариями, чтобы потом было легче разбираться. В Parser комментариями считается любая строка, начинающаяся со знака **#**. До сих пор мы не пользовались этим, но в дальнейшем нам пригодится комментирование кода. Следующая строка – пример комментария:

```
# весь этот текст Parser проигнорирует – это комментарий !!!!
```

Обязательно комментируйте свой код! В идеале он должен быть самодокументирующимся, а человек, читающий его, должен сразу же понимать о чем идет речь, что из чего следует и т.д. Если этого не сделать, то спустя какое-то время вспомнить что делает та или иная функция станет очень сложно даже вам самим, не говоря уже про остальных. Помните об этом!

Что будем делать дальше: на следующем уроке мы научим созданный нами календарь ставить ссылки на числа месяца. А самое главное, мы перейдем к работе с формами и базами данных для создания полноценного новостного раздела.

Урок 4. Шаг второй — переходим к работе с БД

Самое главное – не пугайтесь названия урока, даже если вы никогда не работали с базами данных. Без них просто невозможно построить гибкий, легко настраиваемый сайт. Отказ от использования БД не дает никаких преимуществ разработчику, а наоборот, здорово уменьшает возможности по созданию сайта и быстрому динамическому изменению содержимого. Построение серьезного ресурса без БД – это как рыбалка без удилища: вроде бы и можно кого-то поймать, однако делать это крайне неудобно. Иными словами, если вы пока не умеете работать с БД – обязательно научитесь и активно используйте их в своих проектах. На этом закончим агитацию, будем считать, что мы вас убедили в необходимости использования БД.

Работать с БД на Parser очень удобно. В Parser встроена мощная система взаимодействия с различными СУБД. В настоящее время Parser может работать с MySQL, Oracle, PgSQL, а также с любой СУБД через драйверы ODBC (в т.ч. MS SQL, MS Access). Поскольку исходные коды Parser3 являются открытыми, возможно добавление поддержки любых других знакомых вам СУБД после создания соответствующего драйвера. При этом работа с ними не требует практически никаких дополнительных знаний собственно Parser. Все, что нужно – это подключится к выбранной СУБД и работать, используя SQL в объеме и формате, поддерживаемом СУБД. При передаче SQL-запросов Parser может только заменить апострофы соответствующей конструкцией в зависимости от СУБД, для «защиты от дурака», а все остальное передается, как есть.

Существует еще специальная конструкция для записи больших строковых литералов. Oracle, PgSQL и,

возможно, какие-то серверы, драйверы к которым будут написаны в будущем, не умеют работать с большими строковыми литералами. Если передаваемая, например, из формы, строка будет содержать больше 2000 [Oracle 7.x] или 4000 [Oracle 8.x] букв, сервер выдаст ошибку «слишком длинный литерал». Если пытаться хитрить, комбинируя «2000букв» + «2000букв», то также будет выдана ошибка «слишком длинная сумма». Для хранения таких конструкций используется тип данных CLOB[Oracle] и OID[PgSQL], а для того, чтобы SQL команды были максимально просты, при записи таких строк необходимо лишь добавить управляющий комментарий, который драйвер соответствующего SQL-сервера соответствующим образом обработает:

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

*Слово text в записи /**text**/ – это имя колонки, в которую предназначен следующий за этой конструкцией строковый литерал. Пробелы здесь недопустимы!*

Со всеми возможностями Parser по работе с различными СУБД в рамках данного урока мы знакомиться, конечно же, не будем. Остановимся на MySQL. Почему именно на ней? Прежде всего потому, что она очень распространена, и многие веб-проекты используют именно ее. Кроме того, практически все компании, занимающиеся сетевым хостингом, предоставляют клиентам возможность работы с этой СУБД. Ну и, несомненно, немаловажный фактор – она бесплатна, доступна и легка в освоении.

Давайте определимся, что будем хранить в базе данных. Очевидный ответ : будем хранить новости. Причем таблица СУБД с новостями должна содержать такие поля: уникальный номер новости в базе, который будет формироваться автоматически СУБД, дата внесения новости в базу, по которой мы будем проводить выборку новостей за конкретное число, заголовок новости и собственно ее текст. Просто, без тонкостей и премудростей, однако это эффективно работает.

Есть еще один вопрос, с которым нужно определиться: каким образом новости будут попадать в базу? Можно их заносить и из командной строки СУБД, но это не удобно. В случае если вы предполагаете строить сайт для intranet, есть вариант использовать в качестве СУБД или средства доступа к БД широко распространенную MS Access. Привычный GUI и copy+paste обеспечат вам любовь многих коллег по работе на долгие годы. Для маленьких баз данных это решение может оказаться оптимальным. Мы же предлагаем решение, ориентированное на Internet – создание на сайте раздела администрирования с формой для ввода новостей прямо из браузера.

Постановка задачи закончена, переходим к ее практическому решению. Для дальнейшей работы вам потребуется установленная СУБД MySQL, без которой рассматриваемый здесь пример просто не будет работать.

Прежде всего, средствами MySQL создаем новую базу данных с именем **p3test**, содержащую одну единственную таблицу news с полями **id, date, header, body**:

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

Теперь создадим раздел администрирования, который даст возможность заполнить созданную базу данных новостями. Для этого в корневом каталоге сайта создаем каталог `admin`, а в ней `index.html`, в который пишем следующее:

```
@greeting[]
Администрирование новостей

@body_additional[]
Добавление новостей

@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>

#начало обработки
^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')
            }
            …сообщение добавлено
      }
}{
      …для добавления новости необходимо заполнить все поля формы
}
</center>
```

Также требуется в корневом файле `auto.p` перед методом **main** добавить метод **auto**. Этот метод используется для инициализации глобальных переменных, т.е. переменных, которые будут доступны на всех страницах сайта. В нем мы зададим строку подключения к базе данных, о которой чуть позже.

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

Как видите, структура этой страницы полностью соответствует придуманной нами структуре страниц сайта. Все элементы, как то: приветствие, две части **body**, **footer** и **header** присутствуют. Кстати, вы помните, откуда на этой странице появятся **header** и **footer**? Правильно, из функции **main** корневого `auto.p`).

Незнакомые конструкции только в основной части. Давайте с ней разберемся. В начале обычная HTML форма, с подстановкой текущей даты в поле **date** как значения по умолчанию. Сделано это исключительно для удобства пользователей.

Легкое недоумение может вызвать запись:
`${now.year}-${now.month}-${now.day}`

Фигурные скобки здесь используются для того, чтобы получить строку вида «2001-11-06» (в таком формате мы собираемся хранить дату новости в БД). Если скобки не ставить, то Parser выдаст ошибку при обработке этого кода, поскольку не сможет понять, что нужно делать. Для него «-» будет частью имени. Запомните, если вам нужно четко отделить имя переменной от следующей за ним буквы, скажем «-», как в нашем случае, нужно записать:

`${имя_переменной}-`

И в результате вы получите:

значение_переменной-

Обязательно прочитайте страницу, посвященную правилам составления имен.

Лучшим решением этой проблемы было бы использовать в этом месте конструкцию `^date.sql-string[]`. Попробуйте самостоятельно доработать этот пример, пользуясь справочником. Если не получится – не расстраивайтесь, на следующем уроке мы покажем, как это сделать.

Продолжим. Если вам уже доводилось работать с формами, то вы знаете, что формы передают введенные в них значения на дальнейшую обработку каким-либо скриптам. Здесь обработчиком данных формы будет сама страница, содержащая эту форму. Никаких дополнительных скриптов нам не понадобится.

После закрывающего тега **</form>** начинается блок обработки. Вначале с помощью **if** мы проверяем поля формы на пустоту. Этого можно опять же не делать, но мы хотим создать нечто большее, чем учебный экспонат без практического применения. Для того чтобы осуществить проверку, необходимо получить значения полей этой формы. В Parser это реализуется через статические переменные (поля). Мы просто обращаемся к полям формы, как к статическим полям:

```
$form:поле_формы
```

Полученные таким образом значения полей мы и будем проверять на пустоту с помощью оператора **def** и логического «И»(**&&**). Мы уже проверяли объект на существование в третьем уроке, но там был опущен оператор **def**, поскольку проверяли на пустоту таблицу. Как вы помните, таблица в выражении имеет числовое значение, равное числу строк в ней, поэтому любая непустая таблица считается определенной. Здесь же необходимо использовать **def**, как и в случае проверки на **def** других объектов. Если в поле ничего не было введено, то значение **$form:поле_формы** будет считаться неопределенным (undefined). После того, как все значения полей заполнены, необходимо поместить их в базу данных. Для этого нужно сначала подключиться к базе данных, а затем выполнить запрос SQL для вставки данных в таблицу. Посмотрите, как мы это делаем:

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

Удобство Parser при работе с базами данных состоит в том, что он, за исключением редких случаев, не требует изучать какие-либо дополнительные операторы, кроме тех, которые предусмотрены в самой СУБД. Сессия работы с базой данных находится внутри оператора connect, общий синтаксис которого:

```
^connect[протокол://строка соединения]{методы, передающие запросы SQL}
```

Для MySQL это запишется так:

```
^connect[mysql://пользователь:пароль@хост/база_данных]{…}
```

В фигурных скобках помещаются методы, выполняющие SQL-запросы. При этом любой запрос может возвратить или не возвратить результат (например, в нашем случае нужно просто добавить запись в таблицу БД, не возвращая результат), поэтому Parser предусматривает различные конструкции для создания этих двух типов SQL-запросов. В нашем случае запрос записывается как:

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

Кстати, это статический метод класса **void**, помните про двоеточие?

То, что здесь не выделено цветом, является командами SQL. Ничего сложного здесь нет. Если вы

знакомы с SQL, то больше ничего и не потребуется, а если почему-то пока не знакомы, мы вновь рекомендуем его изучить. Вам это многократно пригодится в дальнейшем. Время, потраченное на это изучение, не пропадет даром.

Оцените все изящество этого варианта взаимодействия с базой данных – Parser обеспечивает прозрачный доступ к СУБД и, за редким исключением, не требует каких-либо дополнительных знаний. При этом, как вы видите, мы можем помещать в запросы SQL еще и данные из нашей формы, пользуясь конструкциями Parser. Возможности этого симбиоза просто безграничны. СУБД решает все задачи, связанные с обработкой данных (она ведь именно для этого и предназначена и очень неплохо с этим справляется), а нам остается только воспользоваться результатами ее работы. Все аналогично и с другими СУБД, с которыми вы можете столкнуться в своей работе.

Теперь у нас есть форма, позволяющая помещать записи в нашу БД. Занесите в нее несколько записей. А теперь давайте их оттуда извлекать, но перед этим неплохо бы немного доработать функцию **calendar**, созданную на предыдущем уроке. Нужно, чтобы в календаре ставились ссылки на дни месяца, а выбранный день передавался как поле формы. Тогда по числам-ссылкам в календаре пользователь будет попадать в архив новостей за выбранный день. Модернизация эта несложная, просто добавим немного HTML в `auto.p` раздела news: **$days.$week_day** в коде **if** обнесем ссылками таким образом:

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

В результате мы получаем возможность использовать наш календарь в качестве меню доступа к новостям за определенный день.

Теперь займемся `/news/index.html`. В него заносим такой код:

```
@greeting[]
Страница новостей,  заходите чаще!

@body_additional[]
<center>Архив новостей за текущий месяц:</center>
<br />
^calendar[]

@body_main[]
$now[^date::now[]]
<b><h1>НОВОСТИ</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 />]
      }{
            За указанный период новостей не найдено.
      }
}
```

Структура обычная. В дополнительной части **body** помещаем меню-календарь вызовом **^calendar[]**

(напомним, что эта функция определена в `auto.p` раздела `news`). Основа информационной части страницы – выборка из базы данных новостей за тот день, по которому щелкнул пользователь (условие **where** в SQL-запросе). Это второй вариант SQL-запроса, при котором результат возвращается. Обратите внимание, здесь результатом запроса будет таблица, с которой в дальнейшем мы будем работать. Поэтому необходимо создать объект класса **table.**

Познакомимся с еще одним конструктором класса **table** – конструктором на базе SQL-запроса. Его логика абсолютно аналогична работе конструктора **^table::load[]**, только источником данных для таблицы является не текстовый файл, как в случае с пунктами меню, а результат работы SQL-запроса – выборка из базы данных:

```
$переменная[^table::sql{код SQL запроса}]
```

Воспользоваться этим конструктором вы можете только внутри оператора **^connect[]**, то есть когда имеется установленное соединение с базой данных, поскольку обработкой SQL-запросов занимается сама СУБД. Результатом будет именованная таблица, имена столбцов которой совпадают с заголовками, возвращаемыми SQL-сервером в ответ на запрос.

*Небольшое отступление. При создании SQL-запросов следует избегать использования конструкций вида select * from… поскольку для постороннего человека, не знающего структуру таблицы, к которой происходит обращение, невозможно понять, какие данные вернутся из БД. Подобные конструкции можно использовать только для отладочных целей, а в окончательном коде лучше всегда явно указывать названия полей таблиц, из которых делается выборка данных.*

Остальная часть кода уже не должна вызывать вопросов: **if** обрабатывает ситуацию, когда поле **day** (выбранный пользователем день на календаре, который передается из функции **calendar**) не определено, то есть человек пришел из другого раздела сайта через меню навигации. Если поле формы **day** определено (**def**), то используется день, переданный посетителем, в противном случае используем текущее число. Далее соединяемся с БД, также как мы это делали, когда добавляли новости, создаем таблицу **$news**, в которую заносим новости за запрошенный день (результат SQL-запроса), после чего с помощью метода **menu** последовательно перебираем строки таблицы **news** и выводим новости, обращаясь к ее полям. Все понятно и знакомо, кроме одного вспомогательного оператора, который служит для специфического вывода текста новости:

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

Отвлекитесь немного и внимательно прочитайте раздел справочника, посвященный операторам taint и untaint*, где подробно описана логика их работы. Это очень важные операторы и вы наверняка столкнетесь с необходимостью их использования. К тому же большой объем работы по обработке данных Parser делает самостоятельно, она не видна на первый взгляд, но понимать логику действий необходимо.*

Прочитали? Теперь продолжим. Зачем он нужен здесь? У нас есть страница для администрирования новостей, и мы хотим разрешить использование тегов HTML в записях. По умолчанию это запрещено, чтобы посторонний человек не мог внести Java-скрипт, например, перенаправляющий пользователя на другой сайт. Как это сделать? Да очень просто: достаточно выборку записей из таблицы преобразовать с помощью оператора **untaint:**

```
^untaint{текст новости}
```

В нашем случае используется значение по умолчанию **[as-is]**, которое означает, что данные будут выведены так, как они есть в базе. Мы можем позволить себе поступить так, поскольку изначально не предполагается доступ обычных пользователей к разделу администрирования, через который добавляются новости.

Теперь можно немного расслабиться – новостной блок нашего сайта завершен. Мы можем добавлять новости и получать их выборку за указанный пользователем день. На этом четвертый урок будем считать оконченным, хотя есть некоторые детали, которые можно доработать, а именно: научить календарь не ставить ссылки на дни после текущего, выводить в заголовке информационной части дату, за которую показываются новости, да и просто реализовать возможность доступа к новостям не только за текущий месяц, но и за предыдущие. Однако это уже задание вам. Знаний, полученных на

предыдущих уроках вполне достаточно, чтобы доработать этот пример под свои требования и желания. Творите!

Подведем итоги четвертого урока.

Что мы сделали: создали раздел администрирования для добавления новостей. Модернизировали функцию, формирующую календарь на текущий месяц. Наполнили раздел новостей данными из БД на основе запроса пользователей либо по умолчанию за текущую дату.

Что узнали:
- механизм взаимодействия Parser с СУБД MySQL
- как осуществлять различные SQL-запросы к БД (статический метод **`sql`** класса **`void`** и конструктор **`sql`** класса **`table`**)
- оператор **`untaint`**

Что надо запомнить: работа с базами данных в Parser осуществляется легко и понятно, нужно только изучить работу самой СУБД. Не отказывайтесь от использования БД в своих работах.

Что будем делать дальше: с разделом новостей закончено, переходим к созданию гостевой книги для нашего сайта, чтобы можно было определять, какова популярность у пользователей созданного совместными усилиями творения.

Урок 5. Пользовательские классы Parser

Во всех предыдущих уроках мы оперировали классами и объектами, предопределенными в Parser. Например, есть уже хорошо знакомый нам класс table, у него существуют свои методы, которые мы широко использовали. Полный список всех методов этого класса можно посмотреть в справочнике. Однако ограничение разработчиков рамками только базовых классов в какой-то момент может стать сдерживающим фактором. «Неспособность не есть благодетель, а есть бессилие...», поэтому для удовлетворения всех потребностей пользователей необходимо иметь возможность создавать собственные (пользовательские) классы объектов со своими методами. На этом уроке мы и создадим средствами Parser новый класс объектов со своими собственными методами.

Объектом, в принципе, может быть все что угодно: форум, гостевая книга, различные разделы и даже целый сайт. Здесь мы подошли к очередному уровню структуризации – на уровне объектов, а не методов. Как мы поступали раньше? Мы выделяли отдельные куски кода в методы и вызывали их, когда они были необходимы. Но в качестве отдельных блоков сайта было бы намного удобнее использовать собственные объекты: для получения форума создаем объект класса «форум», после чего используем его методы, например «удалить сообщение», «показать все сообщения» и поля, например, «количество сообщений». При этом обеспечивается модульный подход на качественно ином уровне, чем простое использование функций. В единую сущность собираются код и данные (методы и поля). Разрозненные ранее методы и переменные объединяются воедино и используются применительно к конкретному объекту – «форуму». В терминах объектно-ориентированного программирования это называется инкапсуляцией. Кроме того, один раз создав класс форум, его объекты можно использовать в различных проектах, абсолютно ничего не меняя.

Работу с пользовательским классом мы покажем на примере гостевой книги, а для начала еще раз напомним порядок работы с объектами в Parser. Сначала необходимо создать объект определенного класса с помощью конструктора, после чего можно вызывать методы объектов этого класса и использовать поля созданного объекта. В случае пользовательского класса ничего не меняется, порядок тот же.

Как всегда начнем с определения того, что нам нужно сделать. Правильная постановка задачи – уже половина успеха. Перед началом создания класса нужно точно определить, что будет делать объект класса, то есть решить, какие у него будут методы. Предположим, что нашими методами будут: показ сообщений гостевой книги, показ формы для добавления записи, а также метод, добавляющий сообщение в гостевую книгу. Хранить сообщения будем в базе данных, так же как и новости.

Если с методами класса все более или менее ясно, то некоторая неясность остается с конструктором

класса, что же он будет делать? Опираясь на прошлые уроки, мы помним, что для того, чтобы начать работать с объектом класса, его необходимо создать, или проинициализировать. Давайте с помощью конструктора будем получать таблицу с сообщениями, а затем в методе показа сообщений будем пользоваться данными этой таблицы.

С целями определились, займемся реализацией. Прежде всего, создадим таблицу **gbook** в базе данных **p3test**:

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

Теперь необходимо познакомиться еще с несколькими понятиями Parser – классом **MAIN** и наследованием. Как уже говорилось, класс является объединяющим понятием для объектов, их методов и полей. Класс **MAIN** объединяет в себя методы и переменные, описанные пользователями в файлах **auto.p** и запрашиваемом документе (например, **index.html**). Каждый следующий уровень вложенности наследует методы, описанные в **auto.p** предыдущих уровней каталога. Эти методы, а также методы, описанные в запрашиваемом документе, становятся статическими функциями класса **MAIN**, а все переменные, созданные в **auto.p** вверх по каталогам и в коде запрошенной страницы, – статическими полями класса **MAIN**.

Для пояснения рассмотрим следующую структуру каталогов:

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

При загрузке страницы `index.html` из каталога `/news/details/` класс **MAIN** будет динамически «собран» из методов, описанных в корневом файле `auto.p`, а также в файлах `auto.p` разделов `/news/` и `/news/details/`. Методы, описанные в `auto.p` раздела `/contacts/`, будут недоступны для страниц из раздела `/news/details/`.

Как «собирается» класс **MAIN** теперь понятно, но, прежде чем приступить к созданию собственного класса, необходимо также выяснить, как из *пользовательского* класса вызывать методы и получать значения переменных класса **MAIN**. Методы класса **MAIN** вызываются как статические функции:

^MAIN:метод[],

а переменные являются статическими полями класса **MAIN**. К ним можно получить доступ так же, как к любым другим статическим полям:

$MAIN:поле

Теперь переходим к практике. В корневой **auto.p** добавляем еще один метод, с помощью которого можно будет не только соединяться с БД, но и передавать ей произвольный SQL-запрос:

@dbconnect[code]

```
^connect[$connect_string]{$code}
# connect_string определяется в методе @auto[]
# $connect_string[mysql://root@localhost/p3test]
```

Метод вынесен в корневой **auto.p** для того, чтобы впоследствии можно было бы легко подключаться к серверу баз данных с любой страницы, поскольку методы из корневого **auto.p** будут наследоваться всегда. Обратите внимание на то, что здесь используется передача методу параметра. В нашем случае он один – **code**, с его помощью мы и будем передавать код, выполняющий SQL-запросы. Параметров может быть и несколько, в этом случае они указываются через точку с запятой.

Дальше в каталоге нашего сайта создаем подкаталог, в которой будем хранить файл с нашим классом, например, **classes**. Далее в этом каталоге создаем файл `gbook.p` (пользовательские файлы мы предлагаем хранить в файлах с расширением имени `.p`) и в него заносим следующий код:

```
@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){
                              $messages.email
                        }{
                              Нет электронного адреса
                        }</b>
                  </td>
                  <td align="right">$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>]
}{
      Гостевая книга пуста.
}

@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>Поле автор обязательно для заполнения</center>
      }
}
```

Посмотрите на код. В первой строке мы говорим, что в этом файле будем хранить пользовательский класс:

@CLASS

В том случае если в качестве родительского выступает другой пользовательский класс, его необходимо подключить, а также объявить базовым. Получится такая конструкция:

@CLASS
имя класса

@USE
файл родительского класса

@BASE
имя родительского класса

Следующей строкой пишем имя нашего класса **gbook**. Необходимо помнить, что Parser чувствителен к регистру букв в именах, поэтому классы **gbook** и **Gbook** являются разными. При этом имя не обязательно должно совпадать с именем файла, в котором храниться пользовательский класс, более того, может быть набрано русскими буквами.

Дальше определяются методы нашего нового класса. Делается это точно так же, как и определение обычных методов, которые мы создавали на предыдущих уроках.

Первый метод **load** будет конструктором нашего класса. При этом надо иметь в виду, что задача конструктора – создать объект. Кроме этого, он может объявить переменные и присвоить им значения. Эти переменные станут полями объекта пользовательского класса. В нашем случае мы при помощи конструктора **sql** класса **table** создаем нужную таблицу. Обратите внимание, что в методах нового класса мы свободно пользуемся методами системных классов и методом **dbconnect** класса **MAIN**:

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

Как уже говорилось выше, поскольку мы находимся за пределами класса **MAIN**, для использования

методов этого класса перед именем необходимо указать класс, к которому эти методы/поля относятся. Делается это таким образом:

```
^имя_класса:метод[параметры]
$имя_класса:переменная
```

В случае, если мы захотим использовать методы/поля другого пользовательского класса, а не класса **MAIN**, необходимо в начале кода выполнять инструкцию:

```
@USE
путь к файлу, описывающему класс
```

Она позволяет использовать модуль, определенный в другом файле. О работе Parser с путями к файлам, рассказано в приложении 1.

Итак, наш новый конструктор будет создавать таблицу с сообщениями, подключаясь к указанной БД. С конструктором разобрались, начинаем описание собственно методов нового класса. Метод **show_messages** нашего класса выводит на экран сообщения из таблицы **gb**, созданной в методе **load**. Строки перебираются при помощи метода **menu** класса **table**. Все знакомо, ничего нового нет и в других методах:

show_form – выводит на экран форму для добавления нового сообщения гостевой книги

test_and_post_message – проверяет, нажата ли кнопка **post**, заполнено ли поле author и, если все условия выполнены, заносит сообщение в базу данных, используя все тот же метод **dbconnect**, определенный в классе **MAIN**

На этом создание пользовательского класса, описывающего методы объектов класса **gbook**, завершено. Его осталось только подключить для использования на нашем сайте. Перед нами стоит задача сообщить Parser, что на некоторой странице мы собираемся использовать свой класс. Для этого в файле **index.html** каталога **gbook** в первой строке напишем следующее:

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

Теперь на этой странице можно создать объект класса **gbook** и использовать затем его методы. Сделаем это в основной информационной части:

```
@body_main[]
Гостевая книга тестового сайта<br />
<hr />

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

# и конечно же не забываем про остальные части
@greeting[]
Оставьте свой след:

@body_additional[]
Нам пишут...
```

Здесь мы уже работаем с объектом созданного пользовательского класса, как с любым другим объектом: создаем его при помощи конструктора класса и вызываем методы, определенные в новом классе. Посмотрите, насколько изящным получилось наше решение. Читабельность кода очевидна и, глядя на этот фрагмент, сразу понятно, что он делает. Все, что относится к гостевой книге, находится в отдельном файле, где описано все, что можно с ней сделать. Если нам понадобится новый метод для работы с гостевой книгой, нужно просто дописать его в файл `gbook.p`. Все очень легко модернизируется, к тому же сразу понятно, где необходимо вносить изменения, если они вдруг понадобились.

В заключение хочется заметить, что изящнее было бы вынести методы вроде **dbconnect** из класса **MAIN** в отдельный класс. Это позволило бы не перегружать класс **MAIN**, улучшилась бы читаемость кода, а также легче стало бы ориентироваться в проекте. Там, где эти нужны методы этого класса, его можно было бы подключать с помощью **@USE**.

Подведем итоги пятого урока.

Что мы сделали: создали свой собственный класс и на основе объекта этого класса создали гостевую книгу на нашем сайте.

Что узнали:

- класс **MAIN**
- создание пользовательского класса
- как передавать параметры методам

Что надо запомнить: класс является «высшей формой» структуризации, поэтому необходимо стремиться к выделению в отдельные классы кубиков, из которых вы будете строить ваши сайты. Это позволяет достичь максимальной понятности логики работы проектов и дает невероятные удобства в дальнейшей работе.

Что делать дальше: на этом создание нашего учебного сайта можно считать завершенным. Конечно, он далек от идеала и использовать его в таком виде не стоит. Для реального использования необходимо выполнить целый ряд доработок: модифицировать календарь в разделе новостей, в гостевой книге организовать проверку введенных данных на корректность и т.д. Но мы и не ставили задачу сделать полнофункциональный проект. Мы просто хотели показать, что работать с Parser совсем не сложно, а производительность от его использования возрастает существенно. Теперь вы сами обладаете всеми необходимыми базовыми знаниями для полноценной работы с Parser, остается только совершенствовать их и приобретать опыт.

Удачи!

Урок 6. Работаем с XML

```
<?xml version="1.0" encoding="windows-1251" ?>
<article>
    <author id="1" />
    <title>Урок 6. Работаем с XML</title>
    <body>
        <para>Представьте, что вам позволено придумывать любые теги
              с любыми атрибутами. То есть вы сами можете определять,
              что означает тот или иной выдуманный вами тег или атрибут.</para>
        <para>Такой код будет содержать данные, …</para>
    </body>
    <links>
        <link href="http://www.parser.ru/docs/lang/xdocclass.htm">Класс
xdoc</link>
        <link href="http://www.parser.ru/docs/lang/xnodeclass.htm">Класс
xnode</link>
    </links>
</article>
```

…но не их форматирование. Подготовкой данных может заняться один человек, а форматированием другой. Им достаточно договориться об используемых тегах и можно приступать к работе… одновременно.

Идея эта не нова, существовали многочисленные библиотеки обработки шаблонов, а многие создавали собственные. Библиотеки были несовместимы между собой, зависели от используемых средств скриптования, порождая разобщенность разработчиков и необходимость тратить силы на изучение

очередной библиотеки вместо того, чтобы заняться делом.

Однако прогресс не стоит на месте, и сейчас мы имеем не зависящие от средства скриптования стандарты **XML** и **XSLT**, позволяющие нам реализовать то, что мы только что представляли. А также стандарты **DOM** и **XPath**, открывающие для нас еще больше возможностей.
Parser полностью поддерживает все эти стандарты.

Сейчас откройте выбранную вами вчера в книжном магазине книгу, описывающую XML и XSLT. Используйте ее как справочник.

Посмотрим, как можно приведенную статью преобразовать из XML в HTML.
Запишем текст из начала статьи в файл `article.xml`
И создадим файл `article.xsl`, в котором определим выдуманные нами теги:

```
<?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">
    Ссылки по теме:
    <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>
```

Данные и шаблон преобразования готовы. Создаем `article.html`, в который заносим следующий код:

```
# входной xdoc документ
$sourceDoc[^xdoc::load[article.xml]]

# преобразование xdoc документа шаблоном article.xsl
$transformedDoc[^sourceDoc.transform[article.xsl]]

# выдача результата в HTML виде
^transformedDoc.string[
    $.method[html]
]
```

Первой строкой мы загружаем XML-файл, получая в **`sourceDoc`** его DOM-представление.

Конструкция похожа на загрузку таблицы, помните **^table::load[…]**? Только в этот раз мы загружаем не таблицу (получая объект класса **table**), а XML-документ (получаем объект класса **xdoc**).

Второй строкой мы преобразуем входной документ по шаблону `article.xsl`.
Из входного документа получаем выходной, применяя XSLT преобразование, описанное в шаблоне.

Последней строкой мы выдаем пользователю текст выходного документа в HTML формате (параметр **method** со значением **html**).
Здесь можно задать все параметры, допустимые для тега **<xsl:output … />**.
Рекомендуем также задать параметр "без отступов" (параметр ***indent*** *со значением* ***no***:
$.indent[no] *), чтобы избежать известной проблемы с пустым местом перед* **</td>**.

Обратившись теперь к этой странице, пользователь получит результат преобразования:

```
<html>
<head><title>Урок 6. Работаем с XML</title></head>
<body>
<p>Представьте, что вам позволено придумывать любые теги
с любыми атрибутами. То есть вы сами можете определять,
что означает тот или иной выдуманный вами тег или атрибут.
</p>
<p>Такой код будет содержать данные, …
</p>
Ссылки по теме:
<ul>
<li><a href="http://www.parser.ru/docs/xdocclass.htm">Класс xdoc</a></li>
<li><a href="http://www.parser.ru/docs/xnodeclass.htm">Класс xnode</a></li>
</ul>
</body>
</html>
```

Как вы заметили, тег **<author … />** никак не был определен, как следствие, информация об авторе статьи не появилась в выходном HTML. Со временем, когда вы решите где и как будете хранить и показывать данные об авторах, достаточно будет дополнить шаблон — исправлять данные статей не потребуется.

Внимание: если вы не хотите, чтобы пользователи вашего сервера имели доступ к `.xml` *и* `.xsl` *файлам, храните эти файлы вне веб-пространства*
*(***^xdoc::create[/../directory_outside_of_web_space/article.xml]***), или запретите к ним доступ средствами веб-сервера (пример запрета доступа к* `.p` *файлам здесь: «Установка Parser на веб-сервер Apache. CGI скрипт»).*

Подведем итоги шестого урока.

Что мы сделали: создали заготовку для публикации информации в XML формате с последующим XSLT преобразованием в HTML.

Что узнали:

- класс **xdoc**
- как загружать XML, делать XSLT преобразования, выводить объекты класса **xdoc** в виде HTML

Что надо запомнить: что надо купить книжку по XML, XSLT и DOM.

Что делать дальше: читать эту книжку и экспериментировать с примерами из нее, радуясь тому, что на свете есть хорошие стандарты. А также почитать о **postprocess**, и придумать, как его приспособить, чтобы обращение к XML-файлу вызывало его преобразование в HTML.

Конструкции языка Parser3

Переменные

Переменные могут хранить данные следующих типов:
- строка (string);
- число (int/double);
- истина/ложь;
- хеш (ассоциативный массив);
- класс объектов;
- объект класса (в т.ч. пользовательского);
- код;
- выражение.

Для использования переменных не требуется их заранее объявлять.

В зависимости от того, что будет содержать переменная, для присвоения ей значения используются различные типы скобок:

$имя_переменной[строка]	переменной присваивается строковое значение (объект класса string) или произвольный объект некоторого класса
$имя_переменной(выражение)	переменной присваивается число или результат математического выражения
$имя_переменной{код}	переменной присваивается фрагмент кода, который будет выполнен при обращении к переменной

Для получения значения переменных используется обращение к имени переменной:

$имя_переменной – получение значения переменной

Примеры

Код	*Результат*
`$string[2+2]` `$string`	**2+2**
`$number(2*2)` `$number`	**4**

`$i(0)`
`$code{$i}`
`$i(1)`
`$code`
	1

`$i(0)`
`$string[$i]`
`$i(1)`
`$string`
	0

В качестве части имени может быть использовано...

...значение другой переменной:
```
$superman[value of superman variable]
$part[man]
$super$part
```
Возвратит: **value of superman variable**

```
$name[picture]
${name}.gif
```
Возвратит строку **picture.gif**, а не значение поля **gif** объекта **picture**.

...результат работы кода:
```
$field.[b^eval(2+3)]
```
Возвратит значение поля **b5** объекта **field**.

Хеш (ассоциативный массив)

Хеш, или ассоциативный массив — позволяет хранить ассоциации между строковыми ключами и произвольными значениями. Создание хеша происходит автоматически при таком присваивании переменной значения или вызове метода:

```
$имя[
    $.ключ1[значение]
    $.ключ2[значение]
    . . .
    $.ключN[значение]
]
```

или

```
^метод[
    $.ключ1[значение]
    $.ключ2[значение]
    . . .
    $.ключN[значение]
]
```

Также можно создать пустой копию другого хеша, см. «Класс hash, create. Создание пустого и копирование хеша».

Получение значений ключей хеша:
```
$имя.ключ
```

Хеш позволяет создавать многомерные структуры, например, **hash of hash**, где значениями ключей хеша выступают другие хеши.

```
$имя[
    $.ключ1_уровня1[$.ключ1_уровня2[значение]]
    . . .
    $.ключN_уровня1[$.ключN_уровня2[значение]]
]
```

Объект класса

Создание объекта
```
^класс::конструктор[параметры]
```

Конструктор создает объект класса, наделяя его полями и методами класса. Параметры конструкторов подробно описаны в соответствующем разделе.
Примечание: результат работы конструктора — созданный объект, обычный результат работы метода игнорируется (никуда не попадает).

Вызов метода
```
^объект.метод[параметры]
```

Вызов метода класса, к которому принадлежит объект. Параметры конструкторов подробно описаны в соответствующем разделе.

Если не указан объект, то конструкция является вызовом метода текущего класса (если у класса нет метода с таким именем, будет вызван метод базового класса) или оператора. При совпадении имен вызывается оператор.

Методы бывают статические и динамические.

Динамический метод — код выполняется в контексте объекта (экземпляра класса).

Статический метод — код выполняется в контексте самого класса, то есть метод работает не с конкретным объектом класса, а со всем классом (например, классы **MAIN**, **math**, **mail**)

Значение поля объекта

```
$объект.поле
```

Получение значения поля объекта.

Получение полей объекта в виде хеша *[3.4.0]*

```
$хеш[^hash::create[$объект]]
```

Будет создан хеш со всеми полями объекта.

Системное поле объекта: CLASS

$объект.CLASS — хранит ссылку на класс объекта.

Это необходимо при задании контекста компиляции кода (см. «process. Компиляция и исполнение строки».

Системное поле класса: CLASS_NAME *[3.2.2]*

$объект.CLASS_NAME — хранит имя класса объекта.

Пример

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

Выведет 'string'.

Статические поля и методы

Вызов статического метода

```
^класс:метод[параметры]
```

Вызов статического метода класса.

Примечание: точно так же вызываются динамические методы родительского класса (см. Создание пользовательского класса*).*

Значение статического поля

```
$класс:поле
```

Получение значения статического поля класса.

Задание статического поля

```
$класс:поле[значение]
```

Задание значения статического поля класса.

Определяемые пользоватем классы и операторы

Файл в таком формате определяет пользовательский класс:

```
@CLASS
имя_класса

# необязательно
@USE
файл_с_родительским классом

# необязательно
@OPTIONS    [3.3.0]
locals
partial

# необязательно
# нельзя наследоваться от системных классов [3.4.0]
@BASE
имя_родительского_класса

# так рекомендуется называть метод-конструктор класса
@create[параметры]

# далее следуют определения методов класса
@method1[параметры]
...
```

Модуль можно подключить (см. «Подключение модулей») к произвольному файлу — там появится возможность использования определенного здесь класса.

Если происходит обращение к неизвестному классу, вызывается метод **autouse** класса **MAIN**, и имя класса передается единственным параметром этому методу. ***[3.4.0]***

Если не указать **@CLASS**, файл определит ряд дополнительных операторов.

Если определен метод...
@auto[]

...он будет выполнен автоматически при загрузке класса как статический метод (так называемый статический конструктор). Используется для инициализации статических полей (переменных) класса. *Примечание: результат работы метода игнорируется — никуда не попадает.*

У метода **auto** может быть объявлен параметр:
@auto[filespec]
В этом параметре Parser передаст полное имя файла, содержащего метод.

В Parser создаваемые классы наследуют методы классов, от которых были унаследованы. Унаследованные методы можно переопределить.

В том случае, когда в качестве родительского класса выступает другой пользовательский класс, необходимо подключить модуль, в котором он находится, а также объявить класс базовым (**@BASE**).

Для того, чтобы пользоваться методами и полями родительских классов, необходимо использовать следующие конструкции:

^класс:метод[параметры] – вызов метода родительского класса *(примечание: хотя такой синтаксис вызова метода и похож на синтаксис вызова статического метода, фактически, в случае динамического метода, происходит динамический вызов метода родительского класса)*, для обращения к своему

ближайшему родительскому классу (базовому классу) можно использовать конструкции **^BASE::конструктор[параметры]** и **^BASE:метод[параметры]**.

С помощью **@OPTIONS** можно определить дополнительное поведение класса. ***[3.3.0]***
Так, указанная опция **locals** автоматически объявит локальными все переменные во всех методах определяемого класса. Если она указана, то для обращения к полям объекта или класса необходимо пользоваться системной переменной self.

С помощью опции **partial** можно разрешить последующую подгрузку методов в класс. Если впоследствии будет сделан use файла, в котором указано такое-же имя класса и эта-же опция, то вместо создания нового класса с таким-же именем, описанные в подключаемом файле методы будут добавлены к ранее загруженному классу. Опция может быть удобна для условного добавления в класс громоздких и редкоиспользуемых методов. После создания класса с использованием данной опции возможно лишь добавление методов классу, но не изменение его родительского класса.

Работа с переменными в статических методах

Поиск значения переменной происходит в:
- в списке локальных переменных;
- в текущем классе или его родителях.

Запись значения переменной производится в уже имеющуюся переменную (см. область поиска выше), если таковая имеется. В противном случае создается новая переменная (поле) в текущем классе.

Работа с переменными в динамических методах

Поиск значения переменной происходит в:
- в списке локальных переменных;
- в текущем объекте и его классе;
- в родительских объектах и их классах.

Запись значения переменной производится в уже имеющуюся переменную (см. область поиска выше), если таковая имеется. В противном случае создается новая переменная (поле) в текущем объекте.

