Студия Лебедева представляет...

...язык скриптования сайтов Parser3.

Автор технологии Parser и Parser 3:
Константин Моршнев | https://artlebedev.ru/moko/

Авторы Parser 3:
Александр Петросян (PAF) | http://paf.design.ru
Михаил Петрушин (Misha v.3) | http://misha.design.ru

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

Оглавление

Как работать с документацией 12

Принятые обозначения 12

Введение 12

Урок 1. Меню навигации 14

Урок 2. Меню навигации и структура страниц 17

Урок 3. Первый шаг. Раздел новостей 22

Урок 4. Второй шаг. Переход к работе с БД 27

Урок 5. Пользовательские классы Parser 32

Урок 6. Работа с XML 37

Конструкции языка Parser 3 39

Переменные ... 39
Хеш (ассоциативный массив) ... 41
Массив ... 41
Объект класса ... 42
Статические поля и методы ... 43
Определяемые пользователем классы ... 43
Определяемые пользователем методы и операторы ... 46
Передача параметров ... 49
Свойства ... 49

Литералы 52

Строковые литералы ... 52
Числовые литералы ... 53
Логические литералы ... 53
Литералы в выражениях ... 53

Операторы 54

Операторы в выражениях и их приоритеты ... 54
- **def. Проверка определенности объекта** ... 55
- **in. Проверка наличия документа в каталоге** ... 55
- **is. Проверка типа** ... 55
- **−f и −d. Проверка существования файла и каталога** ... 56
- **Комментарии к частям выражения** ... 56

eval. Вычисление математических выражений ... 57
Операторы ветвления ... 57

if. Выбор одного варианта из двух ... 57
switch. Выбор одного варианта из нескольких ... 58
Циклы ... 58
break. Выход из цикла ... 58
continue. Переход к следующей итерации цикла ... 59
for. Цикл с заданным числом повторов ... 59
while. Цикл с условием ... 59
cache. Сохранение результатов работы кода ... 60
connect. Подключение к базе данных ... 61
process. Компиляция и исполнение строки ... 61
rem. Вставка комментария ... 63
return. Возврат из метода ... 63
sleep. Задержка выполнения программы ... 63
use. Подключение модулей ... 64
Внешние и внутренние данные ... 64
taint. Задание преобразований данных ... 65
untaint. Задание преобразований данных ... 70
apply-taint. Применение преобразований данных ... 71
Обработка ошибок ... 72
try. Перехват и обработка ошибок ... 72
throw. Сообщение об ошибке ... 73
@unhandled_exception. Вывод необработанных ошибок ... 74
Системные ошибки ... 76
Операторы, определяемые пользователем ... 77
Кодировки 77
Класс MAIN, обработка запроса 78
array (класс) 79
Конструкторы ... 79
create. Создание массива с заданными значениями или пустого массива ... 79
copy. Копирование массива или хеша ... 79
sql. Создание массива на основе выборки из базы данных ... 80
Поля ... 81
Методы ... 82
add. Добавление элементов из другого массива или хеша с перезаписью ... 82
append. Добавление элементов в конец массива ... 82
at. Доступ к элементу массива по порядковому номеру ... 82
compact. Удаление неинициализированных элементов ... 83
contains. Проверка существования элемента по индексу ... 83
count. Количество элементов массива ... 83
delete. Удаление элемента массива ... 83
for. Перебор всех элементов массива ... 84
foreach. Перебор элементов массива ... 84
insert. Вставка элементов в указанную позицию массива ... 85
join. Добавление элементов другого массива или хеша ... 85
keys. Список индексов массива ... 85
left. Получение первых n элементов массива ... 85
mid. Получение диапазона элементов массива ... 86
pop. Удаление и возврат последнего элемента массива ... 86
push. Добавление элемента в конец массива ... 86
remove. Удаление элемента со сдвигом ... 86
reverse. Обратный порядок элементов ... 87

right. Получение последних n элементов массива ... 87
select. Отбор элементов ... 87
set. Установка значения элемента массива ... 88
sort. Сортировка массива ... 88
bool (класс) 89
console (класс) 89
Статическое поле ... 89
Чтение строки ... 89
Запись строки ... 89
cookie (класс) 89
Статические поля ... 89
Чтение ... 89
Запись ... 90
fields. Все cookie ... 90
curl (класс) 91
Статические методы ... 91
info. Информация о последнем запросе ... 91
load. Загрузка файла с удаленного сервера ... 92
options. Задание опций для сессии ... 93
session. Создание сессии ... 93
version. Возврат текущей версии cURL ... 94
Опции работы с библиотекой cURL ... 94
date (класс) 97
Конструкторы ... 97
create. Дата или время в стандартном формате для СУБД ... 97
create. Дата в формате ISO 8601 ... 98
create. Копирование даты ... 98
create. Относительная дата ... 98
create. Произвольная дата ... 99
now. Текущая дата ... 99
today. Дата на начало текущего дня ... 99
unix-timestamp. Дата и время в Unix-формате ... 99
Поля ... 100
Методы ... 100
int, double. Преобразование даты в число ... 100
gmt-string. Вывод даты в виде строки в формате RFC 822 ... 100
iso-string. Вывод даты в виде строки в формате ISO 8601 ... 101
last-day. Получение последнего дня месяца ... 101
roll. Сдвиг даты ... 101
sql-string. Преобразование даты в вид, стандартный для СУБД ... 102
unix-timestamp. Преобразование даты и времени в Unix-формат ... 102
Статические методы ... 102
calendar. Создание календаря на заданную неделю месяца ... 102
calendar. Создание календаря на заданный месяц ... 103
last-day. Получение последнего дня месяца ... 103
roll. Установка временной зоны по умолчанию ... 104
double, int (классы) 104
Методы ... 104
format. Вывод числа в заданном формате ... 104

inc, dec, mul, div, mod. Простые операции над числами ... 105
int, double, bool. Преобразование объектов в числа или bool ... 105
Статические методы ... 106
sql. Получение числа из базы данных ... 106
env (класс) 106
Статические поля ... 106
fields. Все переменные окружения ... 106
PARSER_VERSION. Получение версии Parser ... 107
Получение значения переменной окружения ... 107
Получение значения поля запроса ... 107
file (класс) 107
Конструкторы ... 108
base64. Декодирование из Base64 ... 108
cgi и exec. Исполнение программы ... 109
create. Создание файла ... 111
load. Загрузка файла с диска или HTTP-сервера ... 111
sql. Загрузка файла с SQL-сервера ... 112
stat. Получение информации о файле ... 113
Поля ... 113
Методы ... 114
base64. Кодирование в Base64 ... 114
crc32. Подсчет контрольной суммы файла ... 115
md5. MD5-отпечаток файла ... 115
save. Сохранение файла на диске ... 115
sql-string. Сохранение файла на SQL-сервере ... 116
Статические методы ... 116
base64. Кодирование в Base64 ... 116
basename. Имя файла без пути ... 116
copy. Копирование файла ... 116
crc32. Подсчет контрольной суммы файла ... 117
delete. Удаление файла с диска ... 117
dirname. Путь к файлу ... 117
find. Поиск файла на диске ... 117
fullpath. Полное имя файла от корня веб-пространства ... 118
justext. Расширение имени файла ... 118
justname. Имя файла без расширения ... 118
list. Получение оглавления каталога ... 119
lock. Эксклюзивное выполнение кода ... 119
md5. MD5-отпечаток файла ... 120
move. Перемещение или переименование файла ... 120
form (класс) 120
Получение значения поля формы ... 121
Статические поля ... 121
elements. Массивы всех полей формы ... 121
fields. Все поля формы ... 122
files. Получение множества файлов ... 122
imap. Получение координат нажатия в ISMAP ... 122
qtail. Получение остатка строки запроса ... 123
tables. Получение множества значений поля ... 123
hash (класс) 124
Конструкторы ... 124
create. Создание пустого хеша и копирование хеша ... 124

sql. Создание хеша на основе выборки из базы данных ... 124
Поля ... 126
Использование хеша вместо таблицы ... 126
Методы ... 126
at. Доступ к элементу хеша по индексу ... 126
contains. Проверка существования ключа ... 127
count. Количество ключей хеша ... 127
delete. Удаление пары «ключ / значение» ... 127
foreach. Перебор элементов хеша ... 127
keys. Список ключей хеша ... 128
rename. Переименовывание ключей хеша ... 129
reverse. Обратный порядок элементов ... 129
select. Отбор элементов ... 129
set. Установка значения по индексу ... 130
sort. Сортировка хеша ... 130
Работа с множествами ... 131
add. Сложение хешей ... 131
intersection. Пересечение хешей ... 131
intersects. Определение наличия пересечения хешей ... 132
sub. Вычитание хешей ... 132
union. Объединение хешей ... 132
hashfile (класс) 133
Конструктор ... 133
open. Открытие или создание ... 133
Чтение ... 134
Запись ... 134
Методы ... 134
cleanup. Удаление устаревших записей ... 134
delete. Удаление пары «ключ / значение» ... 134
delete. Удаление файлов данных с диска ... 134
foreach. Перебор ключей хеша ... 135
hash. Получение обычного хеша ... 135
release. Сохранение изменений и снятие блокировок ... 135
image (класс) 135
Конструкторы ... 135
create. Создание объекта с заданными размерами ... 135
load. Создание объекта на основе графического файла в формате GIF ... 136
measure. Создание объекта на основе существующего графического файла ... 136
Поля ... 137
Методы ... 138
gif. Кодирование объектов класса image в формат GIF ... 138
html. Вывод изображения ... 138
Методы рисования ... 138
Тип и ширина линий ... 139
arc. Рисование дуги ... 139
bar. Рисование закрашенных прямоугольников ... 139
circle. Рисование неокрашенной окружности ... 139
copy. Копирование фрагментов изображений ... 140
fill. Закрашивание одноцветной области изображения ... 140
font. Загрузка файла шрифта для нанесения надписей на изображение ... 141
length. Получение длины надписи в пикселях ... 142
line. Рисование линии на изображении ... 142
pixel. Работа с точками изображения ... 142

polybar. Рисование окрашенных многоугольников по координатам узлов ... 142
polygon. Рисование неокрашенных многоугольников по координатам узлов ... 143
polyline. Рисование ломаных линий по координатам узлов ... 143
rectangle. Рисование незакрашенных прямоугольников ... 144
replace. Замена цвета в области, заданной таблицей координат ... 144
sector. Рисование сектора ... 144
text. Нанесение надписей на изображение ... 145

inet (класс) 145

Статические методы ... 145
aton. Преобразование строки с IP-адресом в число ... 145
hostname. Имя хоста ... 145
ip2name. Определение домена по IP-адресу ... 145
name2ip. Определение IP-адреса домена ... 146
ntoa. Преобразование числа в строку с IP-адресом ... 146

junction (класс) 147

json (класс) 148

Статические методы ... 149
parse. Преобразование JSON-строки в хеш ... 149
string. Преобразование объекта Parser в JSON-строку ... 150

mail (класс) 152

Статические методы ... 153
send. Отправка сообщения по электронной почте ... 153

math (класс) 155

Статические поля ... 155
Статические методы ... 155
abs, sign. Операции со знаком ... 155
convert. Конвертирование из одной системы счисления в другую ... 155
crc32. Подсчет контрольной суммы строки ... 156
crypt. Хеширование паролей ... 156
degrees, radians. Преобразования градусы — радианы ... 158
digest. Криптографическое хеширование ... 158
exp, log, log10. Логарифмические функции ... 158
md5. MD5-отпечаток строки ... 158
pow. Возведение числа в степень ... 159
random. Случайное число ... 159
round, floor, ceiling. Округления ... 159
sha1. Хеш строки по алгоритму SHA1 ... 159
sin, asin, cos, acos, tan, atan, atan2. Тригонометрические функции ... 160
sqrt. Квадратный корень числа ... 160
trunc, frac. Операции с целой/дробной частью числа ... 160
uid64. 64-битный уникальный идентификатор ... 160
uuid. Универсальный уникальный идентификатор версии 4 ... 161
uuid7. Универсальный уникальный идентификатор версии 7 ... 161

memcached (класс) 162

Конструктор ... 163
open. Открытие ... 163
Чтение ... 163
Запись ... 163
Методы ... 164

add. Добавление записи ... 164
clear. Удаление всех данных с сервера ... 164
delete. Удаление записи ... 164
mget. Получение множества значений ... 164
release. Закрытие соединения с сервером ... 164
Параметры соединения ... 164
memory (класс) 165
Статические методы ... 165
auto-compact. Автоматическая сборка мусора ... 165
compact. Сборка мусора ... 165
reflection (класс) 166
Статические методы ... 166
base. Родительский класс объекта ... 166
base_name. Имя родительского класса объекта ... 166
class. Класс объекта ... 166
class_alias. Создание псевдонима класса ... 166
class_by_name. Получение класса по имени ... 166
class_name. Имя класса объекта ... 166
classes. Список классов ... 167
copy. Копирование объекта ... 167
create. Создание объекта ... 167
def. Проверка существования класса ... 167
delete. Удаление поля объекта ... 167
dynamical. Тип вызова метода ... 168
field. Получение значения поля объекта ... 168
fields. Список полей объекта ... 168
fields_reference. Ссылка на поля объекта ... 168
filename. Получение имени файла ... 169
is. Проверка типа ... 169
method. Получение метода объекта ... 169
method_info. Информация о методе ... 170
methods. Список методов класса ... 171
mixin. Дополнение типа ... 171
stack. Стек вызовов методов ... 172
tainting. Преобразования строки ... 172
uid. Уникальный идентификатор объекта ... 173
regex (класс) 174
Конструктор ... 174
create. Создание нового объекта ... 174
Поля ... 174
request (класс) 174
Статические поля ... 175
argv. Аргументы командной строки ... 175
body. Получение текста запроса ... 175
body-charset, post-charset. Получение кодировки пришедшего POST-запроса ... 175
body-file, post-body. Тело содержимого запроса ... 175
charset. Задание кодировки документов на сервере ... 175
document-root. Корень веб-пространства ... 176
headers. Получение заголовков HTTP-запроса ... 176
method. Получение метода HTTP-запроса ... 176
path. Получение пути запроса ... 176
query. Получение параметров строки запроса ... 177
uri. Получение URI запроса ... 177

response (класс) 177

Статические поля ... 177

Заголовки HTTP-ответа ... 177
body. Задание нового тела ответа ... 178
charset. Задание кодировки ответа ... 178
download. Задание нового тела ответа ... 179
headers. Заданные заголовки HTTP-ответа ... 179