Примечание: старайтесь всячески избегать использования полей класса не из методов класса, кроме простейших случаев! По-возможности, общайтесь с объектом только через его методы.

Системное поле класса: CLASS

$имя_класса:CLASS – хранит ссылку на класс объекта.

Это удобно при задании контекста компиляции кода (см. «process. Компиляция и исполнение строки».

По этой ссылке также доступны статические поля класса, пример:

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

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

Этот код напечатает значение **$cookie:field**.

Системное поле класса: CLASS_NAME ***[3.2.2]***

$объект.CLASS_NAME – хранит имя класса объекта.

Пример

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

Выведет 'string'.

Методы и определяемые пользователем операторы

```
@имя[параметры]
тело

@имя[параметры][локальные;переменные]
тело
```

Метод, это блок кода, имеющий имя, принимающий параметры, и возвращающий результат. Имена параметров метода перечисляются через точку с запятой. Метод также может иметь локальные переменные, которые необходимо объявить в заголовке метода, после объявления параметров, имена разделяются точкой с запятой.

Локальные переменные видны в пределах оператора или метода, и изнутри вызываемых ими операторов и методов, см. ниже **$caller**.

При описании метода можно пользоваться не только параметрами или локальными переменными, а также любыми другими именами, при этом вы будете работать с полями класса, или полями объекта, в зависимости от того, как был вызван определенный вами метод, статически, или динамически.

В Parser вы можете расширить базовый набор операторов, операторами в Parser считаются методы класса MAIN.
Важно: операторы, это методы класса MAIN, но в отличие от методов других классов, их можно вызвать из любого класса просто по имени, т.е. можно писать **^include[…]**, *вместо громоздкого* **^MAIN:include[…]**.

Системная переменная: self

Все методы и операторы имеют локальную переменную **self**, она хранит ссылку на текущий объект, в статических методах хранит то же, что и **$CLASS**.

Пример:

```
@main[]
$a[Статическое поле ^$a класса MAIN]
^test[Параметр метода]

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

Выведет:

```
$a — Параметр метода
$self.a — Статическое поле $a класса MAIN
```

Системная переменная: result

Все методы и операторы имеют локальную переменную **result**. Если ей присвоить какое-то значение, то именно оно будет результатом выполнения метода. Значение переменной **result** можно считывать и использовать в вычислениях.

Пример:

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

@sum[a;b]
^eval($a+$b)
$result[Ничего не скажу!]
```

Здесь клиенту будет выдана строка **Ничего не скажу!**, а не результат сложения двух чисел.

Системная переменная: result, явное определение *[3.1.5]*

Если в методе явно объявить локальную переменную **result**, это укажет Parser, что нужно проигнорировать все пробельные символы и считать ошибочными любые не пробельные символы, если их явно не присваивают переменной **result**.

Пример:

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

Здесь клиенту будет выдано либо значение из колонки **$resultcol**, либо значение **$notfound**.
Важно: в приведенном примере **не будет** выдано ни одного символа перевода строки, пробела или табуляции.
Важно: попытка заменить $notfound текстом, написанным прямо в теле метода, приведен к ошибке. Чтобы Parser не воспринимал написанные в теле метода не пробельные символы как ошибку их нужно явно присваивать переменной result.

Системная переменная: caller

Все методы и операторы имеют локальную переменную **caller**, которая хранит «контекст вызова» метода или оператора.

Через нее можно:

- узнать, кто вызвал вызвавший описываемый метод или оператор, обратившись к **$caller.self**;
- считать — **$caller.считать**, или записать — **$caller.записать[значение]** переменную, как будто вы находитесь в том месте, откуда вызвали описываемый метод или оператор.

Например вам нужен оператор, похожий на системный **for**, но чем-то отличающийся от него. Вы можете написать его сами, воспользовавшись возможностью менять локальную переменную с именем, переданным вам, в **контексте вызова вашего оператора**.

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

Теперь такой вызов...

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

...напечатает «**1 3 5 7 9** », обратите внимание, что изменяется **локальная переменная метода somewhere.**

Примечание: возможность узнать контекст вызова удобна для задания контекста компиляции кода (см. «process. Компиляция и исполнение строки».

Системная переменная: locals, явное определение *[3.3.0]*

Если в методе явно объявить локальную переменную **locals**, это будет равносильно объявлению всех переменных, используемых в нем локальными.
Для обращения к переменных класса или объекта в этом случае необходимо использовать **self**.

Передача параметров

Параметры могут передаваться в разных скобках и, соответственно, будут по-разному обрабатываться:

(выражение)	— вычисление параметра происходит при каждом обращении к нему внутри вызова метода
[код]	— вычисление параметра происходит один раз перед вызовом метода
{код}	— вычисление происходит при каждом обращении к параметру внутри вызова метода

Пример на различие скобок, в которых передаются параметры:

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

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

Здесь хорошо видно, что в первом случае код был вычислен один раз перед вызовом метода **sum**, и методу передался результат кода – число 30. Во втором случае вычисление кода происходило при каждом обращении к параметру, поэтому результат менялся в зависимости от значения счетчика.

Параметров может быть сколь угодно много или не быть совсем. Если в однотипных скобках несколько параметров, то они могут отделяться друг от друга точкой с запятой. Допустимы любые комбинации различных типов параметров.

Например...

```
^if(условие){когда да;когда нет}
```

...эквивалентно...

```
^if(условие){когда да}{когда нет}
```

Свойства

```
@GET_имя[]
код, выдает значение

@SET_имя[value]
код, обрабатывает новое $value

@GET_DEFAULT[]    [3.3.0]
@GET_DEFAULT[имя] [3.3.0]
код, обрабатывающий обращения к несуществующему полю

@GET[]      [3.3.0]
@GET[тип обращения] [3.4.0]
код, обрабатывающий обращения к объекту/классу в определённых контекстах вызова
```

Можно определить default getter (**@GET_DEFAULT[]**) — метод, который будет вызываться при обращении к несуществующим полям. Имя поля, к которому пытались обратиться, передаётся единственным параметром этому методу.
*Важно: с этим методом нельзя работать как с обычным «**свойством**», при попытке написать $DEFAULT будет выдано сообщение об ошибке.*

У пользовательских классов можно определить специальное свойство **@GET[]**, которое будет

вызываться при обращении к классу/объекту этого класса в определённых контекстах вызова, например: в скалярном контексте, в выражении и т.п. Если у этого свойства определён параметр, то через него будет передан **тип обращения**, который может принимать одно из следующих значений: **def**, **expression**, **bool**, **double**, **hash**, **table** или **file**.

Примечание: при обычном присваивании вида $a[$b] метод @GET[] не вызывается.

Так названные методы задают «**свойство**», которым можно пользоваться, как обычной переменной:

пишем	*происходит*
$имя	**^GET_имя[]**
$имя[значение]	**^SET_имя[значение]**

Примечание: если не требуется возможность записи или чтения свойства, соответствующий метод можно не определять.
Важно: нельзя иметь и свойство и переменную с одним именем.

Пример: возраст и e-mail

Возьмем человека — хранить удобно дату рождения, а выводить частенько нужно возраст. Нужен e-mail, но можно позабыть проверить его на корректность.

Пусть людьми занимается класс, его свойства «возраст» и «e-mail» позволят спрятать ненужные детали, сделав код проще и нагляднее:

```
@USE
/person.p

@main[]
$person[^person::create[
    $.name[Василий Пупкин]
    $.birthday[^date::create(2000;8;5)]
]]
# можно менять, но значение проверят
$person.email[vasya@pupkin.ru]
$person.name ($person.email), возраст: $person.age<br />
```

Выведет:
**Василий Пупкин (vasya@pupkin.ru), возраст: 5
** (с ходом времени возраст будет увеличиваться)

При этом менять возраст человека нельзя:

```
# это вызовет ошибку!
$person.age(99)
```

Также нельзя присваивать свойству **email** некорректные значения:

```
# это вызовет ошибку!
$person.email[vasya#pupkin.ru]
```

Определение класса person

Чтобы вышеописанный пример сработал, нужно определить класс **person** и его свойства.
В корне веб-пространства в файл `person.p` поместим это:

```
@CLASS
person

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

# свойство «возраст»
@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)]
# числовое значение логического выражения: истина=1; ложь=0
$result(^if($birthday>$today)(0)($today.year - $birthday.year -
($today<$celebday)))

# свойство «e-mail»
@SET_email[value]
^if(!^Lib:isEmail[$value]){
    ^throw[email.invalid;Некорректный e-mail: '$value']
}
# имя переменной не должно совпадать с именем свойства!
$private_email[$value]

@GET_email[]
$private_email
```

*Примечание: метод **isEmail**, как и ряд других полезных методов и операторов можно скачать по следующему адресу: http://www.parser.ru/examples/lib/.*
Примечание: классы лучше помещать в отдельное удобное место и при подключении не указывать путь, см. **$CLASS_PATH**.

Пример: класс, расширяющий функционал системного класса table

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

Значение в выражении: ^eval($t)<br />
^^t.count: ^t.count[]<br />
Выводим содержимое пользовательского объекта: ^print[$t]<br />


<br />
Копируем объект и выводим ^^c.count[]:
$c[^MyTable::create[$t]]
^c.count[]<br />
Удаляем 2 строки, начиная со строки с offset=1 и выводим содержимое
пользовательского объекта:
^c.remove(1;2)
^print[$c]<br />


<br />
Создаём объект системного класса table на основании объекта класса MyTable и
выводим ^^z.count[]:
$z[^table::create[$t]]
^z.count[]<br />


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

Определение класса MyTable

```
@CLASS
MyTable

@create[uParam]
```

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

# метод возвращающий значение объекта в разных контекстах вызова
@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']}
}


# метод обрабатывает обращения к "столбцам"
@GET_DEFAULT[sName]
$result[$t.$sName]


# для всех существующих методов нужно написать wrapper-ы
@count[]
^t.count[]

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


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

Литералы

Строковые литералы

В коде Parser могут использоваться любые буквы, включая русские. Следующие символы являются служебными:

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

Чтобы отменить специальное действие этих символов, их необходимо предварять символом **^**. Например, для получения в тексте символа **$** нужно записать **^$**.

Кроме того, допустимо использовать код символа:
^#20 – пробел
^#XX – **XX** hex код буквы

Числовые литералы

Запись числовых литералов допускается в следующем виде:

1
-8
(целое)

1.23
-4.56
(дробное)

1E3 равно **1000**
-2E-6 равно **-0.000002**
(форма так называемой научной записи чисел с плавающей запятой, формат: мантисса**E**порядок)

0xA8 равно **168**
(форма шестнадцатиричной записи целого числа)

Примечание: регистр букв не важен.

Логические литералы

В выражения Parser можно использовать логические литералы
true
false

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

Литералы в выражениях

Если строка содержит пробелы
или начинается с цифры, ***[3.1.5]***
то в выражении ее нужно заключать в кавычки или апострофы.

Пример
```
^if($name eq Вася){...}
```
Здесь **Вася** – строка, которая не содержит пробелов, поэтому ее можно не заключать в кавычки или апострофы.

```
^if($name eq "Вася Пупкин"){...}
```
Здесь строка содержит пробелы, поэтому заключена в кавычки.

Операторы

Операторы в выражениях и их приоритеты

Оператор	Значение	Приоритет	Комментарий
()	Группировка частей выражения	1 (*высший)*	
!	Логическая операция NOT	2	
~	Побитовая инверсия (NOT)	3	
+	Одиночный плюс	4	
-	Одиночный минус	4	
*	Умножение	5	
/	Деление	5	*Внимание,*
\	Целочисленное деление	5	*деление на ноль*
%	Остаток от деления	5	*дает ошибку* number.zerodivision.
+	Сложение	6	
-	Вычитание	6	
<<	Побитовый сдвиг влево	7	
>>	Побитовый сдвиг вправо	7	
&	Побитовая операция AND	8	
|	Побитовая операция OR	9	
!|	Побитовая операция XOR	10	
is	Проверка типа	11	
def	Определен ли объект?	11	
in	Находится ли текущий документ в каталоге?	11	
-f	Существует ли файл?	11	
-d	Существует ли каталог?	11	
==	Равно	12	
!=	Неравно	12	
eq	Строки равны	12	
ne	Строки не равны	12	
<	Число меньше	13	
>	Число больше	13	
<=	Число меньше или равно	13	
>=	Число больше или равно	13	
lt	Строка меньше	13	
gt	Строка больше	13	
le	Строка меньше или равна	13	
ge	Строка больше или равна	13	
&&	Логическая операция AND	14	*второй операнд не вычисляется, если первый — ложь*
||	Логическая операция OR	16	*второй операнд не вычисляется, если первый — истина*
!||	Логическая операция XOR	16 (*низший)*	

def. Проверка определенности объекта

Оператор возвращает булевое значение (истина/ложь) и отвечает на вопрос «определен ли объект?» Проверяемым объектом может любой объект Parser: таблица, строка, файл, объект пользовательского класса и т.д.

def объект
не определенными (не **def**) считаются пустая строка, пустая таблица, пустой хеш и код.

Пример

```
$str[Это определенная строка]
^if(def $str){
    Строка определена
}{
    Строка не определена
}
```

*Важно: для проверки «содержит ли переменная код» и «определен ли метод» используйте оператор **is**, а не **def**. Таким образом.* `^if(def $hash.delete){-}{в hash нет элемента «delete»}`.

in. Проверка, находится ли документ в каталоге

in "/каталог/"

Возвращает результат "истина/ложь" в зависимости от того, находится ли текщий документ в указанном каталоге.

Пример

```
^if(in "/news/"){
      Мы в разделе новостей
}{
      <a href="/news/">Новости</a>
}
```

-f и -d. Проверка существования файла и каталога

-f имя_файла – проверка существования файла на диске.
-d имя_каталога – проверка существования каталога на диске.

Возвращает результат "истина/ложь" в зависимости от того, существует ли указанный файл или каталог по заданному пути.

Пример

```
^if(-f "/index.html"){
      Главная страница существует
}{
      Главная страница не существует
}
```

is. Проверка типа

объект is тип

Возвращает результат "истина/ложь" в зависимости от того, относится ли левый операнд к заданному типу.
Полезно использовать этот оператор в случае, если переменная может содержать единственное значение или набор значений (хеш), а также для проверки определенности методов.

Тип – имя типа, им может быть системное имя (**hash**, **junction**, ...), или имя пользовательского класса.

Простая проверка типа

```
@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]
Работаем с датой:<br />
День:  $day<br />
Месяц: $month<br />
Год:   $year<br />
```

В этом примере в зависимости от типа переменной **$date** либо выполняется синтаксический анализ строки, либо методу **print_date** передаются поля объекта класса **date**:

Проверка определенности метода

Значение **$имя_метода**, это тоже **junction**, поэтому проверять существование метода необходимо также оператором **is**, а не **def**:

```
@body[]
тело

@main[]
Старт
^if($body is junction){
   ^body[]
}{
   Метод «body» не определен!
}
Финиш
```

Внимание: с помощью данной проверки невозможно определить наличие в переменное кода, т.к. любое обращение к нему вызывает его выполнение.

Комментарии к частям выражения

Допустимо использование комментариев к частям математического выражения, которые начинаются со знака **#**, и продолжаются до конца строки исходного файла или до конца выражения.

Пример

```
^if(
    $age>=$MINIMUM_AGE  # не слишком молод
    && $age<=$MAXIMUM_AGE  # и не слишком стар
){
    Годен
}
```

Внимание: настоятельно советуем задавать комментарии к частям сложного математического выражения. Бывает, что даже вам самим через какое-то время бывает трудно в них разобраться.

eval. Вычисление математических выражений

```
^eval(математическое выражение)
^eval(математическое выражение)[форматная строка]
```

Оператор **eval** вычисляет математическое выражение и позволяет вывести результат в нужном виде, задаваемом форматной строкой (см. Форматные строки).

Пример

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

вернет: **16.67**.

Внимание: настоятельно советуем задавать комментарии к частям сложного математического выражения (см. «Комментарии к частям выражения»).

Операторы ветвления

Операторы этого типа позволяют принимать решение о выполнении тех или иных действий в зависимости от ситуации.

В Parser существует два оператора ветвлений: **if**, проверяющий условие и выполняющий одну из указанных веток, и **switch**, выполняющий поиск необходимой ветки, соответствующей заданной строке или значению заданного выражения.

if. Выбор одного варианта из двух

```
^if(логическое выражение){код, если значение логического выражения «истина»}
^if(логическое выражение){
      код, если значение логического выражения «истина»
}{
      код, если значение логического выражения «ложь»
}
```

Оператор вычисляет значение логического выражения. Затем, в зависимости от полученного результата, либо выполняется или не выполняется код (первый вариант использования оператора **if**), либо для исполнения выбирается код, соответствующий полученному значению логического выражения (второй вариант). На код не накладывается никаких ограничений, в том числе внутри него может содержаться еще один или несколько операторов **if**.

Внимание: настоятельно советуем задавать комментарии к частям сложного логического выражения (см. «Комментарии к частям выражения»).

switch. Выбор одного варианта из нескольких

```
^switch[строка для сравнения]{
      ^case[вариант1]{действие для 1}
      ^case[вариант2]{действие для 2}
      ^case[вариант3;вариант 4]{действие для 3 или 4}
         …
      ^case[DEFAULT]{вариант по умолчанию}
}

^switch(математическое выражение){
      ^case(вариант1){действие для 1}
      ^case(вариант2){действие для 2}
      ^case(вариант3;вариант 4){действие для 3 или 4}
         …
      ^case[DEFAULT]{вариант по умолчанию}
}
```

Оператор **switch** сравнивает строку или результат математического выражения со значениями, перечисленными в **case**. В случае совпадения выполняется код, соответствующий совпавшему значению. Если совпадений нет, выполняется код, соответствующий значению **DEFAULT** (пишется только заглавными буквами).

Если код для **DEFAULT** не определен и нет совпадений со значениями, перечисленными в **case**, ни один из вариантов кода, присутствующих в операторе **switch**, выполнен не будет.

Пример

```
^switch[$color]{
      ^case[red]{Необходимо остановиться и подумать о вечном…}
      ^case[yellow]{Настало время собраться и приготовиться!}
      ^case[green]{Покажи им, кто король дороги!}
      ^case[DEFAULT]{Вы дальтоник, или это вовсе не светофор.}
}
```

Циклы

Цикл – процесс многократного выполнения некоторой последовательности действий.

В Parser существует два оператора циклов: **for**, в котором количество повторов тела цикла ограничивается заданными значениями счетчика, и **while**, где количество повторов зависит от выполнения условия. Для того, чтобы избежать зацикливания, в Parser встроен механизм обнаружения бесконечных циклов. Бесконечным считается цикл, тело которого выполняется более 20 000 раз.

for. Цикл с заданным числом повторов

```
^for[счетчик](от;до){тело}
^for[счетчик](от;до){тело}[разделитель]
^for[счетчик](от;до){тело}{разделитель}
```

Оператор **for** повторяет тело цикла, перебирая значения счетчика **от** начального значения **до** конечного. С каждым выполнением тела значение счетчика автоматически увеличивается на 1.

Счетчик – имя переменной, которая является счетчиком цикла;

От и **до** – начальное и конечное значения счетчика, математические выражения, задающие соответственно начало и конец диапазона значений, принимаемых счетчиком. Если конечное значение счетчика меньше начального, тело цикла не выполнится ни разу;

Разделитель – строка или код, который вставляется перед каждым непустым не первым телом.

Замечание: поскольку имена счетчиков могут повторяться, полезно объявлять их локальными переменными метода, где используется цикл `for`.
Замечание: если разделитель задан в виде кода, то этот код выполняется после следующего не пустого тела цикла.

В любой момент можно принудительно выйти из цикла с помошью оператора **break**, или принудительно закончить текущую итерацию и перейти к следующей с помощью оператора **continue**. ***[3.2.2]***

Пример

```
^for[week](1;4){
      <a href="/news/archive.html?week=$week">Новости за неделю №$week</a>
}[<br />]
```

Пример выводит ссылки на недели с первой по четвертую, после очередной строки ставится тег перевода строки.

while. Цикл с условием

```
^while(условие){тело}
^while(условие){тело}[разделитель]   [3.1.5]
^while(условие){тело}{разделитель}   [3.1.5]
```

Оператор **while** повторяет тело цикла, пока условие истинно. Если условие заведомо ложно, тело цикла не выполнится ни разу.

Разделитель – строка или код, который вставляется перед каждым непустым не первым телом.

Замечание: если разделитель задан в виде кода, то этот код выполняется после следующего не пустого тела цикла.

В любой момент можно принудительно выйти из цикла с помошью оператора **break**, или принудительно закончить текущую итерацию и перейти к следующей с помощью оператора **continue**. ***[3.2.2]***

Пример

```
$little_negros(10)
^while($little_negros > 0){
      <p>$little_negros негритят пошли купаться в море.
      Один из них ^little_negros.dec[] утоп,
      ему срубили гроб, и вот вам результат —
      $little_negros негритят.</p>
}[<br />]
```

break. Выход из цикла

```
^break[]
```

Оператор **break** может быть использован внутри циклов (**for**, **while**, **menu**, **foreach**) для их принудительного прерывания.
Использование оператора вне цикла недопустимо и приводит к ошибке.

continue. Переход к следующей итерации цикла

```
^continue[]
```

Оператор **continue** может быть использован внутри циклов (**for**, **while**, **menu**, **foreach**) для их принудительного прерывания текущей итерации цикла и переходу к следующей.

Использование оператора вне цикла недопустимо и приводит к ошибке.

connect. Подключение к базе данных

^connect[строка подключения]{код}

Оператор **connect** осуществляет подключение к серверу баз. Код оператора обрабатывается Parser, работая с базой данных в рамках установленного подключения.

Parser (в виде модуля к Apache или IIS) кеширует соединения с SQL-серверами, и повторный SQL запрос на соединение с той же строкой подключения не производится, а соединение берется из кеша, если оно еще действительно.

Вариант CGI также кеширует соединение, но только на один http запрос (обработка одного документа), поэтому явно допустимы конструкции вида:

^connect[строка подключения]{…первый SQL запрос…}
^connect[строка подключения]{…второй SQL запрос…}

При этом не будет двух соединений, и это полезно, когда, скажем, изредка соединение нужно, и заранее неизвестно нужно или нет – заранее его можно не делать, а делать визуально многократно, зная, что оно фактически не разрывается.

Передать SQL-запрос БД может один из следующих методов или конструкторов языка Parser:

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

Замечание: для работы оператора connect необходимо наличие настроенного драйвера баз данных (см. раздел Настройка).

Форматы строки соединения для поддерживаемых серверов баз данных описаны в приложении.

Пример

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

use. Подключение модулей

^use[файл]

Оператор позволяет использовать модуль из указанного файла. Если путь к файлу начинается с "/", то считается, что это путь от корня веб-пространства. В любом другом случае Parser будет искать модуль по путям, определенным в переменной **$CLASS_PATH** в Конфигурационном методе.

Для подключения модулей также можно воспользоваться конструкцией:

```
@USE
имя файла 1
имя файла 2
…
```

Разница между этими конструкциями в том, что использование **@USE** подключает файлы с модулями до начала выполнения кода, в то время как оператор use может быть вызван непосредственно из тела программы, например:

```
^if(условие){
	^use[модуль1]
}{
	^use[модуль2]
}
```

Замечание: попытки подключить уже подключенные ранее модули не приводят к повторным считываниям файлов с диска. Однако нужно иметь ввиду, что для этого полное имя файла подключаемого модуля должно с точностью до символа совпадать с именем файла уже подключенного ранее модуля. В случае написания **^use[module.p]** *и* **^use[sub/../module.p]** *будет считаться, что идет попытка подключить разные модули.*

Крайне рекомендуем использовать возможность сохранения результатов работы кода, используя оператор **use** для подключения необходимых модулей в коде оператора **cache**.

cache. Сохранение результатов работы кода

```
^cache[файл](число секунд){код}
^cache[файл](число секунд){код}{обработчик проблем}    [3.1.2]
^cache[файл][дата устаревания]{код}
^cache[файл][дата устаревания]{код}{обработчик проблем}    [3.1.2]
^cache[] = дата устаревания    [3.1.5]
```

Оператор сохраняет строку, которая получится в результате работы кода. При последующих вызовах обычно происходит считывание ранее сохраненного результата, вместо повторного вычисления, что сильно экономит время обработки запроса и снижает нагрузку на ваши сервера.

Крайне рекомендуется подключать модули (**^use[…]**) изнутри **кода** оператора **cache**, а не делать это статически (**@USE**).
По-возможности, работайте с базой данных (**^connect[…]**) также внутри **кода** оператора **cache** – вы существенно снизите нагрузку на ваш SQL-сервер и повысите производительность ваших сайтов.

Файл – имя файла-кеша. Если такой файл существует и не устарел, то его содержимое выдается клиенту, если не существует – выполняется код, и результат сохраняется в файл с указанным именем.

Число секунд – время хранения результата работы кода в секундах. Если это число равно нулю, то результат не сохраняется, а файл с ранее сохраненным результатом уничтожается.

Дата устаревания – время, до которого хранится результата работы кода. Если она в прошлом, то результат не сохраняется, а файл с предыдущим сохраненным результатом уничтожается.

Код – код, результат которого будет сохранен.

Обработчик проблем – здесь можно обработать проблему, если она возникнет в **коде**. В этом отношении оператор похож на **try**, см. раздел «Обработка ошибок». В отличие от **try**, можно задать **$exception.handled[cache]=1** – это дает указание Parser обработать ошибку особенным образом: достать из **файла** ранее сохраненный результат работы **кода**, проигнорировав тот факт, что этот результат устарел.

Внутри **кода** допустимы команды, изменяющие время хранения результата работы кода:

```
^cache(число секунд)
^cache[дата устаревания]
```

Берется минимальное время хранения кода.
Текущую дату устаревания можно узнать, вызвав
$expire_date[^cache[]]

Пример

```
^cache[/data/cache/test1](5){
```

```
	Нажимайте reload, меняется каждые 5 секунд: ^math:random(100)
}
```

Изменение времени хранения

```
^cache[/data/cache/test2](5){
	по ходу работы вы выяснили,
	что страницу сохранять не нужно: ^cache(0)
}
```

process. Компиляция и исполнение строки

```
^process{строка}
^process[контекст]{строка}
^process[контекст]{строка}[опции]
```
[3.1.2]

Строка будет скомпилирована и выполнена как код на Parser, в указанном **контексте**, или в текущем контексте.
В качестве **контекста** можно указать объект или класс, но **не метод** (т.е. если вы внутри вашего метода вызовите process, то внутри выполняемого с помощью process кода не будут доступны локальные переменные вызывающего метода).

Удобно использовать, если какие-то части кода или собственные методы необходимо хранить не в файлах `.html`, которые обрабатываются Parser, а в каких-то других, или базе данных.

Также можно указать ряд **опций** (хеш):
- **$.main[**новое имя для метода **main**, описанного в коде **строки]**
- **$.file[**имя файла, из которого взята данная **строка]**
- **$.lineno(**номер строки в файле, откуда взята данная **строка.** *может быть отрицательным***)**

Простые примеры

```
^process{@extra[]
	Здоровья прежде всего…
}
```
Метод **extra** будет добавлен к текущему классу, и его можно будет вызывать в дальнейшем.

```
^process[$engine:CLASS]{@start[]
	Мотор…
}
```
Метод **start** будет добавлен к пользовательскому классу **engine**.

```
$running_man[^man::create[Вася]]
^process[$running_man]{
    Имя: $name<br />
}
```
Код будет выполнен в контексте объекта **$running_man**, соответственно, может воспользоваться полем **name** этого объекта, выдаст «**Вася**».

Оператор include

```
@include[filename][file]
$file[^file::load[text;$filename]]
^process[$caller.self]{^taint[as-is][$file.text]}[
	$.file[$filename]
]
```
Код загружает указанный файл и выполняет его в контексте объекта/класса, вызвавшего **include**.
Опция **file** позволяет указать имя файла, откуда был загружен код. Если возникнет ошибка, вы увидите это «имя файла».
Важно: контекст вызова не включает локальные переменные и параметры вызывающего метода!

Сложный пример

Часто удобно поместить компилируемый код в некоторый метод с именем, вычисляющимся по ходу

работы:

```
# это исходный код, обратите внимание, на ^^
$source_code[2*2=^^eval(2*2)]
# по ходу работы выясняется, что необходимо создать метод с именем method1
$method_name[method1]
# компилируем исходный код, помещяем его в новый метод
^process{$source_code}[
      $.main[$method_name]
]
...
# далее по коду можно вызывать метод method1
^method1[]
```

Данный пример будет продолжать работать, даже если в **$source_code** будет определен ряд методов, поскольку **опция main** задает новое имя методу **main**.

rem. Вставка комментария

^rem{комментарий}

Весь код, содержащийся внутри оператора, не будет выполнен. Используется для комментирования блоков кода.

Внешние и внутренние данные

Создавая код на Parser, мы имеем дело с двумя видами данных. Один из них — это все то, что написано самим кодером. Второй – данные, получаемые кодом извне, а именно из форм, переменных окружения, файлов, от SQL-серверов, из cookies и т.п. Все то, что создано кодером, не нуждается в проверке на корректность. Вместе с тем, когда данные поступают, например, от пользователей через поля форм, выводить их «**as-is**» (как есть) может быть потенциально опасно. Возникает необходимость преобразования таких данных по определенным правилам. Большую часть работы по подобным преобразованиям Parser выполняет автоматически, не требуя вмешательства со стороны. Например, если присутствует вывод данных, введенных через поле формы, то в них символы **< >** автоматически будут заменены на **<** и **>**. Иногда наоборот бывает необходимо позволить вывод таких данных именно в том виде, в котором они поступили.

Для Parser «свой» код, т.е. тот, который набрал кодер, считается **clean** («чистым»). Все данные, которые поступают извне, считаются **tainted** («грязными» или «окрашенными»).

код Parser — этот код создан скриптовальщиком, поэтому никаких вопросов не вызывает;

$form:field — здесь должны быть выведены данные, введенные пользователем через форму;

$my_table[^table::sql{запрос}] — здесь данные поступают из БД.

В случае с **$form:field**, поступившие **tainted** данные будут автоматически преобразованы и некоторые символы заменятся в соответствии с внутренней таблицей замен Parser. После этого они станут **clean** («чистыми»), и их «окрашенность» исчезнет. Здесь неявно выполняется операция **untaint** (снять «окраску»). Автоматическое преобразование данных происходит в тот момент, когда эти данные будут выводиться. Так, в случае с помещением данных, поступивших из БД, в переменную **$my_table**, преобразование выполнится в тот момент, когда данные будут в каком-либо виде выданы во внешнюю среду (переданы браузеру, сохранены в файл или базу данных).

Вместе с тем, бывают ситуации, когда необходимости в таком преобразовании нет, либо данные нужно преобразовать по другим правилам, чем это делает Parser по умолчанию. Например, нам нужно разрешить пользователю вводить HTML-теги через поле формы для дополнительного форматирования текста. Но, так как это чревато неприятностями (ввод Java-скрипта в гостевой книге может перенаправлять пользователей с вашего сайта на вражеский), Parser сам заменит «нежелательные» символы в соответствии со своими правилами. Решение – использование оператора **untaint**.

untaint, taint. Преобразование данных

```
^untaint{код}
^untaint[вид преобразования]{код}
^taint[текст]
^taint[вид преобразования][текст]
```

С помощью механизма автоматических преобразований Parser защищает вашу систему от вторжения извне и «по умолчанию» делает это хорошо. Этот механизм работает даже тогда, когда вы ни разу не написали в вашем коде операторов **taint/untaint**. Когда вы вмешиваетесь в работоспособность этого механизма с помощью данных операторов (особенно используя вид преобразования **as-is**), вы можете создать уязвимость в вашей системе, поэтому делать это нужно внимательно и обязательно разобравшись как же именно он работает.

Оператор **taint** помечает весь переданный ему **текст**, как нуждающийся в преобразовании заданного **вида**.
Если вид преобразования не указан, оператор **taint** помечает **текст** как **tainted** (неопределенно «грязный», без указания вида преобразования). Для помеченного таким образом текста будут применяться такие же правила преобразований как для текста, пришедшего извне (из полей формы, из базы данных, из файла, из cookies и т.п.).

Оператор **untaint** выполняет переданный ему **код** и помечает, как нуждающиеся в преобразовании заданного **вида**, только неопределённо «грязные» части результата выполнения кода (т.е. те, которые не являлись частью кода на Parser, написанного разработчиком в теле документов, а поступили извне или которые были помечены как **tainted** с помощью оператора **taint** без первого параметра). Он не трогает те части, для которых уже задан конкретный вид преобразования.
Если вид преобразования не указан, оператор **untaint** помечает неопределённо «грязные» части результата выполнения кода как **as-is**.

Данные операторы лишь делают пометки в тексте о виде преобразования, который Parser-у нужно будет произвести **позже**, но не производят его сиюминутно. Сами преобразования Parser выполняет при выдаче текста в браузер, перед выдачей SQL-серверу, при сохранении в файл, при отправке письма и т.п.

Для простоты можно представить себе, что вокруг всех букв, пришедших извне написано **^taint[пришедшее извне]**, а вокруг всех букв, набранных вами в теле страницы **^taint[optimized-as-is][написанное вами]**.

В некоторых случаях результаты работы **^taint[вид преобразования][текст]** и **^untaint[вид преобразования]{текст}** одинаковые: это происходит тогда, когда весь обрабатываемый текст является неопределённо «грязным» (например **$form:field**). Однако будьте внимательны: применение к неопределённо «грязному» тексту этих операторов без первого параметра даст совершенно разные результаты, т.к. опущенные значения первого параметра у них различны.

Схема автоматического преобразования Parser при выдаче данных в браузер — **optimized-html** и в общем виде можно представить весь набираемый разработчиком код следующим образом:
^untaint[optimized-html]{весь код, набранный разработчиком}
Это означает что если вы напишите в теле страницы **$form:field** (без всяких **taint/untaint**), то даже если кто-то обратится к ней с параметром «**?field=</html>**», то это не «поломает» страницу из-за досрочно выведенного в неё закрывающего тега **</html>**, т.к. содержимое **$form:field** неопределённо «грязное» и поэтом к нему будет применено автоматическое преобразование **optimized-html**, с помощью которого символы < и > будут заменены на < и > соответственно.

Аналогично работают и другие автоматические преобразования, например если при составлении SQL запроса вы напишите (опять же без использования **taint/untaint**):
^string:sql{SELECT name FROM table WHERE uid = '$form:uid'}
то злоумышленник не сможет выполнить SQL injection, передав в качестве параметра например «**?uid=' OR 1=1 OR '**», т.к. Parser, перед выдачей SQL серверу текста запроса, заэкранирует в

пришедшем от пользователя **$form:uid** одинарные кавычки.

Текст, написанный разработчиком в теле страниц, также подвергается автоматическому преобразованию. В нём Parser выполняет оптимизацию пробельных символов (пробел, табуляция, перевод строки). Идущие подряд перечисленные символы заменяются только одним, который встречается в коде первым. Т.е. если вы напишите в тексте страницы несколько идущих подряд пробельных символов, перед выдачей их в браузер посетителю, от них останется только первый символ. Если в каких-то случаях нужно отключить эту оптимизацию (например для выдачи в **<pre/>**), то вы должны сделать это явно, например написав вокруг текста:

```
<pre>
^taint[as-is][
  Я
   достаю
         из широких штанин
  дубликатом
            бесценного груза.
  Читайте,
          завидуйте,
                    я —
                       гражданин
  Советского Союза.
]
</pre>
```

В данном случае нужно писать именно **taint**, а не **untaint**, т.к. буквы, написанные в тексте страницы разработчиком, являются «чистыми» и поэтому **untaint** не окажет на них никакого влияния.

Пример

```
$clean[<br />]
# предыдущая запись эквивалентна следующей: $clean[^taint[optimized-as-is][<br
/>]]
$tainted[^taint[<br />]]

Строки: ^if($clean eq $tainted){совпадают}{не совпадают}<br />

«Грязные» данные — '$tainted'<br />
«Чистые» данные — '$clean'<br />
```

В данном примере видно, что несмотря на то, что сравнение сообщает об эквивалентности строк, при выводе их в браузер результат различен: «чистая» строка выводится без преобразований, а в «грязной» строке символы < и > заменены на < и > соответственно.