Статический метод ... 179

clear. Отмена задания новых заголовков HTTP-ответа ... 179

status (класс) 179

Поля ... 180

memory. Информация о памяти под контролем сборщика мусора ... 180
log-filename. Путь к журналу ошибок ... 181
mode. Режим работы ... 181
pid. Идентификатор процесса ... 181
rusage. Информация о затраченных ресурсах ... 181
tid. Идентификатор потока ... 183

string (класс) 183

Статические методы ... 184

base64. Декодирование из Base64 ... 184
idna. Декодирование из IDNA ... 184
js-unescape. Декодирование, аналогичное функции unescape в JavaScript ... 185
sql. Получение строки из базы данных ... 185
unescape. Декодирование JavaScript- или URI-кодирования ... 186

Методы ... 186

base64. Кодирование в Base64 ... 186
format. Вывод числа в заданном формате ... 187
int, double, bool. Преобразование строки в число или bool ... 187
idna. Кодирование в IDNA ... 188
js-escape. Кодирование, аналогичное функции escape в JavaScript ... 188
left, right. Подстрока слева и справа ... 188
length. Длина строки ... 189
match. Поиск подстроки по шаблону ... 189
match. Замена подстроки, соответствующей шаблону ... 190
mid. Подстрока с заданной позиции ... 190
pos. Получение позиции подстроки ... 191
replace. Замена подстрок в строке ... 191
save. Сохранение строки в файл ... 191
split. Разбиение строки ... 192
trim. Отсечение букв с концов строки ... 193
upper, lower. Преобразование регистра строки ... 193

table (класс) 194

Конструкторы ... 194

create. Создание объекта на основе заданной таблицы ... 194
create. Копирование существующей таблицы ... 194
load. Загрузка таблицы с диска или HTTP-сервера ... 195
sql. Выборка таблицы из базы данных ... 195

Опции формата файла ... 196

Опции копирования и поиска ... 196

Получение содержимого столбца ... 197

Изменение содержимого столбца ... 197

Получение содержимого текущей строки в виде хеша ... 197
Методы ... 198
append. Добавление строки в таблицу ... 198
array. Преобразование таблицы в массив ... 198
cells. Получение значений столбцов текущей строки таблицы ... 199
columns. Получение структуры таблицы ... 199
count. Количество строк в таблице ... 199
csv-string. Преобразование в строку в формате CSV ... 199
delete. Удаление текущей строки ... 200
flip. Транспонирование таблицы ... 200
foreach. Последовательный перебор всех строк таблицы ... 200
hash. Преобразование таблицы в хеш с заданными ключами ... 201
insert. Вставка строки в таблицу ... 203
join. Объединение двух таблиц ... 203
locate. Поиск в таблице ... 203
menu. Последовательный перебор всех строк таблицы ... 204
offset и line. Получение смещения указателя текущей строки ... 204
offset. Смещение указателя текущей строки ... 205
rename. Изменение названия столбца ... 205
save. Сохранение таблицы в файл ... 205
select. Отбор записей ... 206
sort. Сортировка данных таблицы ... 206
void (класс) 207
Статический метод ... 207
sql. Запрос к БД, не возвращающий результата ... 207
xdoc (класс) 208
Конструкторы ... 208
create. Создание документа на основе заданного XML ... 208
create. Создание нового пустого документа ... 208
create. Создание документа на основе файла ... 209
parser://метод/параметр. Чтение XML из произвольного источника ... 209
Параметр создания нового документа. Базовый путь ... 209
Методы ... 210
DOM ... 210
load. Загрузка XML с диска, HTTP-сервера или иного источника ... 210
file. Преобразование документа в объект класса file ... 211
save. Сохранение документа в файл ... 211
string. Преобразование документа в строку ... 211
transform. XSL-преобразование ... 212
Поля ... 212
DOM ... 212
search-namespaces. Хеш пространств имен для поиска ... 213
Параметры преобразования документа в текст ... 213
xnode (класс) 214
Методы ... 215
DOM ... 215
select. XPath-поиск узлов ... 215
selectSingle. XPath-поиск одного узла ... 216
selectString. Вычисление строчного XPath-запроса ... 216
selectNumber. Вычисление числового XPath-запроса ... 217
selectBool. Вычисление логического XPath-запроса ... 217
Поля ... 217

DOM ... 217
Константы ... 218
DOM. nodeType ... 218
Установка и настройка Parser 3 219
Конфигурационный файл ... 220
Конфигурационный метод ... 221
Описание формата файла, описывающего кодировку ... 222
Установка Parser на веб-сервер как CGI ... 223
Установка Parser на веб-сервер Apache как модуля сервера ... 224
Установка Parser на веб-сервер IIS 8.0 или новее ... 225
Подобие mod_rewrite ... 225
Использование Parser в качестве веб-сервера ... 225
Использование Parser в качестве интерпретатора скриптов ... 226
Получение исходных кодов 227
Сборка под Linux и другие Unix-подобные системы ... 228
Сборка под Windows ... 228
Приложение 1. Пути к файлам и каталогам, работа с HTTP-серверами 229
Переменная CLASS_PATH ... 229
Приложение 2. Форматные строки преобразования числа в строку 229
Приложение 3. Формат строки подключения оператора connect 230
Для MySQL ... 230
Для SQLite ... 231
Для ODBC ... 232
Для PostgreSQL ... 233
Для Oracle ... 234
ClientCharset. Параметр подключения — кодировка общения с SQL-сервером ... 235
Приложение 4. Perl-совместимые регулярные выражения 235
Приложение 5. Как правильно назначить имя переменной, функции, классу 237
Приложение 6. Как бороться с ошибками и разбираться в чужом коде 238
Приложение 7. SQL-серверы, работа с IN/OUT-переменными 239

Индекс 240

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 трудно следить за тем, к чему относится та или иная скобка, и эта возможность существенно облегчит разработку. Также поможет в работе и выделение цветом конструкций языка. Читать и редактировать код станет намного проще.

Учебная часть построена в виде уроков, в начале каждого из которых содержится рабочий код. Его легко 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 даст необходимый запас 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.

[3.4.0] — 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 читателей данной книги, это, несомненно: «А что это вообще такое?» Итак, сейчас расскажем. Для начала — несколько предположений.

Первое, очень важное. Читатель уже имеет представление о том, что такое HTML. Если данное сочетание букв еще 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 познакомиться с Parser на практических примерах, поэтому будем считать, что у читателей под рукой уже есть установленный Parser 3. Теория, как известно, без практики мертва. Как установить и настроить программу, подробно рассказано в отдельном разделе.

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

Вроде бы не очень много. Все остальное — это уже наша забота!

Parser...

Parser появился на свет в 1997 году в Студии Артемия Лебедева (artlebedev.ru). Целью его создания было облегчение труда 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. Именно поэтому большинство интернет-проектов студии делаются на Parser. Точнее, на Parser 3 — текущей (третьей) версии языка.

Parser проще в использовании, чем что-либо, созданное для подобных целей. Но эта простота не означает примитивности. Она позволяет использовать Parser и опытным программистам, и тем людям, которые далеки от программирования. Parser дает возможность создавать красивые, полноценные сайты быстро, эффективно и профессионально.

Идея Parser довольно проста. В HTML-страницы внедряются специальные конструкции, обрабатываемые на сервере перед тем, как страницы увидит пользователь. Программа сама доделывает работу по окончательному формированию и оформлению сложного документа. Это похоже на собирание из конструктора, в котором есть готовые модули для всех обычных целей. При необходимости выполнить нестандартные действия легко создать собственные модули. Ничего невозможного нет, при этом все делается просто и быстро.

Parser!

Подведем итоги. Что же в итоге дает разработчику Parser? Он предлагает переменные, циклы, условия и т. д. — в общем, 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 пропадает необходимость повторять одни и те же инструкции по несколько раз, но появляется возможность формирования динамических страниц в зависимости от действий пользователя, работы с базами данных и XML, а также внешними HTTP-серверами, изменения дизайна страниц за считаные минуты. И все это без обычного в подобных случаях сложного программирования.

Страницы формируются из отдельных законченных объектов, а разработчик просто говорит Parser, какие из них, сколько, куда и в какой последовательности поставить. Остальное будет сделано автоматически. При этом сам проект станет логичным и понятным за счет структуризации.

Очень скоро читатель сможет делать все то, что раньше могли позволить себе лишь профессионалы, использующие достаточно сложные языки программирования, которые требуют месяцев (если не лет) изучения и практики.

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

Parser поддерживает JSON, что делает его удобным инструментом для создания серверной части одностраничных приложений (SPA) на JavaScript. С помощью Parser можно быстро реализовать требуемый для работы сайта программный интерфейс (API).

Впрочем, перечислять преимущества можно долго, но, может быть, мы и так уже убедили 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.

Приступаем? Вперед!

Урок 1. Меню навигации

Начнем с самого начала. Итак, мы собираемся сделать сайт (узел, сервер). Первым делом необходимо уяснить, каким образом на сайте будет упорядочена та или иная информация. Сколько будет категорий, подразделов и 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 понятной, легкоузнаваемой, единообразной и удобной 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. Тем, у кого есть опыт создания сайтов, наверняка приходилось сталкиваться с проблемой создания грамотной навигации.

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

Решение следующее. 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 него. 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 создании таблицы вручную необходимо это учитывать. Для таблиц по умолчанию применяется формат tab-delimited.

2) В том же каталоге, где содержится `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 — это... правильно! Он самый!
3) В файл `auto.p` вставим следующий код:

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

<nav>
      ^sections.menu{
            <a href="$sections.uri">$sections.name</a>
      }[|]
</nav>
```

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 все получилось именно так, как описано выше, погружение в мир динамических файлов можно считать состоявшимся. Не за горами использование баз данных для формирования страниц и многое другое.

Однако не будем радоваться раньше времени. Сперва разберемся, что же мы сделали, чтобы добиться такого результата. Предлагаем взглянуть 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**. И это только наше решение, как его назвать. Вполне допустимы имена вида **a_ну_ка_вставь_меню_быстро**. Но читаться такая программа будет хуже, впрочем, кому как понятнее.

Жизненно необходимо давать методам простые, понятные имена. Они должны точно соответствовать тому, что именуемый объект будет хранить и делать. Настоятельно рекомендуется крайне внимательно относиться к именам, чтобы сохранить нервы и время себе, а также всем тем, кому впоследствии придется разбираться в написанном коде. Имена могут быть написаны как латинскими, так и русскими символами, главное — соблюдать единообразие: все или по-русски, или по-английски.

Идем дальше.

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

Это ключевая строка нашего кода. Она достаточно большая, поэтому лучше разобрать ее по частям.

Строка начинается символом **$** (доллар) и следующим сразу за ним именем **sections**. Так в Parser обозначаются переменные. Это надо запомнить. Все просто: видим в тексте `$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 то же самое:

$var[…] – присваиваем

$var – получаем

Подробнее — см. 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 добавлять или удалять из нее строки. Поскольку переменная может хранить любые данные, необходимо указать, что значение, присвоенное переменной, 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. При необходимости легко определить, к какому классу относится автомобиль, поскольку их все 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 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. Так же и в Parser: каждый объект относится к определенному классу, объект класса создается конструктором этого класса и наделяется 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. Это базовая терминология, ее нужно усвоить перед дальнейшим изучением.

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]
```

Подробнее — см. 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:

```
<nav>
        ^sections.menu{
```

```
            <a href="$sections.uri">$sections.name</a>
      }[|]
</nav>
```

Мы формируем HTML-меню, в каждый пункт которого помещаем значения, содержащиеся в полях нашей таблицы **$sections**: **uri** — адрес и **name** — имя. При помощи метода **menu** мы автоматически перебираем все строки таблицы. В первом параметре, переданном в фигурных скобках, задан код, который нужно выполнить для каждой строки таблицы. Во втором параметре, переданном в прямых скобках, задана строка, разделяющая пункты меню. 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 нарушается. Все просто и красиво.

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 объект 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. Теперь 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]]
<nav>
      ^sections.menu{
            ^navigation_item[]
      }[|]
<nav>
```

```
@navigation_item[]
^if($sections.uri eq $request:uri){
      <b>$sections.name</b>
}{
      <a href="$sections.uri">$sections.name</a>
}
```

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

```
^if(условие){код, если условие «истина»}{код, если условие «ложь»}
```

Что она делает, понять несложно. В круглых скобках задается условие, и 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 проверять на равенство две текстовые 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, 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, если нет – ставим.
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. Разобьем их на следующие блоки: **header** – верхняя часть страницы, **navigation** – наше меню, **content** – основной информационный блок и **footer** – нижняя часть страницы. Страницы многих сайтов имеют похожую структуру.

Footer будет для всех страниц одинаковым, **header** – для всех страниц одинаковым по стилю, но с разными текстовыми строками – заголовками страницы, а **content** будет разным у всех страниц и сохранит только общую структуру (вертикальный информационный блок).

Каждая из страниц будет иметь следующую структуру:

header
navigation content
footer

Так же как и в случае с меню, опишем каждый из этих блоков методом (функцией) на Parser. Разберемся подробнее с каждым блоком.

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

```
@footer[]
<footer>
	$now[^date::now[]]
	Powered by Parser3 1997-$now.year
</footer>
```

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 непонятным, нужно вернуться 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 отличающиеся части заголовков страниц (в нашем случае — строку-приветствие).

Дополняем `auto.p` следующим кодом:

```
@header[]
<title>Parser3: ^greeting[]</title>
<style>
	html {
		height: 100%;
	}
	body {
		display: flex;
		flex-direction: column;
		height: 100%;
		margin: 0;
	}
	nav, #content, footer {
		padding: 10px;
	}
	#content {
		flex: 1 100%;
	}
</style>
```

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 с одинаковой структурой. Как это работает?

В самом начале файла `auto.p` определим функцию **@main[]**. Функция с таким названием всегда исполняется первой. Включим в нее вызовы функций, формирующих части страниц.

В начале `auto.p` пишем:

```
@main[]
<html>
	<head>
		^header[]
	</head>
	<body>
	^navigation[]
	<div id="content">
		^content[]
	</div>
	^footer[]
```

```
    </body>
</html>
```

А для получения уникальных заголовков 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 вызовется функция **header**, из которой вызовется 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 будут выполнены функции **navigation**, **content** и **footer** из **main**.

В результате мы получим страницу, у которой будут все необходимые элементы, а в верхней части дополнительно появится наше уникальное приветствие. Переопределяемые функции носят название виртуальных. Из файла `auto.p` мы 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.

Определение функции **content**, так же как и в случае с **greeting**, вставим в страницы:

```
@content[]
Главная страница сайта
```

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 многократно писать одни и те же фрагменты HTML-кода. Привычные HTML-страницы трансформируются приблизительно в следующее (примерное содержание `index.html` файла для главной страницы):

```
@greeting[]
Добро пожаловать!

@content[]
Главная страница сайта
```

Просто и понятно, содержание разложено по методам и легкодоступно для изменения. 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 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, нам 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

@content[]
Страница контактов тестового сайта

@footer[]
<footer>
      Обращайтесь к нам!
<footer>
```

изменит привычный **footer** на обозначенный выше, т. е., если Parser найдет 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.

Ранее мы использовали в нашем коде конструкцию **$request:uri**. *Она отличается по синтаксису от всего того, с чем мы имели дело до этого. Что же это такое? Внешне похоже на* **$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. На техническом языке 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 на следующем уроке. При этом можно сразу же вызывать их также без создания каких-либо объектов с помощью конструкторов. Обращение к статическим полям и вызов статических методов оформляется с помощью двоеточия. 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 99; это статический метод класса* **math**.

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

Подведем итоги второго урока.

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

Что узнали: познакомились с ветвлением кода, научились 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` допустимо определить функцию **`main`**, которая выполняется автоматически. Любая из функций способна вызывать другие функции. 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, 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. Так же 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 мы расскажем, 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`. 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. 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]
```

```
	]
]
<style>
	.calendar th, .calendar td {text-align: center;}
	.calendar .weekday {background: silver;}
	.calendar .week5day, .calendar .week6day {color: red;}
	.calendar .today {font-weight: bold;}
</style>
$now[^date::now[]]
$days[^date:calendar[rus]($now.year;$now.month)]
<table class="calendar">
	<tr>
		<th colspan="7">
			<b>$calendar_locale.month_names.[$now.month]</b>
		</th>
	</tr>
	<tr>
		^for[i](0;6){
			<th class="weekday week${i}day">
				$calendar_locale.day_names.$i
			</th>
		}
	</tr>
	^days.menu{
		<tr>
			^for[i](0;6){
				^if($days.$i){
					<td class="^if($days.$i == $now.day){today}">
						$days.$i
					</td>
				}{
					<td></td>
				}
			}
		</tr>
	}
</table>
```

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

Самая объемная часть кода, начинающаяся с определения **$calendar_locale**, оказалась незнакомой. В приведенной структуре мы определяем какие-то данные для календаря, напоминающие таблицу. То, что определено как **$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 с его номером в году (3 — март), название дня недели — с его порядковым номером, а также связываем шестнадцатеричное 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. Внимание! Мы создали вложенный 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 июня 2023 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 порядковые номера дней недели и календарь на июнь 2023 г.

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 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. Сформулируем задачу подробнее. 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 при обращении к нему как к переменной. Этим мы и воспользуемся:

```
^for[i](0;6){
```

```
        <th class="weekday week${i}day">
                $calendar_locale.day_names.$i
        </th>
}
```

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

week${i}day** — для названия стиля дня недели; если написать **week$iday, Parser поймет это как обращение к переменной **$iday**; фигурными скобками возможно отделить имя переменной от следующих за ним букв;
$calendar_locale.day_names.$i — Для названия дня недели.

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

В этом месте может возникнуть вопрос: «Зачем мы задаем стиль дням недели?» 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 реализовать при помощи цикла **for**, с которым мы только что познакомились:

```
^for[i](0;6){…}
```

Для проверки столбцов на пустоту для вывода столбцов без чисел используем оператор **if**.

```
^if($days.$i){
        …
}{
        <td></td>
}
```

При этом в условии **if** мы ни с чем не сравниваем значение **$days.$i**. Так осуществляется проверка на неравенство нулю.

Parser 3 это условие понимает так:

«Если значение **$days.$i** больше нуля, то выполняется код **{…}**, если нет, то выводится пустая ячейка таблицы серого цвета».

Основная часть работы выполнена. Осталось только выделить текущее число. Решается это использованием еще одного **if**, где в качестве условия задается сравнение текущего значения таблицы **days** с текущим числом (**$days.$i==$now.day**):

```
<td class="^if($days.$i == $now.day){today}">
        $days.$i
</td>
```

Здесь мы проверяем на равенство два числа, поэтому задействуем 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[i](0;6){
            ^if($days.week_day){
                  ^if($month.$week_day==$date.day){ стиль с жирным шрифтом }
                  число
            }{
                  пустой столбец
            }
      }
}
```

Этот пример явно не назовешь простым. В нем используются вложенные друг в друга конструкции. Однако эта демонстрация позволяет оценить возможность комбинирования различных средств Parser для решения конкретной задачи.

Тому, кто хочет убедиться в работоспособности данного модуля, достаточно создать в разделе `/news/` файл `test.html` всего с одной строчкой **^calendar[]**. Осталось лишь обратиться к этому файлу из браузера и полюбоваться на результат 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, необходимый для построения связи одних объектов с другими; в нашем случае в качестве объектов 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; можно обратиться непосредственно 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 комментарием 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!
```

Нужно обязательно комментировать свой код! В идеале человек, читающий его, должен сразу же понимать, о чем идет речь, что из чего следует и т. д. Иначе спустя какое-то время вспомнить, что делает та или иная функция, станет очень сложно даже автору, не говоря уже про остальных.

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. Второй шаг. Переход к работе с БД

Самое главное — не пугаться названия урока даже при отсутствии опыта работы с базами данных. Без них просто невозможно построить гибкий и легконаполняемый 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, советуем обязательно научиться работать с БД, активно используя в проектах. На этом закончим агитацию, будем считать, что мы убедили всех 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 (MariaDB), Postgres, SQLite, Oracle, а также с любой СУБД через драйверы ODBC (в т. ч. MS SQL, MS Access). Поскольку исходные коды Parser 3 являются открытыми, возможно добавление поддержки любых других знакомых разработчику СУБД после создания соответствующего драйвера. При этом работа с ними не требует практически никаких дополнительных знаний собственно Parser. Все, что нужно, — это подключиться к выбранной СУБД и работать, используя SQL в объеме и формате, поддерживаемом СУБД. При передаче SQL-запросов Parser может только экранировать апострофы соответствующей конструкцией в зависимости от СУБД для защиты от уязвимостей, а все остальное передается как есть.

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 этой СУБД? 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.

Теперь определимся с тем, что именно мы 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 неудобно. Мы же предлагаем решение, ориентированное на интернет, — создание на сайте раздела администрирования с формой для ввода новостей прямо из браузера.

Постановка задачи закончена, переходим к ее практическому решению. Далее нам 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 нем — `index.html`, в который пишем следующее:

```
@greeting[]
Администрирование новостей
```

```
@content[]
<h1>Добавление новостей</h1>

$now[^date::now[]]

<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
}
```

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]
```

Структура этой страницы полностью соответствует придуманной нами структуре страниц сайта. Все элементы, как то: приветствие, две части **body**, **footer** и **header**, присутствуют. Теперь нужно вспомнить, откуда на этой странице появятся **header** и **footer**. Конечно, из функции **main** корневого `auto.p`).

Незнакомые конструкции встречаются только в основной части. Разберемся с ней. 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. При необходимости четко отделить имя переменной от следующего за ним символа, скажем «-», как в нашем случае, нужно записать:

```
${имя_переменной}-
```

И в результате получится:

значение_переменной-

Настоятельно рекомендуем изучить Приложение 5, посвященное правилам составления имен.

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

Продолжим. Те, кому уже доводилось работать с формами, знают, 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 мы помним, 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. Вот как мы это сделаем:

```
^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**, поэтому нужно помнить про двоеточие.

То, что здесь не выделено цветом, является командами SQL. Ничего сложного здесь нет. Знакомым с SQL больше ничего и не потребуется объяснять, а тем, кто его почему-то пока не знает, еще раз рекомендуем изучить этот язык. В дальнейшем это многократно пригодится, а потраченное время точно не пропадет даром.

Продемонстрированный вариант взаимодействия с базой данных весьма изящен — Parser обеспечивает прозрачный доступ к СУБД и, за редким исключением, не требует каких-либо дополнительных знаний. При этом 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. Аналогично происходит работа и с другими СУБД.

Теперь у нас есть форма, позволяющая помещать записи в нашу БД. Занесем в нее несколько записей. Теперь нам предстоит их оттуда извлечь, 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: **$days.$i** в коде **if** обнесем ссылками таким образом:

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

@content[]

^calendar[]

$now[^date::now[]]

$day(^if(def $form:day){
	$form:day
}{
	$now.day
})

<h1>Новости за $day число</h1>

^connect[$connect_string]{
	$news[^table::sql{select
		date, header, body
	from
		news
	where
		date='${now.year}-${now.month}-$day'
	}]
	^if($news){
		^news.menu{
			<article>
				$news.date
				<h2>$news.header</h2>
				^taint[as-is][$news.body]
			</article>
		}
```

```
        }{
                За указанный период новостей не найдено.
        }
}
```

Структура обычная. Сверху размещаем 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 выполнения SQL-запроса, при котором возвращается результат. 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**, основанным на результате выборки из БД. 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 можно 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 постороннему человеку, не знающему структуру таблицы, к которой происходит обращение, невозможно понять, какие поля 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 используется текущее число. Далее соединяемся с БД так же, 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:

```
^taint[as-is][$news.body]
```

*В этом месте советуем отвлечься и внимательно прочесть раздел справочника «Внешние и внутренние данные», где подробно описана логика работы операторов **taint** и **untaint**. Это важные операторы, и каждый рано или поздно столкнется с необходимостью их использования. К тому же большую часть обработки данных Parser делает самостоятельно, она не видна на первый взгляд, но понимать логику действий необходимо.*

Зачем здесь нужен оператор **taint**? 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 JavaScript, например, перенаправляющий пользователя на другой сайт. Разрешить использование тегов HTML очень просто: достаточно пометить текст доверенным с помощью оператора **taint**:

```
^taint[as-is][текст новости]
```

В нашем случае используется значение **as-is**, которое означает, что данные будут выведены так, как они хранятся 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: работа над новостным блоком завершена. 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 останется в качестве «домашнего задания». Знаний, полученных на предыдущих уроках, вполне достаточно, чтобы доработать этот пример под свои требования и желания.

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:

- Как на Parser работать с СУБД MySQL.
- Как делать различные SQL-запросы к БД (статический метод **sql** класса **void** и конструктор **sql** класса **table**).
- Для чего используется оператор **taint**.

Что надо запомнить: работа с базами данных в Parser осуществляется легко и понятно, нужно только изучить работу самой СУБД.