Пример

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

$tainted[$form:field]
«Грязные» данные — $tainted<br />
«Чистые» данные — ^untaint{$tainted}
```

В квадратных скобках оператора **untaint** задается вид выполняемого преобразования. Здесь мы опускаем квадратные скобки в операторе **untaint** и используем значение преобразования по умолчанию **[as-is]**.
Обратите внимание: если оператор **untaint** без указания вида преобразования полностью

эквивалентен оператору **untaint** с указанием вида преобразования **as-is**, то для оператора **taint** не существует такого вида преобразования, который равнозначен оператору **taint** без указания оного.

Пример

```
Пример ^^taint.<br />
$town[Москва]
<a href="town.html?town=^taint[uri][$town]">$town</a>
```

В результате данные, хранящиеся в переменной **town**, будут преобразованы к типу URI и позже, при выводе в браузер, русские буквы будут заменены на шестнадцатеричные коды символов и представлены в виде `%XX`.

Пример

```
Вывод данных полученных от пользователя на странице, сохранение их
в БД и создание на их основе XML<br />
Вы указали: '$form:field'
^connect[$SQL.connect-string]{
      ^void:sql{INSERT INTO news SET (body) VALUES ('$form:field')}
}
$doc[^xdoc::create{<?xml version="1.0" encoding="WINDOWS-1251"?>
<root>
      <data>$form:field</data>
</root>
}]
```

В данном случае ни **taint** ни **untaint** использовать не нужно вовсе, т.к. необходимые преобразования будут сделаны автоматически, причем при выводе в браузер будет сделано преобразование **optimized-html**, при выдаче SQL серверу — **sql**, а при формировании XML — **xml**. Обратите внимание на то, что при сохранении данных в БД в административном интерфейсе, также не требуется писать **taint/untaint** в SQL запросах.

Пример

```
Выдача даннных (могут содержать теги), пришедших от пользователя или из БД
в форму для редактирования<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>
```

В данном случае сработает автоматическое преобразование **optimized-html**, т.к. данные, пришедшие из БД или от пользователя являются «грязными».
Поэтому встретившиеся в данных теги не «поломают» страницу. Имейте в виду, что если в данных есть идущие подряд пробельные символы, то они будут оптимизированы при выдаче в браузер.

Пример

```
Выдача данных с тегами из БД, помещённых туда администратором:<br />
^connect[$SQL.connect-string]{
      $body[^string:sql{SELECT body FROM news WHERE news_id = $id}]
}
^taint[as-is][$body]
```

В данном случае необходимо использовать **taint** с видом преобразования **as-is** (или **untaint** с таким же видом преобразования или без указания оного), т.к. требуется, чтобы теги в тексте новости, помещённые туда администратором, были выданы именно как теги и в них не было произведено никаких преобразований. Ни в коем случае нельзя выводить подобным образом данные из БД, помещённые туда от посетителей сайта (например данные гостевых книг, форумов и т.д.).

Пример

```
Выдача даннных (могут содержать теги), пришедших от пользователя или из БД
в форму для редактирования с сохранением пробельных символов<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>
```

В данном случае нужно использовать **taint** с видом преобразования **html** (или **untaint** с таким же видом преобразования), чтобы встретившиеся в данных теги не «поломали» страницу ***и*** чтобы отключить оптимизацию пробельных символов.

Из примеров выше можно заметить, что нам пришлось использовать оператор **taint** лишь трижды: один раз для того, чтобы разрешить отображать теги в тексте из БД и помещённом туда администратором, второй раз чтобы отключить оптимизацию пробельных символов и третий раз чтобы выдать ссылку с query string содержащей русские буквы таким образом, чтобы эти буквы были закодированы.
Во всех остальных случаях мы вообще не использовали ни **taint** ни **untaint** и Parser сам всё сделал хорошо.

Запомните: в подавляющем большинстве случаев использовать данные операторы не нужно!

Как мы уже отметили, в приведённых примерах ни разу не был использован оператор **untaint**, поэтому возникает вопрос, для чего он вообще нужен? Я знаю ему буквально пару практических применений.

Во первых иногда его использование позволяет уменьшить количество операторов **taint** в коде, например при выводе данных в форму, содержащую много полей и необходимостью отключить оптимизацию пробельных символов. В этом случае вместо того, чтобы писать **^taint[html][...]** вокруг вывода содержимого каждой textarea (как в примере выше), можно написать один раз **^untaint[html]{...}** вокруг всей формы.

Пример

```
Выдача даннных (могут содержать теги), пришедших от пользователя или из БД
в большую форму для редактирования с сохранением пробельных символов<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>Заголовок:</b><br />
```

```
            <textarea name="title">$data.title</textarea>
      </p>
      <p>
            <b>Анонс:</b><br />
            <textarea name="lead">$data.lead</textarea>
      </p>
      <p>
            <b>Текст новости:</b><br />
            <textarea name="body">$data.body</textarea>
      </p>
}
```

И во вторых — когда нам нужно выдать в браузер xml, а не html (например для ajax, RSS, SOAP и т.п.). В этом случае автоматическое преобразование (**optimized-html**) не подходит и вокруг всего кода нужно написать **^untaint[optimized-xml]{...}** и расслабиться :)

Преобразование заключается в замене одних символов на другие в соответствии с внутренними таблицами преобразований. Предусмотрены следующие виды преобразований:

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

optimized-as-is
optimized-xml
optimized-html

Таблицы преобразований

as-is изменений в тексте не делается

file-spec символы * ? ' " < > |
преобразуются в "_XX",
где XX – код символа в шестнадцатеричной форме

uri символы за исключением цифр, строчных и прописных латинских букв, а также следующих символов: _ - . "
преобразуется в %XX где XX – код символа в шестнадцатеричной форме

http-header то же, что и URI

mail-header если известен charset (если неизвестен, не будут работать up/low case), то фрагмент, начиная с первой буквы с восьмым битом и до конца строки, будет представлен в подобном виде:
Subject: Re: parser3: =?koi8-r?Q?=D3=C5=CD=C9=CE=C1=D2?=

sql в зависимости от SQL-сервера
для Oracle, ODBC и SQLite меняется ' на ''
для PgSQL делается средствами самого PgSQL
для MySQL делается средствами самого MySQL

для выполнения данного преобразования необходимо чтобы код, в результате работы которого преобразование должно выполниться, находился внутри оператора **^connect[]{}.**

js " преобразуется в \"
' преобразуется в \'
\ преобразуется в \\
символ конца строки преобразуется в \n
символ с кодом 0xFF предваряется \

parser-code служебные символы предваряются символом ^

regex символы
\ ^ $. [ ] | ( ) ? * + { } -
предваряются символом
\
[3.1.5]

xml & преобразуется в &
> преобразуется в >
< преобразуется в <
" преобразуется в "
' преобразуется в '

html & преобразуется в &
> преобразуется в >
< преобразуется в <
" преобразуется в "

optimized-as-is
optimized-xml
optimized-html дополнительно к заменам выполняется оптимизация по "white spaces" (символы пробела, табуляция, перевода строки).

Идущие подряд перечисленные символы заменяются только одним, который встречается в коде первым

Ряд **taint** преобразований Parser делает автоматически, так, имена и пути файлов всегда автоматически **file-spec** преобразуются, и когда вы пишите...

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

...Parser выполняет...

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

Аналогично, при задании HTTP-заголовков и заголовков писем, происходят **http-header** и **mail-header** преобразования соответственно. А при DOM-операциях текстовые параметры всех методов автоматически **xml** преобразуются.

Также Parser выполняет ряд автоматических **untaint** преобразований:

вид	*что преобразуется*
sql	тело SQL-запроса
xml	XML код при создании объекта класса **xdoc**
optimized-html	результат страницы, отдаваемый в браузер
regex	шаблоны-регулярные выражения
parser-code	тело оператора process

Обработка ошибок

Человек несовершенен. Вы должны быть готовы к тому, что вместо ожидаемого, на экране вашего компьютера появится сообщение об ошибке. Избежать этого, к сожалению, почти невозможно. На начальном этапе сообщения об ошибках будут довольно частыми. Основной причиной ошибок сначала, вероятнее всего, будут непарные скобки (помните, мы говорили о текстовых редакторах, помогающих их контролировать?) и ошибки в записи конструкций Parser.

Если в ходе работы возникла ошибка, обычно обработка страницы прекращается, происходит откат (rollback) по всем активным на момент ошибки SQL-соединениям, и, вместо того вывода, который должен был попасть пользователю, вызывается метод **unhandled_exception**, ему передается информация об ошибке и стек вызовов, приведших к ошибке, и выдаются результаты его работы. Также ошибка записывается в журнал ошибок веб-сервера.

Однако часто желательно перехватить возникшую ошибку, и сделать нечто полезное вместо ошибочного кода.
Допустим, вы хотите проверить на правильность XML код, полученный из ненадежного источника. Здесь прерывание обработки страницы вам совершенно ни к чему, наоборот — вы ожидаете ошибку определенного типа и хотите ее обработать.
Parser с радостью идет навстречу, и дает вам в руки мощный инструмент: оператор **try**.

При сложной обработке данных часто выясняется, что имеет место ошибка в методе, вызванном из другого, а тот, в свою очередь, из третьего. Как в такой ситуации просто сообщить и обработать ошибку?
Используйте оператор **throw**, чтобы сообщить об ошибке, и обработайте ошибку на верхнем уровне — и вам не придется проверять ее на всех уровнях вложенности вызовов методов.

Также во многих ситуациях сам Parser или его системные классы сообщают о ошибках, см. «Системные ошибки».

try. Перехват и обработка ошибок

```
^try{код, ошибки которого попадают...}{...в этот обработчик в виде $exception}
^try{код, ошибки которого попадают...}{...в этот обработчик в виде $exception}{а
тут код, который в любом случае выполнится в конце}   [3.3.0]
```

Если по ходу работы **кода** возникла ошибка, создается переменная **$exception**, и управление передается **обработчику**.

Если указан третий параметр (finally), то он в любом случае будет выполнен после завершения обработки тела или обработчика исключений, даже если исключение не будет перехвачено.

$exception, это такой **hash**:

$exception.type	строка, тип ошибки. Определен ряд системных типов, также тип ошибки может быть задан в операторе **throw**.
$exception.source	строка, источник ошибки (ошибочное имя файла, метода, ...)
$exception.file **$exception.lineno** **$exception.colno**	файл, содержащий **source**, номера строки и колонки в нем
$exception.comment	комментарий к ошибке, по-английски
$exception.handled	истина или ложь, флаг «обработана ли ошибка» необходимо зажечь этот флаг в **обработчике**, если вы обработали переданную вам ошибку

Обработчик обязан сообщить Parser, что данную ошибку он обработал, для чего **только** для нужных типов ошибок он должен зажечь флаг:
$exception.handled(true)

Если обработчик не зажег этого флага, ошибка считается **необработанной**, и передается следующему обработчику, если он есть.

Если ошибка так и не будет обработана, если есть, вызывается метод **unhandled_exception** и ему передается информация об ошибке, стек вызовов, приведших к ошибке, и выдаются результаты его работы. А также производится запись в журнал ошибок веб-сервера.

Пример

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

throw. Сообщение об ошибке

^throw[type] *[3.3.0]*
^throw[type;source]
^throw[type;source;comment]
^throw[хеш]

Оператор **throw** сообщает об ошибке типа **type**, произошедшей по вине **source**, с комментарием **comment**.

Эта ошибка может быть перехвачена и обработана при помощи оператора **try**.

Не перехватывайте ошибки **только** для их красивого вывода, пусть этим централизованно займется метод **unhandled_exception**, вызываемый Parser если ни одного обработчика ошибки так и не будет найдено. Кроме прочего, произойдет запись в журнал ошибок веб-сервера, который можно регулярно просматривать на предмет имевших место проблем.

Пример

```
@method[command]
```

```
^switch[$command]{
	^case[add]{
		добавляем...
	}
	^case[delete]{
		удаляем...
	}
	^case[DEFAULT]{
		^throw[bad.command;$command;Wrong command $command, good are
add&delete]
		^rem{
			допустим также следующий формат вызова оператора throw
			^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){
		$exception.handled(true)
		Неправильная команда '$exception.source', задана
		в файле $exception.file, в $exception.lineno строке.
	}
}
```

Результатом работы примера будет

```
Неправильная команда 'format c:', задана
в файле c:/parser3tests/www/htdocs/throw.html, в 15 строке.
```

Обращаем ваше внимание на то, что пользователи вашего сайта не должны увидеть технические подробности в сообщениях об ошибках, тем более содержащие пути к файлам, это некрасиво и ненадежно.
*Вывод **$exception.file** дан в качестве примера и настоятельно не рекомендуется к использованию на промышленных серверах — только для отладки.*

@unhandled_exception. Вывод необработанных ошибок

Если ошибка так и не была обработана ни одним обработчиком (см. оператор **try**), Parser вызывает метод **unhandled_exception**, ему передается информация об ошибке и стек вызовов, приведших к ошибке, и выдаются результаты его работы. Также ошибка записывается в журнал ошибок веб-сервера.

Хороший тон, это оформить сообщение об ошибке в общем дизайне вашего сайта.
А также проверить, и **не** показывать технические подробности вашим посетителям.

Рекомендуем поместить этот метод в Конфигурационный файл сайта.

Имеется возможность предотвратить запись ошибки в журнал ошибок, для чего **только** для нужных ошибок можно зажечь флаг:
`$exception.handled(true)` ***[3.1.4]***

Пример

```
@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 />
      }
}
```

Системные ошибки

type	Пример возникновения	Описание
parser.compile	**^test[}**	Ошибка компиляции кода. Непарная скобка, и т.п.
parser.runtime	**^if(0).**	Методу передано неправильное количество параметров или не тех типов, и т.п.
parser.interrupted		Загрузка страницы прервалась (пользователь остановил загрузку страницы или истекло время ожидания)
number.zerodivision	**^eval(1/0)**, **^eval(1\0)** или **^eval(1%0)**	Деление или остаток от деления на ноль
number.format	**^eval(abc*5)**	Преобразование к числу нечисловых данных
file.missing	**^file:delete[skdfjs.delme]**	Файл отсутствует
file.access	**^table::load[.]**	Нет доступа к файлу
file.read		Ошибка чтения файла
file.execute		Ошибка выполнения внешней программы, например отсутствующий CGI заголовок при выполнении ^file::cgi[...]
date.range	**^date::create(1950;1;1)**	Дата не принадлежит поддерживаемому
pcre.execute	**^строка.match[((\w)]**	Ошибка компиляции или выполнения PCRE шаблона
image.format	**^image::measure[index.html]**	Файл изображения имеет неправильный формат (возможно, расширение имени не соответствует содержимому, или файл пуст?)
sql.connect	**^connect[mysql://baduser:pass@host/db]{}**	Сервер баз данных не может быть найден или временно недоступен
sql.execute	**^void:sql{bad select}**	Ошибка исполнения SQL запроса
xml	**^xdoc::create{<forgot?>}**	Ошибочный XML код или операция
smtp.connect		SMTP сервер не может быть найден или временно недоступен
smtp.execute		Ошибка отправки письма по SMTP протоколу
email.format		Ошибка в email адресе: адрес пустой или содержит неправильные символы
email.send		Ошибка запуска почтовой программы
http.host	**^file::load[http://notfound/there]**	Сервер не найден
http.connect	**^file::load[http://not_accepting/there]**	Сервер найден, но не принимает соединение
http.response	**^file::load[http://ok/there]**	Сервер найден, соединение принял, но выдал некорректный ответ (нет статуса, заголовка)
http.status	**^file::load[http://ok/there]**	Сервер выдал ответ со статусом не равным 200 (не успешное выполнение запроса)
http.timeout		Загрузка документа с HTTP-сервера не завершилась в отведенное для нее время

Операторы, определяемые пользователем

Иногда вам будет казаться, что каких-то операторов в языке не хватает. Parser позволяет вам определить собственные операторы, которые затем можно будет использовать наравне с системными.

Операторами в Parser считаются методы класса MAIN, добавляя новые методы в этот класс вы расширяете базовый набор операторов.

Внимание: при описании оператора можно использовать и не локальные переменные, при этом вы будете читать и записывать в поля класса MAIN.

Пользовательские операторы могут определяться и в отдельных текстовых файлах без заголовка

@CLASS, которые подключаются к нужным разделам сайта. Если в таком файле определить оператор (написав, скажем, **@include[]**), то при обращении **^include[…]** всегда будет вызываться пользовательский оператор.

Будьте внимательны! Если определить оператор, одноименный с системным, то всегда будет вызываться пользовательский. При этом системный оператор вызвать нельзя никак. Стоит делать как можно меньше пользовательских операторов, используя вместо них статические методы пользовательских классов.

Создавать классы и пользоваться их методами гораздо удобнее, чем пользовательскими операторами. Простой пример: есть несколько разделов сайта, и для каждого из них нужно сделать раздел помощи. Создав несколько файлов, описывающих разные классы, можно получить одноименные методы разных классов. Вызывая методы как статические, мы имеем совершенно ясную картину, что к какому разделу относится:

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

Примеры

Поместите этот код…

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

… в файл `operators.p`, в корень вашего веб-сайта.

Там, где вам необходимы дополнительные операторы, подключите этот модуль. Например, в корневом `auto.p`, напишите…

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

…теперь не только в любой странице, но, что главное, в любом вашем классе можно будет воспользоваться конструкцией

```
^default[$form:name;Аноним]
```

Подробности в разделе Создание методов и пользовательских операторов.

Кодировки

Уверены, наличие разных кодировок доставляет вам такое же удовольствие, как и нам.

В Parser встроена возможность прозрачного перекодирования документов из кодировки, используемой на сервере в кодировку посетителя и обратно.

Parser перекодирует

- данные форм;
- строки при преобразовании вида uri;
- текстовый результат обработки страницы.

Кодировку, используемую вами в документах на сервере, вы задаете в поле **$request:charset**.
Кодировку, желаемую вами в результате — в **$response:charset**.
Сделать это необходимо в одном из **auto** методов.

Рекомендуем задавать кодировку результата в HTTP заголовке **content-type**, чтобы программа просмотра страниц знала о ней, и пользователю вашего сервера не нужно было переключать ее вручную.

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

Кодировку текста отправляемых вами писем можно задать отличной от кодировки результата, см. **^mail:send[…]**.

При работе с базами данных необходимо задать кодировку, в которой общаться с SQL-сервером, см. Формат строки подключения.

Список допустимых кодировок определяется в Конфигурационном файле.
По умолчанию везде используется кодировка **UTF-8**.

*Примечание: если при перекодировании **из** UTF-8 какой-то символ не указан в таблице перекодирования, вместо этого символа создается последовательность* **&#DDDD;** *где* **DDDD** *это Unicode данного символа в десятичной системе счисления.* ***[3.0.8]***

*Примечание: если при перекодировании **в** UTF-8 какой-то символ не указан в таблице перекодирования, вместо этого символа создается последовательность* **%HH** *где* **HH** *это шестнадцатиричный код данного символа.* ***[3.1.4]***

Примечание: имя кодировки нечувствительно к регистру. ***[3.1]***

Класс MAIN, обработка запроса

Parser обрабатывает запрошенный документ следующим образом:

1.
Считываются, компилируются и инициализируются Конфигурационный файл; затем все файлы с именем `auto.p`, поиск которых производится начиная от корня веб-пространства, и ниже по структуре каталогов, вплоть до каталога с запрошенным документом; наконец, сам запрошенный документ.
Все вместе они составляют определение класса **MAIN**.
Инициализация заключается в вызове метода **auto** каждого загруженного файла. Если определение этого метода содержит параметр, при вызове в нем будет передано имя загруженного файла.
Примечание: результат работы метода посетителю не выводится.

2.
Затем вызывается без параметров метод **main** класса **MAIN**.
Т.е. в любом из перечисленных файлов может быть определен метод **main**, и будет вызван тот, который был определен последним — скажем, определение этого метода в запрошенном документе перекрывает все остальные возможные определения, и будет вызван именно он.
Результат работы этого метода будет передан пользователю, если не был определен метод **postprocess**.
Если в файле не определен ни один метод, то все его тело считается определением метода **main**.
Примечание: задание **$response:body[нестандартного ответа]** *переопределяет текст, получаемый пользователем.*

3.
Если в классе **MAIN** определен метод **postprocess**, то результат работы метода **main** передается единственным параметром этому методу, и пользователь получит уже его результат работы.
Таким образом вы получаете возможность «дополнительной полировки» результата работы вашего кода.

Простой пример

Добавление такого определения, скажем, в файл `auto.p` в корне вашего веб-пространства…

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

…приведет к замене одних слов на другие в результатах обработки всех страниц.

Не забудьте проверить тип **body**, ведь там может быть файл.

Bool (класс)

Объектами класса **bool** являются логические значения **true** и **false**.

Создание объекта класса **bool**:
```
$bool(true)
```

Console (класс)

Класс предназначен для создания простых интерактивных служб, работающих в текстовом построчном режиме.

Работать такие службы могут в паре со стандартной UNIX программой `inetd`.

Например, можно на Parser реализовать news-сервер (NNTP).
Добавьте такую строку в ваш `/etc/inetd.conf` и перезапустите `inetd`:
```
nntp stream tcp nowait учетная_запись /путь/к/parser3 /путь/к/parser3
/путь/к/nntp.p
```

В скрипте `nntp.p` опишите ваш NNTP сервер.
Что даст возможность людям его использовать — `nntp://ваш_сервер`.

Статическое поле

Чтение строки

$console:line

Такая конструкция считывает строку с консоли.

Запись строки

$console:line[текст]

Такая конструкция выводит строку на консоль.

Cookie (класс)

Класс предназначен для работы с HTTP **cookies**.

Статические поля

Чтение

$cookie:имя_cookie
Возвращает значение **cookie** с указанным именем.

Пример

```
$cookie:my_cookie
```
Считывается и выдается значение **cookie** с именем **my_cookie**.

*Примечание: записанные значения доступны для чтения **сразу**.*

Запись

$cookie:имя[значение]
Сохраняет в **cookie** с указанным именем указанное значение на 90 дней.

Пример

```
$cookie:user[Петя]
```

Создаст **cookie** с именем **user** и запишет в него значение **Петя**. Созданный **cookie** будет храниться на диске пользователя 90 дней.

*Примечание: записанное значение **сразу** доступно для чтения, но это не дает гарантии, что оно будет принято и записано браузером (например в случае если у посетителя **cookie** отключены или блокируются файрволом).*

```
$cookie:имя[
      $.value[значение]
      $.expires(число дней)
]
$cookie:имя[
      $.value[значение]
      $.expires[session]
]
$cookie:имя[
      $.value[значение]
      $.domain[имя домена]
]
$cookie:имя[
      $.value[значение]
      $.path[подраздел]
]
$cookie:имя[
      $.value[значение]
      $.httponly(true)      [3.2.2]
      $.secure(true)        [3.2.2]
]
```

Сохраняет в **cookie** с указанным именем.

Необязательные модификаторы:
$.expires(число дней) – задает число дней (может быть дробным, 1.5=полтора дня), на которое сохраняется **cookie**;
$.expires[session] – создает сеансовый **cookie** (**cookie** не будет сохранятся, а уничтожится с закрытием окна браузера);
$.expires[$date] – задает дату и время, до которой будет храниться **cookie**, здесь **$date** – переменная типа **date**;
$.domain[имя домена] – задает cookie в домен с указанным именем;
$.path[подраздел] – задает cookie только на определенный подраздел сайта.

При записи **cookie** можно указать bool параметр, при это будет сформирован **http** заголовок в котором у **cookie** указан этот параметр без значения. Это может использоваться, например, для задания параметра httponly.

Пример

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

Создаст на две недели **cookie** с именем **login_name** и запишет в него значение **guest**.

fields. Все cookie

```
$cookie:fields
```

Такая конструкция возвращает хеш со всеми **cookie**.

Пример

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

Пример выведет на экран все доступные **cookie** и соответствующие им значения.

Date (класс)

Класс **date** предназначен для работы с датами. Возможные варианты использования – календари, всевозможные проверки, основывающиеся на датах и т.п.

Диапазон возможных значений – от 01.01.1970 до 01.01.2038 года.
Не забывайте, что в нашем календарном времени есть разрывы и нахлесты: во многих странах принято так-называемое «летнее» время, когда весной часы переводят вперед, а осенью назад.
Скажем, в Москве не бывает времени «02:00, 31 марта 2002», а время «02:00, 27 октября 2002» бывает дважды.

Числовое значение объекта класса **date** равно числу суток с EPOCH (01.01.1970 00:00:00, UTC) до даты, заданной в объекте. Этим значением полезно пользоваться для вычисления относительной даты, например:

```
# проверка "обновлен ли файл позже, чем неделю назад?"
^if($last_update > $now-7){
	новый
}{
	старый
}
```

Число суток может быть дробным, скажем, полтора дня = **1.5**.

Обычно класс оперирует локальными датой и временем, однако можно узнать значение хранимой им даты/времени в произвольном часовом поясе, см. **^date.roll[TZ;…]**.

Для общения между компьютерами, работающими в разных часовых поясах, удобно обмениваться значениями даты/времени, не зависящими от пояса – здесь очень удобен UNIX формат, представляющий собой число секунд, прошедших с EPOCH.

Unix формат можно использовать в JavaScript и ряде других языков сценариев, работающих в браузере.

Parser полностью поддерживает работу с UNIX форматом дат.

Конструкторы

create. Относительная дата

```
^date::create(количество суток после EPOCH)
```

Конструктор с одним параметром предназначен для задания **относительных** значений дат. Имея объект класса **date**, можно сформировать новый объект того же класса с датой, смещенной относительно исходной.

Пример

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

В примере получается дата, на неделю большая текущей.

Параметр конструктора не обязательно должен быть целым числом.

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

create. Произвольная дата

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

Создает объект класса **date**, содержащий значение произвольной даты с точностью до секунды. Обязательными параметрами конструктора являются значения года и месяца. Параметры конструктора **day, hour**, **minute**, **second** являются необязательными, если не заданы, подставляются первый день, нулевые час, минута, секунда.

Пример

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

В результате выполнения данного кода, создается объект класса **date**, значения полей которого соответствуют времени появления президента на телевизионном экране в комнате с веб-сервером.

create. Дата или время в стандартном для СУБД формате

```
^date::create[год]
^date::create[год-месяц]
^date::create[год-месяц-день]
^date::create[год-месяц-день часов]
^date::create[год-месяц-день часов:минут]
^date::create[год-месяц-день часов:минут:секунд]
^date::create[год-месяц-день часов:минут:секунд.миллисекунд]        [3.1.3]
^date::create[часов:минут]
^date::create[часов:минут:секунд]
```

Создает объект класса **date**, содержащий значение произвольной даты и/или времени с точностью до секунды. Обязательными частями строки-параметра являются значение **года** или **часа** и **минуты**. **месяц**, **день**, **часов**, **минут**, **секунд**, **миллисекунд** являются необязательными, если не заданы, подставляются первый день, нулевые час, минута, секунда или текущий день.
Замечание: значение **миллисекунд** *игнорируется.*

Удобно использовать этот конструктор для работы с датами, полученными из базы данных, ведь из запроса вы получите значения полей с датой, временем или и датой и временем в виде строк.

Пример

```
# считаем новыми статьи за последние 3 дня
$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){новая}
    <br />
}
```

Внимание пользователям Oracle: чтобы получать дату и время в удобном формате, в строке соединения с сервером укажите формат даты и времени, рекомендованный в Приложении 3.

create. Копирование даты

```
^date::create[объект класса date]
```

Копирует объект класса **date**.

Пример

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

В примере получается дата, на месяц меньше текущей.

now. Текущая дата

```
^date::now[]
^date::now(смещение в днях)
```

Конструктор создает объект класса **date**, содержащий значение текущей даты с точностью до секунды, используя системное время сервера. Если указано, то плюс **смещение в днях**.

Используется локальное время той машины, где работает Parser (локальное время сервера). Для того, чтобы узнать время в другом часовом поясе, используйте **^date.roll[TZ;…]**.

Пример

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

В результате выполнения данного кода, создается объект класса **date**, содержащий значение текущей даты, а на экран будет выведен номер текущего месяца.

unix-timestamp. Дата и время в UNIX формате

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

Конструктор создает объект класса **date**, содержащий значение, соответствующее переданному числовому значению в UNIX формате (см. также краткое описание).

Поля

Через поля объектов класса date могут быть получены следующие величины:

$date.month месяц
$date.year год
$date.day день
$date.hour часы
$date.minute минуты
$date.second секунды
$date.weekday день недели (0 — воскресенье, 1 — понедельник, ...)
$date.week номер недели в году (согласно стандарту ISO 8601) ***[3.1.5]***
$date.weekyear год, к которому принадлежит неделя (согласно стандарту ISO 8601) ***[3.2.2]***
$date.yearday день года (0 — 1-ое января, 1 — 2-ое января, ...)
$date.daylightsaving 1 — летнее время, 0 — стандартное время
$date.TZ часовой пояс; содержит значение, если эта дата была получена **^date.roll[TZ;...]** ***[3.1.1]***

Пример

```
$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
```

В результате выполнения данного кода, создается объект класса **date**, содержащий значение текущей даты, а на экран будет выведено значение:

```
год
месяц
день
час
минута
секунда
день недели
```

Методы

roll. Сдвиг даты

```
^date.roll[year](смещение)
^date.roll[month](смещение)
^date.roll[day](смещение)
^date.roll[TZ][новый часовой пояс]     [3.1.1]
```

С помощью этого метода можно увеличивать/уменьшать значения полей **year**, **month**, **day** объектов класса **date**.

Также можно узнать дату/время, соответствующие хранящимся в объекте класса **date** в другом часовой поясе, задав системное имя нового часового пояса. Список имен см. в документации на вашу операционную систему, ключевые слова: «Переменная окружения TZ».

Пример сдвига месяца

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

В данном примере мы присваиваем переменной **$today** значение текущей даты и затем уменьшаем

номер текущего месяца на единицу. В результате мы получаем номер предыдущего месяца.

Пример сдвига часового пояса

```
@main[]
$now[^date::now[]]
^show[]
^show[Москва;MSK-3MSD]
^show[Амстердам;MET-1DST]
^show[Лондон;GMT0BST]
^show[Нью-Йорк;EST5EDT]
^show[Чикаго;CST6CDT]
^show[Денвер;MST7MDT]
^show[Лос-Анжелес;PST8PDT]

@show[town;TZ]
^if(def $town){
    $town
    ^now.roll[TZ;$TZ]
}{
    Локальное время сервера
}
<br />
$now.year/$now.month/$now.day, $now.hour ч. $now.minute мин.<hr />
```

sql-string. Преобразование даты к виду, стандартному для СУБД

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

Метод преобразует дату к виду **ГГГГ-ММ-ДД ЧЧ:ММ:СС**, который принят для хранения дат в СУБД. Использование данного метода позволяет вносить в базы данных значения дат без дополнительных преобразований.

Пример

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

Получаем строку вида '**2001-11-30 13:09:56**' с текущей датой и временем, которую сразу помещаем в колонку таблицы СУБД. Без использования данного метода пришлось бы выполнять необходимое форматирование вручную. Обратите внимание, данный метод не формирует кавычки, их требуется задавать вручную.

unix-timestamp. Преобразование даты и времени к UNIX формату

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

Преобразует дату и время к значению в UNIX формате (см. краткое описание).

last-day. Получение последнего дня месяца

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

Возвращает последний день месяца.

Пример

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

Возвратит: **29**.

gmt-string. Вывод даты в виде строки в формате RFC 822

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

Метод преобразует дату к строке в формате RFC 822 (**Fri, 23 Mar 2001 09:32:23 GMT**).

В большинстве случаев Parser сам преобразует дату к этому виду (например при формировании HTTP заголовков: `$response:expires[^date::now(+1)]`) и вам не нужно предпринимать никаких действий, однако иногда (например при формировании RSS лент) данный метод может быть востребован.

Статические методы

calendar. Создание календаря на заданную неделю месяца (copy)

```
^date:calendar[rus|eng](год;месяц;день)
```

Метод формирует таблицу с календарем на одну неделю заданного месяца года. Для определения недели используется параметр **день**. Параметр **rus|eng** также как и в предыдущем методе определяет формат календаря. С параметром **rus** дни недели начинаются с понедельника, с **eng** – с воскресенья.

Пример

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

В результате в переменную **$week_of_month** будет помещена таблица с календарем на ту неделю ноября 2001 года, которая содержит 30-е число. Формат таблицы следующий:

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

calendar. Создание календаря на заданный месяц

```
^date:calendar[rus|eng](год;месяц)
```

Метод формирует таблицу с календарем на заданный месяц года. Параметр **rus|eng** определяет формат календаря. С параметром **rus** дни недели начинаются с понедельника, с **eng** – с воскресенья.

Пример

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

В результате в переменную **$calendar_month** будет помещена таблица с календарем на январь 2005 года:

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

В результате работы метода формируется новый объект класса – **table** со столбцами 0...6 плюс столбцы **week** и **year**, в которых выводится номер недели согласно стандарту ISO 8601 и год, к которому она относится.

last-day. Получение последнего дня месяца

^date:last-day(год;месяц)

Возвращает последний день месяца указанного года и месяца.

Пример

```
^date:last-day(2008;02)
```

Возвратит: **29**.

Double, int (классы)

Объектами классов **double** и **int** являются вещественные и целые числа, как заданные пользователем, так и полученные в результате вычислений или преобразований. Числа, относящиеся к классу **double**, имеют представление в формате с плавающей точкой. Диапазон значений зависит от платформы, но, как правило:

для **double**	от **1.7E-308**	до **1.7E+308**
для **int**	от **-2147483648**	до **2147483647**

Класс **double** обычно имеет 15 значащих цифр и не гарантирует сохранение цифр в последних разрядах. Точное количество значащих цифр зависит от используемой вами платформы.

Методы

int, double, bool. Преобразование объектов к числам или bool

^имя.int[]	или	**^имя.int(default)**
^имя.double[]	или	**^имя.double(default)**
^имя.bool[]	или	**^имя.bool(true|false)**

Преобразуют значение переменной **$имя** к целому, вещественному числу или bool соответственно, и возвращает это значение. При преобразовании вещественного числа к целому дробная часть отбрасывается.

Можно задать значение по умолчанию, которое будет получено, если преобразование невозможно. Значение по умолчанию можно использовать при обработке данных, получаемых интерактивно от пользователей. Это позволит избежать появления текстовых значений в математических выражениях при вводе некорректных данных, например, строки вместо ожидаемого числа.

Внимание: пустая строка и строка состоящая только из "white spaces" (символы пробела, табуляция, перевода строки) считается нулем или false.

Примеры

```
$str[Штука]
^str.int(1024)
```

Выведет число **1024**, поскольку объект **str** нельзя преобразовать к классу **int**.

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

Выведет число **1**, поскольку дробная часть будет отброшена.

```
^if(^form:search_in_text.bool(false)){
      ...ищем в тексте...
}
```

inc, dec, mul, div, mod. Простые операции над числами

^имя.inc[] – увеличивает значение переменной на 1 или **число**
^имя.inc(число)
^имя.dec[] – уменьшает значение переменной на 1 или **число**
^имя.dec(число)
^имя.mul(число) – умножает значение переменной на **число**
^имя.div(число) – делит значение переменной на **число**
^имя.mod(число) – помещает в переменную остаток от деления ее значения на **число**

Пример

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

Пример возвратит **4.5** и эквивалентен записи **$var((5+7-3)/4*2)**.

format. Вывод числа в заданном формате

^имя.format[форматная строка]

Метод выводит значение переменной в заданном формате (см. Форматные строки).

Если выводить не пользоваться **format**, и выводить число просто так:
$имя
то для чисел с нулевой дробной частью выполняется
^имя.format[%.0f] ***[3.1.5]***
для остальных
^имя.format[%g]

Примеры

```
$var(15.67678678)
^var.format[%.2f]
```

Возвратит: **15.68**

```
$var(0x123)
^var.format[0x%04X]
```

Возвратит: **0x0123**

Статические методы

sql. Получение числа из базы данных

^int:sql{запрос}
^int:sql{запрос}[$.limit(1) $.offset(o) $.default(выражение)]
^double:sql{запрос}
^double:sql{запрос}[$.limit(1) $.offset(o) $.default(выражение)]

Возвращает число, полученное в результате SQL-запроса к серверу баз данных. Запрос должен возвращать значение из одного столбца одной строки.

Запрос – запрос к базе данных, написанный на языке SQL
$.offset(o) – отбросить первые o записей выборки

если ответ SQL-сервера был пуст (0 записей), то будет...
$.default{код} ...выполнен указанный **код**, и число, которое он возвратит, будет результатом метода;
$.default(выражение) ...вычислено указанное **выражение**, и оно будет результатом метода.

Для работы этого метода необходимо установленное соединение с сервером базы данных (см. оператор **connect**).

Пример

```
^connect[строка подключения]{
      ^int:sql{select count(*) from news}
}
```

Вернет количество записей в таблице **news**.

Env (класс)

Класс предназначен для получения значения переменных окружения. Со списком стандартных переменных окружения можно ознакомиться по адресу http://www.w3c.org/cgi. Веб-сервер Apache задает значения ряда дополнительных переменных.

Статические поля. Получение значения переменной окружения

$env:переменная_окружения

Возвращает значение указанной переменной окружения.

Пример

```
$env:REMOTE_ADDR
```

Возвратит IP-адрес машины, с которой был запрошен документ.

Получение значения поля запроса

$env:HTTP_ПОЛЕ_ЗАПРОСА

Такая конструкция возвращает значение поля запроса, передаваемое браузером веб-серверу (по HTTP протоколу).

Пример

```
^if(^env:HTTP_USER_AGENT.pos[MSIE]>=0){
    Пользователь, вероятно, использует Microsoft Internet Explorer<br />
}
```

Поля запроса имеют имена в верхнем регистре и начинающиеся с **HTTP_**, и знаки '-' в них заменены на '_'.
Подробности в документации на ваш веб-сервер.

Получение версии Parser

$env:PARSER_VERSION

Такая конструкция возвращает полную версию Parser с указанием платформы.

Например...