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 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 очевидно, 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. С помощью конструктора получим 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 себе 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 соединяться с БД 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. Обратим особое 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 котором будем хранить файл с нашим классом, например **classes**. Далее в этом каталоге создаем файл `gbook.p` (пользовательские файлы предлагаем хранить в файлах с расширением `.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{
		<div class="message">
			<div class="author">
				$messages.author
				^if(def $messages.email){
					$messages.email
				}{
					Нет электронного адреса
				}
			</div>
			<div class="date">
				$messages.date
			</div>
			<div class="body">
				$messages.body
			</div>
		</div>
	}
}{
	<p>Гостевая книга пуста.</p>
}

@show_form[]
<hr/>

$date[^date::now[]]
<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>

@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]
      }{
            <p>Поле "Автор" обязательно для заполнения</p>
      }
}
```

Разберем код подробнее. 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 хранится пользовательский класс, более того, может быть набрано кириллицей.

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. 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 информационной части:

```
@content[]
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 заголовок
@greeting[]
Оставьте свой след
```

Здесь мы уже работаем с объектом пользовательского класса **gbook** как с любым другим объектом: создаем его при помощи конструктора класса и вызываем методы, определенные в этом классе. Решение получилось крайне изящным! Читаемость кода очевидна, и при взгляде на этот фрагмент сразу понятно, что он делает. Все, что относится к гостевой книге, находится в отдельном файле, где описаны все возможные действия с ней. 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 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:

- Как добавляются поля и методы в класс **MAIN**.
- Как создать пользовательский класс и как его использовать.

Что надо запомнить: класс является «высшей формой» структуризации, поэтому необходимо стремиться к выделению в отдельные классы кубиков, из которых будут строиться сайты. Это позволяет достичь максимальной понятности логики работы проектов и обеспечивает 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 наши читатели обладают всеми необходимыми базовыми знаниями для полноценной работы с Parser, остается только совершенствовать их и приобретать опыт.

Урок 6. Работа с XML

```
<?xml version="1.0" encoding="windows-1251" ?>
<article>
    <author id="1" />
    <title>Урок 6. Работаем с XML</title>
    <body>
        <para>Представим, что разработчикам позволено придумывать любые теги
              с любыми атрибутами. То есть они сами могут определять,
              что означает тот или иной выдуманный 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.

А сейчас самое время открыть выбранную вчера в книжном магазине книгу, описывающую XML и XSLT,

и использовать ее как справочни.

Посмотрим, как можно преобразовать приведенную статью 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: **^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 формате HTML (параметр **method** со значением **html**).

Здесь можно задать все параметры, допустимые для тега **<xsl:output … />**.
Рекомендуем также установить режим «без отступов» (задав 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>Представим, что разработчикам позволено придумывать любые теги
с любыми атрибутами. То есть они сами могут определять,
что означает тот или иной выдуманный 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 видим, тег `<author … />` никак не был определен, как следствие, информация об авторе статьи не появилась в выходном HTML. Со временем, когда будет решено, где и как будут показываться данные об авторах, достаточно будет дополнить шаблон — исправлять данные статей не потребуется.

Внимание: если нужно сделать так, чтобы посетители сайта не имели доступа к XML- и XSL-файлам, эти файлы следует хранить вне веб-пространства (`^xdoc::create[/../directory_outside_of_web_space/article.xml]`*) или запретить к ним доступ средствами веб-сервера (пример запрета доступа к p-файлам описан в разделе: «Установка Parser на веб-сервер Apache. CGI-скрипт»).*

Подведем итоги шестого урока.

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

Что узнали:

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

Что надо запомнить: купить и прочесть 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 Parser 3

Переменные

Переменные могут хранить данные следующих типов:

- строка (string);
- число (int или double);
- истина или 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 устанавливать связь 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 хеш и сделать копию другого хеша, см. «Создание пустого хеша и копирование хеша». Хеш запоминает порядок, в котором были добавлены элементы.

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]]
]
```

Массив

Массив позволяет хранить упорядоченный набор значений, доступных по числовым индексам. Создание массива происходит автоматически при таком присваивании переменной значений:

```
$имя[значение1;значение2;...;значениеN]
```

Также можно создать пустой массив и сделать копию другого массива или хеша (см. «copy. Копирование массива или хеша»). Получение значения элемента массива по индексу:

```
$имя.индекс
```

В качестве индекса допустимо использование выражения:

```
$имя.(2+2)
```

Присвоение значения элементу массива:

```
$имя.индекс[значение]
$имя.($i*2)[значение]
```

Если задать в массиве все элементы (от нулевого до последнего), получится обычный массив. Но можно инициализировать только часть элементов, тогда получится разреженный массив (с «дырками»). Эта возможность делает массив полностью совместимым с хешем, содержащим числовые

ключи. Массивы позволяют создавать многомерные структуры, где значениями элементов массива выступают другие массивы или хеши:

```
$имя[значение1;$другой массив; $.ключ1[значение]
$.ключ2[значение];...;значениеN]
```

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: созданный объект доступен в переменной $result, и его можно переопределить для возвращения другого объекта.

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 пользователем классы

Файл в таком формате определяет пользовательский класс:

```
@CLASS
имя_класса

# необязательно
@USE
файл_с_родительским классом

# необязательно
@OPTIONS      [3.3.0]
locals
partial
dynamic или static  [3.4.1]

# 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[]
код
```

...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.

*Примечание: метод **auto** не наследуется, благодаря чему не происходит его множественного выполнения, сначала при инициализации родительского класса, а затем — текущего.* ***[3.4.1]***

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]**.

*Примечание: аналогично можно обращаться к свойствам базового класса — **$BASE:свойство** и **$BASE:свойство[значение]**.* ***[3.4.5]***

С помощью **@OPTIONS** можно определить дополнительное поведение класса. ***[3.3.0]***
*Примечание: пробельные символы в конце **@USE**, **@CLASS**, **@BASE**, **@OPTIONS** игнорируются.* ***[3.4.1]***

Так, указанная опция **locals** автоматически объявит локальными все переменные во всех методах определяемого класса. Если она указана, то для записи в поля 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 редко используемых методов. После создания класса с использованием данной опции возможно лишь добавление методов классу, но не изменение его родительского класса.

С помощью опций **static** и **dynamic** можно задать возможный тип вызова определяемых в файле методов класса. По умолчанию описываемые в файле методы могут вызываться как динамически, так и статически, что может быть не всегда безопасно, и эти опции помогут запретить небезопасные вызовы.

Пример
```
@CLASS
```

```
my

@OPTIONS
dynamic

# вызов $object[^my::create[]] будет допустим, а вызов $var[^my:create[]] будет вызывать исключение
@create[]
Код

# вызов ^object.method1[] будет допустим, а вызов ^my:method1[] будет вызывать исключение
@method1[]
Код

# вызов ^my:method2[] будет допустим, а вызов ^object.method2[] будет вызывать исключение
@static:method2[]
Код
```

Работа с переменными в статических методах

Поиск значения переменной (**$name**) происходит:

- в списке локальных переменных;
- текущем классе или его родителях.

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

Область поиска значения может быть сужена указанием **$self.** или **$класс:**.

Работа с переменными в динамических методах

Поиск значения переменной (**$name**) происходит:

- в списке локальных переменных;
- текущем объекте;
- классе текущего объекта или его родителях.

Запись значения переменной (**$name[value]**) производится в уже имеющуюся локальную переменную, если таковая имеется. В противном случае запись происходит в переменную (поле) в текущем объекте. ***[3.4.5]***

Область поиска значения может быть сужена указанием **$self.** или **$класс:**.

Примечание: следует всячески избегать использования полей класса не из методов класса, кроме простейших случаев! По возможности следует общаться 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

$объект.CLASS_NAME — хранит имя класса объекта.

Пример

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

Выведет 'string'.

Определяемые пользователем методы и операторы

```
@имя[параметры]
тело

@имя[параметры][локальные;переменные]
тело

@static:имя[параметры]  [3.4.1]
тело метода класса, который может быть вызван только статически (подробности)

@имя[*параметры]  [3.4.1]
тело

@имя[параметр1;параметр2;*параметры]  [3.4.1]
тело

@имя[параметр1;.именованный_параметр1;.именованный_параметр2[;...]]  [3.5.0]
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 переменных разделяются точкой с запятой.

Локальные переменные видны в пределах оператора или метода, а также 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 работа будет вестись с полями класса или полями объекта, в зависимости от того, как был вызван определенный разработчиком метод — статически или динамически.

В Parser возможно 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[…]**.

В методах, которые могут принимать произвольное число параметров, все «лишние» параметры доступны в виде хеша с числовыми ключами. Ключ 0 соответствует первому «лишнему» параметру.

В методах, которые могут принимать именованные параметры, все такие параметры передаются в виде хэша последним параметром. Именованные параметры инициализируются значениями соответствующих ключей этого хэша. Допускается не указывать значения отдельных параметров или не передавать хэш вовсе; в таком случае значения параметров будут считаться void.

Пример

```
@main[]
call: ^call[a;b;c]
named: ^named[a; $.o2[value] ]

@call[p;*args][k;v]
p=$p
^args.foreach[k;v]{$k=$v}[^#0A]

@named[p;.o1;.o2][k;v]
p=$p
o1=$o1
o2=$o2
```

Выведет:

```
call: p=a
0=b
1=c

named: p=a
o1=
o2=value
```

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

Внимание: каждый метод должен или возвращать результат через $result во всех вариантах своего выполнения, или не использовать $result вовсе. ***[3.4.0]***

Системная переменная: result, явное определение *[3.4.5]*

Если в методе явно объявить локальную переменную **result**, это укажет Parser, что нужно проигнорировать все пробельные символы в коде (фигурных скобках).

Пример

```
@lookup[table;findcol;findvalue][result]
^if(^table.locate[$findcol;$findvalue]){
      $yes[yes found] $yes
}{
      not found
}
```

Здесь посетителю будет выдано либо значение **yes found**, либо значение **not found**.

Важно: в приведенном примере не будет *выдано ни одного символа перевода строки, пробела или табуляции, написанных в коде.*

*Важно: до версии **3.4.5** попытка написать* **not found** *текстом прямо в теле метода приведет к ошибке.*

Системная переменная: caller

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

Через нее можно:

- считать переменную — **$caller.считать** или записать переменную — **$caller.записать[значение]**, как будто мы находимся в том месте, откуда вызвали описываемый метод или оператор;
- узнать, кто вызвал описываемый метод или оператор, обратившись к **$caller.self** и к **$caller.method** *[3.4.5]*;
- узнать имя вызывающего, обратившись к **$caller.method.name** *[3.4.5]*.

Например, нам нужен оператор, похожий на системный **for**, но чем-то отличающийся от него. Его можно написать самостоятельно, воспользовавшись возможностью менять локальную переменную с переданным именем *в контексте вызова 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** ». Следует обратить внимание на то, 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 переменным класса или объекта в этом случае необходимо использовать **self**.

Передача параметров

Параметры могут передаваться в разных скобках, которые определяют способ обработки параметра:

{код} — выполнение кода параметра происходит при каждом обращении к нему внутри вызванного метода;
(выражение) — вычисление значения выражения в параметре происходит при каждом обращении к нему внутри вызванного метода;
[код] — выполнение кода параметра происходит один раз перед вызовом метода.

Пример на различие скобок, в которых передаются параметры:

```
@main[]
$a(20)
$b(10)
^sum[^eval($a+$b)]
<hr />
^sum($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 указано 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, выдающий значение или метод.

@SET_имя[value]
код, обрабатывающий новое значение $value.

@GET_DEFAULT[]    [3.3.0]
@GET_DEFAULT[имя] [3.3.0]
код, обрабатывающий обращения к несуществующим полям или вызовы несуществующих
методов.

@SET_DEFAULT[имя;значение]    [3.4.1]
код, обрабатывающий запись в несуществующие поля.

@GET[]      [3.3.0]
@GET[тип обращения]      [3.4.0]
код, обрабатывающий обращения к объекту или классу в определенных
контекстах вызова.
```

Можно определить default getter (**@GET_DEFAULT[]**) — метод, который будет вызываться при обращении к несуществующим полям. Имя поля, к которому пытались обратиться, передается

единственным параметром этому методу.

Важно: с этим методом нельзя работать как с обычным «свойством», *при попытке написать* **$DEFAULT** *будет выдано сообщение об ошибке.*

Также можно определить default setter (**@SET_DEFAULT[name;value]**) — метод, который будет вызываться при попытках записи в несуществующие поля. Имя поля, к которому пытались обратиться, и записываемое значение будут переданы этому методу.

У пользовательских классов можно определить специальное свойство **@GET[]**, которое будет вызываться при обращении к классу или объекту этого класса в определенных контекстах вызова, например: в скалярном контексте, в выражении и т. п. Если у этого свойства определен 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 эл. почта
Возьмем человека — хранить удобно дату рождения, а выводить часто нужно возраст. Нужна эл. почта, но можно забыть проверить ее на корректность.