```
3.2.0 (compiled on i386-pc-win32)
```

File (класс)

Класс **file** предназначен для работы с файлами. Объекты класса могут быть созданы различными способами:

1. методом POST через поле формы **<form method="post" enctype="multipart/form-data">...<input name="photo" type="file">**.

2. одним из конструкторов класса **file**.

При передачи файлов клиентам (например, методом **mail:send** или через поле **response:body**) необходимо задавать HTTP-заголовок **content-type**. В Parser для определения типа файла по расширению его имени существует таблица **MIME-TYPES**, определенная в Конфигурационном методе (см. главу Настройка). По ней, в зависимости от расширения файла, Parser автоматически определяет

нужный тип данных для передачи в строке **content-type**. Если тип данных не удается определить по таблице, используется тип `application/octet-stream`.

Для проверки существования файлов и каталогов есть специальные операторы.

Конструкторы

load. Загрузка файла с диска или HTTP-сервера

```
^file::load[формат;имя файла]
^file::load[формат;имя файла;опции загрузки]
^file::load[формат;имя файла;новое имя файла]
^file::load[формат;имя файла;новое имя файла;опции загрузки]
```

Загружает файл с диска или HTTP-сервера.

формат – формат представления загружаемого файла. Может быть **text**(текстовый) или **binary**(двоичный). Различие между этими типами в разных символах переноса строк. Для PC эти символы **0D 0A**. При использовании формата **text** при загрузке **0D** отбросится за ненадобностью, при записи методом **save** добавится.

имя файла – имя файла с путем или URL файла на HTTP-сервере.

Необходимо иметь ввиду, что если в конструкторе задан параметр **новое имя файла**, его значение будет присвоено полю **name**. Этим параметром удобно пользоваться при использовании метода **mail:send** для передачи файла под нужным именем.

опции загрузки – см. «Работа с HTTP-серверами».

Если файл был загружен с HTTP-сервера, поля заголовков HTTP-ответа в верхнем регистре доступны как поля объекта класса **file**.
Также доступно поле **tables**, это хеш, ключами которого являются поля заголовки HTTP-ответа в верхнем регистре, а значениями таблицы с единственным столбцом **value**, содержащими все значения одноименных полей HTTP-ответа. ***[3.1.1]***

Пример загрузки файла с диска

```
$f[^file::load[binary;article.txt]]
Файл с именем $f.name имеет размер $f.size и содержит текст:<br />
$f.text
```

Выведет размер, имя и текст файла.

Пример загрузки файла с HTTP-сервера

```
$file[^file::load[text;http://www.parser.ru/;
      $.timeout(5)
]]
Программное обеспечение сервера: $file.SERVER
<hr />
<pre>$file.text</pre>
```

sql. Загрузка файла из SQL-сервера

```
^file::sql{запрос}
^file::sql{запрос}[$.name[имя] $.content-type[пользовательский тип] $.limit(1)
$.offset(o)]
```

Загружает файл из SQL-сервера. Результатом выполнения запроса должна быть одна запись (при необходимости воспользуйтесь опцией `limit`).
Считается, что:

- первая колонка содержит данные файла;
- вторая колонка содержит имя файла;
- третья колонка содержит `content-type` файла (если не указан, он будет определен по таблице **`$MIME-TYPES`**).

Необязательные параметры:
`$.limit(1)` – в ответе заведомо будет содержаться только одна строка; ***[3.3.0]***
`$.offset(o)` – отбросить первые **`o`** записей выборки; ***[3.3.0]***
`$.content-type[пользовательский тип]` – задать пользовательский `content-type`; ***[3.1.4]***
`$.name[имя]` – задать имя файла. ***[3.1.4]***

Имя файла и его `content-type` будет переданы посетителю при **`$response:download`**.

Примечание: пока работает только с MySQL сервером.

stat. Получение информации о файле

`^file::stat[имя файла]`
Объект, созданный этим конструктором, имеет дополнительные поля (объекты класса **`date`**):

`$файл.size` – размер файла в байтах;
`$файл.cdate` – дата создания;
`$файл.mdate` – дата изменения;
`$файл.adate` – дата последнего обращения к файлу.

`имя файла` – имя файла с путем.

Пример

```
$f[^file::stat[some.zip]]
Размер в байтах: $f.size<br />
Год создания: $f.cdate.year<br />
$new_after[^date::now(-3)]
Статус: ^if($f.mdate >= $new_after){новый;старый}
```

cgi и exec. Исполнение программы

```
^file::cgi[имя файла]
^file::cgi[имя файла;env_hash]
^file::cgi[имя файла;env_hash;аргумент1;аргумент2;...]
^file::cgi[формат;имя файла;env_hash;аргумент1;аргумент2;...] [3.2.2]
^file::exec[имя файла]
^file::exec[имя файла;env_hash]
^file::exec[имя файла;env_hash;аргумент1;аргумент2;...]
^file::exec[формат;имя файла;env_hash;аргумент1;аргумент2;...]        [3.2.2]
```

Конструктор **`cgi`** создает объект класса **`file`**, содержащий результат исполнения программы в соответствии со стандартом CGI.
Внимание: все пути в парсере указываются относительно текущего исполняемого файла.
По аналогии при запуске внешнего скрипта текущим каталогом для него является каталог, где находится этот скрипт.

Заголовки, которые выдаст CGI-скрипт, конструктор поместит в поля класса **`file`** в ВЕРХНЕМ регистре. Например, если некий скрипт **`script.pl`**, среди прочего, выдает в заголовке строку **`field:value`**, то после работы конструктора
`$f[^file::cgi[script.pl]]`,
обратившись к **`$f.FIELD`**, получим значение **`value`**.

Конструктор **`exec`** аналогичен **`cgi`**, но не отделяет HTTP-заголовки от текста, возвращаемого скриптом.

`Формат` – формат представления получаемых от скрипта данных. Может быть **`text`** (по умолчанию)

или **binary**. При использовании формата **binary** не будут производиться перекодирования полученных данных в кодировку **$request:charset** и их обрезания по первому нулевому символу.

Имя файла — имя файла с путем.

Объект, созданный этими конструкторами, имеет дополнительные поля:
status — информация о статусе завершении программы (обычно 0 означает, что программа завершилась успешно, не 0 — с ошибкой)
stderr — результат считывания стандартного потока ошибок

Пример:
```
$cgi_file[^file::cgi[new.cgi]]
$cgi_file.text
```

Выведет на экран результаты работы скрипта **new.cgi**.

Необязательные параметры конструкторов:
env_hash — хеш, в котором могут задаваться
- дополнительные переменные окружения, которые впоследствии будут доступны внутри исполняемого скрипта;
- ключ **stdin**, содержащая текст, передаваемый исполняемому скрипту в стандартном потоке ввода;
- ключ **charset**, задающий кодировку, в которой работает скрипт (будут перекодированы данные передаваемые скрипту и получаемые из скрипта). ***[3.1.3]***

Внимание: можно задавать только стандартные CGI переменные окружения и переменные, имена которых начинаются с CGI_ или HTTP_ (допустимы латинские буквы в ВЕРХНЕМ регистре, цифры, подчеркивание, минус).
Внимание: при обработке `HTTP POST` *запроса, при помощи конструкции* **$.stdin[$request:body]** *вы можете передать в стандартный поток ввода скрипта полученные вами POST-данные. [3.0.8, раньше они передавались по-умолчанию]*
Внимание: запускаемому скрипту также передаются все переменные окружения, которые были выставленные http сервером при запуске Parser.

Пример внешнего CGI-скрипта
```
$search[^file::cgi[search.cgi;$.QUERY_STRING[text=$form:q&page=$form:p]]]
```

Пример внешнего скрипта
```
$script[^file::exec[script.pl;$.CGI_INFORMATION[этого мне не хватало]]]
```

Внутри скрипта **script.pl** можно воспользоваться переданной информацией:
```
print "Дополнительная информация: $ENV{CGI_INFORMATION}\n";
```

Пример получения бинарных данных от внешнего скрипта
```
$response:body[^file::exec[binary;getfile.pl;$.CGI_FILENAME[$form:filename]]]
```

Пример передачи нескольких аргументов
Кроме того, вызываемой программе можно передать ряд аргументов, перечислив их через точку с запятой после хеша переменных окружения:
```
$script[^file::exec[script.pl;;длина;ширина]]
```

...или передать методу список аргументов, заданный в виде таблицы с одним столбцом: ***[3.2.2]***
```
$args[^table::create{arg
длина
ширина}]
$script[^file::exec[script.pl;;$args]]
```

Внимание: настоятельно рекомендуется хранить запускаемые скрипты вне веб-пространства, поскольку запуск скрипта с произвольными параметрами может привести к неожиданным результатам.

base64. Декодирование из Base64

`^file::base64[закодированное]`

Декодирует файл из Base64 представления. Для кодирования файла используйте
`^файл.base64[]`

Подробная информация о Base64 доступна здесь: http://www.ietf.org/rfc/rfc2045.txt и здесь http://en.wikipedia.org/wiki/Base64

Пример

```
$encoded[
R0lGODdhyAAyANUAAP////j88fLz8O/v7+v21uns4uTyyd/f397vu9vf1NfsrtDpoM/Pz8rmk8Tj
hr+/v73geK+vr6nWUJ+fn52jkJzQNZXNJ4+Pj39/f3BwcGBgYFBQUEBAQDAwMCAgIBAQEAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACwAAAAAyAAyAAAG/0CAcEgsChkRjHIZ
ORif0Kh0Sq1ar9isdkudaD6gsHj80US46LR6zW5zBxjweE73YAbuvH7P71/kdIFzHxN9hoeIiUQH
HIKOgRx4ipOUlVgPgHQcGUsYGY2CHwyWpKWlE4IbZ1ADE6CDo6ays3wRkE5VD69iorS+v2gMmSAf
q1h/g7jAy8ysHnMdy1nC0M3W1wAZ0JJvz2MY2OG+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" />
```

Выведет...

create. Создание текстового файла

`^file::create[формат;имя;текст]`
`^file::create[формат;имя;текст;опции]` *[3.4.0]*

Создает объект класса **`file`**, с указанными **`именем`** и **`текстовым`** содержимым.

`Формат` – формат представления создаваемого файла. Пока может быть только **`text`**(текстовый) .
`Опции` – хеш, в котором можно указать кодировку для создаваемого текстового файла:
`$.charset`[кодировка]

*Примечание: если файл нужно сохранить на диск **сервера**, есть более простой подход:*
^string.save[…].

Пример выгрузки данных в XML виде

```
#export.html
^connect[строка соединения]{
$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]
}
```

При открытии этого документа произойдет создание файла `export.xml` и браузер предложит посетителю сохранить этот файл.
Получится такой текстовый файл:

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<products>
      <product id="1" name="Бывает &quot;В кавычках&quot;"/>
      <product id="2" name="Johnson&amp;Johnson"/>
</products>
```

Поля

name. Имя файла
$файл.name

Поле содержит имя файла. Объект класса **file** имеет поле **name**, если пользователь закачал файл через поле формы. Также в конструкторе **file::load** может быть указано альтернативное имя файла.

size. Размер файла
$файл.size

Поле содержит размер файла в байтах.

text. Текст файла
$файл.text

Поле содержит текст файла. Использование этого поля позволяет выводить на странице содержимое текстовых файлов или результатов работы **file::cgi** и **file::exec**.

Дополнительная информация о файле
$файл.cdate – дата создания;
$файл.mdate – дата изменения;
$файл.adate – дата последнего обращения к файлу.

Поля доступны если объект получен конструкторами **file::stat** или **file::load** путём загрузки локального файла ***[3.3.0]***

stderr. Текст ошибки выполнения программы
$файл.stderr

При выполнении **file::cgi** и **file::exec** сюда попадает текст из стандартного потока ошибок программы.

status. Статус получения файла
$файл.status

При выполнении **file::cgi** и **file::exec** в поле **status** попадает статус выполнения программы (0=успех).
При выполнении **file::load** с HTTP сервера, сюда попадает статус выполнения HTTP запроса (200=успех).

content-type. MIME-тип файла
$файл.content-type

Поле может содержать MIME-тип файла. При выполнении CGI-скрипта (см. **file::cgi**) MIME-тип может задаваться CGI-скриптом, полем заголовка ответа «content-type». При загрузке (см. **file::load**) или получении информации о файле (см. **file::stat**) MIME-тип определяется по таблице **$MAIN:MIME-TYPES** (см. «Конфигурационный метод»), если в таблице расширение имени файла найдено не будет, будет использован тип «application/octet-stream».

Поля HTTP-ответа
Если файл был загружен с HTTP-сервера, поля заголовков HTTP-ответа доступны, как поля объекта класса **file**:

$файл.ПОЛЕ_HTTP_ОТВЕТА (ЗАГЛАВНЫМИ БУКВАМИ)

Например: **$файл.SERVER**.

Если один заголовок повторяется в ответе несколько раз, все его значения доступны в поле **tables**:

```
$.tables[
   $.HTTP-ЗАГОЛОВОК[таблица значений, единственный столбец value]
]
```

Пример:

```
$f[^file::load[binary;http://www.parser.ru]]
^f.tables.foreach[key;value]{
    $key=^value.menu{$value.value}[|]<br />
}
```

Методы

save. Сохранение файла на диске

^файл.save[формат;имя файла]
^файл.save[формат;имя файла;опции] ***[3.4.0]***

Метод сохраняет объект в файл в заданном формате под указанным именем.

Формат — формат сохранения файла (**text** или **binary**);
Имя файла — имя файла и путь, по которому он будет сохранен.
Опции — хеш, в котором можно указать кодировку для сохраняемого текстового файла:
$.charset[кодировка**]**

Примечание: если файл с указанным именем существует, то он будет молча перезаписан.
Примечание: чтобы добавить текст в файл, используйте ^string.save[append;...].

Пример
```
^archive.save[text;/arch/archive.txt]
```

Пример сохранит объект класса **file** в текстовом формате под именем **archive.txt** в каталог **/arch/**.

sql-string. Сохранение файла на SQL-сервере

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

Выдает строку, которую можно использовать в SQL-запросе. Позволяет сохранить файл в базе данных.

Внимание: на данный момент реализована поддержка только MySQL-сервера.

Пример

```
$name[image.gif]
$file[^file::load[$name]]
^connect[строка соединения]{
      ^void:sql{insert into images (name, bytes) values ('$name', '^file.sql-
string[]')}
}
```

base64. Кодирование в Base64

```
^file.base64[]
```

Метод позволяет преобразовать файл в Base64 форму.
Чтобы преобразовать файл из Base64 к исходному виду, воспользуйтесь
^file::base64[закодированное]

Подробная информация о Base64 доступна здесь: http://www.ietf.org/rfc/rfc2045.txt и здесь http://en.wikipedia.org/wiki/Base64

Пример

```
$original[^file::load[binary;http://www.parser.ru/i/artlebedev.gif]]
<pre>^original.base64[]</pre>
```

Выведет...

**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-отпечаток файла

```
^file.md5[]
```

Для файла будет получен «отпечаток» размером 16 байт.
Выдает его представление в виде строки — байты представлены в шестнадцатиричном виде без разделителей, в нижнем регистре.

Считается, что практически невозможно

- создать две строки, имеющие одинаковый «отпечаток»;
- восстановить исходную строку по ее «отпечатку».

Подробная информация о MD5 доступна здесь: http://www.ietf.org/rfc/rfc1321.txt

crc32. Подсчет контрольной суммы файла

```
^file.crc32[]
```

Для файла будет подсчитана контрольная сумма (CRC32).
Метод выдает её в виде целого числа.

Статические методы

delete. Удаление файла с диска

```
^file:delete[путь]
```

Удаляет указанный файл.
Путь – путь к файлу

Если после удаления в каталоге больше ничего не осталось, каталог тоже удаляется(если это возможно).

Пример

```
^file:delete[story.txt]
```

find. Поиск файла на диске

```
^file:find[файл]
^file:find[файл]{код, если файл не найден}
```

Метод возвращает строку (объект класса **string**), содержащую имя файла с путем от корня веб пространства, если он существует по указанному пути, либо в каталогах более высокого уровня. В противном случае выполняется заданный код, если он указан.

Пример без указания пути

```
<img src="^file:find[header.gif]{/i/header.gif}" />
```

Допустим, этот код расположен в документе `/news/sport/index.html`, здесь ищется файл **header.gif** в каталоге **/news/sport/**, разработанный специально для раздела спортивных новостей. Если он не найден, и не существует **/news/sport/header.gif**, то используется стандартный заголовочный рисунок новостного раздела.

Пример с указанием пути

```
<img src="^file:find[/i/$section/$subsection/header.gif]" />
```

Здесь ищется файл **header.gif** в каталоге **/i/раздел/подраздел/**. Если он не найден, он будет последовательно искаться в каталогах

- **/i/раздел/**
- **/i/**
- **/**

list. Получение оглавления каталога

```
^file:list[путь]
^file:list[путь;фильтр]
```

Формирует таблицу (объект класса **table**) с одним столбцом **name**, содержащим файлы и каталоги по указанному пути, имена которых удовлетворяют шаблону, если он задан.

фильтр – строка с регулярным выражением (см. метод **match** класса **string**) или объект **regex** ***[3.4.0]***. Без указания фильтра будут выведены все найденые по заданному пути файлы

Пример

```
$list[^file:list[/;\.zip^$]]
^list.menu{
      $list.name<br />
}
```

Выведет имена всех архивных файлов с расширением имени **.zip**, находящихся в корневом каталоге веб-сервера.

copy. Копирование файла

```
^file:copy[имя файла источника;имя нового файла]
```

Метод копирует файл.

Внимание: необходимо крайне осторожно относиться к возможности записи в веб-пространстве, поскольку возможностью что-нибудь куда-нибудь записать нередко пользуются современные герострaты.

Пример

```
^file:copy[/path/source.txt;/path/destination.txt]
```

Скопирует файл **source.txt**.

move. Перемещение или переименование файла

```
^file:move[старое имя файла;новое имя файла]
```

Метод переименовывает или перемещает файл и каталог (для платформы Win32 объекты нельзя перемещать через границу диска). Новый каталог создается с правами 775. Каталог старого файла удаляется, если после выполнения метода он остается пустым.

Внимание: необходимо крайне осторожно относиться к возможности записи в веб-пространстве, поскольку возможностью что-нибудь куда-нибудь записать нередко пользуются современные герострaты.

Пример

```
^file:move[/path/file1;/file1]
```

Переместит файл **file1** в корень веб-пространства.

lock. Эксклюзивное выполнение кода

```
^file:lock[имя файла-блокировки]{код}
```

Код не выполняется одновременно, для обеспечения эксклюзивности используется **файл-блокировки**.

Пример

```
^file:lock[/counter.lock]{
    $file[^file::load[text;/counter.txt]]
    $string[^eval($file.text+1)]
    ^string.save[/counter.txt]
}
Количество посещений: $string<br />
```

В отсутствие блокировки, два одновременных обращения к странице могли вызвать увеличение

счетчика... на 1, а не на 2:

- пришел первый;
- пришел второй;
- считал первый, значение счетчика 0;
- считал второй, значение счетчика 0;
- увеличил первый, значение счетчика 1;
- увеличил второй, значение счетчика 1;
- записал первый, значение счетчика 1;
- записал второй **поверх только что записанного первым**, значение счетчика **1, а не 2**.

Внимание: всегда думайте об одновременно приходящих запросах. При работе с базами данных обычно есть встроенные в SQL-сервер средства для их корректной обработки.

Внимание: при использовании более одной блокировке всегда думайте об их взаимном сочетании, чтобы избежать ситуации «А ждет Б, Б ждет А», так-называемого deadlock.

dirname. Путь к файлу

```
^file:dirname[filespec]
```

Из пути к файлу или каталогу (**filespec**) получает только путь.

Пример

```
#имя файла
^file:dirname[/a/some.tar.gz]
#имя каталога...
^file:dirname[/a/b/]
```

Оба вызова выдадут:

```
/a
```

basename. Имя файла без пути

```
^file:basename[filespec]
```

Из полного пути к файлу (**filespec**) получает имя файла с расширением имени, но без пути.

Пример

```
^file:basename[/a/some.tar.gz]
```

...выдаст...

```
some.tar.gz
```

justname. Имя файла без расширения

```
^file:justname[filespec]
```

Из полного пути к файлу (**filespec**) получает имя файла без пути и расширения имени.

Пример

```
^file:justname[/a/some.tar.gz]
```

...выдаст...

```
some.tar
```

justext. Расширение имени файла

```
^file:justext[filespec]
```

Из полного пути к файлу (**filespec**) получает расширение имени файла без точки.

Пример

```
^file:justext[/a/some.tar.gz]
```

...выдаст...

```
gz
```

fullpath. Полное имя файла от корня веб-пространства

```
^file:fullpath[имя файла]
```

Из **имени файла** получает полное имя файла от корня веб-пространства. См. также «Приложение 1. Пути к файлам и каталогам».

Пример: в странице `/document.html` вы создаете ссылку на картинку, но настоящий адрес запрошенного документа может быть иным, скажем, при применении модуля `mod_rewrite` веб-сервера Apache, если поставить относительную ссылку на картинку, она не будет отображена браузером, поскольку браузер относительные пути разбирает относительно к текущему запрашиваемому документу, и ничего не знает про то, что на веб-сервере использован `mod_rewrite`.

Поэтому удобно заменить относительное имя на полное:

```
$image[^image::measure[^file:fullpath[image.gif]]]
^image.html[]
```

Такая конструкция...

```
<img src="/image.gif" width="..." height="..." />
```

...создаст код, содержащий абсолютный путь.

base64. Кодирование в Base64

```
^file:base64[имя файла]
```

Метод позволяет преобразовать файл с указанным именем в Base64 форму.
Чтобы преобразовать файл к исходному виду, воспользуйтесь

```
^file::base64[закодированное]
```

Использование описываемого статического метода полностью равносильно следующему коду (за исключением того, что описываемый метод использует меньше памяти):

```
$f[^file::load[binary;filespec]]
^f.base64[]
```

Имейте в виду, что результат не будет совпадать с результатом работы такого кода:

```
$f[^file::load[binary;filespec]]
^f.text.base64[]
```

т.к. в последнем случае при обращении к полю text содержимое файла будет обрезано по первону нулевому символу и все символы перевода строк будут нормализованы.

md5. MD5-отпечаток файла

```
^file:md5[имя файла]
```

Для файла с указанным именем будет получен «отпечаток» размером 16 байт.
Выдает его представление в виде строки — байты представлены в шестнадцатиричном виде без разделителей, в нижнем регистре.

Считается, что практически невозможно
- создать две строки, имеющие одинаковый «отпечаток»;
- восстановить исходную строку по ее «отпечатку».

Подробная информация о MD5 доступна здесь: http://www.ietf.org/rfc/rfc1321.txt

crc32. Подсчет контрольной суммы файла

```
^file:crc32[имя файла]
```

Для файла с указанным именем будет подсчитана контрольная сумма (CRC32).
Метод выдает её в виде целого числа.

Form (класс)

Класс **form** предназначен для работы с полями форм. Класс имеет статические поля, доступные только для чтения.

Для проверки заполнения формы и редактирования имеющихся записей из базы данных удобно использовать такой подход:

```
^if($edit){
# запись из базы
    $record[^table::sql{... where id=...}]
}{
# новая запись, ошибка при заполнении, необходимо вывести
# поля формы
    $record[$form:fields]
}
<input name="age" value="$record.age" />
```

Статические поля

Получение значения поля формы

```
$form:поле_формы
```

Такая конструкция возвращает значение поля формы. Возвращаемый объект может принадлежать либо классу **file**, если поле формы имеет тип **file**, либо классу **string**. Дальнейшая работа с объектом возможна только методами, определенными для соответствующих классов.

Поле без имени считается имеющим имя **nameless**.
Координаты нажатия пользователем на картинку с атрибутом **ISMAP** доступны через **$form:imap**.

Наобходимо помнить, что если в html используется <input type="image" name="fieldname" />, то при нажатии пользователем на эту кнопку мышью, браузером на сервер передаются координаты места произошедшего события в полях **fieldname.x** и **fieldname.y**.

Пример: текстовое поле, поле типа image и загрузка файла

```
^if(def $form:photo){
```

```
      ^form:photo.save[binary;/upload/photos/beauty.^file:justext[$form:photo.n
ame]]
      Файл $form:photo.name загружен на сервер.
}
^if(def $form:user){
      Пользователь: $form:user<br />
}
^if(def $form:[action.x]){
      Координаты:<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>
```

Сохранит картинку, выбранную пользователем в поле формы и присланную на сервер, в заданном файле.

Пример: безымянное поле

```
<img src="/show.html?123&a=b">
```

Внутри `show.html` строка **123** доступна как **`$form:nameless`**.

imap. Получение координат нажатия в ISMAP

`$form:imap`

Если пользователь нажал на картинку с атрибутом **`ISMAP`**, такая конструкция возвращает хеш с полями **`x`** и **`y`**, в которых доступны координаты нажатия.

Пример

В файле `/go.html` напишите:

```
$clicked[$form:imap]
^if(def $clicked){
    Пользователь нажал на ISMAP ссылке:<br />
    x=$clicked.x<br />
    y=$clicked.y<br />
}
```

В файле `/test.html` напишите:

```
<a href="/go.html?a=b"><img src="map.png" ismap /></a>
```

Откройте в браузере `/test.html` и нажмите мышкой на картинке, это приведет к переходу по адресу... `/go.html?a=b?10,30`

...и вы увидите...

```
    Пользователь нажал на ISMAP ссылке:
    x=10
    y=30
```

qtail. Получение остатка строки запроса

$form:qtail

Возвращает часть **$request:query** после второго **?**.

Пример

Предположим, пользователь запросил такую страницу:
`http://www.mysite.ru/news/article.html?year=2000&month=05&day=27?thisText`

Тогда:
$form:qtail

вернет:
thisText

fields. Все поля формы

$form:fields

Такая конструкция возвращает хеш со всеми полями формы или параметрами, переданными через URL. Имена ключей хеша те же, что и у полей формы, значениями ключей являются значения полей формы.

Пример

```
^form:fields.foreach[field;value]{
      $field — $value
}[<br />]
```

Пример выведет на экран все поля формы и соответствующие значения.
Предположим, что URI страницы
www.mysite.ru/testing/index.html?name=dvoechnik&mark=2. Тогда пример выдаст следующее:

```
name – dvoechnik
mark — 2
```

tables. Получение множества значений поля

$form:tables

Такая конструкция возвращает хеш со всеми полями формы или параметрами, переданными через URL. Имена ключей хеша те же, что и у полей формы, значениями же являются таблицы, см. ниже.

$form:tables.поле_формы

Если поле формы имеет хотя бы одно значение, такая конструкция возвращает таблицу (объект класса **table**) с одним столбцом **field**, содержащим все значения поля. Используется для получения множества значений поля.
Внимание: не забудьте проверить наличие таблицы перед тем, как начать ею оперировать.

Пример

```
Выберите, чем вы увлекаетесь в свободное время:
<form method="POST">
   <p><input type=checkbox name=hobby value="Театр">Театром</p>
   <p><input type=checkbox name=hobby value="Кино">Кино</p>
   <p><input type=checkbox name=hobby value="Книги">Книгами</p>
   <p><input type=submit value="OK"></p>
</form>
$hobby[$form:tables.hobby]
```

```
^if($hobby){
    Ваши хобби:<br />
    ^hobby.menu{
        $hobby.field
    }[<br />]
}{
    Ничего не выбрано
}
```

Пример выведет на экран выбранные варианты или напишет, что ничего не выбрано.

files. Получение множества файлов

$form:files

Такая конструкция возвращает хеш со всеми файлами формы. Имена ключей хеша те же, что и у полей формы, значениями же являются хеши, см. ниже.

$form:files.поле_формы

Если поле формы имеет хотя бы одно значение типа файл, такая конструкция возвращает хеш (объект класса **hash**) с ключами 0, 1, 2... (по количеству переданных файлов), содержащий все файлы с указанным именем. Используется для получения множества файлов с одинаковым именем формы.

Внимание: не забудьте проверить наличие хеша перед тем, как начать им оперировать.

Пример

```
^if(def $form:picture){
      <p>Загружены изображения (^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>Выберите несколько изображений для загрузки:<br />
      <input type="file" name="picture" /><br />
      <input type="file" name="picture" /><br />
      <input type="file" name="picture" /><br />
      <input type="submit" value="Загрузить" />
   </p>
</form>
```

Внимание: загруженные изображения выводятся в произвольном порядке.

Hash (класс)

Класс предназначен для работы с хешами – ассоциативными массивами. Хеш считается определенным (**def**), если он не пустой. Числовым значением хеша является число ключей (значение, возвращаемое методом **^хеш._count[]**).

Конструкторы

Обычно хеши создаются не конструкторами, а так, как описано в разделе "Конструкции языка Parser".

create. Создание пустого и копирование хеша

```
^hash::create[]
^hash::create[существующий хеш или хешфайл]
```

Если параметр не задан, будет создан пустой хеш.
Если указан **существующий хеш** или **хешфайл**, конструктор создает его копию.

Пустой хеш, создаваемый конструктором без параметров, нужен в ситуации, когда необходимо динамически наполнить хеш данными, например:

```
$dyn[^hash::create[]]
^for[i](1;10){
	$dyn.$i[$value]
}
```

Перед выполнением **for** мы определили, что именно наполняем.

Если предполагается интенсивная работа по изменению содержимого хеша, но необходимо сохранить, скажем, значения по умолчанию, например:

```
$pets[
	$.pet[Собака]
	$.food[Косточка]
	$.good[Ошейник]
]
$pets_copy[^hash::create[$pets]]
```

Замечание: поле **_default** *копируется.* ***[3.1.4]***

sql. Создание хеша на основе выборки из базы данных

```
^hash::sql{запрос}
^hash::sql{запрос}[$.limit(n) $.offset(o) $.distinct(true/false)
$.bind[variables hash] $.type[hash|string|table]]
```

Конструктор создает хеш, в котором имена ключей совпадают со значениями первого столбца выборки. Имена столбцов формируют ключи хеша, а значения столбцов – соответствующие этим ключам значения.
Если же запрос возвращает только один столбец, формируется хеш, где значения столбца формируют ключи хеша, и им ставится в соответствие логическое значение **истина**. ***[3.1.2]***

Дополнительные параметры конструктора:

$.limit(n)	получить только **n** записей
$.offset(o)	отбросить первые **o** записей выборки
$.bind[hash] ***[3.1.4]***	связанные переменные, см. «Работа с IN/OUT переменными»
$.distinct(true/false)	**false** или **0**=считать наличие дубликата ошибкой (по-умолчанию); **true** или **1**=выбрать из таблицы записи с уникальным ключом.
$.type[hash/string/table] ***[3.3.0]***	**hash**=значение каждого элемента – хеш (по умолчанию); **string**=значение каждого элемента – строка, при этом вы должны указать ровно **два** столбеца в SQL запросе; **table**=значение каждого элемента – таблица.

По-умолчанию, наличие в ключевом столбце одинаковых значений считается ошибкой, если вам необходимо именно отобрать из результата записи с уникальным ключом, задайте опцию **$.distinct(true)**.
Примечание: имейте в виду, что так между клиентом и сервером передаются лишние данные, и, скорее всего, запрос можно изменить, чтобы необходимая уникальность ключа обеспечивалась SQL-сервером. Если вам необходимы данные и в виде таблицы и в виде хеша, подумайте над использованием ***table::sql*** *в паре с* ***table.hash****.*

Пример hash of hash

В БД содержится таблица `hash_table`:

```
pet   food  aggressive
cat   milk  very
dog   bone  never
```

Выполнение кода...

```
^connect[строка подключения]{
	$hash_of_hash[^hash::sql{
		select
			pet,
			food,
			aggressive
		from
			hash_table
	}]
}
```

...даст хеш такой структуры...

```
$hash_of_hash[
	$.cat[
		$.food[milk]
		$.aggressive[very]
	]
	$.dog[
		$.food[bone]
		$.aggressive[never]
	]
]
```

...из которого можно эффективно извлекать информацию, например, так:

```
$animal[cat]
$animal любит $hash_of_hash.$animal.food
```

Пример hash of bool *[3.1.2]*

В БД содержится таблица `participants`:

```
name
Константин
Александр
```

Выполнение кода...

```
^connect[строка подключения]{
	$participants[^hash::sql{select name from participants}]
}
```

...даст хеш такой структуры...

```
$participants[
	$.Константин(true)
	$.Александр(true)
]
```

...из которого можно эффективно извлекать информацию, например, так:

```
$name[Иван]
$name ^if($participants.$name){участвует}{не участвует} в мероприятии
```

Поля

В качестве поля хеша выступает ключ, по имени которого можно получить значение:

```
$my_hash.key
```

Такая запись возвратит значение, поставленное в соответствие ключу. Если происходит обращение к

несуществующему ключу, будет возвращено значение ключа **_default**, если он задан в хеше.

Присваивание ключу значения добавит или обновит пару ключ/значение в хеш:
$my_hash.key[значение]

Для большей взаимозаменяемости таблиц и хешей поле **fields** хранит ссылку на сам хеш, см. «Использование хеша вместо таблицы».

Использование хеша вместо таблицы

$хеш.fields — сам хеш.

Для большей взаимозаменяемости таблиц и хешей поле **fields** хранит ссылку на сам хеш. См. **table.fields**.

Методы

_keys. Список ключей хеша

^хеш._keys[]
^хеш._keys[имя столбца] ***[3.2.2]***

Метод возвращает таблицу (объект класса **table**), содержащую единственный столбец, где перечислены все ключи хеша (начиная с версии ***3.4.0*** порядок ключей в полученной таблице соответствует порядку добавления элементов в хеш, до этой версии — порядок не определен). Имя столбца — «key» или переданное **имя столбца**.

Пример

```
$man[
      $.name[Вася]
      $.age[22]
      $.sex[m]
]
$tab_keys[^man._keys[]]
^tab_keys.save[keys.txt]
```

Будет создан файл `keys.txt` с такой таблицей:

key
name
age
sex

_count. Количество ключей хеша

```
^хеш._count[]
```

Возвращает количество ключей хеша.

Пример

```
$man[
      $.name[Вася]
      $.age[22]
      $.sex[m]
]
^man._count[]
```

Вернет: **3**.

В выражениях числовое значение хеша равно количеству ключей:

```
^if($man > 2){больше}
```

foreach. Перебор ключей хеша

```
^хеш.foreach[ключ;значение]{тело}
^хеш.foreach[ключ;значение]{тело}[разделитель]
^хеш.foreach[ключ;значение]{тело}{разделитель}
```

Метод аналогичен методу menu класса **table**. Перебирает все ключи хеша и соответствующие им значения (начиная с версии ***3.4.0*** порядок перебора элементов соответствует порядку их добавления в хеш, в ранних версиях — порядок не определен).

ключ – имя переменной, которая возвращает имена ключей
значение – имя переменной, которая возвращает соответствующие значения ключей
тело – код, исполняемый для каждой пары ключ-значение хеша
разделитель – код, который вставляется перед каждым непустым не первым телом

Замечание: если разделитель задан в виде кода, то этот код выполняется после следующего не пустого тела цикла.

В любой момент можно принудительно выйти из цикла с помошью оператора **break**, или принудительно закончить текущую итерацию и перейти к следующей с помощью оператора **continue**. ***[3.2.2]***

Пример

```
$man[
      $.name[Вася]
      $.age[22]
      $.sex[m]
]
^man.foreach[key;value]{
        $key=$value
}[<br />]
```

Выведет на экран:

```
name=Вася
age=22
sex=m
```

delete. Удаление пары ключ/значение

^хеш.delete[ключ]

Метод удаляет из хеша пару ключ/значение.

Пример

```
^man.delete[name]
```

Удалит ключ **name** и связанное с ним значение из хеша **man**.

contains. Проверка существования ключа

^хеш.contains[ключ]

Метод возвращает "**истина**" если в хеше содержится запись с указанным ключём и "**ложь**" в противном случае.

Пример

```
^if(^man.contains[birthday]){
      У посетителя определена дата рождения.
}
```

Работа с множествами

sub. Вычитание хешей

^хеш.sub[хеш-вычитаемое]

Метод вычитает из хеша другой хеш-вычитаемое, удаляя ключи, общие для обоих хешей.

Пример

```
$man[
      $.name[Вася]
      $.age[22]
      $.sex[m]
]
$woman[
      $.name[Маша]
      $.age[20]
]
^man.sub[$woman]
```

В результате в хеше **$man** останется только один ключ **$man.sex** со значением **m**.

add. Сложение хешей

^хеш.add[хеш-слагаемое]

Добавляет к хешу другой хеш-слагаемое, при этом одноименные ключи хеша перезаписываются.

Пример

```
$man[
    $.name[Вася]
    $.age(22)
    $.sex[m]
]
$woman[
```

```
$.name[Маша]
    $.age(20)
    $.smile[да]
]
^man.add[$woman]
```

Новое содержание хеша **$man**:

```
$man[
    $.name[Маша]
    $.age(20)
    $.sex[m]
    $.smile[да]
]
```

Замечание: поле **_default** *добавляется. Если оно было, перезаписывается новым.* ***[3.1.4]***

union. Объединение хешей

```
^хеш_a.union[хеш_b]
```

Метод выполняет объединение двух хешей. Возвращает хеш, содержащий все ключи хеша **a** и те из **b**, которых нет в **a**. Результат необходимо присваивать новому хешу.

Пример

```
$man[
      $.name[Вася]
      $.age[22]
      $.sex[m]
]
$woman[
      $.name[Маша]
      $.age[20]
      $.weight[50]
]
$union_hash[^man.union[$woman]]
```

Получится хеш **$union_hash**:

```
$union_hash[
      $.name[Вася]
      $.age[22]
      $.sex[m]
      $.weight[50]
]
```

intersection. Пересечение хешей

```
^хеш_a.intersection[хеш_b]
```

Метод выполняет пересечение двух хешей. Возвращает хеш, содержащий ключи, принадлежащие как хешу **a**, так и **b**. Результат необходимо присваивать новому хешу.

Пример

```
$man[
      $.name[Вася]
      $.age[22]
      $.sex[m]
]
$woman[
      $.name[Маша]
      $.age[20]
      $.weight[50]
```

```
]
$int_hash[^man.intersection[$woman]]
```

Получится хеш **$int_hash**:

```
$int_hash[
      $.name[Вася]
      $.age[22]
]
```

intersects. Определение наличия пересечения хешей

```
^хеш_a.intersects[хеш_b]
```

Метод определяет наличие пересечения (одинаковых ключей) двух хешей. Возвращает булевое значение "**истина**", если пересечение есть, или "**ложь**" в противном случае.

Пример

```
^if(^man.intersects[$woman]){
      Пересечение есть
}{
      Не пересекаются
}
```

Hashfile (класс)

Класс предназначен для работы с хешами, хранящимися на диске. В отличие от класса **hash** объекты данного класса считаются всегда определенным (**def**) и не имеют числового значения.

Если класс **hash** хранит свои данные в оперативной памяти, **hashfile** хранит их на диске, причем можно отдельно задавать время хранения каждой пары ключ-значение.
Замечание: для хранения одного **hashfile** *используются два файла: .dir и .pag.*
Замечание: существует ограничение на длину строк ключа и значения, в сумме они не должны превышать 8000 байт.

Чтение и запись данных происходит очень быстро — идет работа только с необходимыми фрагментами файлов данных.
На простых задачах **hashfile** работает значительно быстрее баз данных.
Замечание: в один момент времени файл может изменяться только одним скриптом, остальные ждут окончания его работы.

Пример

Допустим, желательно некоторую информацию получить от посетителя на одной странице сайта, и иметь возможность отобразить ее — на другой странице сайта. Причем необходимо, чтобы посетитель не мог ее ни увидеть ни подделать.

Можно поместить информацию в **hashfile**, ассоциировав ее со случайной строкой — идентификатором «сеанса общения с посетителем». Идентификатор сеанса общения можно поместить в **cookie**, данные теперь хранятся на сервере, не видны посетителю и не могут быть им подделаны.

```
# создаем/открываем файл с информацией
$sessions[^hashfile::open[/sessions]]
^if(!def $cookie:sid){
    $cookie:sid[^math:uuid[]]
}
# после этого...

$information_string[произвольное значение]
# ...так запоминаем произвольную $information_string под ключом sid на 2 дня
$sid[$cookie:sid]
$sessions.$sid[$.value[$information_string] $.expires(2)]
```

```
# …а так можем считать сохраненное ранее значение
# если с момента сохранения прошло меньше 2х дней
$sid[$cookie:sid]
$information_string[$sessions.$sid]
```

Конструктор

open. Открытие или создание

^hashfile::open[имя файла]

Открывает имеющийся на диске файл или создает новый.
Для хранения данных в настоящий момент используются два файла, с суффиксами .dir и .pag.

Замечание: в один момент времени файл может изменяться только одним скриптом, остальные ждут окончания его работы. Перед началом изменений скрипт ожидает, чтобы все остальные скрипты перестали читать этот файл.

Замечание: нельзя два раза открыть один и тот же файл.

Чтение

$hashfile.ключ

Возвращает строку, ассоциированную с **ключом**, если эта ассоциация не устарела.

Запись

$hashfile.ключ[строка]

Сохраняет на диск ассоциацию между **ключом** и **строкой**.

```
$hashfile.ключ[
      $.value[строка]
      $.expires(число дней)
]
$hashfile.ключ[
      $.value[строка]
      $.expires[дата]
]
```

Такая запись позволяет указать дату устаревания ассоциации. Можно указать **число дней** или конкретную **дату**.

Необязательные модификаторы:
$.expires(число дней) – задает число дней (может быть дробным, 1.5=полтора дня), на которое сохраняется пара **ключ**/**строка**, 0 дней=навсегда;
$.expires[$date] – задает дату и время, до которой будет храниться ассоциация, здесь **$date** – переменная типа **date**.

Замечание: существует ограничение на длину строк ключа и значения, в сумме они не должны превышать 8000 байт.

Методы

hash. Получение обычного hash

```
^hashfile.hash[]
```

Выдает обычный хеш с данными **hashfile**.

foreach. Перебор ключей хеша

```
^hashfile.foreach[ключ;значение]{тело}
^hashfile.foreach[ключ;значение]{тело}[разделитель]
^hashfile.foreach[ключ;значение]{тело}{разделитель}
```

Перебирает все ключи хеша и соответствующие им значения (порядок перебора не определен). Метод аналогичен **foreach** класса **hash**.

В любой момент можно принудительно выйти из цикла с помошью оператора **break**, или принудительно закончить текущую итерацию и перейти к следующей с помощью оператора **continue**. ***[3.2.2]***

delete. Удаление пары ключ/значение

```
^hashfile.delete[ключ]
```

Метод удаляет из файла пару **ключ/значение**.

Замечание: физического удаления из файла не происходит. Пара с указанным ключём лишь помечается как удалённая и последующая запись новых данных может использовать освободившееся место.

delete. Удаление файлов данных с диска

```
^hashfile.delete[]
```

Удаляет с диска файлы, в которых хранятся данные хеш файла.

cleanup. Удаление устаревших записей

```
^hashfile.cleanup[]
```

Перебираются все пары и удаляются устаревшие.

Замечание: физического удаления из файла не происходит. Устаревшие пары лишь помечается как удалённые и последующие записи новых данных могут использовать освободившееся место.

release. Сохранение изменений и снятие блокировок

```
^hashfile.release[]
```

Все сделанные изменения сохраняются на диске и с файлов снимаются блокировки.
Таким образом хешфайл становится доступен другим процессам. Однако для продолжения работы с ним не требуется производить его повторного открытия — любое обращение к его элементам автоматически откроет файл.

Image (класс)

Класс для работы с графическими изображениями. Объекты класса **image** бывают двух типов. К первому относятся объекты, созданные на основе существующих изображений в поддерживаемых форматах. Ко второму – объекты, формируемые самим Parser.

Из JPEG файлов можно получить EXIF информацию (http://www.exif.org).

Для представления цветов используется схема RGB, в которой каждый оттенок цвета представлен тремя составляющими компонентами (R-красный, G-зеленый, B-синий). Каждая составляющая может принимать значение от 0x00 до 0xFF (0 – 255 в десятичной системе). Итоговый цвет представляет собой целое число вида 0xRRGGBB, где под каждую составляющую компоненту отведено два разряда в указанной последовательности. Формула для вычисления цвета следующая:

```
(R*0x100+G)*0x100+B
```

Так, для белого цвета, у которого все компоненты имеют максимальное значение – FF, данная формула при подстановке дает:

```
(0xFF*0x100+0xFF)*0x100+0xFF = 0xFFFFFF
```

Конструкторы

measure. Создание объекта на основе существующего графического файла

```
^image::measure[файл]
^image::measure[имя файла]
```

Создает объект класса **image**, измеряя размеры существующего графического файла или объекта класса **file** в поддерживаемом формате (сейчас поддерживаются GIF, JPEG и PNG).

Из JPEG файлов также считывается EXIF информация (http://www.exif.org), если она там записана. Большинство современных цифровых фотоаппаратов при записи JPEG файла записывают в него также информацию о снимке, параметрах экспозиции и другую информацию в формате EXIF.

Сама картинка не считывается, основное назначение метода – последующий вызов для созданного объекта метода **html**.

Параметры:
Файл – объект класса **file**
Имя файла – имя файла с путем

Примечание: поддерживается EXIF 1.0, считываются теги из IFD0 и SubIFD.

Пример создания тега IMG с указанием размеров изображения

```
$photo[^image::measure[photo.png]]
^photo.html[]
```

Будет создан объект **photo** класса **image**, на основе готового графического изображения в формате PNG, и выдан тег `IMG`, ссылающийся на данный файл, с указанием `width` и `height`.

Пример работы с EXIF информацией

```
$image[^image::measure[jpg/DSC00003.JPG]]
$exif[$image.exif]
^if($exif){
    Производитель фотоаппарата, модель: $exif.Make $exif.Model<br />
    Время съемки: ^exif.DateTimeOriginal.sql-string[]<br />
    Выдержка: $exif.ExposureTime секунды<br />
    Диафрагма: F$exif.FNumber<br />
    Использовалась вспышка: ^if(def
```

```
$exif.Flash){^if($exif.Flash){да;нет};неизвестно}<br />
}{
    нет EXIF информации<br />
}
```

create. Создание объекта с заданными размерами

```
^image::create(размер X; размер Y)
^image::create(размер X; размер Y; цвет фона)
```

Создает объект класса **image** размером X на Y. В качестве необязательного параметра можно задать произвольный цвет фона. Если этот параметр пропущен, созданное изображение будет иметь белый цвет фона.

Пример

```
$square[^image::create(100;100;0x000000)]
```

Будет создан объект **square** класса **image** размером 100x100 с черным цветом фона.

load. Создание объекта на основе графического файла в формате GIF

```
^image::load[имя_файла.gif]
```

Создает объект класса **image** на основе готового фона. Это дает возможность использовать готовые изображения в формате GIF в качестве подложки для рисования, что может использоваться для создания графиков, графических счетчиков и т.п.

Пример

```
$background[^image::load[counter_background.gif]]
```

Будет создан объект класса **image** на основе готового изображения в формате GIF. Этот объект может впоследствии использоваться для подложки в методах рисования.

Поля

$картинка.src – имя файла
$картинка.width – ширина
$картинка.height – высота
$картинка.exif – хеш с EXIF информацией

Ключами **$картинка.exif** являются названия EXIF тегов, см. спецификацию (http://www.exif.org/specifications.html). Значения бывают типов **string**, **int**, **double**, **date**. Когда тег имеет несколько значений, они считываются в хеш, ключами которого являются цифры (0...количество_значений-1).

Часто используемые EXIF теги (см. подробности в спецификации):

Тег	Тип	Описание
Make	**string**	Производитель фотоаппарата
Model	**string**	Модель фотоаппарата
DateTimeOriginal	**date**	Дата и время съемки
ExposureTime	**double**	Выдержка в секундах
FNumber	**double**	Диафрагменное число F
Flash	**int**	0= не использовалась другие значения=использовалась

Примечание: ключами нестандартных EXIF тегов являются их значения в десятичной системе счисления.

Пример

```
$photo[^image::measure[photo.jpg]]
Имя файла: $photo.src<br />
Ширина изображения в пикселах: $photo.width<br />
Высота изображения в пикселах: $photo.height<br />
$date_time_original[$photo.exif.DateTimeOriginal]
^if(def $date_time_original){
    Снимок сделан ^date_time_original.sql-string[]<br />
}
```

Будет выведено имя файла, а также ширина и высота изображения, хранящегося в этом файле. Если снимок был сделан цифровым фотоаппаратом, вероятно, будет выведена дата и время съемки.

Методы

html. Вывод изображения

```
^картинка.html[]
^картинка.html[хеш]
```

Создает следующий HTML-тег:

```
<img src="$image.src" width="$image.width" height="$image.height" />
```

В качестве параметра методу может быть передан хеш, содержащий дополнительные атрибуты изображения, например **alt** и **border**, задающие надпись, появляющуюся при наведении курсора и ширину рамки.

Замечание: атрибуты изображения можно переопределять.

Пример

```
$photo[^image::measure[myphoto.jpg]]
^photo.html[
      $.border[0]
      $.alt[Это я в молодости...]
]
```

В браузере будет выведена картинка из переменной **$photo**. При наведении курсора будет появляться надпись: **это я в молодости...**

gif. Кодирование объектов класса image в формат GIF

```
^картинка.gif[]
^картинка.gif[имя файла]        [3.1.2]
```

Используется для кодирования созданных Parser объектов класса **image** в формат GIF.
Имя файла будет передано посетителю при **$response:download**.

Внимание: в результате использования этого метода создается новый объект класса **file**, *а не* **image**!

Кроме того, необходимо учитывать тот факт, что цвета выделяются из палитры, и, когда палитра заканчивается, начинается подбор ближайших цветов. В случае создания сложных изображений, особенно с предварительно загруженным фоном, следует иметь в виду последовательность захвата цветов.

Пример

```
$square[^image::create(100;100;0x000000)]
$response:body[^square.gif[]]
```

В браузере будет выведен черный квадрат размером 100 на 100 пикселов.

Методы рисования

Данные методы используются только для объектов класса **image**, созданных с помощью конструкторов **create** и **load**. С их помощью можно рисовать линии и различные геометрические фигуры на изображениях и закрашивать области изображений различными цветами. Это дает возможность создавать динамически изменяемые картинки для графиков, графических счетчиков и т.п.

Отсчет координат для графических объектов ведется с верхнего левого угла, точка с координатами (0:0).

Тип и ширина линий

```
$картинка.line-style[тип линии]
$картинка.line-width(толщина линии)
```

Перед вызовом любых методов рисования можно задавать тип и толщину используемых линий. **Тип линии** задается строкой, где пробелы означают отсутствие точек в линии, а любые другие символы — наличие.

Пример

```
$картинка.line-style[***   ]
$картинка.line-width(2)
```

Для методов рисования будет использоваться пунктирная линия вида:
*** *** *** *** ***
толщиной в два пиксела.

line. Рисование линии на изображении

```
^картинка.line(x0;y0;x1;y1;цвет)
```

Метод рисует на изображении линию из точки с координатами (x0:y0) в точку (x1:y1) заданного цвета.

Пример

```
$square[^image::create(100;100;0x000000)]
^square.line(0;0;100;100;0xFFFFFF)
$response:body[^square.gif[]]
```

В браузере будет выведен черный квадрат размером 100 на 100 пикселов перечеркнутый по диагонали белой линией.

pixel. Работа с точками изображения

```
^картинка.pixel(x;y)
```

Выдает цвет указанной точки изображения. Если координаты попадают за пределы изображения, выдает -1.

```
^картинка.pixel(x;y;цвет)
```

Задает **цвет** указанной точки.

fill. Закрашивание одноцветной области изображения

```
^картинка.fill(x;y;цвет)
```

Метод используется для закрашивания областей изображения, окрашенных в одинаковый цвет, новым цветом. Область закрашивания определяется относительно точки с координатами X и Y.

Пример

```
$square[^image::create(100;100;0x000000)]
^square.line(0;0;100;100;0xFFFFFF)
^square.fill(10;0;0xFFFF00)
$response:body[^square.gif[]]
```

В браузере будет выведен квадрат размером 100 на 100 пикселов, перечеркнутый по диагонали белой линией. Нижняя половина квадрата черная, а верхняя закрашена желтым цветом.

rectangle. Рисование незакрашенный прямоугольников

```
^картинка.rectangle(x0;y0;x1;y1;цвет линии)
```

Метод рисует на изображении незакрашенный прямоугольник по заданным координатам с заданным цветом линии.

Пример

```
$square[^image::create(100;100;0x000000)]
^square.rectangle(5;40;95;60;0xFFFFFF)
$response:body[^square.gif[]]
```

В браузере будет выведен черный квадрат размером 100 на 100 пикселов, внутри которого находится прямоугольник 90 x 20 пикселов, нарисованный линией белого цвета по заданным координатам.

bar. Рисование закрашенных прямоугольников

```
^картинка.bar(x0;y0;x1;y1;цвет прямоугольника)
```

Метод рисует на изображении закрашенный заданным цветом прямоугольник по заданным координатам.

Пример

```
$square[^image::create(100;100;0x000000)]
^square.bar(5;40;95;60;0xFFFFFF)
$response:body[^square.gif[]]
```

В браузере будет выведен черный квадрат размером 100 на 100 пикселов, внутри которого находится белый прямоугольник 90 x 20 пикселов, нарисованный по заданным координатам.

polyline. Рисование ломаных линий по координатам узлов

```
^картинка.polyline(цвет)[таблица с координатами точек]
```

Метод рисует линию по координатам узлов, задаваемым в таблице. Он используется для создания ломаных линий.

Пример

```
$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[]
```

В браузере будет выведена буква F, нарисованная пунктирной линией на белом фоне. В рабочем каталоге будет создан файл **letter.gif**. В этом примере используются объекты класса **image** двух различных типов. В таблице задаются координаты точек ломанной линии. Затем на созданном с помощью конструктора **create** фоне рисуется линия по указанным координатам узлов. Созданный объект класса **image** кодируется в формат GIF. Полученный в результате этого объект класса **file** сохраняется на диск. Затем создается новый объект класса **image** на основе сохраненного файла. Этот объект выводится на экран браузера методом **html**.

polygon. Рисование неокрашенных многоугольников по координатам узлов

^картинка.polygon(цвет линии)[таблица с координатами узлов]

Метод рисует линией заданного цвета многоугольник по координатам узлов, задаваемым в таблице. Последний узел автоматически соединяется с первым.

Пример

```
$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[]]
```

В браузере будет выведен равнобедренный треугольник, нарисованный линией зеленого цвета на черном фоне. В таблице заданы координаты вершин треугольника.

polybar. Рисование окрашенных многоугольников по координатам узлов

^картинка.polybar(цвет многоугольника)[таблица с координатами узлов]

Метод рисует многоугольник заданного цвета по координатам узлов, задаваемым в таблице. Последний узел автоматически соединяется с первым.

Пример

```
$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[]]
```

В браузере будет выведен равнобедренный треугольник зеленого цвета на черном фоне. В таблице заданы координаты вершин треугольника.

replace. Замена цвета в области, заданной таблицей координат

^картинка.replace(старый цвет;новый цвет)[таблица с координатами точек]

Метод используется для замены одного цвета другим в области изображения, заданной с помощью таблицы координат.

Пример

```
$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[]]
```

В браузере будет выведен черный квадрат, перечеркнутый по диагонали белыми линиями, со вписаным в него розовым прямоугольником. Поскольку в методе replace задана замена на розовый цвет только для черного цвета, белые линии не перекрасились.

circle. Рисование неокрашенной окружности

^картинка.circle(center x;center y;радиус;цвет линии)

Метод рисует окружность заданного радиуса линией заданного цвета относительно центра с координатами X и Y.

Пример

```
$square[^image::create(100;100;0x000000)]
^square.circle(50;50;10;0xFFFFFF)
$response:body[^square.gif[]]
```

В браузере будет выведен черный квадрат с окружностью радиусом в десять пикселов, нарисованной линией белого цвета с центром в точке (50;50).

arc. Рисование дуги

^картинка.arc(center x;center y;width;height;start in degrees;end in degrees;color)

Метод рисует дугу с заданными параметрами. Дуга представляет собой часть эллипса (как частный случай окружности) и задается координатами центра X и Y, шириной, высотой, а также начальным и конечным углом, задаваемым в градусах.

Пример

```
$square[^image::create(100;100;0x000000)]
^square.arc(50;50;40;40;0;90;0xFFFFFF)
$response:body[^square.gif[]]
```

В браузере будет выведен черный квадрат с дугой в четверть (от 0 до 90 градусов) окружности радиусом 40 пикселов.

sector. Рисование сектора

^картинка.sector(center x;center y;width;height;start in degrees;end in degrees;color)
Метод рисует сектор с заданными параметрами линией заданного цвета. Параметры метода аналогичны методу **arc**.

Пример

```
$square[^image::create(100;100;0x000000)]
^square.sector(50;50;40;40;0;90;0xFFFFFF)
$response:body[^square.gif[]]
```

В браузере будет выведен черный квадрат с сектором в четверть (от 0 до 90 градусов) окружности радиусом 40 пикселов. Сектор нарисован линией белого цвета.

font. Загрузка файла шрифта для нанесения надписей на изображение

^картинка.font[набор_букв;имя_файла_шрифта.gif](ширина_пробела)
^картинка.font[набор_букв;имя_файла_шрифта.gif](ширина_пробела;ширина_символа)

Помимо методов для рисования, Parser также предусматривает возможность нанесения надписей на рисунки. Для реализации этой возможности требуется наличие специальных файлов с изображением шрифтов. Можно либо использовать готовые файлы шрифтов, либо самостоятельно создавать собственные с нужным набором символов.

После загрузки такого файла с помощью метода **font** набору букв, заданных в параметрах метода, ставятся в соответствие фрагменты изображения из файла. Данный файл должен быть в формате `GIF` с прозрачным фоном и содержать изображение необходимого набора символов в следующем виде:

Пример файла `digits.gif` с изображением цифр:
0
1
2
3
4
5
6
7
8
9

Высота каждого символа определяется как отношение высоты рисунка к количеству букв в наборе. Методу передаются следующие параметры:

Набор букв — перечень символов, входящих в файл шрифта
Имя и путь к файлу шрифта
Ширина символа пробела (в пикселах)
Ширина символа — необязательный параметр

По умолчанию, при загрузке файла шрифта автоматически измеряется ширина всех его символов и при выводе текста используется пропорциональный (proportional) шрифт. Если задать ширину символа, то шрифт будет моноширинным.

Все символы следует располагать непосредственно у левого края изображения.

Пример

```
$square[^image::create(100;100;0x00FF00)]
^square.font[0123456789;digits.gif](0)
```

В данном случае будет загружен файл, содержащий изображения цифр от 0 до 9, и набору цифр от 0 до 9 будет поставлено в соответствие их графическое изображение. После того, как определен шрифт для нанесения надписи, можно использовать метод **text** для нанесения надписей.

text. Нанесение надписей на изображение

```
^картинка.text(x;y)[текст надписи]
```

Метод выводит заданную надпись по указанным координатам (X;Y), используя файл шрифта, предварительно загруженный методом **font**

Пример

```
$square[^image::create(100;100;0x00FF00)]
^square.font[0123456789;digits.gif](0)

^square.text(5;5)[128500]
$response:body[^square.gif[]]
```

В браузере будет выведен зеленый квадрат с надписью «128500» левая верхняя точка которой находится в точке с координатами (5;5).

length. Получение длины надписи в пикселях

```
^картинка.length[текст надписи]
```

Метод вычисляет полную длину надписи в пикселах.

Пример

```
$square[^image::create(100;100;0x00FF00)]
^square.font[0123456789;digits.gif](0)
^square.length[128500]
```

В результате будет вычислена длина надписи «**128500**» в пикселах с учетом пробелов.

copy. Копирование фрагментов изображений

```
^картинка.copy[исходное_изображение](x1;y1;ширина1;высота1;x2;y2)
^картинка.copy[исходное_изображение](x1;y1;ширина1;высота1;x2;y2;ширина2;высота
2;приближение цвета)
```

Метод копирует фрагмент одного **изображения** в другое изображение. Это очень удобно использовать в задачах, подобных расставлению значков на карте. В качестве параметров методу передаются:

1. Исходное **изображение**
2. координаты (X1;Y1) верхнего левого угла копируемого фрагмента
3. **ширина** и **высота** копируемого фрагмента
4. координаты (X2;Y2) по которым будет вставлен копируемый фрагмент
5. в качестве необязательных параметров могут быть заданы новая ширина и высота вставляемого фрагмента (в этом случае происходит масштабирование), а также величина, характеризующая точность передачи цвета. Чем она меньше, тем точнее цветопередача, но количество передаваемых цветом уменьшается и наоборот (по умолчанию равна 150)

Пример

```
$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[]]
```

В данном примере мы создаем два объекта класса **image**. Первый создан на основе существующего GIF файла. Второй – вдвое больший по размеру, чем первый, создается самим Parser, после чего в него мы копируем фрагмент первого размером 20x30 и «растягиваем» этот фрагмент на всю ширину и высоту второго рисунка. Последняя строчка кода выводит увеличенный фрагмент на экран. Данный подход можно применять только для изображений, которые не требуется выводить с хорошим качеством.

Inet (класс)

Класс inet не имеет конструкторов для создания объектов, он обладает только статическими методами.

Статические методы

aton. Преобразование строки с IP адресом в число

```
^inet:aton[строка]
```

Переданное строка с IP адресом будет преобразовано в число. Метод аналогичен функции inet_aton MySQL сервера и perl.

Пример

```
^inet:aton[10.0.0.2]
```
– получаем число 167772162.

ntoa. Преобразование числа в строку с IP адресом

```
^inet:ntoa(число)
```

Переданное число будет преобразовано в строку с IP адресом. Метод аналогичен функции inet_ntoa MySQL сервера и perl.

Пример

```
^inet:ntoa(167772162)
```
– получаем строку '10.0.0.2'

Junction (класс)

Класс предназначен для хранения **кода** и **контекста** его выполнения.
При обращении к переменным, хранящим в себе **junction**, Parser выполняет **код** в сохраненном **контексте**.

Значение типа **junction** появляется в переменной...

...при присваивании ей кода:

```
$junction{Код, присваиваемый переменной: ^do_something[]}
```

...при передачи кода параметром:

```
@somewhere[]
^method{Код, передаваемый параметром: ^do_something_else[]}
...
@method[parameter]
#здесь в $parameter придет junction
```

...при обращении к имени метода класса:

```
$action[$user:edit]
#$action[$user:delete]
^action[параметр]
```

Здесь **$action** хранит ссылку на метод и его класс, вызов **action** теперь аналогичен вызову **^edit[параметр]**.

...при обращении к имени метода объекта:

```
$action[$person.show_info]
^action[full]
```

Здесь **$action** хранит ссылку на метод и его объект, вызов **action** теперь аналогичен вызову **^person.show_info[параметры]**.

Пример junction выражений и кода

```
@possible_reminder[age;have_passport]
^myif($age>=16 && !$have_passport){
    Тебе уже $age лет, пора сходить в милицию.
}

@myif[condition;action][age]
$age(11)
^if($condition){
    $action
}
```

Напоминание: параметр с выражением, это код, вычисляющий выражение, он выполняется — вычисляется выражение — при каждом обращении к параметру внутри вызова.

Здесь оператору **myif** передан код, печатающий, среди прочего, **$age**. Выполнение проверки и кода оператор производит в сохраненном (внутри **$condition** и **$action**) **контексте**, поэтому наличие в **myif** локальной переменной **age** и ее значение никак не влияет на то, что будет проверено и что напечатано.

Пример проверки наличия метода

```
^if($some_method is junction){
    ^some_method[параметр]
}{
    нет метода
}
```

Метод **some_method**, будет вызван только, если определен.

Mail (класс)

Класс предназначен для работы с электронной почтой. Описание настройки Parser для работы этого класса см. Конфигурационный метод.

Статические методы

send. Отправка сообщения по электронной почте

^mail:send[сообщение]

Метод отправляет **сообщение** на заданный адрес электронной почты. Можно указать несколько адресов через запятую.
Пример:

```
^mail:send[
      $.from[Вася <vasya@hotmail.ru>]
      $.to[Петя <petya@hotmail.ru>]
      $.subject[как дела]
      $.text[Как у тебя дела? У меня - изумительно!]
]
```

В результате будет отправлено сообщение для **petya@hotmail.ru** с содержимым "Как у тебя дела? У меня – изумительно!".

сообщение – хеш, в котором могут быть заданы такие ключи:

- **поле_заголовка**
- **text**
- **html**
- **file**
- **charset**
- **options** *[3.1.2]*
- **print-debug** *[3.4.0]*

Внимание: рекомендуется в поле заголовка `errors-to` *задавать адрес, на который может прийти сообщение об ошибке доставки письма. По-умолчанию* «`postmaster`».

charset – если задан этот ключ, то заголовок и текстовые блоки сообщения будут перекодированы в указанную кодировку. По умолчанию сообщение отправляется в кодировке, заданной в **$request:charset** (т.е. не перекодируется).

Пример:

```
$.charset[koi8-r]
```

options – эти опции будут переданы в командную строку программе `sendmail` (только под UNIX).

print-debug – при указании этой опции писомо не будет отправлено, вместо этого будет выведен полный сформированный текст письма, что может быть удобно при отладке сложных html-писем.

Также можно задать все поля заголовка сообщения, передав их значение в таком виде (короткая форма):

```
$.поле_заголовка[строка]
```

или с параметрами (полная форма):

```
$.поле_заголовка[
    $.value[строка]
    $.параметр[строка]
]
```

Примеры:

```
$.from[Вася <vasya@hotmail.ru>]
$.to[Петя <petya@hotmail.ru>]
$.subject[Как у тебя дела? У меня - изумительно!]
$.x-mailer[Parser 3]
```

Кроме заголовка можно передать один или оба текстовых блока: **text**, **html**. А также любое количество блоков **file** и **message** (см. ниже).
Если будет передано оба текстовых блока, будет сформирована секция `MULTIPART/ALTERNATIVE`, при прочтении полученного сообщения современные почтовые клиенты покажут HTML, а устаревшие –

простой текст.
Короткая форма:
```
$.text[строка]
```
Полная форма...
```
$.text[
    $.value[строка]
    $.поле_заголовка[значение]
]
```
...где **value** – значение тестового блока, и можно задать все поля заголовка сообщения, передав их как и в хеше **сообщение** (см. выше).
Внимание: можно не передавать заголовок `content-type`*, он будет сформирован автоматически. Этот заголовок не влияет на перекодирование, а влияет только на ту кодировку, в которой почтовый клиент будет отображать сообщение.*

Отправка HTML. Короткая форма:
```
$.html{строка}
```
Полная форма:
```
$.html[
    $.value{строка}
    $.поле_заголовка[значение]
]
```
Фигурные скобки нужны для переключения вида преобразования по умолчанию на `HTML`.

Вложение файла. Короткая форма:
```
$.file[файл]
```
Полная форма:
```
$.file[
    $.value[файл]
    $.name[имя_файла]
    $.content-id[XYZ]          [3.2.2]
    $.format[uue|base64]       [3.2.2]
    $.поле_заголовка[значение]
]
```
Файл – объект класса **file**, который будет прикреплен к сообщению. MIME-тип данных (`content-type` заголовок части) определяется по таблице **MIME-TYPES** (см. Конфигурационный метод).
Имя_файла – имя, под которым файл будет передан.
По умолчанию файл будет передан в `uuencode` *форме (uue) до версии 3.4.0 и в base64 форме начиная с версии 3.4.0.*

Вложение сообщения:
```
$.message[сообщение]
```
Формат сообщения такой же, как у параметра всего метода.

Вложений может быть несколько, для чего после имени следует добавить целое число. Пример:
```
    $.file
    $.file2
    $.message
    $.message2
```

Пример с альтернативными блоками и вложениями:
```
^mail:send[
    $.from[Вася <vasya@hotmail.ru>]
    $.to[Петя <petya@hotmail.ru>]
    $.subject[как дела]
    $.text[Как у тебя дела? У меня изумительно!]
    $.html{Как у тебя дела? У меня <b>изумительно</b>!
        <br /><img src="cid:pic2" width="100" height="50" border="0" />
    }
    $.file[^file::load[binary;perfect_life1.jpg]]
    $.file2[
        $.value[^file::load[binary;perfect_life2.jpg]]
```

```
$.name[изумительная_жизнь2.jpg]
        $.content-id[pic2]
    ]
]
```

В результате будет отослано сообщение для **petya@hotmail.ru** с содержимым «Как у тебя дела? У меня – изумительно!» в простом тексте и HTML. К сообщению будут приложены две подтверждающие фотографии, на которых...

Math (класс)

Класс math не имеет конструкторов для создания объектов, он обладает только статическими методами и предназначен для вычисления математических выражений. При работе с этим классом необходимо учитывать ограничения на разрядность для класса **double**.

Статические поля

Число Пи

$math:PI – число π
$math:E – число e

Статические методы

abs, sign. Операции со знаком

Выполняют операции со знаком числа.

^math:abs(число) – возвращает абсолютную величину числа (модуль)

^math:sign(число) – возвращает **1**, если число положительное, **-1**, если число отрицательное и **0**, если число равно **0**

Пример

^math:abs(-15.506) – получаем 15.506
^math:sign(-15.506) – получаем -1

round, floor, ceiling. Округления

^math:round(число) – округление до ближайшего целого
^math:floor(число) – округление до целого в меньшую сторону
^math:ceiling(число) – округление до целого в большую сторону

Методы возвращают округленное значение заданного числа класса **double**.

Пример

`^math:round(45.50)` — получаем 46
`^math:floor(45.60)` — получаем 45
`^math:ceiling(45.20)` — получаем 46
`^math:round(-4.5)` — получаем -4
`^math:floor(-4.6)` — получаем -5
`^math:ceiling(-4.20)` — получаем -4

trunc, frac. Операции с целой/дробной частью числа

`^math:trunc(число)` – возвращает целую часть числа
`^math:frac(число)` – возвращает дробную часть числа

Пример

`^math:trunc(85.506)` — получаем 85
`^math:frac(85.506)` — получаем 0.506

degrees, radians. Преобразования градусы-радианы

Методы выполняют преобразования из градусов в радианы и обратно.

`^math:degrees(число радиан)` – возвращает число градусов, соответствующее заданному числу радиан

`^math:radians(число градусов)` – возвращает число радиан, соответствующее заданному числу градусов

Пример

`^math:degrees($math:PI/2)` — получаем 90 (градусов)
`^math:radians(180)` — получаем π

sin, asin, cos, acos, tan, atan. Тригонометрические функции

`^math:sin(радианы)` — синус
`^math:asin(число)` — арксинус
`^math:cos(радианы)` — косинус
`^math:acos(число)` — аркосинус
`^math:tan(радианы)` — тангенс
`^math:atan(число)` — арктангенс

Методы вычисляют значения тригонометрических функций от заданного числа.

Пример

`^math:cos(^math:radians(180))` – получаем **`-1 (cos π = -1)`**.

exp, log, log10. Логарифмические функции

`^math:exp(число)` — экспонента по основанию **e**
`^math:log(число)` — натуральный логарифм
`^math:log10(число)` — десятичный логарифм

Методы вычисляют значения логарифмических функций от заданного числа

Примечание (если вы совсем забыли родную школу):
логарифм по произвольному основанию base вычисляется как log(число)/log(base).

pow. Возведение числа в степень

^math:pow(число;степень)

Возводит число в степень.

Пример
^math:pow(2;10) – получаем 1024 (2^{10} =1024)

sqrt. Квадратный корень числа

^math:sqrt(число)

Вычисляет квадратный корень числа.

Пример
^math:sqrt(16) – получаем 4

Примечание [если вы совсем забыли родную школу]:
корень n-ной степени вычисляется как возведение в степень 1/n.

random. Случайное число

^math:random(верхняя_граница)

Метод возвращает случайное число, попадающее в интервал от 0 до заданного числа, не включая заданное.
Примечание: на некоторых операционных системах выдает псевдослучайное число.

Пример
^math:random(1000)

Получим случайное число из диапазона от 0 до 999.

uuid. Универсальный уникальный идентификатор

^math:uuid[]

Выдает случайную строку вида...
```
22C0983C-E26E-4169-BD07-77ECE9405BA5
```

Примечание: на некоторых операционных системах выдает псевдослучайную строку.

Удобно использовать, когда трудно обеспечить или вообще нецелесообразно использовать сквозную нумерацию объектов.
Например, при распределенных вычислениях.

UUID также известен как GUID.

Пример
В разных филиалах компании собираются заказы, которые периодически отправляются в центральный офис. Чтобы обеспечить уникальность идентификатора заказа используем UUID.

```
# в разных филиалах происходит наполнение таблицы orders и order_details
```

```
# создаем уникальный идентификатор
$order_uuid[^math:uuid[]]

# добавляем запись о заказе
^void:sql{
      insert into orders
          (order_uuid, date_ordered, total)
      values
          ('$order_uuid', '$date_ordered', $total)
}
#цикл по заказанным продуктам вокруг добавления записи о продукте
^void:sql{
      insert into order_details
          (order_uuid, item_id, price)
      values
          ('$order_uuid', $item_id, $price)
}


# с какой-то периодичностью выбирается часть таблицы orders (и order_details)
# отправляется (^mail:send[…]) в центральный офис,
# где части таблиц попадают в общие таблицы orders и order_details
# БЕЗ проблем с повторяющимся order_id
```

Примечание: Parser создает UUID основываясь на случайных числах, а не времени. Параметры:
- *variant = DCE;*
- *version = DCE Security version, with embedded POSIX UIDs.*

В UUID не все биты случайны, и это так и должно быть:
xxxxxxxx-xxxx-4xxx-{8,9,A,B}xxx-xxxxxxxxxxxx

Подробная информация о UUID доступна здесь:
http://www.opengroup.org/onlinepubs/9629399/apdxa.htm

uid64. 64-битный уникальный идентификатор

^math:uid64[]

Выдает случайную строку вида...

```
BA39BAB6340BE370
```

Примечание: на некоторых операционных системах выдает псевдослучайную строку.

См. **^math:uuid[]**.

md5. MD5-отпечаток строки

^math:md5[строка]

Из переданной **строки** получает «отпечаток» размером 16 байт.
Выдает его представление в виде строки — байты представлены в шестнадцатиричном виде без разделителей, в нижнем регистре.

Считается, что практически невозможно
- создать две строки, имеющие одинаковый «отпечаток»;
- восстановить исходную строку по ее «отпечатку».

Пример

В качестве имени cache-файла возьмем «отпечаток» строки **$request:uri**, это обеспечит взаимно-однозначное соответствие имени строке запроса, а также избавит нас от необходимости укорачивать строку запроса и очищать ее от спецсимволов.

```
^cache[$cache_directory/^math:md5[$request:uri]]($cache_time){
    …
}
```

Подробная информация о MD5 доступна здесь: http://www.ietf.org/rfc/rfc1321.txt

crypt. Хеширование паролей

^math:crypt[password;salt]

Хеширует **password** с учетом **salt**.

Параметры:
- **password** — исходная строка;
- **salt** — строка, определяющая алгоритм хеширования и вносящая элемент случайности в результат хеширования, состоит из начала и тела. Начало определяет алгоритм хеширования, тело вносит элемент случайности. Если тело не будет указано, Parser сформирует случайное.

Неразумно хранить пароли пользователей, просто записывая их на диск или в базу данных — ведь если предположить, что злоумышленник украдет файл или таблицу с паролями, он легко сможет ими воспользоваться. Поэтому принято хранить не пароли, а их **хеши** — результат надежного однозначного необратимого преобразования строки пароля. Для проверки введенного пароля к нему применяют то же преобразование, передавая в качестве **salt** сохраненный хеш, а результат сверяют с сохраненным.

Вносить элемент случайности необходимо, поскольку иначе потенциальный злоумышленник может заранее сформировать таблицу хешей многих часто используемых паролей. Вторая причина: элемент случайности вносится на начальном этапе алгоритма хеширования, что существенно осложняет подбор пароля даже при использовании специальных аппаратных ускорителей.

Внимание: обязательно задавайте случайное тело **salt**, или позвольте Parser сделать это за вас, попросту не указывая тело **salt**, указывая только **начало salt**.

Таблица доступных алгоритмов:

Алгоритм	*Описание*	*Начало salt*	*Тело "salt"*
MD5	встроен в Parser, доступен на всех платформах	**$apr1$**	до 8 случайных букв (в любом регистре) или цифр
MD5	если поддерживается операционной системой (UNIX)	**1**	до 8 случайных букв (в любом регистре) или цифр
DES	если поддерживается операционной системой (UNIX)	(нет)	2 случайных буквы (в любом регистре) или цифры
другие	какие поддерживаются операционной системой (UNIX)	читайте документацию на вашу операционную систему, функция `crypt`	читайте документацию на вашу операционную систему, функция `crypt`

Внимание: в Parser для использования в тексте символа '$' его необходимо предварить символом '^'.

Примечание: Веб-сервер Apache допускает в файлах с паролями (`.htpasswd`) использовать хеши, сформированные по любому из алгоритмов, представленных в таблице, включая алгоритм, встроенный в Parser.

Пример создания .htpasswd файла

```
@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]
```

Пример проверки пароля

```
$right[123]
$from_user[123]
$crypted[^math:crypt[$right;^$apr1^$]]
#обратите внимание на то, что $crypted при каждом обращении разный
$crypted<br />
^if(^math:crypt[$from_user;$crypted] eq $crypted){
    Казнить нельзя, помиловать
}{
    Казнить, нельзя помиловать
}
```

Подробная информация о MD5 доступна здесь: http://www.ietf.org/rfc/rfc1321.txt

crc32. Подсчет контрольной суммы строки

```
^math:crc32[строка]
```

Для переданной строки будет подсчитана контрольная сумма (CRC32).
Метод выдает её в виде целого числа.

sha1. Хеш строки по алгоритму SHA1

```
^math:sha1[строка]
```

Для переданной строки будет вычислен хеш по алгоритму SHA1.

Memory (класс)

Класс предназначен для работы с памятью Parser.
Его использование поможет вам экономить память в ваших скриптах.

Для любознательных: в Parser используется известный и хорошо зарекомендовавший себя консервативный сборщик мусора Boehm-Demers-Weiser, см.
http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

Статический метод

compact. Сборка мусора

```
^memory:compact[]
```

Собирает так называемый «мусор» в памяти, освобождая ее для повторного использования вашим кодом.
Мусором считается память, более не используемая вашим кодом, т.е. та, на которую в вашем коде нет ссылок.

Например,

```
$table[^table::sql{запрос}]
$table[]
# освободит память, занимаемую результатом выполнения SQL-запроса
^memory:compact[]
```

Parser не собирает мусор автоматически, полагаясь в данном вопросе на кодера: поставьте вызов **compact** в той точке (точках), где ожидаете наибольшей выгоды, например, перед XSL преобразованием.

$status:memory поможет вам в отладке и поиске мест, наиболее выгодных для сборки мусора.

*Важно: необходимо как можно более интенсивно использовать локальные переменные, и обнулить глобальные, которые вам не будут нужны для дальнейшей работы кода. Это поможет **compact** освободить больше.*

Важно: не гарантируется, что будет освобождена абсолютно вся неиспользуемая память.

Reflection (класс)

Класс предназначен для получения информации о классах и их методах.

Статические методы

create. Создание объекта

```
^reflection:create[имя класса;имя конструктора]
^reflection:create[имя класса;имя конструктора;па;рам;етры]
```

Создаёт объект указанного класса, вызывая конструктор с указанным именем.
Использовать этот метод удобно, если необходимо создать объект класса, имя которого находится в переменной.

Замечание: передать конструктору можно не более 100 параметров.

classes. Список классов

```
^reflection:classes[]
```

Возвращает хеш со списком всех классов, доступных на момент вызова.
Ключами хеша являются имена классов, значениями могут быть строки **methoded** (для классов, содержащих методы) или **void**.

class. Класс объекта

```
^reflection:class[объект]
```

Возвращает класс объекта (аналогично **$объект.CLASS**).

class_name. Имя класс объекта

```
^reflection:class_name[объект]
```

Возвращает имя класса объекта (аналогично **$объект.CLASS_NAME**).

base. Родительский класс объекта

```
^reflection:base[класс]
^reflection:base[объект]
```

Возвращает базовый класс объекта или класса (если он есть) или **void**.

base_name. Имя родительского класса объекта

```
^reflection:base_name[класс]
^reflection:base_name[объект]
```

Возвращает имя базового класса объекта или класса (если он есть) или **void**.

methods. Список методов класса

```
^reflection:methods[имя класса]
```

Возвращает хеш со всеми методами указанного класса.
Ключами хеша являются имена методов, значениями — строки **native** (для системных классов) или **parser** (для классов, созданых пользователем).

method_info. Информация о методе

```
^reflection:method_info[имя класса;имя метода]
```

Возвращает хеш с параметрами указанного метода указанного класса.

Для методов системных классов возвращается хеш следующего вида:

```
$хеш[
    $.inherited[имя класса, в котором метод был определён]
    $.min_params(минимальное необходимое количество параметров метода)
    $.max_params(максимальное допустимое количество параметров метода)
    $.call_type[допустимый тип вызова метода: static, dynamic или any]
]
```

Для методов пользовательских классов возвращается хеш следующего вида:

```
$хеш[
    $.inherited[имя класса, в котором метод был определён]
    $.0[имя первого параметра метода]
    $.1[имя второго параметра метода]
    ...
]
```

fields. Список полей объекта

```
^reflection:fields[класс]
^reflection:fields[объект]
```

Для класса метод возвращает хеш со списоком статическимх полей, для объекта — со списком динамических полей.

dynamical. Тип вызова метода

```
^reflection:dynamical[]
^reflection:dynamical[класс]
^reflection:dynamical[объект]
```

При вызове без параметров возвращает логическое значение **true**, если метод, из которого был вызван метод **^reflection:dynamical[]**, был вызван динамически и **false**, если он был вызван статически.

Если методу в качестве параметра был передан объект или класс, то метод возвращает логическое значение **true**, если был передан динамический объект и **false**, если был передан класс.

Метод удобно использовать внутри методов классов чтобы узнать, как именно был вызван метод —

динамически или статически.

Regex (класс)

Класс предназначен для работы с *регулярными выражениями*, совместимыми с PCRE (Perl Compatible Regular Expressions).
Частичный перевод описания PCRE приведен в Приложении 4.

Объект класса regex всегда считается определенным (**def**). Числовым значением объекта класса regex является размер скомпилированного шаблона в байтах.

Конструктор

create. Создание нового объекта

```
^regex::create[шаблон]
^regex::create[шаблон][опции поиска]
```

Шаблон — это строка с *регулярным выражением*, совместимым с PCRE (Perl compatible regular expressions).
Частичный перевод описания PCRE приведен в Приложении 4.