Пусть людьми занимается класс, его свойства «**возраст**» и **email** 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 течением 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 следующий код:

```
@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[$value]){
    ^throw[email.invalid;Некорректная эл. почта: '$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: 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 функциональность 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 две строки, начиная со строки с offset=1, и выводим содержимое
пользовательского объекта:
^c.remove(1;2)
^print[$c]<br />


<br />
Создаем 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 обертки-wrappers
@count[]
^t.count[]

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


# добавляем новую функциональность
@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 символа.

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 шестнадцатеричной записи целого числа)

Примечание: регистр букв не важен.

Логические литералы

В выражениях Parser можно использовать логические литералы
true
false

Пример

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

Литералы в выражениях

Если строка содержит пробелы или начинается с цифры, 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, деление на ноль*
%	Остаток от деления	5	*дает ошибку* number.zerodivision.
\	Целочисленное деление	5	*Операнды преобразуются в тип Int.*
+	Сложение	6	
-	Вычитание	6	
<<	Побитовый сдвиг влево	7	*Операнды*
>>	Побитовый сдвиг вправо	7	*всех битовых операторов*
&	Побитовая операция AND	8	*автоматически*
|	Побитовая операция OR	9	*преобразуются*
!|	Побитовая операция XOR	10	*в тип Int.*
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	
!=	Не равно	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 булево значение «истина / ложь» и отвечает на вопрос «Определен ли объект?» Проверяемым объектом может быть 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 условий «содержит ли переменная код» и «определен ли метод» используется оператор **is**, *а не* **def**.

Замечание: хеш, содержащий только значение по умолчанию, считается определенным ***[3.4.5]***.

in. Проверка наличия документа в каталоге

in "/каталог/"

Возвращает результат «истина / ложь» в зависимости от того, находится ли текущий документ в указанном каталоге.

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

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 так же – 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 переменной кода, т. к. любое обращение к нему вызывает его выполнение. Для такой проверки следует использовать **^reflection:is[]**.*

-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
}
```

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 оставлять комментарии к частям сложного математического выражения. Бывает, что даже разработчику 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 преобразования числа в строку»).

Пример

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

вернет: **16.67**.

Внимание: настоятельно советуем оставлять 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, которая соответствует заданной строке или значению заданного выражения.

if. Выбор одного варианта из двух

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

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

Оператор **if** вычисляет значение логического выражения. Затем, в зависимости от полученного результата, выполняется код для значения «истина» или код для значения «ложь». На код не накладывается никаких ограничений, в том числе внутри него может содержаться еще один или несколько операторов **if**.

```
^if(логическое выражение 1){
      код, если значение выражения 1 «истина»
}(логическое выражение 2){
      код, если значение выражения 2 «истина»
}…(логическое выражение N){
      код, если значение выражения N «истина»
}{
      код, если значение выражения N «ложь»
}     [3.4.1]
```

Оператор вычисляет значение логического выражения. Если выражение истинно, выполняется код для значения «истина». Иначе осуществляется переход к следующему логическому выражению и процесс повторяется.

Внимание: настоятельно советуем оставлять 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. Во избежание зацикливания в Parser встроен механизм обнаружения бесконечных циклов. По умолчанию бесконечным считается цикл, тело которого выполняется более 20 000 раз.

Кроме операторов цикла, для повторения выполнения кода можно использовать метод перебора строк таблицы **menu** и метод перебора элементов хеша **foreach**.

break. Выход из цикла

```
^break[]
^break(условие)    [3.4.5]
```

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

Вызов **^break(условие)** эквивалентен **^if(условие){ ^break[] }**.

continue. Переход к следующей итерации цикла

```
^continue[]
^continue(условие)
```
[3.4.5]

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 **parser.continue**.

Вызов **^continue(условие)** эквивалентен **^if(условие){ ^continue[] }**.

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 не пустым 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 помощью 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]
^while(условие){тело}{разделитель}
```

Оператор **while** повторяет тело цикла, пока условие истинно. Если оно изначально имеет значение «ложь», тело цикла не выполнится ни разу.

Разделитель — строка или код, который вставляется перед каждым не пустым 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 помощью оператора **break** или принудительно закончить текущую итерацию и перейти к следующей с помощью оператора **continue**.

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 />]
```

cache. Сохранение результатов работы кода

```
^cache[файл]
^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 серверы.

Крайне рекомендуется подключать модули (**^use[…]**) изнутри **кода** оператора **cache**, а не делать это статически (**@USE**). По возможности следует работать с базой данных (**^connect[…]**) также внутри **кода** оператора **cache** — это существенно снизит нагрузку на SQL-сервер и повысит производительность 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 — дата и время, до которого хранится результат 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]** — это дает указание Parser обработать ошибку особенным образом: достать из **файла** ранее сохраненный результат работы **кода**, проигнорировав тот факт, что этот результат устарел.

Для принудительного удаления файла-кеша можно использовать:
```
^cache[файл]
```

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 выяснилось,
	что страницу сохранять не нужно: ^cache(0)
}
```

connect. Подключение к базе данных

```
^connect[строка подключения]{код}
```

Оператор **connect** осуществляет подключение к серверу баз данных. После подключения Parser обрабатывает переданный оператору код, который может работать с базой данных в рамках установленного соединения.

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. Это полезно, когда, например, заранее неизвестно, потребуется соединение или нет — можно делать его 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}]
}
```

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 внутри

метода будет вызван **process**, то внутри выполняемого с помощью process кода не будут доступны локальные переменные вызывающего метода).

Этот метод 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 в 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]**
- **$.lineno(**номер строки в файле, откуда взята данная **строка**, — *может быть отрицательным***)**
- **$.replace(true|false)**

Внимание: начиная с версии 3.4.3 в случае создания класса с именем, которое уже существует у ранее загруженного или созданного класса, выдается исключение. Вернуть поведение без исключений можно с помощью указания вновь появившейся опции ***$.replace(true)***.

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 или класса, вызвавшего **include**. Опция **file** позволяет указать имя файла, откуда был загружен код. Если возникнет ошибка, будет отображено это «**имя файла**».

Важно: контекст вызова не включает локальные переменные и параметры вызывающего метода!

Сложный пример

Часто удобно помещать компилируемый код в некоторый метод с именем, вычисляющимся по ходу работы:

```
# это исходный код, следует обратить внимание, на ^^
$source_code[2*2=^^eval(2*2)]
# по ходу работы выясняется, что необходимо создать метод с именем method1
$method_name[method1]
# компилируем исходный код, помещаем его в новый метод
^process{$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 многострочных блоков кода. Закомментированный некорректный код все равно приведет к ошибке интерпретатора.

return. Возврат из метода

^return[]
^return[результат]

При вызове осуществляет принудительное прерывание выполнения метода на Parser, в котором написан код вызова **^return[]**. Результатом работы метода будет то, что успело вывестись до вызова **^return[]**, или текущее значение переменной **$result**. Вызов **^return[результат]** эквивалентен **$result[результат] ^return[]**. Чтобы вернуть пустую строку, нужно использовать **^return{}**.

Пример

```
@main[]
$exit{ -return- ^return[] }
^check[good]{ $exit }
^check[normal]{ $exit }
^check[bad]{ $exit }
-end-

@check[value;exit]
Value: $value ^if($value eq 'bad'){ $exit } -passed-
```

Выведет:

```
Value: good   -passed-
Value: normal -passed-
Value: bad    -return-
```

*Замечание: код вызова **^return[]** написан в методе **@main[]**, поэтому возврат осуществляется из него. Для этого выполнение метода **@check[]** тоже прерывается, поэтому в выводе отсутствует **passed** для значения **bad**.*

sleep. Задержка выполнения программы

^sleep(секунды)

Метод позволяет приостановить выполнение программы на указанное число секунд (допустимы дробные значения).

use. Подключение модулей

^use[файл]
^use[файл;опции] ***[3.4.3]***

$.replace(true)

Оператор позволяет использовать модуль из указанного файла. Если путь к файлу начинается с косой черты «/», то считается, что это путь от корня веб-пространства (а не от корня диска!). В любом другом случае Parser будет искать модуль сначала относительно файла, из которого происходит подключение модуля, а затем — двигаться по путям, определенным в переменной **$CLASS_PATH** в Конфигурационном методе.

*Внимание: до версии **3.4.1** поиск подключаемых модулей производился не относительно файла, в котором написаны **@USE**/**^use[]**, а осуществлялся либо относительно корня веб-пространства, либо по путям, определенным в **$CLASS_PATH**.*

*Внимание: начиная с версии **3.4.3** в случае загрузки класса с именем, которое уже существует у ранее загруженного класса, выдается исключение. Вернуть поведение без исключения можно с помощью указания вновь появившейся опции **$.replace(true)**.*

Для подключения модулей также можно воспользоваться конструкцией:

```
@USE
имя файла 1
имя файла 2
...
```

Разница между этими конструкциями состоит 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]
}
```

@USE начиная с версии ***3.4.5*** вызывает оператор **^use[]**, который, как и любой другой оператор, можно переопределить. Это позволяет реализовать свою логику загрузки модулей. Реализация логики оператора **^use[]** на Parser доступна по ссылке.

Начиная с версии ***3.4.6***, при указании опции **$.main(true)** перед загрузкой файла будут загружены все существующие `auto.p`, начиная от корня веб-пространства до директории с файлом — то есть так же, как это происходит при обработке запроса к странице.

Замечание: попытки подключить уже подключенные ранее модули не приводят к повторному считыванию файлов с диска.

Внешние и внутренние данные

Создавая код на Parser, мы имеем дело с двумя видами данных. Один из них — это все то, что написано самим разработчиком. Второй — данные, получаемые кодом извне, а именно: из форм, переменных окружения, файлов, от SQL-серверов, из cookies и т. п. Все то, что создано разработчиком, 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 **<** 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 разработчиком, считается **clean** («чистым»).
Все данные, которые поступают извне, считаются **tainted** («грязными» или «окрашенными»).

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

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

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

В случае с **$form:field** поступившие **tainted**-данные будут автоматически преобразованы, и некоторые символы заменятся в соответствии с внутренней таблицей замен Parser. Автоматическое преобразование данных произойдет в тот момент, когда эти данные будут выводиться. Так, в случае с помещением данных, поступивших из БД, в переменную **$my_table**, преобразование выполнится тогда, 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, отличным от того, как 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 нашего сайта на вражеский), Parser сам заменит «нежелательные» символы в соответствии со своими правилами. Решение — использование оператора **taint**.

taint. Задание преобразований данных

```
^taint[текст]
^taint[вид преобразования][текст]
```

С помощью механизма автоматических преобразований Parser защищает сайты от вторжения извне и «по умолчанию» делает это хорошо. Этот механизм работает, даже если в коде нет ни одного оператора **taint**. Вмешиваясь в работоспособность этого механизма с помощью данных операторов (особенно используя вид преобразования **as-is**), легко создать уязвимость в разрабатываемом сайте, поэтому делать это нужно внимательно, 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, базы данных, файла, cookies и т. п.).

Данный оператор лишь делает пометки в тексте о виде преобразования, который Parser должен будет произвести **позже**, но не производят его сиюминутно. Сами преобразования Parser выполняет или при выполнении оператора **apply-taint** или 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 в теле страницы, — **^taint[optimized-as-is][написанное разработчиком]**.

Автоматические преобразования защищают от небезопасных внешних данных. Например, если при составлении SQL-запроса написать (опять же без использования **taint**):
^string:sql{SELECT name FROM table WHERE uid = '$form:uid'},
то злоумышленник не сможет выполнить SQL injection, передав в качестве параметра, например, «**?uid=' OR 1=1 OR '**», т. к. Parser перед выдачей SQL-серверу текста запроса экранирует в пришедшем от пользователя **$form:uid** одинарные кавычки.

Текст, написанный разработчиком в теле страниц, также подвергается автоматическому преобразованию. В нем Parser выполняет оптимизацию пробельных символов (включая пробел, табуляцию, символ перевода строки). Идущие подряд перечисленные символы заменяются только одним, который встречается в коде первым. То есть при вставке в текст страницы нескольких 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 необходимо 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 />
```

Из данного примера видно, что, несмотря на сообщение об эквивалентности строк, при выводе их в браузер результат различен: «чистая» строка выводится без преобразований, а в «грязной» строке символы < и > заменены **<** и **>** соответственно.

Пример

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

В результате данные, хранящиеся в переменной **town**, будут помечены как нуждающиеся в преобразовании к типу URI и позже, при выводе в браузер, кириллические буквы будут заменены шестнадцатеричными кодами 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 />
Указано: '$form:field'
^connect[$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 **taint** 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**.

Обратим внимание на то, что при сохранении данных в БД в административном интерфейсе также не требуется писать **taint** 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. Нужно иметь 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, помещенных 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**, т. к. требуется, чтобы теги в тексте новости, помещенные 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, полученные 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**, чтобы встретившиеся в данных теги не «поломали» страницу и чтобы отключить оптимизацию пробельных символов.

В примерах выше можно заметить, что нам пришлось использовать оператор **taint** лишь трижды: один раз для того, чтобы разрешить отображать теги в тексте из БД, помещенном 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 **taint**, и Parser сам все сделал правильно.

В подавляющем большинстве случаев использовать оператор **taint** не нужно!