Предусмотрены следующие опции поиска:
i – не учитывать регистр;
x – игнорировать символы white space и разрешить **#комментарий до конца строки**;
s – символ **$** считать концом всего текста (опция по умолчанию);
m – символ **$** считать концом строки, но не всего текста;
U – инвертировать «жадность» квантификаторов (они становятся не «жадными», чтобы сделать их «жадными» необходимо поставить после них символ **?**); ***[3.3.0]***
g – найти все вхождения строки (а не только первое);
n – вернуть число с количеством совпадений вместо таблицы; ***[3.2.2]***
' – вычислять значения столбцов **prematch**, **match**, **postmatch**.

Поскольку символы **^** и **$** используются в Parser, в шаблоне вместо символа **^** используется строка **^^**, а вместо символа **$** – строка **^$** (см. Литералы).

Поля

pattern. Текст шаблона
```
$шаблон.pattern
```

Поле содержит строку с исходным текстом регулярного выражения.

options. Опции
```
$шаблон.options
```

Поле содержит строку с исходным текстом опций.

Request (класс)

Класс содержит статические поля, которые позволяют получать информацию, передаваемую браузером веб-серверу (по HTTP протоколу).

Для работы с полями форм (**<FORM>**) и строкой после **?** (`/?name=value?orThisText`) используйте

класс **form**.

Часть информации о запросе доступна через переменные окружения, см. «Получение значения поля запроса».

Статические поля

uri. Получение URI страницы

$request:uri

Возвращает URI документа.

Пример

Предположим, пользователь запросил такую страницу:
`http://www.mysite.ru/news/articles.html?year=2000&month=05&day=27`

Тогда:
$request:uri

вернет:
/news/articles.html?year=2000&month=05&day=27

query. Получение строки запроса

$request:query

Возвращает строку после **?** в URI (значение переменной окружения QUERY_STRING).
Для работы с полями форм (**<FORM>**) и строкой после второго **?** (`/?a=b?thisText`) используйте класс **form**.

Пример

Предположим, пользователь запросил такую страницу:
`http://www.mysite.ru/news/articles.html?year=2000&month=05&day=27`

Тогда:
$request:query

вернет:
year=2000&month=05&day=27

charset. Задание кодировки документов на сервере

$request:charset[кодировка]

Задает **кодировку** документов, обрабатываемых на сервере.
При обработке запроса считается, что в этой кодировке находятся все файлы на сервере.

По умолчанию используется кодировка **UTF-8**.

Список допустимых кодировок определяется Конфигурационным методом.
Рекомендуется определять кодировку документов в Конфигурационном файле.

См. также «Задание кодировки ответа».

post-charset. Получение кодировки пришедшего POST запроса

$request:post-charset

Если в HTTP заголовке content-type пришедшего POST запроса содержится информация о кодировке, то её название доступно в этом поле.
При разборе полей формы подобного POST запроса данные перекодируются из указанной в этом заголовке кодировки, а не из того, что задано в $response:charset.

Внимание: если кодировка, указанная в упомянутом HTTP заголовке не была подключена (например в конфигурационном методе), то будет выдано сообщение об ошибке.

body. Получение текста запроса

$request:body

Получение текста HTTP POST-запроса.

Вариант использования: можно написать свой **XML-RPC** сервер (см. http://www.xmlrpc.com).

document-root. Корень веб-пространства

$request:document-root[/дисковый/путь/к/корню/вашего/веб-пространства]

По-умолчанию, **$request:document-root** равен значению, которое задается в веб-сервере. Однако иногда его удобно заменить.

См. также «Пути к файлам и каталогам».

argv. Аргументы командной строки

$request:argv

Хеш, содержащий аргументы командной строки с ключами 0, 1, 2 и т.д., которым может быть удобно использовать при использовании парсера в качестве интерпретатора скриптов (например при запуске из cron).

$request:argv.0 — содержит имя обрабатываемого файла.

Response (класс)

Класс позволяет дополнять стандартные HTTP-ответы сервера. Класс не имеет конструкторов для создания объектов.

Статические поля

Заголовки HTTP-ответа

$response:поле[значение]
$response:поле

Поле соответствует заголовку HTTP-ответа, выдаваемого Parser. Его можно как задавать, так и считывать. Значением может быть дата, строка или хеш с обязательным ключом **value**. Дата может использоваться и в качестве значения поля и в качестве значения атрибута поля, при этом она будет стандартно отформатирована.

Примечание: при задании значения имя поля преобразуется к нижнему регистру, а при считывании ищется в нижнем регистре.
*Примечание: при выдаче браузеру имя HTTP-заголовка капитализируется (например **content-type** будет преобразован в **Content-Type**).* ***[3.4.0]***

Пример перенаправления браузера на стартовую страницу

```
#работает если администратор веб-сервера правильно настроил передачу параметра SERVER_NAME
#обычно настроено все правильно
$response:location[http://$env:SERVER_NAME/]
```

Другой пример перенаправления браузера на стартовую страницу

```
#работает вне зависимости от правильности SERVER_NAME
$response:refresh[
      $.value(0)
      $.url[/]
]
```

Пример задания заголовка expires в значение «завтра»

```
$response:expires[^date::now(+1)]
```

headers. Заданные заголовки HTTP-ответа

Такая конструкция возвращает хеш со всеми заголовками HTTP-ответа, которые были заданы в коде на данный момент.

Пример

```
$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 />]
```

Пример выведет на экран все заданные ранее заголовки HTTP-ответа.

body. Задание нового тела ответа

$response:body[DATA]

Замещает все тело ответа значением **DATA**.

DATA – строка, файл или хеш параметров.

Ключи хеша параметров: ***[3.1.4]***
file – имя файла на диске (в этом случае Parser поддерживает докачку файлов ***[3.1.4]***);
name – имя файла, которое передать посетителю;
mdate – дата и время изменения файла, которую передать посетителю.

Если передан файл с известным `content-type` (см. поля объекта класса file), этот заголовок передается посетителю.

См. также **$response:download**.

Пример замены всего тела на результат работы скрипта

```
$response:body[^file::cgi[script.cgi]]
```

Заменит весь ответ результатом работы программы `script.cgi`.

Пример выдачи создаваемой картинки

```
$square[^image::create(100;100;0x000000)]
^square.circle(50;50;10;0xFFFFFF)
$response:body[^square.gif[]]
```

В браузере будет выведен черный квадрат с белой окружностью. Кроме того, автоматически будет установлен нужный тип файла (content-type) по таблице **MIME-TYPES**.

download. Задание нового тела ответа

$response:download[DATA]

Идентичен **$response:body**, но выставляет флаг, который браузер воспринимает как «Предложить посетителю сохранить файл на диске».

Браузеры умеют отображать файлы некоторых типов прямо внутри своего окна (например: `.doc`, `.pdf` файлы).
Однако бывает необходимо дать возможность посетителю скачать файл по простому нажатию на ссылку.

Пример: выдача PDF файла

Посетитель заходит на страницу с таким HTML...

```
<a href="/download_documentation.html">Скачать документацию</a>
```

`download_documentation.html`:

```
$response:download[^file::load[binary;documentation.pdf]]
```

...и нажимает на ссылку, браузер предлагает ему Скачать/Запустить.

charset. Задание кодировки ответа

$response:charset[кодировка]

Задает **кодировку** ответа.
После обработки запроса результат перекодируется в эту кодировку.

По умолчанию используется кодировка **UTF-8**.

Список допустимых кодировок определяется Конфигурационным методом.
Рекомендуется определять кодировку документов в Конфигурационном файле.

См. также «Задание кодировки документов на сервере».

Статические методы

clear. Отмена задания новых заголовков HTTP-ответа

^response:clear[]

Метод отменяет все действия по переопределению полей ответа.

Status (класс)

Класс предназначен для анализа текущего состояния скрипта на Parser.
Его использование поможет вам найти узкие места в ваших скриптах.

Если вы используете parser как модуль Apache, вы будете получать сообщение **class not found** при попытке использовать класс status, пока не добавите в httpd.conf строки:

<Location />
Разрешает использовать встроенный класс status
ParserStatusAllowed
</Location>

и не перезапустите Apache.

Поля

rusage. Информация о затраченных ресурсах

Это поле — хеш с информацией о ресурсах сервера, затраченных на данный момент системой на обработку вашего Parser-скрипта.
Не все операционные системы умеют возвращать эти значения (WinNT/Win2000/WinXP умеет все, Win98 умеет только **tv_sec** и **tv_usec** ***[3.0.8]***).

Ключ	*Единица*	*Описание значения*	*Как уменьшить?*
utime	секунда	Чистое время, затраченное текущим процессом (не включает время, когда работали другие задачи)	Упростить манипуляции с данными внутри Parser (улучшить алгоритм, переложить часть действий на SQL-сервер)
stime	секунда	Время, сколько система читала ваши файлы, каталоги, библиотеки	Уменьшить количество и размер необходимых для работы файлов, не подключать ненужные для обработки данного документа модули
maxrss	блок	Память, занимаемая процессом	Уменьшить количество загружаемых ненужных данных. Найти и исправить все «`select * ...`», задав список действительно необходимых полей. Не загружать из SQL-сервера ненужные записи, отфильтровать как можно больше средствами самого SQL-сервера.
		Точное системное время. Позволяет оценить траты времени на ожидание ответа от SQL-, HTTP-, SMTP-серверов.	Упростить SQL запросы, для MySQL воспользуйтесь `EXPLAIN`; для Oracle: `EXPLAIN PLAN` (см. документацию по серверу); для других SQL-серверов: см. их документацию.
		Сколько прошло с Epoch...	
tv_sec	секунда	...целых секунд;	
tv_usec	микросекунда (10E-6)	...еще прошло микросекунд (миллионных долей секунды)	

Рекомендуемый способ анализа

Временно добавьте в конец вашего скрипта вызов...
^rusage[total]

...вот этого метода...

```
@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[]
```

...и проанализируйте журнал.

Для более точного анализа, добавьте вызовы...
^rusage[before XXX]
^rusage[after XXX]

...вокруг интересующего вас блока.

Примечание: для записи журнала не рекомендуется использовать веб-пространство.

WinNT/2K/XP

Под этими OS доступен ряд дополнительных значений:

Ключ	*Единица*	*Описание значения*	*Как уменьшить?*
ReadOperationCount **ReadTransferCount**	штук байт	Количество операций чтения с диска и суммарное количество считанных байт	Уменьшить количество и размер необходимых для работы файлов, не подключать ненужные для обработки данного документа модули. Больше использовать SQL-сервер, меньше файлы.
WriteOperationCount **WriteTransferCount**	штука байт	Количество операций записи на диск и суммарное количество записанных байт	
OtherOperationCount **OtherTransferCount**	штука байт	Количество других операций с диском (не чтения/записи) и суммарное количество переданных байт	
PeakPagefileUsage	байт	Максимальное количество памяти в файле подкачки (swap-файле)	см. комментарий к **maxrss** выше.
QuotaPeakNonPagedPoolUsage **QuotaPeakPagedPoolUsage**			

memory. Информация о памяти под контролем сборщика мусора

Это поле — хеш с информацией о памяти, находящейся под контролем сборщика мусора.

Поле	*Значение (в килобайтах)*	*Детали*
used	Занято	В это число не включен размер служебных данных самого сборщика мусора.
free	Свободно	Свободная память скорее всего фрагментирована.
ever_allocated_since_compact	Было выделено с момента последней сборки мусора. См. **memory:compact**.	Между сборками мусора это число только растет. Факты освобождения памяти без сборки мусора на него не влияют, только сборки мусора.
ever_allocated_since_start	Было выделено за все время обработки запроса	Это число только растет. Ни факты сборки мусора, ни освобождения памяти между сборками мусора на него не влияют.

Рекомендуемый способ анализа

Временно добавьте вызовы...

```
^musage[before XXX]
^musage[after XXX]
```

...вокруг интересующего вас блока вот этого метода...

```
@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[]
```

...и проанализируйте журнал.

*Важно: в ходе работы Parser захватывает у операционной системы дополнительные блоки памяти по мере необходимости. Поэтому есть моменты, когда и **used** и **free**, увеличиваются. Это нормально.*

Примечание: для записи журнала не рекомендуется использовать веб-пространство.

pid. Идентификатор процесса

Идентификатор процесса (process) операционной системы, в котором работает Parser.

tid. Идентификатор потока

Идентификатор потока (thread) операционной системы, в котором работает Parser.

String (класс)

Класс для работы со строками. В выражении строка считается определенной (**def**), если она не пуста. Если в строке содержится число, то при попытке использовать его в математических выражениях содержимое строки будет автоматически преобразовано к **double**. Если строка пуста, ее числовое "значение" в математических выражениях считается нулем.

Создание объекта класса **string**:

```
$str[Строка, которая содержится в объекте]
```

Статические методы

sql. Получение строки из базы данных

```
^string:sql{SQL-запрос}
^string:sql{SQL-запрос}[$.limit(1) $.offset(o) $.default{код} $.bind[variables hash]]
```

Замечание: именно метод, не конструктор!

Возвращает строку, полученную из базы данных через SQL-запрос. Результатом выборки должен быть только один столбец из одной строки. Для работы оператора необходимо установленное соединение с сервером базы данных (см. оператор **connect**).

Необязательные параметры:
$.limit(1) – в ответе заведомо будет содержаться только одна строка;
$.offset(o) – отбросить первые **o** записей выборки;
$.bind[hash] – связанные переменные, см. «Работа с IN/OUT переменными». ***[3.1.4]***
если ответ SQL-сервера был пуст (0 записей), то будет...
$.default{код} ...выполнен указанный **код**, и строка, которую он возвратит, будет результатом метода;
$.default(выражение) ...вычислено указанное **выражение**, и оно будет результатом метода;
$.default[строка] ...будет возвращена указанная **строка**;

Пример

```
^string:sql{select name from company where company_id=$company_id}
```

Используя этот метод, полезно конструировать SQL-запрос так, чтобы в ответе заведомо содержалась одна строка из одного столбца.

base64. Декодирование из Base64

```
^string:base64[закодированное]
```

Замечание: именно метод, не конструктор!

Декодирует строку из Base64 представления. Для кодирования строки используйте
^строка.base64[]

Подробная информация о Base64 доступна здесь: http://www.ietf.org/rfc/rfc2045.txt и здесь http://en.wikipedia.org/wiki/Base64

Пример

```
$encoded[pyAxOTczLiDV7uT/8iDx6/P16Cwg9/LuIKvH5evl7fvlIPDz6uDi4Lsg7eDv6PHg6yDx4OyF]
$original[^string:base64[$encoded]]
$original
```

Выведет...
§ 1973. Ходят слухи, что «Зеленые рукава» написал сам...

js-unescape. Декодирование, аналогичное функции unescape в JavaScript

```
^string:js-unescape[закодированное]
```

Замечание: именно метод, не конструктор!

Метод выполняет преобразование строки аналогичное методу **unescape** описанному в ECMA-262.

Для кодирования воспользуйтесь
^string.js-escape[]

С помощью данного метода вы можете декодировать строки, закодированные в браузере с помощью функции **escape**.

Подробная информация о ECMA-262 доступна здесь:
http://www.ecma-international.org/publications/standards/Ecma-262.htm (B.2.2)

Пример

```
$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%u0432%u0433%u0434]
$original[^string:js-unescape[$escaped]]
$original
```

Выведет...
abcd `+-=~!@#%&*()_ []{}<>:'",./?абвгд

Методы

int, double, bool. Преобразование строки к числу или bool

```
^строка.int[]
^строка.int(значение по умолчанию)
^строка.double[]
^строка.double(значение по умолчанию)
^строка.bool[]
^строка.bool(значение по умолчанию)
```

Преобразуют значение переменной **$строка** к целому, вещественному числу или bool значению соответственно, и возвращает это значение.

Можно задать значение по умолчанию, которое будет получено, если преобразование невозможно или строка пуста или состоит только из "white spaces" (символы пробела, табуляция, перевода строки).

Значение по умолчанию можно использовать при обработке данных, получаемых интерактивно от пользователей. Это позволит избежать появления текстовых значений в математических выражениях при вводе некорректных данных, например, строки вместо ожидаемого числа.

Метод **.bool** умеет преобразовать в **bool** строки, содержащую число (значение 0 будет преобразовано в **false**, не 0 — в **true**), а также строки, содержащие значения "**true**" и "**false**" (без учёта регистра).

Внимание: использование пустой строки в математических выражениях не является ошибкой, ее значение считается нулем.
Внимание: преобразование строки, не являющейся целым числом к целому числу является ошибкой (пример: строка «1.5» не является целым числом).

Примеры использования

```
$str[123]
^str.int[]
```

Выведет число **123**, поскольку объект **str** можно преобразовать к классу **int**.

```
$str[много]
^str.double(-1)
```

Выведет число **-1**, поскольку преобразование невозможно.

```
$str[1]
^if(^str.bool[]){истина}

$str[True]
^if(^str.bool[]){истина}
```

Выведут строки "**истина**".

format. Вывод числа в заданном формате

```
^строка.format[форматная_строка]
```

Метод выводит значение переменной в заданном формате (см. Форматные строки).
Выполняется автоматическое преобразование строки к числу.

Пример

```
$var[15.67678678]
^var.format[%.2f]
```

Возвратит: **15.68**

split. Разбиение строки

```
^строка.split[разделитель]
^строка.split[разделитель;опции разбиения]
^строка.split[разделитель;опции разбиения;имя столбца]        [3.2.2]
```

Разбивает строку на подстроки относительно подстроки-**разделителя** и формирует объект класса **table**, содержащий

- либо таблицу со столбцом, в который помещаются части исходной строки,
- либо безымянную таблицу с частями исходной строки в колонках единственной записи.

Предусмотрены следующие **опции разбиения**:
l – разбить слева направо (по-умолчанию);
r – разбить справа налево;
h – сформировать безымянную таблицу где части исходной строки помещаются горизонтально;
v – сформировать таблицу со столбцом, где части исходной строки помещаются вертикально (по-умолчанию).

Имя столбца при создании вертикальной таблицы – «**piece**» или переданное **имя столбца**.

Пример вертикального разбиения

```
$str[O, сколько нам открытий чудных!…]
$parts[^str.split[нам]]
^parts.save[parts.txt]
```

Создает на диске файл **parts.txt**, содержащий следующее:

```
piece
O, сколько
открытий чудных!…
```

Пример горизонтального разбиения

```
$str[/a/b/c/d]
$parts[^str.split[/;lh]]
$parts.0, $parts.1, $parts.2
```

Выведет:

```
, a, b
```

upper, lower. Преобразование регистра строки

```
^строка.upper[]
^строка.lower[]
```

Переводят строку в верхний или нижний регистр соответственно. Для их работы необходимо, чтобы был задан **$request:charset**.

Пример

```
$str[Москва]
^str.upper[]
```

Вернет: **МОСКВА**.

length. Длина строки

```
^строка.length[]
```

Возвращает длину строки.

Пример

```
$str[O, сколько нам открытий чудных!…]
^str.length[]
```

Вернет: **32**

mid. Подстрока с заданной позиции

```
^строка.mid(P;N)
^строка.mid(P)
```

Возвращает подстроку, которая начинается с позиции **P** и имеет длину **N** (если **N** не задано, то возвращается подстрока с позиции **P** до конца строки). Отсчет **P** начинается с нулевой позиции. Если **P+N** больше длины строки, то будут возвращены все символы строки, начиная с позиции **P**.

Пример

```
$str[О, сколько нам открытий чудных!…]
^str.mid(3;20)
```

Выведет на экран: **сколько нам открытий**

left, right. Подстрока слева и справа

```
^строка.left(N)
^строка.right(N)
```

Методы возвращают **N** первых или последних символов строки соответственно. Если длина строки меньше **N**, то возвращается вся строка.

Пример

```
$str[О, сколько нам открытий чудных!…]
^str.left(10) ^str.right(10)
```

На экран будет выведено: **О, сколько чудных!**…

pos. Получение позиции подстроки

```
^строка.pos[подстрока]
^строка.pos[подстрока](позиция начала поиска)
```
[3.3.0]

Возвращает число **int** – позицию первого символа подстроки в строке (начиная с нуля), или –1, если подстрока не найдена.
Если задан второй параметр, то поиск подстроки будет начинаться с указанной в нем позиции.

Примеры

```
$str[полигон]
^str.pos[гон]
```

Вернет: **4**

```
$str[полигон]
^str.pos[о](2)
```

Вернет: **5**

replace. Замена подстрок в строке

```
^строка.replace[$таблица_подстановок]
```

Эффективно заменяет подстроки в строке в соответствии с **таблицей подстановок**, работает существенно быстрее **match**.

Таблица подстановок – объект класса **table**, содержащая два столбца:
первый – подстрока, которую нужно заменить,
второй – подстрока, которая появится на месте подстроки из первого столбца после замены.

Имена столбцов несущественны, можно называть их from/to, или вообще никак не называть, воспользовавшись **nameless** таблицей.

Пример

```
$s[A magic moment I'll remember!]
Исходная строка: $s<br />
$rep[^table::create{from      to
A An
magic ugly}]
Исковерканная строка: ^s.replace[$rep]
```

Выведет на экран:

```
Исходная строка: A magic moment I'll remember!
Исковерканная строка: An ugly moment I'll remember!
```

save. Сохранение строки в файл

```
^строка.save[имя_файла_с_путем]
^строка.save[append;имя_файла_с_путем]
^строка.save[имя_файла_с_путем;опции]        [3.4.0]
```

Сохраняет или добавляет строку в файл по указанному пути.
При этом с фрагментами строки производятся необходимые преобразования, см. «Преобразование данных».

Для опций доступны следующие значения:
$.charset[кодировка**]**
$.append(true)

Пример

Задача: из SQL-сервера А достать данные, положить в SQL-сервер Б.

Если оба SQL-сервера доступны с какой-то машины, можно так:

```
^connect[A]{
    $data[
#       код, наполняющий data данными из SQL-сервера A
    ]
    ^connect[Б]{
         ^void:sql{insert into table x (x) values ('$data')}
    }
}
```

При этом **$data** в SQL-запросе insert будет правильно обработан по правилам SQL-диалекта сервера Б.

Однако если оба SQL-сервера **не**доступны одновременно с какой-то машины, можно так:

```
^connect[A]{
    $data[
#       код, наполняющий data данными из SQL-сервера A
    ]
    $string[^untaint[sql]{insert into table x (x) values ('$data')}]
    ^connect[локальный фиктивный Б]{
#       это соединение нужно только для того,
#       чтобы задать правила обработки для SQL-диалекта сервера Б
        ^string.save[B-inserts.sql]
    }
}
```

При этом в файл `B-inserts.sql` запишется правильно обработанный SQL-запрос.

match. Поиск подстроки по шаблону

```
^строка.match[шаблон]
^строка.match[шаблон][опции поиска]
```

Осуществляет поиск в строке по шаблону.
Шаблон — это строка с *регулярным выражением*, совместимым с PCRE (Perl compatible regular expressions) или объект класса **regex** ***[3.4.0]***.
Частичный перевод описания PCRE приведен в Приложении 4.

Предусмотрены следующие опции поиска:
i – не учитывать регистр;
x – игнорировать символы white space и разрешить **#комментарий до конца строки**;
s – символ **$** считать концом всего текста (опция по умолчанию);
m – символ **$** считать концом строки, но не всего текста;

U – инвертировать «жадность» квантификаторов (они становятся не «жадными», чтобы сделать их «жадными» необходимо поставить после них символ **?**); ***[3.3.0]***
g – найти все вхождения строки (а не только первое);
n – вернуть число с количеством совпадений вместо таблицы; ***[3.2.2]***
' – вычислять значения столбцов **prematch**, **match**, **postmatch**.

Поскольку символы **^** и **$** используются в Parser, в шаблоне вместо символа **^** используется строка **^^**, а вместо символа **$** – строка **^$** (см. Литералы).

Если указана опция поиска **g**, будет создана **таблица** (объект класса **table**) найденного по шаблону (по одной строке на каждое вхождение). Если опция **g** не была указана, то таблица будет содержать лишь одну строку с первым вхождением. Если не было найдено ни одного совпадения, то результатом операции будет **пустая таблица**. Если указана опция **n** то вместо таблицы с результатами будет возвращаться **число** – количество найденных совпадений.

Таблица совпадений имеет следующие столбцы: **1**, **2**,..., **n**, **prematch**, **match**, **postmatch**, где:

prematch столбец с подстрокой от начала строки до совпадения
match столбец с подстрокой, совпавшей с шаблоном
postmatch столбец с подстрокой, следующей за совпавшей подстрокой до конца строки
1, **2**,..., **n** столбцы с подстроками, соответствующими фрагментам шаблона, заключенным в круглые скобки, **n** – номер открывающей круглой скобки.

*Замечание 1: значения столбцов **prematch**, **match**, **postmatch** вычисляются только если указана опция **'**.*
*Замечание 2: начения столбцов **1**, **2**,..., **n** вычисляются лишь в случае, если в шаблоне указаны круглые скобки.*
Замечание 3: если в шаблоне вам нужно использовать круглые скобки, но не требуется запоминания заключённого в них в результирующей таблице, то вместо них лучше использовать конструкцию (?:)

Примеры использования

```
$str[www.parser.ru?user=admin]
^if(^str.match[
    \? #есть разделитель
    .+ #и есть хоть что-то за ним
][x]){Есть совпадение}{Совпадений нет}
```

Выведет на экран: **Есть совпадение.**

*Внимание: настоятельно советуем задавать комментарии к частям сложного регулярного выражения. Бывает, что даже вам самим через какое-то время бывает трудно в них разобраться. Для этого включите опцию **x**, разрешающую расширенный синтаксис выражений, допускающий комментарии.*

```
$str[www.parser.ru?user=admin]
$mtc[^str.match[(\?.+)]['']]
^mtc.save[match.txt]
```

Создаст файл **match.txt**, содержащий такую таблицу:

prematch	match	postmatch	1
www.parser.ru	?user=admin		?user=admin

match. Замена подстроки, соответствующей шаблону

```
^строка.match[шаблон][опции поиска]{замена}
^строка.match[шаблон][опции поиска][замена] [3.4.0]
```

Осуществляет поиск в строке по шаблону и производит замену совпавшей подстроки на заданную. Механизм поиска устроен так же, как и у предыдущего метода. Внутри кода замены доступна автоматически создаваемая таблица совпадений **match**, которая была рассмотрена выше.

Пример

```
$str[2002.01.01]
^str.match[(\d+)\.(\d+)\.(\d+)][g]{Год $match.1, месяц $match.2, число
$match.3}
```

Выведет: **Год 2002, месяц 01, число 01.**

trim. Отсечение букв с концов строки

```
^строка.trim[]
^строка.trim[откуда]
^строка.trim[откуда;набор]
```

Метод отсекает любые буквы из указанного **набора** с концов строки. По умолчанию отсекаются white spaces с начала и конца строки.
Можно указать, **откуда** именно отсекать буквы, задав одно из значений:

- **both** – отсекать и с начала и с конца;
- **left** или **start** – отсекать с начала;
- **right** или **end** – отсекать с конца.

Пример отсечения white space

```
$name[ Вася ]
"$name"
"^name.trim[]"
```

Выведет...

```
" Вася "
"Вася"
```

Пример отсечения указанных букв

```
$path[/section/subsection/]
^path.trim[right;/]
```

Выведет...

```
/section/subsection
```

base64. Кодирование в Base64

```
^строка.base64[]
```

Метод позволяет преобразовать строку в Base64 форму. Чтобы преобразовать строку из Base64 к исходному виду, воспользуйтесь

```
^string:base64[закодированное]
```

Подробная информация о Base64 доступна здесь: http://www.ietf.org/rfc/rfc2045.txt и здесь http://en.wikipedia.org/wiki/Base64

Пример

```
$original[§ 1973. Ходят слухи, что «Зеленые рукава» написал сам...]
<pre>^original.base64[]</pre>
```

Выведет...
pyAxOTczLiDV7uT/8iDx6/P16Cwg9/LuIKvH5evl7fvlIPDz6uDi4Lsg7eDv6PHg6yDx4OyF

js-escape. Кодирование, аналогичное функции escape в JavaScript

^строка.js-escape[]

Метод выполняет преобразование строки аналогичное методу **escape** описанному в ECMA-262. Чтобы выполнить обратное преобразование, воспользуйтесь
^string:js-unescape[закодированное]

Строки, закодированные данным методом, могут быть раскодированы в браузере с помощью функции **unescape**.

Подробная информация о ECMA-262 доступна здесь:
http://www.ecma-international.org/publications/standards/Ecma-262.htm (B.2.1)

Пример

```
$value[abcd `+-=~!@#%&*()_ []{}<>:'",./?абвгд]
<pre>^value.js-escape[]</pre>
```

Выведет...

```
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 (класс)

Класс предназначен для работы с таблицами.

Таблица считается определенной (**def**), если она не пуста. Числовое значение равно количеству строк таблицы.

Конструкторы

create. Создание объекта на основе заданной таблицы

^table::create{табличные_данные}
^table::create[nameless]{табличные_данные}
^table::create{табличные_данные}[опции формата] ***[3.2.2]***

Конструктор создает объект класса **table**, используя табличные данные, определенные в самом конструкторе.

Табличные данные – данные, представленные в формате tab-delimited, то есть столбцы разделяются символом табуляции, а строки – символом перевода строки. При этом части первой строки, разделенные символом табуляции, рассматриваются как имена столбцов, и создается именованная таблица. Пустые строки игнорируются. Если необходимо получить таблицу без имен столбцов (что не рекомендуется), то перед заданием табличных данных необходимо указать параметр **nameless**. В этом случае столбцы первой строки воспринимаются конструктором как данные таблицы, а в качестве имен столбцов выступят их порядковые номера, начиная с нулевого.

Из опций формата пока доступен только **$.separator**.

Пример

```
$tab[^table::create{name      age
Вова  27
Леша  22
}]
```

Будет создан объект **tab** класса **table**, содержащий таблицу из двух строк с именами столбцов **name** и **age**.

create. Копирование существующей таблицы

```
^table::create[таблица]
^table::create[таблица;опции]
```

Конструктор создает объект класса **table**, копируя данные из другой **таблицы**. Также можно задать ряд опций, контролирующих копирование, см. «Опции копирования».

Пример

```
$orig[^table::create{name
Вася
Коля
Маша
}]

#сдвигает текущую запись таблицы orig на «Коля»
^orig.offset(1)

#копирует, начиная с текущей записи в orig, не больше 10 записей
$copy[^table::create[$orig;
    $.offset[cur]
    $.limit(10)
]]

^copy.menu{$copy.name}[, ]
```

Выведет...

```
Коля, Маша
```

load. Загрузка таблицы с диска или HTTP-сервера

```
^table::load[имя файла]
^table::load[имя файла;опции загрузки]
^table::load[nameless;имя файла]
^table::load[nameless;имя файла;опции загрузки]
```

Конструктор создает объект, используя таблицу, определенную в некотором файле или документе на HTTP-сервере. Данные должны быть представлены в формате tab-delimited (см. **table::create**).

Имя файла – имя файла с путем или URL документа на HTTP-сервере.

Опции загрузки – об основных опция см. раздел «Работа с HTTP-серверами», также доступны дополнительные опции, см. «Опции формата файла».

Использование параметра **nameless** такое же, как и в конструкторе **table::create**.

Пример загрузки таблицы с диска

```
$loaded_table[^table::load[/addresses.cfg]]
```

Пример создает объект класса **table**, содержащий именованную таблицу, определенную в файле **addresses.cfg**, который находится в корневом каталоге веб-сайта.

Пример загрузки таблицы с HTTP-сервера

```
$table[^table::load[nameless;http://www.parser.ru/;
      $.headers[
            $.USER-AGENT[table load example]
      ]
```

```
]]
Количество строк: ^table.count[]
<hr />
<pre>$table.0</pre>
```

sql. Выборка таблицы из базы данных

```
^table::sql{SQL-запрос}
^table::sql{SQL-запрос }[$.limit(n) $.offset(o) $.bind[variables hash]]
```

Конструктор создает объект класса **table**, содержащий таблицу, полученную в результате выборки из базы данных.
Для использования конструктора необходимо установленное соединение с сервером базы данных (см. оператор **connect**).

SQL-запрос – запрос на выборку из базы данных

Возможно использование дополнительных параметров конструктора:
$.limit(n) – получить не более **n** записей;
$.offset(o) – отбросить первые **o** записей выборки;
$.bind[hash] – связанные переменные, см. «Работа с IN/OUT переменными» ***[3.1.4]***.

Пример

```
$sql_table[^table::sql{select * from news}]
```

В результате будет создан объект, содержащий все записи из таблицы **news**.

Примечание: всегда указывайте конкретный список необходимых вам полей.
Использование «» крайне не рекомендуется, поскольку постороннему читателю (или вам самим через некоторое время) непонятно, что же за поля будут извлечены. Кроме того, так можно извлечь лишние поля (скажем, добавившиеся в ходе развития проекта), что повлечет ненужные расходы на их извлечение и хранение.*

Опции формата файла

При загрузке и записи файла можно задать символы-разделители столбцов и символы, обрамляющие значения столбцов.

Опция	*По-умолчанию*	*Описание*
$.separator[символ**]**	табуляция	Задает символ, разделитель столбцов
$.encloser[символ**]**	нет	Задает символ, обрамляющий значение столбца.

Пример загрузки .txt файла, созданного Miscrosoft Excel

Excel умеет сохранять данные в простой текстовый файл, разделенный табуляциями:
Файл|Сохранить как... Текст (Разделенный табуляциями) (`.txt`).
Данные сохраняются в следующем формате:

name	*description*
"ООО ""Петров и партнеры"""	Текст

(Значения ряда столбцов обрамляется кавычками, которые внутри самого значения удваиваются)

Чтобы считать такой файл, необходимо указать соответствующую опцию загрузки:

```
$companies[^table::load[companies.txt;
      $.encloser["]
]]
$companies.name
```

Parser также может работать и с `.csv` файлами, достаточно указать опцию:

```
$.separator[^;]
```

Опции копирования и поиска

При копировании записей из одной таблицы в другую, см...
table::create *[3.0.7]*
table.join *[3.0.7]*
и при поиске, см...
table.locate *[3.0.8]*

можно задать хеш **опций**:

$.offset(количество строк)	пропустить указанное **количество строк** таблицы;
$.offset[cur]	с текущей строки **таблицы**;
$.limit(максимум)	**максимум** строк, которые можно обработать;
$.reverse(1/0)	**1**=в обратном порядке. *[3.0.8]*

Получение содержимого столбца

$таблица.поле
Возвращает содержимое столбца **поле** из текущей строки таблицы.

Пример

```
$tab.name
```

Пример вернет значение, определенное в колонке **name** текущей строки таблицы.

Получение содержимого текущей строки в виде хеша

$таблица.fields – содержимое текущей строки таблицы в виде хеша (не доступно для **nameless** таблиц).

Возвращает содержимое текущей строки таблицы в виде хеша. При этом имена столбцов становятся ключами хеша, а значения столбцов – соответствующими значениями ключей.

Использовать этот метод необходимо, если имена столбцов совпадают с именами методов или конструкторов класса table. В таком случае получить их значения напрямую нельзя – Parser будет выдавать сообщение об ошибке. Если необходимо работать с полями, называющимися именно так, можно воспользоваться полем **fields**, и далее работать уже не с таблицей, а с хешем.

Пример

```
$tab[^table::create{menu      line
yes   first
no    second
}]

$tab_hash[$tab.fields]
$tab_hash.menu
$tab_hash.line
```

В результате будут выведены значения полей **menu** и **line** (имена которых совпадают с именами методов класса **table**) как значения ключей хеша **tab_hash**.

Методы

save. Сохранение таблицы в файл

```
^таблица.save[путь]
^таблица.save[путь;опции]          [3.1.2]
^таблица.save[nameless;путь]
^таблица.save[nameless;путь;опции] [3.1.2]
^таблица.save[append;путь]         [3.3.0]
^таблица.save[append;путь;опции]   [3.3.0]
```

Сохраняет таблицу в текстовый файл в формате tab-delimited.
Использование опции **nameless** сохраняет таблицу без имен столбцов.
При использование опции **append** таблица сохраняется с именами столбцов только в том случае, если файла ещё не существует.
Также доступны опции записи, см. «Опции формата файла», позволяющие, например, сохранить файл в `.csv` формате, для последующей загрузки данных в программы, которые понимают такой формат (Miscrosoft Excel).

Пример

```
^conf.save[/conf/old_conf.txt]
```

Таблица **$conf** будет сохранена в текстовом файле **old_conf.txt** в каталоге **/conf/**.

count. Количество строк в таблице

```
^таблица.count[]
```

Выдает количество строк в таблице (**int**).

Пример

```
$goods[^table::create{  pos   good        price
1     Монитор     1000
2     Системный блок    1500
3     Клавиатура  15}
]
Количество: ^goods.count[]
```

Выведет:

```
Количество: 3
```

В выражениях числовое значение таблицы равно количеству строк:

```
^if($goods > 2){больше}
```

menu. Последовательный перебор всех строк таблицы

```
^таблица.menu{код}
^таблица.menu{код}[разделитель]
^таблица.menu{код}{разделитель}
```

Метод **menu** выполняет код для каждой строки таблицы, последовательно перебирая все строки.

Разделитель – код, который вставляется перед каждым непустым не первым телом. Разделитель в квадратных скобках вычисляется один раз, в фигурных – много раз по ходу вызова.

Замечание: если разделитель задан в виде кода, то этот код выполняется после следующего не пустого тела цикла.

В любой момент можно принудительно выйти из цикла с помошью оператора **break**, или принудительно закончить текущую итерацию и перейти к следующей с помощью оператора

`continue`. *[3.2.2]*

Пример

```
$goods[^table::create{  pos   good        price
1     Монитор     1000
2     Системный блок    1500
3     Клавиатура  15}
]
<table border=1>
^goods.menu{
      <tr>
            <td>$goods.pos</td>
            <td>$goods.good</td>
            <td>$goods.price</td>
      </tr>
}
</table>
```

Пример выводит все содержимое таблицы **$goods** в виде HTML-таблицы.

append. Добавление данных в таблицу

```
^таблица.append{табличные данные}
^таблица.append[табличные данные]    [3.4.0]
```

Метод добавляет запись в конец таблицы. Формат представления **данных** – tab-delimited.
Табличные данные должны иметь такую же структуру, как и таблица, в которую добавляются данные.

Пример

```
$stuff[^table::create{name    pos
Alexander   boss
Sergey      coder
}]

^stuff.append{Nikolay   designer}
^stuff.save[stuff.txt]
```

Пример добавит в таблицу **$stuff** новую запись и сохранит таблицу в файл **stuff.txt**.

offset. Смещение указателя текущей строки

```
^таблица.offset(число)
^таблица.offset[cur|set](число)
```

Смещает указатель текущей строки на указанное **число** вниз. Если аргумент метода отрицательный, то указатель перемещается вверх. Смещение указателя осуществляется циклически, то есть, достигнув последней строки таблицы, указатель возвращается на первую.

Необязательный параметр:
cur – смещает указатель относительно текущей строки
set – смещает указатель относительно первой строки

Пример

```
<table border="1">
^goods.offset(-1)
      <tr>
            <td>$goods.pos</td>
            <td>$goods.good</td>
            <td>$goods.price</td>
      </tr>
```

```
</table>
```

Результатом выполнения кода будет HTML-таблица, содержащая последнюю строку таблицы из предыдущего примера (метод **menu**).

offset и line. Получение смещения указателя текущей строки

```
^таблица.offset[]
```

Метод **offset** без параметров возвращает текущее смещение указателя текущей строки от начала таблицы.

Пример

```
$men[^table::create{name
Вася
Петя
Сережа
}]
^men.menu{
    ^men.offset[] — $men.name
}[<br />]
```

Выдаст:

```
0 — Вася
1 — Петя
2 — Сережа
```

Людям более привычно считать записи, начиная с единицы. Для удобного вывода нумерованных списков имеется метод **line**:

```
^таблица.line[]
```

Он позволяет сразу получить номер записи из таблицы в привычном виде, когда номер первой строки равен единице. Если в примере использовать **^men.line[]**, то нумерация будет идти от одного до трех.

sort. Сортировка данных таблицы

```
^таблица.sort{функция сортировки_по_строке}
^таблица.sort{функция_сортировки_по_строке}[направление_сортировки]
^таблица.sort(функция сортировки_по_числу)
^таблица.sort(функция_сортировки_по_числу)[направление_сортировки]
```

Метод осуществляет сортировку таблицы по указанной функции.

Функция сортировки – произвольная функция, по текущему значению которой принимается решение о положении строки в отсортированной таблице. Значением функции может быть строка (значения сравниваются в лексикографическом порядке) или число (значения сравниваются как действительные числа).

Направление сортировки – параметр, задающий направление сортировки. Может быть:
desc – по убыванию
asc – по возрастанию
По умолчанию используется сортировка по возрастанию.

Пример

```
$men[^table::create{name      age
Serge 26
Alex  20
```

```
Mishka      29
}]
^men.sort{$men.name}
^men.menu{
    $men.name: $men.age
}[<br />]
```

В результате записи таблицы **$men** будут отсортированы по столбцу **name** (по строке имени):

```
Alex: 20
Mishka: 29
Serge: 26
```

А можно отсортировать по столбцу **age** (по числу прожитых лет) по убыванию (**desc**), измените в примере вызов **sort** на такой...