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

as-is
file-spec
HTTP-header
mail-header
uri
sql
js
json ***[3.4.1]***
parser-code ***[3.4.0]***
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: **_ - . "** преобразуются 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 символ **'** меняется на **''** – для Postgres символы **'** и **** предваряются символом **** – для MySQL символы **' "** и **** предваряются символом ****, символы с кодами **0x00 0x0A 0x0D** преобразуются, соответственно, в **\0 \n \r** Для выполнения данного преобразования необходимо, чтобы код, в результате работы которого преобразование должно выполниться, находился внутри оператора **^connect[]{}**
js	Символ **"** преобразуется в **\"** Символ **'** преобразуется в **\'** Символ **** преобразуется в **** Символ конца строки преобразуется в **\n** Символ с кодом 0xFF предваряется символом ****
json	Символы **" \ /** предваряются символом **** Символ конца строки преобразуется в **\n** Символ табуляции преобразуется в **\t** Служебные символы с кодами **0x08 0x0C 0x0D** преобразуются в **\b \f \r** в случае вывода не в UTF-8 все unicode-символы преобразуются в **\uXXXX**
parser-code	Служебные символы предваряются символом **^**
regex	Символы **\ ^ $. [ ] | ( ) ? * + { } -** предваряются символом ****
xml	Символ **&** преобразуется в **&** Символ **>** преобразуется в **>** Символ **<** преобразуется в **<** Символ **"** преобразуется в **"** Символ **'** преобразуется в **'**
html	Символ **&** преобразуется в **&** Символ **>** преобразуется в **>** Символ **<** преобразуется в **<** Символ **"** преобразуется в **"** Символ **'** преобразуется в **'** ***[3.5.0]***
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 разработчик пишет:

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

Parser выполняет...

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

Аналогично при задании HTTP-заголовков и заголовков писем происходят преобразования **HTTP-header** и **mail-header** соответственно. А при DOM-операциях применяется таблица преобразований **file-spec** (см. выше) и таблица преобразований **xml**.

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**

untaint. Задание преобразований данных

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

Оператор **untaint** выполняет переданный ему **код** и помечает как нуждающиеся в преобразовании заданного **вида** только неопределенно «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 неопределенно «грязные» части результата выполнения кода как **as-is**.

Данный оператор лишь делает пометки в тексте о виде преобразования, которое Parser должен будет произвести **позже**, но не производит его сиюминутно. Сами преобразования Parser выполняет или при выполнении оператора **apply-taint**, или при выдаче текста в браузер, перед выдачей SQL-серверу, при сохранении в файл, при отправке письма и т. п.

В некоторых случаях результаты работы **^taint[вид преобразования][текст]** и **^untaint[вид преобразования]{текст}** одинаковые: это происходит тогда, когда весь обрабатываемый текст является неопределенно «грязным» (например, **$form:field**). Однако нужно быть внимательным: применение к неопределенно «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, как набранный разработчиком, так и пришедший извне}**

Это означает, что, если написать в теле страницы **$form:field** (без всяких **taint/untaint**), то даже при обращении к ней с параметром «**?field=</html>**», это не «поломает» страницу из-за досрочно выведенного в нее закрывающего тега **</html>**, т. к. содержимое **$form:field** неопределенно «грязное», и поэтому к нему будет применено автоматическое преобразование **optimized-html**, с помощью которого символы < и > будут заменены **<** и **>** соответственно.

Пример

```
<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]**.

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.

Может возникнуть вопрос, для чего вообще нужен оператор **untaint**. Продемонстрируем пару вариантов его практического применения.

Во-первых, иногда его использование позволяет уменьшить количество операторов **taint** в коде, например, при выводе данных в форму, содержащую много полей, и при необходимости отключить оптимизацию пробельных символов. В этом случае вместо того, чтобы писать **^taint[html][...]** вокруг вывода содержимого каждой textarea, 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, поэтому вокруг всего кода нужно написать **^untaint[optimized-xml]{...}** и расслабиться :)

apply-taint. Применение преобразований данных

^apply-taint[текст] ***[3.4.1]***
^apply-taint[вид преобразования][текст] ***[3.4.1]***

Оператор **apply-taint** выполняет сиюминутное преобразование всех фрагментов в строке. Неопределенно «грязные» фрагменты преобразуются в указанный вид преобразования (по умолчанию **as-is**).

Пример

```
Пример, иллюстрирующий разницу между ^taint и ^untaint.<br />
$s[?  ^taint[?]   ^taint[uri][?]    ^taint[file-spec][?]]
<pre>^apply-taint[uri][$s]
^apply-taint[uri][^taint[as-is][$s]]
^apply-taint[uri][^untaint{$s}]
^apply-taint[uri][^untaint[uri]{$s}]</pre>

Выведет:
?    %3F    %3F    _3F
?    ?    ?    ?
?    ?    %3F    _3F
?    %3F    %3F    _3F
```

Обработка ошибок

Человек несовершенен. Нужно быть готовым к тому, что вместо ожидаемого на экране 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 (выше 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, а также выдаются результаты его работы. Помимо этого, ошибка записывается в журнал ошибок веб-сервера.

Однако часто желательно перехватить возникшую ошибку и сделать нечто полезное вместо ошибочного кода. Допустим, нужно проверить на правильность XML-код, полученный из ненадежного источника. Здесь прерывание обработки страницы совершенно ни к чему, наоборот — разработчик ожидает ошибку определенного типа и хочет ее обработать. Parser с радостью идет навстречу и предоставляет для этого мощный инструмент: оператор **try**.

При сложной обработке данных часто выясняется, что имеет место ошибка в методе, вызванном из другого, а тем, в свою очередь, из третьего. Как в такой ситуации просто сообщить и обработать ошибку? Следует использовать оператор **throw**, чтобы сообщить об ошибке, и обрабатывать ошибку на верхнем уровне — и тогда ее не придется исправлять 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 переданная ошибка уже обработана

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**, ему передается информация об ошибке, стек вызовов, приведших к ошибке, и выдаются результаты его работы. Помимо этого, 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 нужно перехватывать 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 строке $exception.lineno.
      }
}
```

Результатом работы примера будет

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

Обращаем внимание на то, что посетители сайтов не должны видеть 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, а также выдаются результаты работы метода. Помимо этого, ошибка записывается в журнал ошибок.

Хорошим тоном является оформление сообщения об ошибке в общем дизайне сайта, а также проверка и скрытие технических подробностей от посетителей.

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

Тип	Пример возникновения	Описание
parser.compile	^test[}	Ошибка компиляции кода: непарная скобка и т. п.
parser.runtime	^if(0).	Методу передано неправильное количество параметров, параметры неверных 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 нечисловых данных в числа.
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(100000;1;1)	Дата выходит за границы диапазона.
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 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 найден или временно недоступен.
smtp.execute		Ошибка отправки письма по SMTP-протоколу.
email.format		Ошибка в адресе эл. почты: 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 (неуспешное выполнение запроса).
http.timeout		Загрузка документа с HTTP-сервера не завершилась в отведенное для нее время.
curl.host	^curl:load[$.url[http://notfound/there]]	Сервер не найден.
curl.connect	^curl:load[$.url[http://not_accepting/there]]	Сервер найден, но не принимает соединение.
curl.status	^curl:load[$.url[http://ok/there]]	Сервер выдал ответ со статусом, не равным 200 (неуспешное выполнение запроса).
curl.ssl	^curl:load[$.url[https://not_accepting/there]]	Сервер найден, но не принимает соединение по причине ошибок с сертификатом.
curl.timeout		Загрузка документа с сервера не завершилась в отведенное для нее время.
curl.fail		Другая ошибка при общении

Операторы, определяемые пользователем

Иногда оказывается, что каких-то операторов в языке не хватает. Parser позволяет определить собственные операторы, которые затем можно будет использовать наравне с системными. Операторами в Parser считаются методы класса **MAIN**. Добавляя новые методы в этот класс, мы расширяем базовый набор операторов.

*Внимание! При описании оператора можно использовать и не локальные переменные, при этом происходит чтение и запись 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.

Внимание! 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
Поместим этот код:

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

в файл `operators.p`, в корень сайта.

Там, где необходимы дополнительные операторы, нужно подключить этот модуль. Например, в корневом `auto.p` напишем:

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

Теперь не только на любой странице, но, главное, в любом пользовательском классе можно будет воспользоваться конструкцией

```
^default[$form:name;Аноним]
```

Подробности — в разделе «Определяемые пользователем методы и операторы».

Кодировки

Уверены, наличие разных кодировок доставит разработчикам 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 в документах на сервере, надо задать в поле **$request:charset**.

Желаемую кодировку результата — в **$response:charset**.
Сделать это необходимо в одном из **auto**-методов.

Рекомендуем задавать кодировку результата в HTTP-заголовке **content-type**, чтобы браузер знал о ней и посетителям сайта не нужно было переключать ее вручную.

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

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

При работе с базами данных необходимо задать кодировку, в которой следует общаться с SQL-сервером, см. Приложение 3. Формат строки подключения оператора connect.

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 юникод 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 шестнадцатеричный код данного символа.* ***[3.1.4]***

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

Класс MAIN, обработка запроса

Parser обрабатывает запрошенный документ описанным ниже 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 разработчик получает возможность «дополнительной полировки» результата работы написанного кода.

Простой пример
Добавление такого определения, скажем, в файл `auto.p` в корне веб-пространства сайта:

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

приведет к замене одних слов другими в результатах обработки всех страниц.Важно не забыть проверить тип **body**, ведь там может быть файл.

array (класс)

Класс предназначен для работы с обычными или разреженными массивами. Массив считается определенным (**def**), если в нем есть хотя бы один инициализированный элемент. Числовым значением массива является количество инициализированных элементов (значение, возвращаемое методом **^массив.count[]**).

Конструкторы

Иногда массив удобнее создавать не с помощью конструктора, а так, как это описано в разделе «Конструкции языка Parser. Массив».

create. Создание массива с заданными значениями или пустого массива

```
^array::create[элемент1;элемент2;...]
^array::create[]
```

Если указаны элементы, они будут добавлены в массив. Если параметры не заданы, будет создан пустой массив. В качестве элементов используются любые значения: числа, строки, массивы или хеши.

Пример создания массива с элементами

```
$a[^array::create[M](20)[N]]
```

В этом случае массив **$a** будет содержать элементы с индексами 0, 1 и 2 со значениями M, 20 и N соответственно.

Пустой массив, создаваемый конструктором без параметров, нужен в ситуации, когда необходимо динамически наполнить массив данными, например:

```
$dyn[^array::create[]]
^for[i](1;10){
      $dyn.$i[$value]
}
```

Перед выполнением **for** мы определили, что именно наполняем.

Если предполагается изменение содержимого массива, но необходимо сохранить исходные значения, то это можно сделать с помощью конструктора копирования:

```
$dyn_copy[^array::copy[$dyn]]
```

copy. Копирование массива или хеша

```
^array::copy[существующий массив или хеш]
```

Конструктор создает копию указанного массива или хеша. При копировании хеша создается массив, в котором элементы размещены по числовым ключам хеша.

Пример копирования хеша

```
$hash[
	$.0[яблоко] $.2[банан]
]
$array[^array::copy[$hash]]
```

В результате массив **$array** будет содержать элементы с индексами 0 (яблоко) и 2 (банан).

sql. Создание массива на основе выборки из базы данных

```
^array::sql{запрос}
^array::sql{запрос}[$.sparse(false|true) $.limit(n) $.offset(n)
$.distinct(true|false) $.bind[variables hash] $.type[hash|string|table]]
```

Конструктор на основе результата SQL-запроса создает по умолчанию обычный массив. Если задан параметр **$.sparse(true)**, создается сразреженный массив, в котором индексы элементов совпадают со значениями первого столбца выборки. По умолчанию каждый элемент массива — это хеш, где имена столбцов используются в качестве ключей, а соответствующие им значения — это данные столбцов.

Дополнительные параметры конструктора:

$.sparse(false|true)	**false** или **0** = создать обычный массив. Значения строк выборки пос (по умолчанию) **true** или **1** = создать разреженный массив. Первая колонка данных д будут размещены значения (аналогично ^hash::sql{})
$.limit(n)	получить только **n** записей
$.offset(n)	отбросить первые **n** записей выборки
$.bind[hash]	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.
$.type[hash/string/table]	**hash** = значение каждого элемента — хеш (по умолчанию); **string** = значение каждого элемента — строка, при этом нужно указа **$.sparse(true)**) столбца в SQL-запросе; **table** = значение каждого элемента — таблица со всеми колонками р

По умолчанию наличие в ключевом столбце одинаковых значений считается ошибкой. Если наличие дубликатов допустимо, следует задать опцию **$.distinct(true)**. В этом случае в качестве данных для каждого ключа используется первая встреченная строка, последующие строки с тем же ключом игнорируются без ошибки. А если задать **$.type[table]**, для каждого ключа формируется таблица со всеми записями, имеющими этот ключ.