```
^men.sort($men.age)[desc]
```

...получится...

```
Mishka: 29
Serge: 26
Alex: 20
```

join. Объединение двух таблиц

```
^таблица1.join[таблица2]
^таблица1.join[таблица2;опции]
```

Метод добавляет в конец **таблицы1** записи из **таблицы2**. При этом из **таблицы2** будет взято значение из столбца, одноименного столбцу **таблицы1**, или пустая строка, если такой столбец не найден. Также можно задать ряд опций, контролирующих добавление, см. «Опции копирования».

Пример

```
^stuff.join[$just_hired_people]
```

Все записи таблицы **$just_hired_people** будут добавлены в таблицу **$stuff**.

flip. Транспонирование таблицы

```
^таблица.flip[]
```

Создает новую **nameless** таблицу с записями, полученными в результате транспонирования исходной таблицы. Иными словами, метод превращает столбцы исходной таблицы в строки, а строки в столбцы.

Пример

```
$emergency[^table::create{id  number
fire  01
police      02
ambulance   03
gas         04
}]

$fliped[^emergency.flip[]]
^fliped.save[fliped.txt]
```

В результате выполнения кода в файл **fliped.txt** будет сохранена такая таблица:

0	1	2	3
fire	police	ambulance	gas
01	02	03	04

locate. Поиск в таблице

```
^таблица.locate[столбец;искомое_значение]
^таблица.locate(логическое_выражение)
^таблица.locate[столбец;искомое_значение;опции]
^таблица.locate(логическое_выражение)[опции]
```

Метод ищет в указанном **столбце** значение, равное **искомому** и возвращает логическое значение «истина/ложь» в зависимости от успеха поиска. В случае если искомое значение найдено, строка, его содержащая, делается текущей. Если искомое значение найдено не было, указатель текущей строки не меняется.
Второй вариант вызова метода ищет первую запись, для которой истинно **логическое выражение**.

Также можно задать ряд опций, контролирующих поиск, см. «Опции поиска».

Поиск чувствителен к регистру букв.

Пример

```
$stuff[^table::create{name          pos   status
Александр   босс  1
Сергей      технолог    1
Тема        арт-директор      2
}]
^if(^stuff.locate[name;Тема]){
    Запись найдена в строке номер ^stuff.line[].<br />$stuff.name:
$stuff.pos<br />
}{
    Запись не найдена
}
```

На экран будет выведено:

```
Запись найдена в строке номер 3.
Тема: арт-директор
```

Подставьте такой поиск в пример...

```
^stuff.locate($stuff.status>1)
```

...и будет найдена первая запись со статусом, большим 1.

select. Отбор записей

```
^таблица.select(критерий_отбора)
```

Метод последовательно перебирает все строки таблицы, применяя к ним выражение **критерий_отбора**, те строки, которые подпали под заданный **критерий** (логическое выражение было истинно), помещаются в результат, которым является таблица с такой же структурой, что и входная.

Пример

```
$men[^table::create{name      age
Serge 26
Alex  20
Mishka      29
}]

$thoseAbove20[^men.select($men.age>20)]
```

В **$thoseAbove20** попадут строки с **Serge** и **Mishka**.

hash. Преобразование таблицы к хешу с заданными ключами

```
^таблица.hash[ключ]
^таблица.hash[ключ][опции]
^таблица.hash[ключ][столбец значений]
^таблица.hash[ключ][столбец значений][опции]
^таблица.hash[ключ][таблица со столбцами значений]
^таблица.hash[ключ][таблица со столбцами значений][опции]
```

Ключ может быть задан, как:
- **[строка]** – название столбца, значение которого считается ключом;
- **{код}** – результат исполнения которого считается ключом;
- **(математическое выражение)** – результат вычисления которого считается ключом.

С опциями по умолчанию метод преобразует таблицу к хешу вида:

```
$хеш[
        $.значение_ключа[
            $.название_столбца[значение_столбца]
            ...
        ]
    ...
]
```

Иными словами, метод создает хеш, в котором ключами являются значения, описанные параметром **ключ**. При этом каждому ключу ставится в соответствие хеш, в котором для всех столбцов таблицы хранятся ассоциации «название столбца – значение столбца в записи».

Если задан столбец значений, то каждому ключу будет соответствовать хеш с одной ассоциацией «название столбца – значение столбца в записи».

Кроме того, можно задать несколько столбцов значений, для этого необходимо передать дополнительным параметром таблицу, в которой перечислены все необходимые столбцы.

Опции – хеш с опциями преобразования.

$.type[hash/string/table] ***[3.2.2]***	**hash**=значение каждого элемента – хеш (по умолчанию); **string**=значение каждого элемента – строка, при этом вы должны указать **один** столбец значений; **table**=значение каждого элемента – таблица при этом вы не можете указать **столбец значений** или **таблица со столбцами значений.** Это сделано для экономии рессурсов, т.к. в результирующем хеше создаются таблицы со ссылками на строки таблиц уже расположенных в памяти, таким образом копирования строк таблиц с их содержимым не происходит.
$.distinct(0/1)	**0**=наличие в ключевом столбце одинаковых значений считается ошибкой (по-умолчанию); **1**=выбрать из таблицы записи с уникальным ключом.
$.distinct[tables] ***[3.0.8]***	создать хеш из таблиц, содержащих строки с ключом. Это устаревший ключ, который равносилен одновременному заданию **$.distinct(1)** и **$.type[table]**.

Пример

Есть список товаров, в котором каждый товар имеет наименование и уникальный код – **id**. Есть прайс-лист товаров, имеющихся в наличии. Вместо названия товара используется **id** товара из списка товаров. Все это хранится в двух таблицах. Подобные таблицы называются связанными. Нам нужно получить данные в виде «товар – цена», т.е. получить данные сразу из двух таблиц.

Реализация:

```
# это таблица с нашими товарами
$product_list[^table::create{id     name
1     хлеб
```

```
2     колбаса
3     масло
4     водка
}]

# это таблица с ценами товаров
$price_list[^table::create{id price
1     6.50
2     70.00
3     60.85
}]

#hash таблицы с ценами по полю id
$price_list_hash[^price_list.hash[id]]

#перебираем записи таблицы с товарами
^product_list.menu{
    $product_price[$price_list_hash.[$product_list.id].price]
#   проверяем — есть ли цена на товар в нашем hash
    ^if($product_price){
#       печатаем название товара и его цену
        $product_list.name — $product_price<br />
    }{
#       а у этого товара нет цены, т.е. его нет в наличии
        $product_list.name — нет в наличии<br />
    }
}
```

В результате получим:

```
хлеб — 6.50
колбаса — 70.00
масло — 60.85
водка — нет в наличии
```

columns. Получение структуры таблицы.

^таблица.columns[]
^таблица.columns[имя столбца] ***[3.2.2]***

Метод создает именованную таблицу из одного столбца, содержащего заголовки столбцов исходной таблицы.
Имя столбца — «**column**» или переданное **имя столбца**.

Пример

```
$columns_table[^stuff.columns[]]
```

Void (класс)

Класс предназначен для работы с «пустыми» объектами. Он не имеет конструкторов, объекты этого класса создаются автоматически, например, когда вы обращаетесь к несуществующей переменной.

Методы

int, double, bool

```
^объект.int[]
^объект.int(значение по умолчанию)
^объект.double[]
^объект.double(значение по умолчанию)
^объект.bool[]
^объект.bool(значение по умолчанию)
```

В случае отсутствия объекта эти методы выдают либо 0, либо **значение по умолчанию**. Эти методы работают, когда выполняется преобразование к **int/double** или **bool** не определенных заранее объектов, скажем, полей форм (см. класс **form**).

Пример

```
^form:number.int[]
```

Если поле **number** определено, то есть было передано, его значение просто преобразуется к классу **int**. Если же это поле не определено, то есть его просто нет, несуществующее значение, относящееся к классу **void**, приравняется к **0**, и ничего страшного не произойдет – код будет выполняться дальше.

length. Длина «строки»

```
^объект.length[]
```

В случае отсутствия объекта этот метод выдает 0.

Пример

```
^if(^form:password.length[]<$MIN_PASSWORD_LENGTH){
    Длина введенного пароля меньше $MIN_PASSWORD_LENGTH
}
```

Если поле **password** определено, то есть было передано, вычислится его длина, и будет проверена. Это обычный вызов метода **^строка.length[]**. Если же это поле не определено, то есть его просто нет, длина несуществующего значения, относящегося к классу **void**, считается равным **0**, и ничего страшного не произойдет – эта длина будет успешно проверена.

pos. Получение позиции подстроки

```
^объект.pos[подстрока]
^объект.pos[подстрока](позиция начала поиска)    [3.3.0]
```

В случае отсутствия объекта этот метод выдает –1.

Пример

```
^if(^form:email.pos[@]>0){
    Может быть...
}
```

Если поле **email** определено, то есть было передано – это обычный вызов метода **^строка.pos[]**. Если же это поле не определено, то есть его просто нет, считается что в несуществующем значении, относящемся к классу **void**, никакие подстроки не существуют, и ничего страшного не произойдет – будет успешно выполнена проверка.

left, right, mid. Получение подстроки

```
^строка.left(N)
^строка.right(N)
^строка.mid(P;N)
^строка.mid(P)
```

В случае отсутствия объекта эти методы выдают пустую подстроку.

Пример

```
^form:email.left(50)
```

Если поле **email** определено, то есть было передано — это обычный вызов метода **^строка.left[]**. Если же это поле не определено, то есть его просто нет, считается что в несуществующем значении, относящемся к классу **void**, никакие подстроки не существуют, и ничего страшного не произойдет – будет успешно выдана пустая строка.

Статический метод

sql. Запрос к БД, не возвращающий результат

```
^void:sql{SQL-запрос}
^void:sql{SQL-запрос}[$.bind[variables hash]] [3.1.4]
```

Осуществляет выполнение SQL-запроса, который не возвращает результат (операции по управлению данными в базе данных).
Для работы этого метода необходимо установленное соединение с сервером базы данных (см. оператор **connect**).

Возможно использование дополнительного параметра конструктора:
$.bind[hash] – связанные переменные, см. «Работа с IN/OUT переменными» ***[3.1.4]***.

Пример

```
^connect[строка подключения]{
      ^void:sql{create table users (id int, name text, email text)}
}
```

В результате выполнения этого кода в базе данных будет создана таблица `users`, при этом запрос не вернет никакого результата. Пример дан для СУБД MySQL.

XDoc (класс)

Класс предназначен для работы с древовидными структурами данных в паре с **xnode**, и поддерживает считывание файлов в **XML** формате и запись в XML (http://www.w3.org/XML) и HTML, а также **XSLT** (http://www.w3.org/TR/xslt) трансформацию.

Работа с деревом производится в **DOM**-модели (http://www.w3.org/DOM), доступен DOM1 и ряд возможностей DOM2.

Класс реализует DOM-интерфейс Document и является наследником класса **xnode**.

Ошибки DOM-операций (интерфейс DOMException) преобразуются в исключения **xml**-типа.

Конструкторы

create. Создание документа на основе заданного XML

```
^xdoc::create{XML-код}
^xdoc::create[базовый_путь]{XML-код}
```

Конструктор создает объект класса **xdoc** из **XML-кода**. Возможно задание **базового пути**.

Пример

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
текст
</document>}]
$response:body[^document.string[]]
```

create. Создание нового пустого документа

```
^xdoc::create[имя_тега]
^xdoc::create[базовый_путь;имя_тега]
```

Конструктор создает объект класса **xdoc**, состоящий из единственного тега **имя_тега**. Возможно задание **Базового пути**.

Пример

```
$document[^xdoc::create[document]]
$paraNode[^document.createElement[para]]
$addedNode[^document.documentElement.appendChild[$paraNode]]
$response:body[^document.string[]]
```

create. Создание документа на основе файла

```
^xdoc::create[файл]
```

Конструктор создает объект класса **xdoc**, состоящий из **XML-кода** содержащегося в файле.

Пример

```
$file[^file::load[binary;http://server/data.xml;
      $.timeout(10)
]]

$xdoc[^xdoc::create[$file]]
$response:body[^xdoc.string[]]
```

load. Загрузка XML с диска, HTTP-сервера или иного источника

```
^xdoc::load[имя файла]
```

Конструктор загружает XML-код из некоторого файла или адреса на HTTP-сервере и создает на его основе объект класса **xdoc**.
Parser может считать XML из произвольного источника, см. раздел «Чтение XML из произвольного источника».

имя файла – имя файла с путем или URL файла на HTTP-сервере.

Пример загрузки XML-документа с диска

```
$xdoc[^xdoc::load[article.xml]]
$response:body[^xdoc.string[]]
```

Пример загрузки XML-документа с HTTP-сервера

```
$xdoc[^xdoc::load[http://www.cbr.ru/scripts/XML_daily.asp]]
На
    ^xdoc.selectString[string(/ValCurs/@Date)]
курс валюты
    $node[^xdoc.selectSingle[/ValCurs/Valute[CharCode='USD']]]
    "^node.selectString[string(Name)]"
равен
    ^node.selectString[string(Value)]
<hr />
<pre>^taint[^xdoc.string[]]</pre>
```

parser://метод/параметр. Чтение XML из произвольного источника

Parser может считать XML из произвольного источника.
Везде, где можно считать XML, можно задать адрес документа вида...
`parser://метод/параметр`

Считывание документа по такому адресу приводит чтению результата работы метода Parser, **`^метод[/параметр]`**.

Пример хранения XSL шаблонов в базе данных

```
@main[]
...
# к этому моменту в $xdoc находится документ, который хотим преобразовать
^xdoc.transform[parser://xsl_database/main.xsl]

@xsl_database[name]
^string:sql{select text from xsl where name='$name'}
```

Причем относительные ссылки будут обработаны точно также, как если бы файлы читались с диска. Скажем, если `parser://xsl_database/main.xsl` ссылается на `utils/common.xsl`, будет загружен документ
`parser://xsl_database/utils/common.xsl`, для чего будет вызван метод **`^xsl_database[/utils/common.xsl]`**.

Параметр создания нового документа: Базовый путь

В конструкторах нового документа можно задать **`Базовый путь`**.

По действию он аналогичен заданию атрибута

```
<...
    xmlns:xml="http://www.w3.org/XML/1998/namespace"
    xml:base="базовый URI" ...
```

Отличаясь тем, что пути задаются стандартным для Parser способом (см. «Приложение 1. Пути к файлам и каталогам»), что куда удобнее задания полного дискового пути, включающего путь к веб-пространству. По умолчанию равен пути к текущему обрабатываемому документу.
Внимание: символ «/» на конце пути обязателен.

Пример

```
$sheet[^xdoc::create[/xsl/]{<?xml version="1.0" encoding="$request:charset"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:import href="import.xsl"/>
</xsl:stylesheet>
}]
```

Здесь файл `import.xsl`, будет считан из каталога `/xsl/`.

Методы

DOM

DOM1-интерфейс Document:

```
$Element[^документ.createElement[tagName]]
$DocumentFragment[^документ.createDocumentFragment[]]
$Text[^документ.createTextNode[data]]
$Comment[^документ.createComment[data]]
$CDATASection[^документ.createCDATASection[data]]
$ProcessingInstruction[^документ.createProcessingInstruction[target;data]]
$Attr[^документ.createAttribute[name]]
$EntityReference[^документ.createEntityReference[name]]
$NodeList[^документ.getElementsByTagName[tagname]]
```

DOM2-интерфейс Document:

```
$Node[^документ.importNode[importedNode](deep)]
$Element[^документ.createElementNS[namespaceURI;qualifiedName]]  [3.1.1]
$Attr[^документ.createAttributeNS[namespaceURI;qualifiedName]]  [3.1.1]
$NodeList[^документ.getElementsByTagNameNS[namespaceURI;localName]]
$Element[^документ.getElementById[elementId]]
```

В Parser
- DOM-интерфейсы Node и Element и их производные реализованы в классе **xnode**;
- DOM-интерфейс NodeList — класс **hash** с ключами 0, 1, ...;
- DOM-тип DOMString — класс **string**;
- DOM-тип boolean — логическое значение: 0=ложь, 1=истина.

Подробная спецификация DOM1 доступна здесь: http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

Подробная спецификация DOM2 доступна здесь: http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

string. Преобразование документа в строку

```
^документ.string[]
^документ.string[Параметры_преобразования_в_текст]
```

Преобразует документ в текстовую форму. Возможно задание **параметров_преобразования**.
По умолчанию создается XML-представление документа с заголовком **<?xml ... ?>** (можно отключить вывод заголовка, задав соответствующий параметр).

Результат выдается посетителю **as-is**. ***[3.1.4]***

Пример

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
строка1<br/>
строка2<br/>
</document>}]

^document.string[
    $.method[html]
]
```

save. Сохранение документа в файл

```
^документ.save[путь]
^документ.save[путь;Параметры_преобразования_в_текст]
```

Сохраняет документ в текстовый файл. Возможно задание **параметров_преобразования** в текст. По умолчанию создается XML-представление документа с заголовком **<?xml ... ?>** (можно отключить вывод заголовка, задав соответствующий параметр).

Путь – путь к файлу.

Пример

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
строка1<br/>
строка2<br/>
</document>}]

^document.save[saved.xml]
```

file. Преобразование документа к объекту класса file

```
^документ.file[Параметры_преобразования_в_текст]
```

Преобразует документ к типу **file**. Возможно задание **параметров_преобразования** в текст. По умолчанию создается XML-представление документа с заголовком **<?xml ... ?>** (можно отключить вывод заголовка, задав соответствующий параметр).

Пример

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
строка1<br/>
строка2<br/>
</document>}]

$response:body[^document.file[]]
```

transform. XSL преобразование

```
^документ.transform[шаблон]
^документ.transform[шаблон][XSLT-параметры]
```

Осуществляет XSL-преобразование **документа** по **шаблону**. Возможно задание **XSLT-параметров**.

Шаблон – или **путь_к_файлу_с_шаблоном**, или **xdoc** документ.
Parser может считать XML из произвольного источника, см. раздел «Чтение XML из произвольного источника».

XSLT-параметры – хеш строк, доступных из шаблона через **<xsl:param ... />**.

Внимание: Parser (в виде модуля к Apache или IIS) кеширует результат компиляции **файла_с_шаблоном** *во внутреннюю форму, повторная компиляция не производится, а скомпилированный шаблон берется из кеша. Вариант CGI также кеширует шаблон, но только на один запрос. Шаблон перекомпилируется при изменении даты файлов шаблона.*

Пример (см. также «Урок 6. Работаем с XML»)

```
# входной xdoc документ
$sourceDoc[^xdoc::load[article.xml]]

# преобразование xdoc документа шаблоном article.xsl
```

```
$transformedDoc[^sourceDoc.transform[article.xsl]]

# выдача результата в HTML виде
^transformedDoc.string[
    $.method[html]
]
```

Если **шаблон** не считывается с диска, а создается динамически, важным вопросом становится «а откуда загрузятся **<xsl:import href="some.xsl"/>**?», обратите внимание на возможность задания базового пути: «Параметр создания нового документа: Базовый путь».

Параметры преобразования документа в текст

В ряде методов можно задать хеш **Параметры_преобразования_в_текст**.

Они идентичны атрибутам элемента **<xsl:output ... />**.
Исключением являются атрибуты **doctype-public** и **doctype-system**, которые так задать нельзя.
Пока также является исключением **cdata-section-elements**.

По умолчанию текст создается в кодировке **$request:charset**, однако в XML заголовке или в элементе **meta** для HTML-метода Parser указывает кодировку **$response:charset**. Такое поведение можно изменить, явно указав кодировку в **<xsl:output ... />** или соответствующем параметре преобразования. ***[3.1.2]***

При создании объекта класса **file** можно задать параметр **media-type**, при задании нового тела ответа заголовок ответа **content-type** получит значение этого параметра.

Пример

```
# выдаст документ в HTML-представлении без отступов
^document.string[
    $.method[html]
    $.indent[no]
]
```

Выдача XHTML

Если необходимо выдать XHTML, следует использовать такие атрибуты элемента **<xsl:stylesheet ... />**:

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

Обратите внимание на указание **xmlns** без префикса — так необходимо делать, чтобы все создаваемые в шаблоне элементы без префикса попадали в пространство имен `xhtml`. Необходимо задавать **xmlns** без префикса в каждом `.xsl` файле, этот параметр не распространяется на включаемые файлы.

А также необходимо задать такие атрибуты **<xsl:output ... />**:

```
<xsl:output
      doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
      doctype-system="DTD/xhtml1-strict.dtd"
      />
```

Внимание: не задавайте атрибут method. XHTML это разновидность метода xml, включающаяся при использовании следующих doctype:

```
      -//W3C//DTD XHTML 1.0 Strict//EN
      -//W3C//DTD XHTML 1.0 Frameset//EN
      -//W3C//DTD XHTML 1.0 Transitional//EN
```

Поля

DOM

DOM1-интерфейс Document:

```
$DocumentType[$документ.doctype]
$Element[$документ.documentElement]
```

В Parser DOM-интерфейсы Node и Element и их производные реализованы в классе **xnode**.

Подробная спецификация DOM1 доступна здесь: http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

search-namespaces. Хеш пространств имен для поиска

```
$документ.search-namespaces
```

Для использования префиксов пространств имен в методах **xnode.select*** необходимо заранее эти префиксы определить в данном хеше.

Здесь
- ключи — префиксы пространств имен,
- значения — их URI.

Добавление нескольких префиксов

```
$xdoc[^xdoc::create{<?xml version="1.0"?>
<document xmlns:s="urn:special">
      <s:code xmlns:o="urn:other" o:attr="123">давай поиграем в прятки</s:code>
</document>
}]
^xdoc.search-namespaces.add[
   $.s[urn:special]
   $.o[urn:other]
]
^xdoc.selectString[string(//s:code[@o:attr=123])]
```

Добавление одного префикса

```
$xdoc.search-namespaces.s[urn:special]
```

XNode (класс)

Класс предназначен для работы с древовидными структурами данных в паре с **xdoc**, поддерживает **XPath** (http://www.w3.org/TR/xpath) запросы.

Класс реализует DOM-интерфейсы Node и Element и их производные.
Класс напрямую не создается, используются соответствующие методы класса **xdoc**.

Вместо DOM-интерфейса NamedNodeMap в Parser используется класс **hash**.

Методы

DOM

DOM1-интерфейс Node:

```
$Node[^узел.insertBefore[$newChild;$refChild]]
$Node[^узел.replaceChild[$newChild;$oldChild]]
$Node[^узел.removeChild[$oldChild]]
$Node[^узел.appendChild[$newChild]]
^if(^узел.hasChildNodes[]){…}
$Node[^узел.cloneNode(deep)]
```

DOM1-интерфейс Element:

```
^узел.getAttribute[name]
^узел.setAttribute[name;value]
^узел.removeAttribute[name]
$Attr[^узел.getAttributeNode[name]]
$Attr[^узел.setAttributeNode[$newAttr]]
$Attr[^узел.removeAttributeNode[$oldAttr]]
$NodeList[^узел.getElementsByTagName[name]]
^узел.normalize[]
```

DOM2-интерфейс Element:

```
$строка[^узел.getAttributeNS[namespaceURI;localName]]   [3.1.1]
^узел.setAttributeNS[namespaceURI;qualifiedName;value]   [3.1.1]
^узел.removeAttributeNS[namespaceURI;localName]   [3.1.1]
$Attr[^узел.getAttributeNodeNS[namespaceURI;localName]]   [3.1.1]
$Attr[^узел.setAttributeNodeNS[$newAttr]]   [3.1.1]
$NodeList[^узел.getElementsByTagNameNS[namespaceURI;localName]]
^if(^узел.hasAttribute[name]){…}   [3.1.1]
^if(^узел.hasAttributeNS[namespaceURI;localName]){…}   [3.1.1]
^if(^узел.hasAttributes[]){…}   [3.2.2]
```

В Parser
- DOM-интерфейс NodeList – класс **hash** с ключами 0, 1, ...;
- DOM-тип DOMString – класс **string**;
- DOM-тип boolean – логическое значение: 0=ложь, 1=истина.

Подробная спецификация DOM1 доступна здесь: http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

Подробная спецификация DOM2 доступна здесь: http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

select. XPath поиск узлов

```
$NodeList[^узел.select[XPath-запрос]]
```

Выдает список узлов, найденных в контексте **узла** по заданному **XPath-запросу**. Если запрос не вернул подходящих узлов, выдается пустой список.

Для использования в **запросе** префиксов пространств имен необходимо их заранее определить, см. **$xdoc.search-namespaces**.

Пример

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
```

```
<t/><t/>
</document>}]

# результат=список из двух элементов "t"
$list[^d.select[/document/t]]
# перебираем найденные листы:
#   этот код будет работать
#   даже если запрос не найдет ни одного листа
^for[i](0;$list-1){
    $node[$list.$i]
    Имя: $node.nodeName<br />
    Тип: $node.nodeType<br />
}
```

В Parser DOM-интерфейс NodeList – класс **hash** с ключами 0, 1, ...

Подробная спецификация XPath доступна здесь: http://www.w3.org/TR/xpath

selectSingle. XPath поиск одного узла

^узел.selectSingle[XPath-запрос]

Выдает узел, найденный в контексте **узла** по заданному **XPath-запросу**. Если запрос не нашел подходящего узла, выдается **void**. Если запрос выдал больше, чем один узел, выдается ошибка.

Для использования в **запросе** префиксов пространств имен необходимо их заранее определить, см. **$xdoc.search-namespaces**.

Пример

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<t attr="привет" n="123"/>}]

# результат=один элемент "t"
$element[^d.selectSingle[t]]
# результат=2 (количество атрибутов <t>)
Количество атрибутов: ^element.attributes._count[]<br />
```

Подробная спецификация XPath доступна здесь: http://www.w3.org/TR/xpath

selectString. Вычисление строчного XPath запроса

^узел.selectString[XPath-запрос]

Выдает результат выполнения **XPath-запроса** в контексте **узла**, если это строка. Если не строка, выдается ошибка типа **parser.runtime**.

Для использования в **запросе** префиксов пространств имен необходимо их заранее определить, см. **$xdoc.search-namespaces**.

Пример

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<t attr="привет" n="123"/>}]

# результат=привет
^d.selectString[string(t/@attr)]
```

Подробная спецификация XPath доступна здесь: http://www.w3.org/TR/xpath

selectNumber. Вычисление числового XPath запроса

^узел.selectNumber[XPath-запрос]

Выдает результат выполнения **XPath-запроса** в контексте **узла**, если это число. Если не число, выдается ошибка типа **parser.runtime**.

Для использования в **запросе** префиксов пространств имен необходимо их заранее определить, см. **$xdoc.search-namespaces**.

Пример

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<t attr="привет" n="123"/>}]

#результат=124
^d.selectNumber[number(/t/@n)+1]<br />
#результат=4
^d.selectNumber[2*2]<br />
```

Подробная спецификация XPath доступна здесь: http://www.w3.org/TR/xpath

selectBool. Вычисление логического XPath запроса

^узел.selectBool[XPath-запрос]

Выдает результат выполнения **XPath-запроса** в контексте **узла**, если это логическое значение. Если не логическое значение, выдается ошибка типа **parser.runtime**.

Для использования в **запросе** префиксов пространств имен необходимо их заранее определить, см. **$xdoc.search-namespaces**.

Пример

```
$d[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<t attr="привет" n="123"/>}]

^if(^d.selectBool[/t/@n > 10]){
    /t/@n больше 10
}{
    не больше
}
```

Подробная спецификация XPath доступна здесь: http://www.w3.org/TR/xpath

Поля

DOM

DOM1-интерфейс Node:

```
$узел.nodeName
$узел.nodeValue
$узел.nodeValue[новое значение]      [3.1.2]
^if($узел.nodeType == $xnode:ELEMENT_NODE){…}
$Node[$узел.parentNode]
$NodeList[$узел.childNodes]
$Node[$узел.firstChild]
$Node[$узел.lastChild]
$Node[$узел.previousSibling]
$Node[$узел.nextSibling]
$NodeList[$узел_типа_ELEMENT.attributes]
$Document[$node.ownerDocument]
```

DOM2-интерфейс Node:

```
$узел.prefix
$узел.namespaceURI
```

DOM1-интерфейс Element:

```
$узел_типа_ELEMENT.tagName
```

DOM1-интерфейс Attr:

```
$узел_типа_ATTRIBUTE.name
^if($узел_типа_ATTRIBUTE.specified){…}
$узел_типа_ATTRIBUTE.value
```

DOM1-интерфейс ProcessingInstruction:

```
$узел_типа_PROCESSING_INSTRUCTION.target
$узел_типа_PROCESSING_INSTRUCTION.data
```

DOM1-интерфейс DocumentType:

```
$узел_типа_DOCUMENT_TYPE.name
$узел_типа_DOCUMENT_TYPE.entities
$узел_типа_DOCUMENT_TYPE.notations
```

DOM1-интерфейс Notation:

```
$узел_типа_NOTATION.publicId
$узел_типа_NOTATION.systemId
```

В Parser

- DOM-интерфейс NodeList — класс **hash** с ключами 0, 1, ...;
- DOM-тип DOMString — класс **string**;
- DOM-тип boolean — логическое значение: 0=ложь, 1=истина.

Подробная спецификация DOM1 доступна здесь: http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

Подробная спецификация DOM2 доступна здесь: http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

Константы

DOM. nodeType

DOM-элементы бывают разных типов, тип элемента хранится в integer поле **nodeType**.
В классе **xdoc** имеются следующие константы, удобные для проверки значения этого поля:

```
$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
```

Пример

```
^if($node.nodeType == $xnode:ELEMENT_NODE){
    <$node.tagName />
}
```

Приложение 1. Пути к файлам и каталогам, работа с HTTP-серверами

Для доступа к файлам и каталогам в Parser можно использовать абсолютный или относительный путь.

Абсолютный путь начинается слешем, а файл ищется от корня веб-пространства. Файл по относительному пути ищется от каталога, в котором находится запрошенный документ.

Пример абсолютного пути:
`/news/archive/20020127/sport.html`

Пример относительного пути:
относительно каталога `/news/archive`…
`20020127/sport.html`

При записи файлов необходимые каталоги создаются автоматически.

Внимание: корень веб-пространства, переданный веб-сервером, можно изменить: см. «Корень веб-пространства».

*Внимание: Parser преобразует пути к языку **file-spec** (см. «Внешние и внутренние данные»).*

Также методы…
- **file::load**
- **table::load**

…может работать с внешними HTTP-серверами, если имя загружаемого документа содержит префикс
`http://`

Этим методам также можно задать дополнительные опции загрузки документа по HTTP, это хеш,

ключами которого могут быть:

Опция	*По-умолчанию*	*Значение*
$.charset[кодировка**]**	берется из заголовка HTTP-ответа	Кодировка документов на удале сервере. В эту кодировку переко, строка запроса, и из этой кодирс перекодируется ответ. ***[3.1.0]*** Также данная опция доступна и загрузке локальных текстовых ф ***[3.2.2]***
$.timeout(секунд**)**	2 секунды	Время ожидания ответа HTTP сер секундах. Операция загрузки дол завершена за это время, иначе в
$.method[HTTP-МЕТОД**]**	`GET`	Название HTTP-метода должно указано в верхнем регистре. Название метода можно указыв нижнем регистре ***[3.3.1]***
$.enctype[CONTENT-TYPE**]**	application/x-www-form-urlencoded	Допустимые значение: application/x-www-form-urlencode или multipart/form-data. Последнее должно быть исполь вместе с методом `POST` в случа отправляете удалённому серве ***[3.3.1]***
$.form[** **$.поле[строка**]** **$.поле[**файл**]** **$.поле[**$таблица**]** … **]**	отсутствует	Параметры запроса. Для `GET` зап будут переданы в `?строке_зап` запросов с другим **method**, пара будут переданы с `Content-type: application form-urlencoded` Значением может являться строк из одного столбца или файл ***[3.3*** *Предпочтительно задавать пара запросам именно при помощи* **$** *не передавать их в* `?параметра` *самостоятельно.* *Однако можно передавать их и* ***[3.1.5]***
$.body[текст**]**	отсутствует	Текст тела запроса (нельзя совме **form** и методом **GET**) ***[3.1.2]***
$.cookies[** **$.имя[значение**]** … **]**	отсутствует	Хеш, содержащий список cookie необходимо передать удалённом серверу. ***[3.2.3]***
$.headers[** **$.HTTP-ЗАГОЛОВОК[значение**]** … **]**	`$.User-Agent[parser3]`	Хеш, содержащий дополнительн заголовки, которые необходимо на HTTP-сервер Значение HTTP-заголовка может дата, строка или хеш с обязатель ключом **value**. Дата может использоваться и в к значения поля и в качестве значе атрибута поля, при этом она буд
$.any-status(true)	false/0	Логическое: допустим ли статус равный 200. Если ЛОЖЬ, и будет статус, не равный 200, возникне системная ошибка `http.status` ***[3.0.8]***

Для ^**file::load[**…**]** также можно дополнительные опции загрузки ***[3.0.8]***, это хеш, ключами которого могут быть:

Опция	***По-умолчанию***	***Значение***
$.offset(смещение**)**	0	Загрузить данные начиная с этого смещения (в байтах).
$.limit(ограничение**)**	-1	Загрузить не более данного количес байт.

Переменная CLASS_PATH

В конфигурационном методе может быть задана переменная или таблица **CLASS_PATH**, в которой задается путь (пути) к каталогу с файлами классов. Если имя подключаемого модуля – относительно, то файл ищется по **CLASS_PATH**, (если **CLASS_PATH** таблица, то каталоги в ней перебираются снизу вверх).

Пример таблицы **CLASS_PATH**:

```
$CLASS_PATH[^table::create{path
/classes/common
/classes/specific
}]
```

Теперь по относительному пути `my/class.p` поиск файла будет проходит в таком порядке:
`/classes/specific/my/class.p`
`/classes/common/my/class.p`

Приложение 2. Форматные строки преобразования числа в строку

Форматная строка определяет форму представления значения числа. В общем случае она имеет следующий вид:

%Длина.ТочностьТип

Тип – определяет способ преобразования числа в строку.

Существуют следующие типы:

d – десятичное целое число со знаком

u – десятичное целое число без знака

o – восьмеричное целое число без знака

x – шестнадцатиричное целое число без знака; для вывода цифр, больших 9, используются буквы a, b, c, d, e, f

X – шестнадцатиричное целое число без знака; для вывода цифр, больших 9, используются буквы A, B, C, D, E, F

f – действительное число

Точность – точность представления дробной части, т. е. количество знаков после запятой. Если для отображения дробной части значения требуется больше знаков, то значение округляется. Обычно точность указывают в том случае, если используется тип преобразования **f**. Для других типов указывать точность не рекомендуется. Если точность не указана, то для типа преобразования **f** она по умолчанию

принимается равной **6**. Если указана точность **0**, то число выводится без дробной части

Длина – количество знаков, отводимое для значения. Может получиться так, что для отображения полученного значения требуется меньше символов, чем указано в блоке **Длина**. Например, указана длина **10**, а получено значение **123**. В этом случае слева к значению будет приписано семь пробелов. Если нужно, чтобы слева приписывались не пробелы, а нули, следует в начале блока **Длина** поместить **0**, например, написать не **10**, а **010**. Блок **Длина** может отсутствовать, тогда для значения будет отведено ровно столько символов, сколько требуется для его отображения.

Приложение 3. Формат строки подключения оператора connect

Строка подключения обрабатывается драйвером базы данных для Parser3.

Для MySQL