Пример array of hash

В БД содержится таблица `test_table`:

pet	*food*	*aggressive*
cat	milk	very
dog	bone	never

Выполнение кода:

```
^connect[строка подключения]{
	$array_of_hash[^array::sql{
		select
			pet,
			food,
			aggressive
		from
			test_table
	}]
```

```
}
```

даст массив такой структуры:

```
$array_of_hash[
            $.pet[cat]
            $.food[milk]
            $.aggressive[very]
;
            $.pet[dog]
            $.food[bone]
            $.aggressive[never]
]
```

из которого можно извлекать информацию, например, так:

```
$animal[$array_of_hash.0]
$animal.pet любит $animal.food
```

Пример array of table

В БД содержится таблица `students`

```
name  subject     grade
Mike  Math  5
Alex  Math  4
Ivan  History     5
```

Выполнение кода:

```
^connect[строка подключения]{
      $students[^array::sql{select grade, name, subject from students}[
$.sparse(1) $.distinct(true) $.type[table] ]]
}
```

даст разреженный массив такой структуры:

```
$students[^array::create[]]
$students.4[^table::create{grade    name  subject
4     Alex  Math}]
$students.5[^table::create{grade    name  subject
5     Make  Math
5     Ivan  History}]
```

из которого можно извлекать информацию, например, так:

```
Оценок отлично: ^students.5.count[]
```

Поля

В качестве поля массива выступает числовой индекс, используемый для получения значения элемента по этому индексу:

```
$массив.индекс
```

Такая запись возвратит значение, находящееся по заданному индексу. В качестве индекса допустимо использование выражения:

```
$массив.(2+2)
```

Присваивание значения по индексу добавит или обновит элемент в массиве:

```
$массив.индекс[значение]
$массив.($i*2)[значение]
```

Методы

add. Добавление элементов из другого массива или хеша с перезаписью

```
^массив.add[другой_массив_или_хеш]
```

Метод добавляет элементы из переданного массива или хеша в текущий массив. Если имеются элементы с одинаковыми индексами, то значениями из переданного массива или хеша будут перезаписаны значения в текущем массиве.

Пример

```
$data[name;age;gender]
$more[phone;address]
^data.add[$more]
^data.add[ $.4[email] ]
```

Новое содержание массива **$data**:

```
$man[phone;address;gender;<дырка>;email]
```

append. Добавление элементов в конец массива

```
^массив.append[элемент1;элемент2;...]
```

Метод добавляет указанные элементы в конец массива. Количество добавляемых элементов соответствует количеству переданных параметров.

Пример

```
$data[name;age;gender]
^data.append[phone;address]
```

Новое содержание массива **$data**:

```
$man[name;age;gender;phone;address]
```

at. Доступ к элементу массива по порядковому номеру

```
^массив._at(число|-число)
^массив._at[first|last]
^массив.at(число|-число)
^массив.at[first|last]
^массив.at(число|-число)[key|value|hash]
^массив.at[first|last;key|value|hash]
```

Метод возвращает значение элемента массива по указанному порядковому номеру. Здесь порядковый номер означает позицию элемента в массиве, при этом неинициализированные элементы («дырки») пропускаются. Нумерация начинается с 0. Отрицательное число означает отсчет с конца массива, где -1 — последний элемент.

Также можно использовать строки first или last для получения первого или последнего элемента массива соответственно.

Опциональный второй параметр определяет возвращаемый результат:

value — вернется значение элемента (по умолчанию);
key — вернется ключ элемента;
hash — вернется хеш из одного элемента.

compact. Удаление неинициализированных элементов

```
^массив.compact[]
^массив.compact[undef]
```

Метод удаляет из массива неинициализированные элементы («дырки»), сдвигая последующие элементы влево и уплотняя массив. Вызов без параметров (**^массив.compact[]**) удаляет только «дырки». Если указан параметр **undef** (**^массив.compact[undef]**), метод дополнительно удаляет элементы, содержащие пустые значения (например, пустые строки, пустые хеши и т. д.).

contains. Проверка существования элемента по индексу

```
^массив.contains(индекс)
```

Метод возвращает булево значение **«истина»**, если в массиве инициализирован элемент по заданному индексу, и **«ложь»** в противном случае.

Пример

```
^if(^data.contains(99)){
      По индексу 99 есть данные.
}
```

count. Количество элементов массива

```
^массив.count[]
^массив.count[all]
```

Метод возвращает количество инициализированных элементов в массиве. Если указан параметр all, метод вернет общее количество элементов, включая неинициализированные («дырки»).

Пример

```
$man[Вася;22;m]
^man.count[]
```

Вернет: **3**.

В выражениях числовое значение хеша равно количеству ключей:

```
^if($man > 2){больше}
```

delete. Удаление элемента массива

```
^массив.delete(индекс)
^массив.delete[]
```

Метод удаляет элемент массива, находящийся по указанному индексу. После удаления на месте элемента остается «дырка» (неинициализированный элемент). При вызове без параметра удаляются все элементы массива.

Пример

```
$data[name;age;gender]
^data.delete(1)
```

Новое содержание массива **$data**:

```
$data[age;<дырка>;gender]
```

for. Перебор всех элементов массива

```
^массив.for[индекс;значение]{тело}
^массив.for[индекс;значение]{тело}[разделитель]
^массив.for[индекс;значение]{тело}{разделитель}
```

Метод перебирает все элементы массива, включая неинициализированные («дырки»). Он аналогичен методу **foreach**, но итерируется по всем индексам массива.

Пример

```
$man[Вася;22;m]
^man.delete(1)

^man.for[key;value]{
        $key=$value
}[<br />]
```

Выведет на экран:

```
0=Вася
1=
2=m
```

foreach. Перебор элементов массива

```
^массив.foreach[индекс;значение]{тело}
^массив.foreach[индекс;значение]{тело}[разделитель]
^массив.foreach[индекс;значение]{тело}{разделитель}
```

Метод перебирает все инициализированные элементы массива, передавая в тело цикла индекс и значение каждого элемента. Неинициализированные элементы («дырки») пропускаются.
Параметры метода:

индекс —имя переменной, в которую будет помещен индекс текущего элемента (может быть пустым);
значение —имя переменной, в которую будет помещено значение текущего элемента (может быть пустым);
тело —код, выполняемый для каждого элемента массива;
разделитель —код, который вставляется перед каждым непустым и не первым телом цикла.

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 помощью оператора **break** или принудительно закончить текущую итерацию и перейти к следующей с помощью оператора **continue**.
Метод аналогичен методу **foreach** класса hash, но работает с массивами и их числовыми индексами.

Пример

```
$man[Вася;22;m]
^man.delete(1)

^man.foreach[key;value]{
        $key=$value
}[<br />]
```

Выведет на экран:

```
0=Вася
2=m
```

insert. Вставка элементов в указанную позицию массива

```
^массив.insert(индекс)[элемент1;элемент2;...]
```

Метод вставляет элементы в массив, начиная с заданного индекса. Начиная с этого индекса, элементы сдвигаются вправо, чтобы освободить место для новых элементов.

Пример

```
$data[name;age;gender]
^data.insert(1)[phone;address]
```

Новое содержание массива **$data**:

```
$man[name;phone;address;age;gender]
```

join. Добавление элементов другого массива или хеша

```
^массив.join[другой_массив_или_хеш]
```

Метод добавляет инициализированные элементы переданного массива или значения элементов хеша в конец массива. Индексы элементов не учитываются; элементы добавляются в порядке их следования.

Пример

```
$data[name;age;gender]
^data.join[ $.a[phone] $.0[address] ]
```

Новое содержание массива **$data**:

```
$man[name;age;gender;phone;address]
```

keys. Список индексов массива

```
^массив.keys[]
^массив.keys[имя столбца]
```

Метод возвращает таблицу (объект класса **table**), содержащую единственный столбец, где перечислены все определенные индексы массива в порядке возрастания. Имя столбца — **key** или переданное **имя столбца**.

left. Получение первых n элементов массива

```
^массив.left(n)
```

Метод возвращает новый массив, содержащий первые n инициализированных элементов текущего массива. Неинициализированные элементы («дырки») пропускаются при выборе элементов.

Пример

```
$data[name;age;gender]
$part[^data.left(2)]
```

Содержание массива **$part**:

```
$part[name;age]
```

mid. Получение диапазона элементов массива

^массив.mid(n;m)

Метод возвращает новый массив, содержащий m инициализированных элементов, начиная с позиции n. Позиции считаются по определенным элементам, неинициализированные элементы («дырки») пропускаются при подсчете.

Пример

```
$data[name;age;gender]
$part[^data.mid(1;1)]
```

Содержание массива **$part**:

```
$part[age]
```

pop. Удаление и возврат последнего элемента массива

^массив.pop[]

Метод удаляет последний инициализированный элемент массива и возвращает его значение.

Пример

```
$data[name;age;gender]
$last[^data.pop[]]
```

Новое содержание массива **$data**:

```
$man[name;age]
```

Значение переменной **$last**:

```
gender
```

push. Добавление элемента в конец массива

^массив.push[значение]

Метод добавляет указанный элемент в конец массива. Этот метод эквивалентен методу append и может использоваться для последовательного добавления элементов в массив.

Пример

```
$data[name;age;gender]
^data.push[phone]
```

Новое содержание массива **$data**:

```
$man[name;age;gender;phone]
```

remove. Удаление элемента со сдвигом

^массив.remove(индекс)

Метод удаляет элемент массива, находящийся по указанному индексу. Элементы, следующие за удаленным, сдвигаются влево, заполняя образовавшееся место.

Пример

```
$data[name;age;gender]
^data.remove(1)
```

Новое содержание массива **$data**:

```
$data[age;gender]
```

reverse. Обратный порядок элементов

```
^массив.reverse[]
```

Метод возвращает новый массив, в котором инициализированные элементы идут в порядке, обратном порядку их расположения в текущем массиве.

Пример

```
$man[Вася;22;m]
$man[^man.reverse[]]
^man.foreach[key;value]{
        $key=$value
}[<br />]
```

Выведет на экран:

```
0=m
1=22
2=Вася
```

right. Получение последних n элементов массива

```
^массив.right(n)
```

Метод возвращает новый массив, содержащий последние n инициализированных элементов текущего массива. Неинициализированные элементы («дырки») пропускаются при выборе элементов

Пример

```
$data[name;age;gender]
$part[^data.right(2)]
```

Содержание массива **$part**:

```
$part[age;gender]
```

select. Отбор элементов

```
^массив.select[ключ;значение](критерий_отбора)
^массив.select[ключ;значение](критерий_отбора)[опции]
```

Метод последовательно перебирает все определенные элементы массива, применяя к ним выражение **критерий_отбора**. Элементы, подпавшие под заданный **критерий** (логическое выражение было истинно), помещаются в результат, которым является новый массив.

Можно задать хеш опций:

$.limit(максимум)	максимальное число элементов, которые можно отобрать;
$.reverse(false|true)	**true** = перебирать элементы в обратном порядке.

Пример

```
$men[ $.name[Serge] $.age(26); $.name[Alex] $.age(20); $.name[Misha] $.age(29);
$.name[Denis] $.age(30)]

$thoseAbove20[^men.select[;m]($m.age > 20)[ $.limit(2) ]]
```

В **$thoseAbove20** попадут элементы **Serge** и **Misha**.

set. Установка значения элемента массива

```
^массив.set[first|last][значение]
^массив.set(число)[значение]
```

Метод присваивает **значение** существующему элементу массива по указанному порядковому номеру:

first – устанавливает значение первого инициализированного элемента массива;
last – устанавливает значение последнего инициализированного элемента массива;
число – порядковый номер элемента, которому будет присвоено значение; неинициализированные элементы («дырки») пропускаются при подсчете.

sort. Сортировка массива

```
^иассив.sort[ключ;значение]{функция_сортировки_по_строке}
^иассив.sort[ключ;значение]{функция_сортировки_по_строке}[направление_сортировки]
^иассив.sort[ключ;значение](функция_сортировки_по_числу)
^иассив.sort[ключ;значение](функция_сортировки_по_числу)[направление_сортировки]
```

Метод осуществляет сортировку определенных элементов в массиве по указанной функции.

Функция сортировки – произвольная функция, по текущему значению которой принимается решение о положении поля в отсортированном массиве. Значением функции может быть строка (значения сравниваются в лексикографическом порядке) или число (значения сравниваются как действительные числа).

Направление сортировки – параметр, задающий направление сортировки. Принимает значения:
desc – по убыванию;
asc – по возрастанию.
По умолчанию используется сортировка по возрастанию.

Пример

```
$men[ $.name[Serge] $.age(26); $.name[Alex] $.age(20); $.name[Misha] $.age(29)]
^men.sort[;m]{$m.name}
^men.foreach[;m]{
    $m.name: $m.age
}[<br />]
```

В результате записи массива **$men** будут отсортированы по строке с именем:

```
Alex: 20
Misha: 29
Serge: 26
```

А можно отсортировать записи хеша по числу прожитых лет по убыванию (**desc**), если изменить в примере вызов **sort** на такой:

```
^men.sort[;m]($m.age)[desc]
```

...получится...

```
Misha: 29
Serge: 26
Alex: 20
```

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).
Для этого нужно добавить такую строку в `/etc/inetd.conf` и перезапустить `inetd`:
```
nntp stream tcp nowait учетная_запись /путь/к/parser3 /путь/к/parser3
/путь/к/nntp.p
```

В скрипте `nntp.p` надо описать NNTP-сервер. Это 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.

Примечание: записанные значения доступны для чтения сразу.

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

Запись

```
$cookie:имя[значение]

$cookie:имя[
	$.value[значение]
	...необязательные модификаторы...
]
```

Сохраняет значение в **cookie** с указанным именем. По умолчанию указанное значение сохраняется 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* cookies *отключены или блокируются файерволом).*

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 будут сохраняться, а уничтожатся с закрытием окна браузера);
$.expires[$date] — задает дату и время, до которой будут 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;
$.HTTPonly(true) — если указан ключ с **bool**-значением, то будет сформирован **HTTP**-заголовок в котором у **cookie** этот параметр указан без значения; это может использоваться, например, для задания параметров **HTTPonly** или **secure**.