```
mysql://user:password@host[:port]|[/unix/socket]/database?
      charset=значение& [значением может быть направление перекодирования для MySQL 3.x/4.0
или название кодировки для MySQL 4.1+]
      ClientCharset=кодировка& [3.1.2]
      timeout=3&
      compress=0&
      named_pipe=1&
      autocommit=1&
      multi_statements=0 [3.3.0]
```

Необязательные параметры:
`Port` – номер порта сервера баз данных. Можно использовать выражение:
`user:password@имя_хоста:номер_порта/database`,

А можно вместо `имени_хоста` и `номера_порта` передать путь к UNIX сокету в квадратных скобках (UNIX socket – это некий магический набор символов (путь), который вам расскажет администратор MySQL, если он – это не вы. Через этот сокет может идти общение с сервером):
`user:password@[/unix/socket]/database`

`charset` – сразу после соединения выполняет команду «`SET CHARACTER SET значение`»;
`ClientCharset` – задает кодировку, в которой необходимо общаться с SQL-сервером, перекодированием занимается драйвер;
`timeout` – задает значение параметра `Connect timeout` в секундах;
`compress` – режим сжатия трафика между сервером и клиентом;
`named_pipe` – использование именованных каналов для соединения с сервером MySQL, работающим под управлением Windows NT;
`autocommit` – если установлен в 0, то после соединения выполняет команду «`SET AUTOCOMMIT=0`» (в документации по MySQL следует прочтать, как работает autocommit, в том числе какие команды вызывают COMMIT);
`multi_statements` – если установлен в 1, то текст SQL запроса может содержать несколько инструкций, разделённых символом ';' (*символ ";" необходимо предварять символом "^"*).

Пример: перекодирование средствами SQL сервера (рекомендуется, требуется MySQL 4.1 или выше)

MySQL сервер версии 4.1 и выше имеет богатые возможности по перекодированию данных, поэтому в случае его использования рекомендуется задействовать именно их, используя опцию `charset`, а не заниматься перекодированием средствами драйвера с помощью опции `ClientCharset`. В случае, если вы используете версию MySQL 4.1 и выше, вы даже можете в разных таблицах хранить данные в разных кодировках, хотя мы считаем, что в этом случае лучше всего хранить данные в кодировке `UTF-8`.

Допустим, данные в вашей базе хранятся в кодировке `UTF-8`, а сайт работает в кодировке `windows-1251`, в этом случае нужно использовать следующую строку подключения:
`mysql://user:password@host/database?charset=cp1251`

В этом случае сразу после соединения SQL серверу будет выдана команда «**`SET CHARACTER SET cp1251`**» и сервер сам будет перекодировать принимаемые данные из кодировки `cp1251` в кодировку, в которой данных хранятся у него в таблице и обратно.

Внимание: в данном случае вы должны указать кодировку, в которой работает сайт.
Внимание: данная опция выполняет команду MySQL, поэтому необходимо использовать названия кодировок MySQL сервера, которые отличаются от названий кодировок Parser, определяемых вами в конфигурационном файле.

Пример: база в koi8-r, страницы в windows-1251, перекодирование SQL сервером (рекомендуется, MySQL 3.x и 4.0)

MySQL сервер версии 3.x и 4.0 не умеет перекодировать данные произвольным образом, однако умеет перекодировать их для наиболее распространённого для русского языка случая, когда символы в БД хранятся в кодировке `koi8-r`, а сайт работает в кодировке `windows-1251`.

В этом случае данные можно перекодировать средствами драйвера, задав опцию `ClientCharset=koi8-r` (см. ниже), однако лучше это делать средствами SQL сервера, использовав следующую строку подключения:
`mysql://user:password@host/database?charset=cp1251_koi8`

В этом случае сразу после соединения SQL серверу будет выдана команда «**`SET CHARACTER SET cp1251_koi8`**», однако MySQL сервер данных версий трактует её иначе, а именно: он будет приходящие данные перекодировать из кодировки `cp1251` в кодировку `koi8-r` и обратно.

Внимание: вы указываете опцию, указывающую направление перекодирования, при этом других значений данной опции, например koi8_cp1251, MySQL сервер не имеет.

Пример: база в windows-1251, страницы в koi8-r, перекодирование драйвером (работает со всеми версиями MySQL сервера)

В некоторых редких случаях бывает, что невозможно использовать функции перекодирования, предоставляемые MySQL сервером. Тогда можно задействовать механизмы перекодирования драйвера, используя опцию `ClientCharset`.
Допустим, данные в вашей базе хранятся в кодировке `windows-1251`, а сайт работает в кодировке `koi8-r`, в этом случае можно использовать такую строку подключения:
`mysql://user:password@host/database?ClientCharset=windows-1251`

В этом случае отправляемые SQL серверу данные будут перекодироваться драйвером из кодировки **`$request:charset`** (в данном примере `koi8-r`) в кодировку `windows-1251`, а принимаемые от SQL сервера данные — обратно.

Внимание: в данном случае вы должны указать кодировку, в которой данные хранятся в базе данных.
Внимание: в данной опции вы должны указывать названия кодировок Parser, которые определяются вами в конфигурационном файле.

Для SQLite

```
sqlite://path-to-DB-file?
      ClientCharset=UTF-8& [3.3.0]
      autocommit=1& [3.3.0]
      multi_statements=0 [3.3.0]
```

Путь к файлу с базой данных задаётся относительно document_root, кроме того в качестве пути к файлу драйвер понимает специальные значения **:memory:** и **:temporary:**. В первом случае на сессию будет создаваться временная база данных в памяти, а во втором случае — на диске.

`autocommit` — по умолчанию SQLite автоматически выполняет `COMMIT` после каждого успешно выполненного запроса. Если указать опцию `autocommit=0`, то такое поведение будет изменено, и Parser в начале оператора **connect** будет выдавать комманду `BEGIN`, а в конце — `COMMIT` или `ROLLBACK`. Таким образом все запросы, написанные внутри одного оператора `connect` будут выполняться в рамках одной транзакции;
`multi_statements` — если установлен в 1, то текст SQL запроса может содержать несколько инструкций, разделённых символом ';' (*символ ";" необходимо предварять символом "^"*);
`ClientCharset` — по умолчанию драйвер перекодирует все отправляемые текстовые данные в UTF-8 и обратно (числа и BLOB-ы не перекодируются), однако в некоторых случаях, если у вас есть БД, содержащая данные в иной кодировке (что в применении к SQLite некорректно), используя данную опцию вы можете задать кодировку, в которую драйвер будет производить перекодирование данных при общении с SQL-сервером.

Примеры

Для работы с базой данных **my.db** которая располагается в директории **data**, находящейся рядом с директорией, на которую указывает document_root, строку подключения стоит написать так:
`sqlite://../data/my.db`

Для работы с временной базой данных, расположенной в памяти и без autocommit, строку подключения стоит написать так:
`sqlite://:memory:?autocommit=0`

Для ODBC

```
odbc://строка_соединения_смотрите_документацию_по_ODBC?
      ClientCharset=кодировка& [3.1.2]
      autocommit=1& [3.3.0]
      SQL=MSSQL|FireBird|Pervasive [3.3.0]
```

`ClientCharset` — задает кодировку, в которой необходимо общаться с SQL-сервером, перекодированием занимается драйвер;
`autocommit` — по умолчанию Parser автоматически выполняет `COMMIT` после каждого успешно выполненного запроса. Если указать опцию `autocommit=0`, то такое поведение будет изменено и все запросы, написанные внутри одного оператора **connect** будут выполняться в рамках одной транзакции.
`SQL` — если указана, то Parser будет использовать специфику для указанного сервера при модифицировании запросов с `limit/offset`. В настоящий момент драйвер понимает только значения `MSSQL`, `Pervasive` и `FireBird`. Для первых двух серверов SQL-запрос модифицируется путём добавления в него «`TOP (limit+offset)`», для последнего — «`FIRST (limit) SKIP (offset)`».

Рекомендуем этот сайт, здесь собраны строки соединения ко всевозможным базам данных: www.connectionstrings.com.

Внимание: при работе с MS-SQL при языковой настройке отличной от английской возникают неудобства при форматировании дат и чисел — SQL сервер форматирует их согласно языковой настройке, что обычно совершенно неудобно при их программной обработке. Настоятельно

рекомендуем сразу после соединения с сервером выполнить команду переключения языковой настройки в us_english, что обеспечит поддержку дат в ANSI SQL92 формате и чисел с десятичным разделителем «точка»:

^void:sql{SET LANGUAGE us_english}

Примеры

MS-SQL:

```
odbc://DRIVER={SQL
Server}^;SERVER=сервер^;DATABASE=база^;UID=пользователь^;PWD=пароль
```

Microsoft Access (`.mdb` файл):

```
odbc://Driver={Microsoft Access Driver (*.mdb)}^;Dbq=C:\полный\путь\к\файлу.mdb
```

Ссылка на **системный** источник данных, созданный в Пуск|Настройки|Панель управления|Источники данных (ODBC).

```
odbc://DSN=dsn^;UID=пользователь^;PWD=пароль
```

Замечание: В коде Parser символ ";" в строке подключения к БД необходимо предварять символом "^".

Пример

Допустим вы храните данные в MS-SQL сервере в кодировке windows-1251, строку подключения стоит написать так:

```
odbc://DRIVER={SQL
Server}^;SERVER=сервер;UID=пользователь^;PWD=пароль?ClientCharset=windows-
1251&SQL=MSSQL
```

Для PostgreSQL

```
pgsql://user:password@host[:port]|[local]/database?
      charset=значение&
      ClientCharset=кодировка&  [3.1.2]
      autocommit=1&  [3.3.0]
      datestyle=ISO,SQL,Postgres,European,US,German
            по умолчанию ISO
```

Необязательные параметры:
`port` – номер порта.

Можно задать:
`user:password@host:port/database`,

а можно:
`user:password@local/database`

В этом случае произойдет соединение с сервером, расположенным на локальной машине.

`charset` – сразу после соединения выполняет команду «`SET CLIENT ENCODING=значение`»;
`ClientCharset` – задает кодировку, в которой необходимо общаться с SQL-сервером, перекодированием занимается драйвер;
`autocommit` – по умолчанию Parser автоматически выполняет `COMMIT` после каждого успешно выполненного запроса. Если указать опцию `autocommit=0`, то такое поведение будет изменено и все запросы, написанные внутри одного оператора **`connect`** будут выполняться в рамках одной транзакции;
`datestyle` – если задан этот параметр, то при соединении с сервером драйвер выполнит команду «`SET DATESTYLE=значение`»

Пример: перекодирование средствами SQL сервера (рекомендуется)

Допустим данные в вашей базе хранятся в кодировке `UTF-8`, а сайт работает в кодировке `windows-1251`, в этом случае нужно использовать следующую строку подключения:`pgsql://user:password@host/database?charset=win`

В этом случае сразу после соединения SQL серверу будет выдана команда «**`SET CLIENT ENCODING=win`**» и сервер сам будет перекодировать принимаемые данные из кодировки `win` в кодировку, в которой данных хранятся у него в таблице и обратно.

Внимание: в данном случае вы должны указать кодировку, в которой работает сайт.
Внимание: данная опция выполняет команду PgSQL, поэтому необходимо использовать названия кодировок PgSQL сервера, которые отличаются от названий кодировок Parser, определяемых вами в конфигурационном файле.

Пример: перекодирование драйвером (работает со всеми версиями PgSQL сервера)

В некоторых редких случаях бывает, что невозможно использовать функции перекодирования, предоставляемые PgSQL сервером. Тогда можно задействовать механизмы перекодирования драйвера, используя опцию `ClientCharset`.
Допустим, данные в вашей базе хранятся в кодировке `windows-1251`, а сайт работает в кодировке `koi8-r`, в этом случае можно использовать такую строку подключения:
`pgsql://user:password@host/database?ClientCharset=windows-1251`

В этом случае отправляемые SQL серверу данные будут перекодироваться драйвером из кодировки **`$request:charset`** (в данном примере `koi8-r`) в кодировку `windows-1251`, а принимаемые от SQL сервера данные – обратно.

Внимание: в данном случае вы должны указать кодировку, в которой данные хранятся в базе данных.
Внимание: в данной опции вы должны указывать названия кодировок Parser, которые определяются вами в конфигурационном файле.

Для Oracle

```
oracle://user:password@service?
      ClientCharset=кодировка&  [3.1.2]
      LowerCaseColumnNames=0&  [3.1.1]
      DisableQueryModification=0&  [3.3.0]
      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` – задает кодировку, в которой необходимо общаться с SQL-сервером, перекодированием занимается драйвер.

Если имена колонок в запросе **`select`** не взять в кавычки, Oracle преобразует их к ВЕРХНЕМУ регистру. По-умолчанию, Parser преобразует их к нижнему регистру. Указав параметр `LowerCaseColumnNames=0` можно отключить преобразование в нижний регистр.

При выполнении запроса с `limit/offset` драйвер модифицирует текст запроса для отсечения

ненужных данных средствами SQL сервера. Однако в случае проблем это поведение можно отключить с помощью параметра `DisableQueryModification=1`.

Информацию по остальным параметрам можно найти в документации по «Environment variables» для Oracle. Однако, мы рекомендуем всегда задавать параметры `NLS_LANG` и `NLS_DATE_FORMAT` такими, какие указаны выше.

Пример

Допустим данные в вашей базе хранятся в кодировке windows-1251, строку подключения стоит написать так:

```
oracle://user:password@service?ClientCharset=windows-
1251&NLS_LANG=RUSSIAN_AMERICA.CL8MSWIN1251&NLS_DATE_FORMAT=YYYY-MM-DD
HH24:MI:SS
```

ClientCharset. Параметр подключения — кодировка общения с SQL-сервером

Параметр `ClientCharset` определяет кодировку, в которой необходимо общаться с SQL-сервером. Если параметр не указан, Parser считает, что общение с SQL-сервером идет в кодировке **`$request:charset`**.

Список допустимых кодировок определяется в Конфигурационном файле.

Приложение 4. Perl-совместимые регулярные выражения

Подробную информацию по Perl-совместимым регулярные выражениям (Perl Compatible Regular Expressions, PCRE) можно найти в документации к Perl (см. http://perldoc.perl.org/perlre.html), в документации к использованной в Parser библиотеке PCRE 2.09 (см. http://www.pcre.org/man.txt), а также в большом количестве специальной литературы, содержащей помимо всего остального много практических примеров. Особенно детально использование регулярных выражений описано в книге Дж. Фридла «Регулярные выражения» издательства «O'Reilly» (ISBN 1-56592-257-3), перевод книги на русский язык: издательство «Питер» (ISBN: 5-272-00331-4, второе издание; ISBN: 5-318-00056-8 первое издание).

Краткое описание, которое приводится тут, имеет справочный характер.

Регулярное выражение — это шаблон для поиска подстроки, который должен совпасть с подстрокой слева направо в строке поиска. Большинство символов в этом шаблоне представлены сами собою, и при поиске просто проверяется наличие этих символов в строке поиска в заданной последовательности. В качестве простейшего примера можно привести шаблон для поиска «**шустрая лиса**», который должен совпасть с аналогичным набором символов в строке поиска. Мощь регулярных выражений состоит в том, что помимо обычных символов, они позволяют включать в шаблоны альтернативные варианты выбора и повторяющиеся фрагменты с помощью метасимволов. Эти метасимволы ничего не значат сами по себе, но при использовании их в регулярных выражениях, они обрабатываются особым образом.

Существует два различных набора метасимволов:

1. Распознаваемые в любой части шаблона, не заключенной в квадратные скобки
2. Распознаваемые в частях шаблона, заключенных в квадратные скобки

К метасимволам, распознаваемым вне квадратных скобок относятся следующие:

****	общее обозначение для escape-последовательностей. Имеют различное использование, рассмотрены ниже
^	совпадает с началом фрагмента для поиска или перед началом строки в многострочном режиме
$	совпадает с концом фрагмента для поиска или перед концом строки в многострочном режиме
.	символьный класс, содержащий все символы. Этот метасимвол, совпадает с любым символом кроме символа новой строки по умолчанию.
[...]	символьный класс. Совпадение происходит с любым элементом из заданного в квадратных скобках списка
|	метасимвол означающий «или». Позволяет объединить несколько регулярных выражений в одно, совпадающее с любым из выражений-компонентов
(...)	ограничение подстроки поиска в общем шаблоне поиска
?	совпадает с одним необязательным символом
*****	совпадает с неограниченным количеством любых необязательных символов, указанных слева
+	совпадает с неограниченным количеством символов, указанный слева. Для совпадения требуется хотя бы один произвольный символ
{мин, макс}	интервальный квантификатор – требуется минимум экземпляров, допускается максимум экземпляров.

Часть шаблона, заключенная в квадратные скобки называется символьным классом. В описании символьного класса можно использовать только следующие метасимволы:

****	Escape – символ
^	Инвертированный символьный класс, мета-символ обязательно должен быть первым символом в описании класса. Совпадение будет происходить с любыми символами, не входящими в символьный класс
-	Используется для обозначения интервала символов
[...]	Ограничитель символьного класса

Использование метасимвола «****».

Обратный слеш имеет несколько вариантов использования. В случае если вслед за ним следует символ, не обозначающий букву алфавита, обратный слеш выполняет функцию экранирования и отменяет специальное значение, которое может иметь этот символ. Такое использование этого метасимвола возможно как внутри символьного класса, так и вне его. В качестве примера, если необходимо найти символ «*****», то используется следующая запись в шаблоне «*****». В случае необходимости экранировать сам символ «****» используется запись «****».

Второй вариант использования этого мета-символа – для описания управляющих символов в шаблоне. Можно использовать следующие escape-последовательности:

\a сигнал
\cx «control-x», где x – любой символ
\e ASCII-символ escape
\f подача бумаги
\n новая строка
\r возврат курсора
\t табуляция
\xhh шестнадцатиричный код символа hh
\ddd восьмиричный код символа ddd

Третий вариант – для определения специфических символьных классов

\d любое десятичное число [0-9]
\s пропуск, обычно [\f\n\r\t] Первый символ квадратных скобках – пробел
\w символ слова, обычно [a-zA-Z0-9_]
\D \S \W отрицание \d \s \w

Четвертый вариант – для обозначения мнимых символов. В PCRE существуют символы, которые соответствуют не какой-либо литере или литерам, а означают выполнение определенного условия, поэтому в английском языке они называются утверждениями (assertion). Их можно рассматривать как мнимые символы нулевого размера, расположенные на границе между реальными символами в точке, соответствующей определенному условию. Эти утверждения не могут использоваться в символьных классах (**\b** имеет дополнительное значение и обозначает возврат каретки внутри символьного класса)

\b граница слова
\B отсутствие границы слова
\A «истинное» начало строки
\Z «истинный» конец строки или позиция перед символом начала новой строки, расположенного в «истинном» конце строки
\z «истинный» конец строки

Приложение 5. Как правильно назначить имя переменной, функции, классу

Имя должно быть понятно как минимум вам самим, а как идеал – любому человеку, читающему ваш код. Имя может быть набрано русскими или латинскими буквами, главное – единообразие. Рекомендуем все же пользоваться английским (а вдруг вас ждет мировое признание?). Слова в именах лучше использовать в единственном числе. Если есть необходимость – пользуйтесь составными именами вида **column_color**. Глядя на такое имя можно сразу понять, что оно означает.

Parser чувствителен к регистру!

$Parser и **$parser** – разные переменные!

Есть определенные ограничения на использование в именах символов. Для Parser имя всегда заканчивается перед:
пробелом
табуляцией
переводом строки
;] }) " < > # + * / % & | = ! ' , ?
в выражениях заканчивается и перед "**-**"

Код:
```
$var[значение_из_переменной]
$var>text
```

выдаст на экран:
```
значение_из_переменной>text
```

т.е. символ '>' Parser считает окончанием имени переменной **$var** и подставляет ее значение, поэтому вышеуказанные символы не следует использовать при составлении имен.

Если есть необходимость сразу после значения переменной (т.е. без пробела, который является концом имени) вывести символ, который не указан выше (например, нам нужно поставить точку сразу после значения переменной) используется следующий синтаксис:
${var}.text

даст:
значение_из_переменной.text

Нельзя (!) пользоваться в именах символами "**.**", "**:**", "**^**" поскольку они будут расцениваться как часть кода Parser, что приведет к ошибкам при обработке вашего кода.

Все остальные символы использовать в именах, в принципе, можно, но лучше всего отказаться от использования в именах каких-либо служебных и специальных символов кроме случаев крайней необходимости (в практике не встречаются), за исключением знака подчеркивания, который не используется Parser и достаточно нагляден при использовании в именах.

Приложение 6. Как бороться с ошибками и разбираться в чужом коде

Для начала вдумчиво прочитайте сообщение об ошибке. В нем содержится имя файла, вызвавшего ошибку и номер строки в нем. Здесь требуется внимательность при написании кода и справочник. Всегда помните о том, что Parser оперирует объектной моделью, поэтому внимательно следите за тем, с объектом какого класса вы работаете. Некоторые методы возвращают объекты других классов!

Так, например, некоторые методы класса **date** возвращают объект класса **table**. Попытка вызвать для этого объекта методы класса **date** приведет к ошибке. Нельзя вызывать методы классов для объектов, которые к этим классам не принадлежат. Впрочем, этот этап вам удастся преодолеть довольно быстро. Еще одна категория ошибок – ошибки в логике работы самого кода. Это уже сложнее, и придется запастись терпением. Обязательно давайте грамотные имена переменным, методам, классам и комментируйте код.

Если и в этом случае не удастся понять причины неверной работы – попробуйте обратиться к справочнику. «Если ничего не помогает – прочтите, наконец, инструкцию...» Последняя стадия в поиске ошибок – вы близки к сумасшествию, пляшете вокруг компьютера с бубном, а код все равно не работает. Здесь остается только обратиться за помощью к тем, кто пока разбирается в Parser чуть лучше, чем вы. Задайте свой вопрос на форуме, посвященном работе на этом языке и вам постараются ответить. Вы не одиноки! Удачи вам!

Приложение 7. SQL сервера, работа с IN/OUT переменными

При работе с SQL сервером Oracle поддерживается работа со связанными переменными (bind variables), поддерживаются `IN`, `OUT` и `IN/OUT` переменные, которые связываются с передаваемым в запрос хешем.

При прямом использовании конструкций `CALL` и `EXECUTE` в некоторых версиях Oracle имеются известные проблемы, рекомендуем пользоваться PL/SQL оберткой (`begin …; end;`), не забывайте экранировать знак «;».

Примечание: значение типа **void** *соответствует* `NULL`*. Во втором примере* **days** *имеет начальное значение* `NULL`*.*

Пример использования IN переменных

```
#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)
    ]
]
```

Пример использования IN и OUT переменных

```
#procedure read_user_ban_days(user_id in number, days out number)

$variables[
    $.user_id(7319)
#несмотря на то, что параметр OUT, все равно необходимо его передать
#его текущее значение будет проигнорировано
    $.days[]
]

^void:sql{begin read_user_ban_days(:user_id, :days)^; end^;}[
    $.bind[$variables]
]

Пользователь выключен на $variables.days!
```

Установка и настройка Parser

Parser3 доступен в нескольких вариантах:
- CGI скрипт (и интепретатор),
- модуль к веб-серверу Apache 1.3,
- ISAPI расширение веб-сервера Microsoft Internet Information Server 4.0 или новее.

Дополнительно можно установить драйверы для различных SQL-серверов (сейчас доступны для MySQL, PgSQL, Oracle, ODBC).

Описание каталогов и файлов :
`parser3[.exe]` – CGI скрипт (и интерпретатор)
`ApacheModuleParser3.dll` – модуль к веб-серверу Apache 1.3
`parser3isapi.dll` – ISAPI расширение веб-сервера IIS 4.0 или новее

`auto.p.dist` – пример Конфигурационного файла

`parser3.charsets/` – каталог с файлами таблиц кодировок:

`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]

Поскольку исходные коды являются открытыми, вы можете сами собрать Parser (см. Сборка Parser их исходных кодов) и написать свой SQL-драйвер.
Доступны скомпилированные версии Parser и его SQL-драйверов под ряд платформ (см. http://www.parser.ru/download/).
Внимание: в целях безопасности они скомпилированы так, что могут читать и исполнять только файлы, принадлежащие тому же пользователю/группе пользователей, от имени которых работает сам Parser.

Как подключаются конфигурационные файлы?

Для CGI скрипта (`parser3[.exe]`):
конфигурационный файл считывается из файла, заданного переменной окружения `CGI_PARSER_CONFIG`,

Если переменная не задана, ищется в том же каталоге, где расположен сам CGI скрипт.

Для модуля к Apache путь к конфигурационному файлу задается директивой:

```
#can be in .htaccess
ParserConfig полный_путь
```

Для ISAPI расширения (`parser3isapi.dll`):
конфигурационный файл `auto.p` ищется в том же каталоге, где расположен сам файл.

Конфигурационный файл

Пример файла включен в поставку (см. `auto.p.dist`).

Этот файл — основной, с которого начинается сборка класса **MAIN**. Может содержать Конфигурационный метод, который выполняется первым, до метода **auto**, и задает важные системные параметры.

После выполнения конфигурационного метода можно задать кодировку ответа и кодировку, в которой набран код (по умолчанию в обоих случаях используется кодировка **UTF-8**):

Рекомендуемый код:

```
@auto[]
#source/client charsets
$request:charset[windows-1251]
$response:charset[windows-1251]
$response:content-type[
      $.value[text/html]
      $.charset[$response:charset]
]
```

Примечание: для корректной работы методов **upper** *и* **lower** *класса* **string** *с национальными языками (в том числе русским) необходимо корректное задание* **$request:charset**.

Также здесь рекомендуется определить путь к классам вашего сайта:

```
$CLASS_PATH[/../classes]
```

И строку соединения с SQL-сервером, используемым на вашем сайте (пример для ODBC):

```
$SQL.connect-string[odbc://DSN=www_mydomain_ru^;UID=user^;PWD=password]
```

Примечание: вашем коде вы будете использовать ее так:

```
^connect[$SQL.connect-string]{…}
```

Советуем поместить сюда же определение метода **unhandled_exception**, который будет выводить сообщение о возможных проблемах на вашем сайте.

Внимание: конечно, Конфигурационный файл можно не использовать, а Конфигурационный метод поместить в файл `auto.p` *в корне веб-пространства, однако в разных местах размещения сервера (например: отладочная версия и основной сервер) конфигурации скорее всего будут различными, и очень удобно, когда эти различия находятся в отдельном файле и вне веб-пространства.*

Конфигурационный метод

Если в файле определен метод **conf**, он выполняется первым, до **auto**, и задает важные системные параметры:

- файлы, описывающие кодировки символов,
- ограничение на размер HTTP POST-запроса,
- сервер/программу отправки почты,
- SQL-драйвера и их параметры,
- таблицу соответствия расширения имени файла и его mime-типа.

Рекомендуется поместить этот метод в Конфигурационный файл.

Определение метода:
@conf[filespec]
filespec – полное имя файла, содержащего метод.

Всегда доступна и не нуждается в загрузке файла кодировка **UTF-8**, являющаяся для Parser кодировкой по умолчанию.
Чтобы сделать доступными для использования Parser другие кодировки, необходимо указать файлы их описывающие, делается это так:

```
$CHARSETS[
	$.windows-1251[/полный/путь/к/windows-1251.cfg]
	...
]
```

См. Описание формата файла, описывающего кодировку.

Максимальный размер POST данных:

```
$LIMITS[
	$.post_max_size(10*0x400*0x400)
]
```

Параметр отправки писем (см. **^mail:send[**...**]**)...

...под Windows и UNIX (под UNIX ***[3.1.2]***) адрес SMTP-сервера

```
$MAIL[
	$.SMTP[mail.office.design.ru]
]
```

...под UNIX в safe-mode версиях, настроить программу отправки можно только при сборке Parser из исходных кодов, в бинарных версиях, распространяемых с сайта `parser.ru`, задана команда

```
	/usr/sbin/sendmail -i -t -f postmaster
```

Только в unsafe-mode версиях можно задать программу отправки почты самому:

```
$MAIL[
	$.sendmail[/custom/mail/sending/program params]
]
```

и, по умолчанию, используется эта...

```
	/usr/sbin/sendmail -t -i -f postmaster
```

...или эта...

```
	/usr/lib/sendmail -t -i -f postmaster
```

...команда, в зависимости от вашей системы.
При отправке письма вместо «`postmaster`» будет подставлен адрес отправителя из письма из обязательного поля заголовка «`from`».

Также можно задать таблицу SQL-драйверов:

```
$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
}]
]
```

В колонке **client** таблицы **drivers** допустимы параметры клиентской библиотеке, отделяемые знаком **?** от имени файла библиотеки, в таком виде:
имя1=значение1&имя2=значение2&...
а также ***имя+=значение***.
Эти переменные будут занесены(**=**) или добавлены к имеющемуся значению(**+=**) в программное окружение (environment) перед инициализацией библиотеки. В частности, удобно добавить путь к Oracle библиотекам здесь, если этого не было сделано в системном программном окружении (system environment).

Таблица типов файлов:

```
#файл, создаваемый ^file::load[…],
#при выдаче в $response:body задаст этот $response:content-type
$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
}]
```

Расширения имен файлов в таблице должны быть написаны в нижнем регистре. Поиск по таблице нечувствителен к регистру, т.е. файл `FACE.GIF` получит mime-тип `image/gif`.

Описание формата файла, описывающего кодировку

Данные в формате tab-delimited со следующими столбцами:

`char` – символ, или его код, заданный в десятичной или шестнадцатеричной форме (0xHH) в той кодировке, которую определяет этот файл.

`white-space`, **`digit`**, **`hex-digit`**, **`letter`**, **`word`** – набор флажков, задающих класс этого символа. Пустое содержимое означает непринадлежность символа к этому классу, непустое [например, 'x'] – принадлежность.

Подробнее о символьных классах см. описание регулярных выражений в литературе.

`lowercase` – если символ имеет пару в нижнем регистре, то символ или код парного символа. Скажем, у буквы 'W' есть парная 'w'. Используется в регулярных выражениях для поиска, не чувствительного к регистру символов, а также в методах **`lower`** и **`upper`** класса **`string`**.

`unicode1` – основной Unicode код символа. Если совпадает с кодом символа, то можно не указывать. Скажем, у буквы 'W' он совпадает, а у буквы 'Я' – нет.

`unicode2` – дополнительный Unicode символа, если имеется.

Установка Parser на веб-сервер Apache, CGI скрипт

Для установки Parser необходимо внести изменения в основной конфигурационный файл веб-сервера, или, если у вас нет к нему доступа, необходима возможность использовать `.htaccess` файлы.

По-умолчанию, в установке Apache возможность использования `.htaccess` отключена.
Если она вам необходима, разрешите их использовать (по крайней мере, задавать `FileInfo`). Для чего в основном конфигурационном файле веб-сервера (обычно `httpd.conf`) в секцию `<virtualhost ...>` вашего сайта, или вне ее — для всех сайтов, добавьте директивы:

```
<Directory /путь/к/вашему/веб/пространству>
AllowOverride FileInfo
</Directory>
```

Parser3 самостоятельно выполняет необходимые перекодирования, так что для Русского Apache добавьте в основной конфигурационный файл веб-сервера (обычно `httpd.conf`) строку:

```
CharsetDisable On
```

запрещающую использование возможностей перекодирования Русского Apache для вашего сервера. Если возможности изменить основной конфигурационный файл веб-сервера у вас нет, добавьте эту строку в `.htaccess` файл.

Поместите файл с исполняемым кодом Parser (в текущей версии, `parser3`) в каталог для CGI-скриптов (закачивать файл по ftp нужно в режиме binary, а не text). Дайте ему права на выполнение, которые можно уточнить у вашего хостинг-провайдера (обычно необходимые права — 755).
Под `Win32`: если вы используете версию Parser с поддержкой XML, в этот же каталог распакуйте XML библиотеки.

Добавьте в файл `.htaccess` вашего сайта (или в `httpd.conf` в секцию `<virtualhost ...>` вашего сайта, или вне ее — для всех сайтов) блоки:

```
# назначение обработчиком .html страниц
AddHandler parser3-handler html
Action parser3-handler /cgi-bin/parser3

# запрет на доступ к .p файлам, в основном, к auto.p
<Files ~ "\.p$">
  Order allow,deny
  Deny from all
</Files>
```

Если вас не устраивает расположение конфигурационного файла по умолчанию (см. Установка и настройка Parser), вы можете задать его явно:

```
# задание переменной окружения с путем к auto.p
SetEnv CGI_PARSER_CONFIG /путь/к/файлу/auto.p
```

Замечание: для этого необходим модуль `mod_env`, который по умолчанию установлен.

Об ошибках Parser делает записи в журнал ошибок `parser3.log`, который, по умолчанию, расположен в том же каталоге, где и CGI-скрипт Parser. Если у Parser нет возможности сделать запись в данный файл, об ошибке будет сообщено в стандартный поток ошибок, и запись об ошибке попадет в журнал ошибок веб-сервера. Если вас не устраивает расположение журнала ошибок `parser3.log`, вы можете задать его явно:

```
# задание переменной окружения с путем к parser3.log
SetEnv CGI_PARSER_LOG /путь/к/файлу/parser3.log
```

Замечание: для этого необходим модуль `mod_env`, который по умолчанию установлен.

Установка Parser на веб-сервер Apache 1.3, модуль сервера

Для установки Parser необходимо внести изменения в основной конфигурационный файл веб-сервера, или, если у вас нет к нему доступа, необходима возможность использовать `.htaccess` файлы.

По-умолчанию, в установке Apache возможность использования `.htaccess` отключена.
Если она вам необходима, разрешите их использовать (по крайней мере, задавать `FileInfo`). Для чего в основном конфигурационном файле веб-сервера (обычно `httpd.conf`) в секцию вашего `<virtualhost ...>` вашего сайта, или вне ее — для всех сайтов, добавьте директивы:

```
<Directory /путь/к/вашему/веб/пространству>
AllowOverride FileInfo
</Directory>
```

Parser3 самостоятельно выполняет необходимые перекодирования, так что для Русского Apache добавьте в основной конфигурационный файл веб-сервера (обычно `httpd.conf`) строку:

```
CharsetDisable On
```

запрещающую использование возможностей перекодирования Русского Apache для вашего сервера.
Если возможности изменить основной конфигурационный файл веб-сервера у вас нет, добавьте эту строку в `.htaccess` файл.

Под UNIX:
Необходимо собрать Parser из исходных кодов, задав ключ `--with-apache13` у скрипта `configure`, при `make` на экране появится инструкция по дальнейшей сборке Apache из его исходных кодов.
Внимание: на некоторых системах стандартный `make` *не работает с make-файлами Parser3, воспользуйтесь GNU вариантом:* `gmake`.

Под Win32:
Поместите файлы с исполняемым кодом модуля Parser (в текущей версии, `ApacheModuleParser3.dll`) в произвольный каталог.
Если вы используете версию Parser с поддержкой XML, в каталог, указанный в переменной окружения `PATH` (например, C:\WinNT), распакуйте XML библиотеки.
Добавьте в файл `httpd.conf` после имеющихся строк `LoadModule`:

```
# динамическая загрузка модуля
LoadModule parser3_module x:\path\to\ApacheModuleParser3.dll
```

Внимание: если необходимо, поместите сопутствующие `.dll` *файлы в тот же каталог.*

А после имеющихся строк `AddModule` (если не имеются, не добавляйте):

```
# добавление модуля к списку активных модулей
AddModule mod_parser3.c
```

Внимание: до 3.1.1 версии Parser: ***`AddModule mod_parser3.C (на конце .C большая)`***

Добавьте в файл `.htaccess` вашего сайта (или в `httpd.conf` в секцию `<virtualhost ...>` вашего сайта, или вне ее — для всех сайтов) блоки:

```
# назначение обработчиком .html страниц
AddHandler parser3-handler html

# задание Конфигурационного файла
ParserConfig x:\path\to\parser3\config\auto.p

# запрет на доступ к .p файлам, в основном, к auto.p
<Files ~ "\.p$">
    Order allow,deny
    Deny from all
</Files>
```

Установка Parser на веб-сервер IIS 5.0 или новее

Поместите файлы с исполняемым кодом Parser в произвольный каталог.
Если вы используете версию Parser с поддержкой XML, в каталог, указанный в переменной окружения PATH (например, C:\WinNT), распакуйте XML библиотеки.

После чего назначьте Parser обработчиком `.html` страниц:
1. Запустите Management Console, нажмите на правую кнопку мыши на названии вашего веб-сервера и выберите **Properties**.
2. Перейдите на вкладку **Home directory** и в разделе **Application settings** нажмите на кнопку **Configuration**...
3. В появившемся окне нажмите на кнопку **Add**.
4. В поле **Executable** введите полный путь к файлу `parser3.exe` или `parser3isapi.dll`.
5. В поле **Extension** введите строку **.html**.
6. Включите опцию **Check that file exists**.
7. Нажмите на кнопку **OK**.

Подобие mod_rewrite

Для веб-сервера IIS встроенного подобия Apache модулю `mod_rewrite` (см. также http://www.egoroff.spb.ru/portfolio/apache/mod_rewrite.html) нет, есть только модули, разработанные сторонними компаниями.

Однако можно назначить произвольную страницу `handler.html` в качестве обработчика 404 ошибки (рекомендуем ее же назначить обработчиком ошибок 403.14 и 405).

Оригинальный запрос при этом будет доступен в **$request:uri**.

К сожалению, при обработке `POST` запросов к адресам, в которых не указано имя документа (.../), **IIS не передает** тело POST запроса CGI скриптам.
Возможный вариант выхода из ситуации: задавать для таких страниц
<form action="form.html"...
и перехватывать неизбежную ошибку отсутствия файла `form.html` в **@unhandled_exception**, и подавлять ее запись в журнал ошибок.

Использование Parser в качестве интерпретатора скриптов

```
/путь/к/parser3 файл_со_скриптом
x:\путь\к\parser3 файл_со_скриптом
```

Выполнять скрипты можно и без веб-сервера, достаточно запустить интерпретатор Parser, передав ему в командной строке параметр — имя скрипта. При этом корнем веб-пространства считается текущий каталог.

При этом ошибки попадут в стандартный поток ошибок, который можно перенаправить в желаемый файл так:
```
команда 2>>error_log
```
Внимание: не забывайте его время от времени очищать.

На UNIX можно также использовать стандартный подход с заданием команды запуска интерпретатора в первой строке скрипта:
```
#!/путь/к/parser3
#ваш код
Проверка: ^eval(2*2)
```
Не забудьте зажечь биты атрибута, разрешающие исполнение владельцу и группе. Команда:
`chmod ug+x файл`

Сборка Parser из исходных кодов

Загрузите из CVS исходные коды Parser3 и необходимых дополнительных модулей. Для этого выполните следующую команду:
`cvs -d :pserver:anonymous@cvs.parser.ru:/parser3project login`
Пароль пустой.

```
cvs -d :pserver:anonymous@cvs.parser.ru:/parser3project get -r имя_ветки
имя_модуля
```

`Имя_ветки` — если не указывать **`-r`**, вы получите текущую разрабатываемую версию (HEAD). Для получения стабильной версии, забирайте ветку «`release_3_X_XXXX`».

`Имя модуля`:
Имя основного модуля: `parser3`.

Модуль с SQL драйверами: `sql`.
Сейчас в нем доступны каталоги:

```
sql/mysql
sql/pgsql
sql/oracle
sql/odbc
sql/sqlite
```

Дла сборки SQL драйверов необходимо наличие исходников Parser3 и, т.к. .h файлы ищутся по относительным путям, структура каталогов должна быть следующей:

```
parser3project  <- директория, где вы решили положить исходники
  |
  +-parser3     <- исходники парсера
  |
  +-sql
     +-mysql    <- исходники драйвера mysql
     +-...      <- исходники других необходимых вам драйверов
```

Для компиляции под UNIX...

...варианта Parser в виде модуля Apache 1.3 необходимо сначала в каталоге с исходными кодами Apache исполнить команду
`./configure`
и только после этого собирать Parser.

Для компиляции под Win32...

...необходим каталог:
`win32/tools`

...SQL драйверов необходимы каталоги:

```
win32/sql/mysql
win32/sql/pgsql
win32/sql/oracle
win32/sql/sqlite
```

...варианта Parser, работающего с XML, в файле `parser3/src/include/pa_config_fixed.h` необходима директива
`#define XML`

...варианта Parser, принимающего письма по электронной почте, в файле

`parser3/src/include/pa_config_fixed.h` необходима директива
`#define WITH_MAILRECEIVE`

Пользователи *UNIX/Cygwin*, инструкции по компиляции и установке читайте в файлах INSTALL каждого модуля.

Для компиляции под Win32 используйте *Microsoft Visual Studio.NET (2003 или новее)*, используйте файлы `.sln` каждого модуля. Распаковывайте все модули в каталог `parser3project`, находящийся в корне (важно!) диска.

Индекс

– B –

– C –

– – –

– D –

– E –

– – –

– F –

– G –

– H –

– I –

– J –

– K –

– L –

– M –

– N –

– O –

– P –

– Q –

– R –

– S –

– T –

– U –

– V –

– W –

– X –

– Y –

– Z –