Пример

```
$cookie:user[Петя]
```

Создаст **cookie** с именем **user** и запишет туда значение «**Петя**». **Cookie** будут храниться на диске посетителя 90 дней.

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.

curl (класс)

Класс предназначен для работы с серверами по протоколам HTTP и HTTPS с использованием библиотеки **libcurl**.

Статические методы

info. Информация о последнем запросе

```
^curl:info[название]
^curl:info[]
```

Метод возвращает информацию о последнем запросе. Результатом является либо конкретное значение, либо хеш со всеми значениями.

Поддерживаемые значения параметра **название**:

Название	Тип	Аналог в libcurl	Описание
appconnect_time	double	CURLINFO_APPCONNECT_TIME	Время до установки SSL-соединен
connect_time	double	CURLINFO_CONNECT_TIME	Время до установки соединения.
content_length_download	double	CURLINFO_CONTENT_LENGTH_DOWNLOAD	Значение заголовка Content-le полученных данных.
content_length_upload	double	CURLINFO_CONTENT_LENGTH_UPLOAD	Значение заголовка Content-le переданных данных.
content_type	string	CURLINFO_CONTENT_TYPE	Значение заголовка Content-ty
effective_url	string	CURLINFO_EFFECTIVE_URL	Последний использованный URL.
header_size	int	CURLINFO_HEADER_SIZE	Размер всех заголовков в байтах.
httpauth_avail	int	CURLINFO_HTTPAUTH_AVAIL	Доступные методы HTTP-аутентификации.
namelookup_time	double	CURLINFO_NAMELOOKUP_TIME	Время от начала до завершения определения IP-адреса по имени.
num_connects	int	CURLINFO_NUM_CONNECTS	Число успешных соединений, потребовавшихся для предыдуще запроса.
os_errno	int	CURLINFO_OS_ERRNO	Код errno последней ошибки соединения.
pretransfer_time	double	CURLINFO_PRETRANSFER_TIME	Время от начала запроса до начал передачи данных.
primary_ip	string	CURLINFO_PRIMARY_IP	IP-адрес последнего соединения.
proxyauth_avail	int	CURLINFO_PROXYAUTH_AVAIL	Доступные методы HTTP-аутентификации прокси-сервера.
redirect_count	string	CURLINFO_REDIRECT_COUNT	Общее число совершенных перех по редиректам.
redirect_time	double	CURLINFO_REDIRECT_TIME	Время, потребовавшееся для совершения редиректов до финального соединения.
redirect_url	string	CURLINFO_REDIRECT_URL	URL, по которому был бы соверш переход, если бы был включен переход по редиректам.
request_size	int	CURLINFO_REQUEST_SIZE	Размер совершенных HTTP-запро в байтах.
response_code	int	CURLINFO_RESPONSE_CODE	Последний полученный код HTTP ответа.
size_download	double	CURLINFO_SIZE_DOWNLOAD	Размер полученных данных.
size_upload	double	CURLINFO_SIZE_UPLOAD	Размер переданных данных.
speed_download	double	CURLINFO_SPEED_DOWNLOAD	Средняя скорость получения данн
speed_upload	double	CURLINFO_SPEED_UPLOAD	Средняя скорость передачи данны
ssl_verifyresult	int	CURLINFO_SSL_VERIFYRESULT	Результат проверки SSL-сертифик
starttransfer_time	double	**CURLINFO_STARTTRANSFER_TIME**	Время от начала запроса до начал получения данных.
total_time	double	**CURLINFO_TOTAL_TIME**	Общее время последнего запроса

load. Загрузка файла с удаленного сервера

^curl:load[]
^curl:load[опции]

Метод выполняет загрузку файла с удаленного сервера. В рамках сессии этот метод может быть вызван несколько раз с разными параметрами (или вообще без них). Пришедшие **cookies** помещаются в поле **cookies** в виде таблицы со столбцами **name**, **value**, **expires**, **max-age**, **domain**, **path**, **HTTPonly** и **secure**. ***[3.4.3]***

Также доступно поле **tables** — это хеш, ключами которого являются поля заголовков HTTP-ответа в верхнем регистре, а значениями — таблицы с единственным столбцом **value**, содержащим все значения одноименных полей HTTP-ответа. ***[3.4.5]***

Пример
$file[^curl:load[

```
      $.url[HTTPs://store.artlebedev.ru/]
      $.useragent[Parser3]
      $.timeout(10)
      $.ssl_verifypeer(0)
]]
```

options. Задание опций для сессии

^curl:options[опции]

Метод может быть вызван только внутри сессии. Все последующие вызовы метода загрузки файлов в рамках данной сессии будут использовать установленные этим методом опции до тех пор, пока они не будут переопределены другим вызовом данного метода или перекрыты в методе загрузки файла. Но если есть необходимость задать путь к библиотеке **libcurl**, это нужно сделать до начала использования curl.

Пример

```
^curl:options[
      $.library[libcurl.so.4]
]
^curl:session{
      ^curl:options[
            $.charset[UTF-8]
            $.timeout(10)
      ]
      ...
}
```

session. Создание сессии

^curl:session{код}

Метод создает cURL-сессию. Код метода обрабатывается Parser, позволяя работать с удаленным сервером. Внутри сессии могут быть установлены общие опции и сделано несколько вызовов метода загрузки файла. Если удаленный сервер поддерживает **keep-alive**, то все запросы к нему будут сделаны в рамках одной установленной HTTP-сессии.

Пример

```
^curl:session{
      ^curl:options[
            $.url[HTTPs://store.artlebedev.ru/]
            $.charset[UTF-8]
            $.timeout(10)
            $.ssl_verifypeer(0)
      ]

      $file1[^curl:load[
            $.url[HTTPs://store.artlebedev.ru/login/]
            $.postfields[Username=^taint[uri][$form:login]&Password=^taint[uri]
[$form:password]&btnSubmit=^taint[uri][Войти]]
      ]]

      $file2[^curl:load[]]
}
```

version. Возврат текущей версии cURL

^curl:version[]

Метод возвращает строку, содержащую версию используемой библиотеки cURL.

Опции работы с библиотекой cURL

В качестве опций у методов **^curl:options[]** и **^curl:load[]** можно указывать любую из опций, доступную в библиотеке **libcurl**, установленной в системе (см. документацию). Имена опций нужно писать в нижнем регистре и без префикса **CURLOPT_**.

Кроме этого, поддерживаются следующие опции Parser:

Опция	По умолчанию	Значение
`$.library[/путь/к/libcurl.so]`	unix — `libcurl.so` win32 — `libcurl.dll`	Имя или полный дисковый путь динамической библиотеки `libcurl` в системе. Задается вызовом `^curl:options[]` до начала
`$.charset[кодировка]`	Соответствует `$request:charset`	Кодировка документов на удаленном сервере. В эту кодировку перекодируется строка запроса. Из этой кодировки перекодируется ответ сервера, если в HTTP-ответе не указана кодировка.
`$.response-charset[кодировка]`	Берется из заголовка HTTP-ответа	Принудительно указывает, в какой кодировке был получен ответ от сервера
`$.name[имя файла]`	`NONAME.DAT`	Имя файла создаваемого объекта класса `file`.
`$.mode[text|binary]`	`text`	Тип создаваемого объекта класса `file`.
`$.content-type[CONTENT-TYPE]`	Берется из заголовка HTTP-ответа	Поле `content-type` создаваемого объекта класса `file`.

Поддерживаемые опции libcurl в алфавитном порядке:

Название	Тип	Аналог в libcurl	Описание
accept_encoding	string	CURLOPT_ACCEPT_ENCODING	Метод упаковки ответа: `gzip` или `deflate`. (Старое название параметра – `encoding` – тоже поддерживается).
autoreferer	int	CURLOPT_AUTOREFERER	Автоматическое создание заголовка `Referer`.
cainfo	string	CURLOPT_CAINFO	См. документацию по `libcurl`.
capath	string	CURLOPT_CAPATH	См. документацию по `libcurl`.
connecttimeout	int	CURLOPT_CONNECTTIMEOUT	Тай-маут ожидания соединения в секундах.
connecttimeout_ms	int	CURLOPT_CONNECTTIMEOUT_MS	Тайм-аут ожидания соединения в миллисекундах.
cookie	string	CURLOPT_COOKIE	Строка с `cookies`.
cookielist	string	CURLOPT_COOKIELIST	Строка с `cookies` в формате, соответствующем документации `libcurl` (отличном от формата `cookie` в Parser).
cookiesession	int	CURLOPT_COOKIESESSION	Поставить `cookies` на всю сессию.
copypostfields	string, file	CURLOPT_COPYPOSTFIELDS	Тело пост-запроса (с копированием).
crlfile	string	CURLOPT_CRLFILE	См. документацию по `libcurl`.
customrequest	string	CURLOPT_CUSTOMREQUEST	Другой HTTP-метод.
failonerror	int	CURLOPT_FAILONERROR	Выдавать ошибку, если HTTP-статус больше или равен 400.
followlocation	int	CURLOPT_FOLLOWLOCATION	Обрабатывать редиректы в ответе сервера.
forbid_reuse	int	CURLOPT_FORBID_REUSE	См. документацию по `libcurl`.
fresh_connect	int	CURLOPT_FRESH_CONNECT	Создавать новое соединение при каждом запросе в сессии.
http_version	string	CURLOPT_HTTP_VERSION	Версия HTTP-протокола. Допустимые значения: `1.0`, `1.1`, `2`, `2.0`, `2TLS`, `2ONLY`.
http_content_decoding	int	CURLOPT_HTTP_CONTENT_DECODIN	См. документацию по `libcurl`.
http_transfer_decoding	int	CURLOPT_HTTP_TRANSFER_DECODI	См. документацию по `libcurl`.
httpauth	int	CURLOPT_HTTPAUTH	Тип HTTP-авторизации CURLAUTH_NONE = 0, CURLAUTH_BASIC = (1<<0), CURLAUTH_DIGEST = (1<<1), CURLAUTH_GSSNEGOTIATE = (1<<2), CURLAUTH_NTLM = (1<<3), CURLAUTH_DIGEST_IE = (1<<4), CURLAUTH_NTLM_WB = (1<<5), CURLAUTH_ONLY = (1<<31), CURLAUTH_ANY = (~CURLAUTH_DIGEST_IE), CURLAUTH_ANYSAFE = (~(CURLAUTH_BASIC|CURLAUTH_DIGEST_IE)).
httpget	int	CURLOPT_HTTPGET	Передать запрос методом `GET`.
httpheader	hash	CURLOPT_HTTPHEADER	HTTP-заголовки запроса.
httppost	hash	CURLOPT_HTTPPOST	Поля пост-запроса, заданные аналогично полю `form` для `file::load`.
httpproxytunnel	int	CURLOPT_HTTPPROXYTUNNEL	Включить тунелирование запросов через прокси.
ignore_content_length	int	CURLOPT_IGNORE_CONTENT_LENGTH	Игнорировать заголовок `Content-Length` ответа сервера.
interface	string	CURLOPT_INTERFACE	Имя сетевого интерфейса.
ipresolve	int	CURLOPT_IPRESOLVE	`1` – использовать IPv4 (по умолчанию), `2` – использовать IPv6.
issuercert	string	CURLOPT_ISSUERCERT	Имя файла с сертификатом CA.
keypasswd	string	CURLOPT_KEYPASSWD	Пароль для ключа (`passphrase`).

localport	int	CURLOPT_LOCALPORT	Локальный порт.
low_speed_limit	int	CURLOPT_LOW_SPEED_LIMIT	Минимальная скорость передачи, Б/сек.
low_speed_time	int	CURLOPT_LOW_SPEED_TIME	Максимальное время, когда скорость передачи может быть меньше `low_speed_limit`.
maxconnects	int	CURLOPT_MAXCONNECTS	Максимальное количество постоянных соединений в рамках сессии.
maxfilesize	int	CURLOPT_MAXFILESIZE	Максимальный размер ответа в байтах.
maxredirs	int	CURLOPT_MAXREDIRS	Максимальное число редиректов.
nobody	int	CURLOPT_NOBODY	Передать запрос методом `HEAD`.
password	string	CURLOPT_PASSWORD	Пароль.
port	int	CURLOPT_PORT	Порт.
post	int	CURLOPT_POST	Передать запрос методом `POST`.
postfields	string, file	CURLOPT_POSTFIELDS	Тело пост-запроса.
postredir	int	CURLOPT_POSTREDIR	См. документацию по `libcurl`.
proxy	string	CURLOPT_PROXY	Адрес прокси-сервера.
proxyauth	int	CURLOPT_PROXYAUTH	Тип авторизации (см. параметр httpauth).
proxyport	int	CURLOPT_PROXYPORT	Порт прокси-сервера.
proxytype	int	CURLOPT_PROXYTYPE	Тип прокси: `0` — `HTTP`, `1` — `HTTP_1_0`, `4` — `SOCKS4`, `5` — `SOCKS5`, `6` — `SOCKS4A`, `7` — `SOCKS5_HOSTNAME`.
proxyuserpwd	string	CURLOPT_PROXYUSERPWD	Имя пользователя и пароль для прокси.
range	string	CURLOPT_RANGE	Вернуть части ответа, находящиеся в указанном диапазоне.
referer	string	CURLOPT_REFERER	Заголовок `Referer`.
ssl_cipher_list	string	CURLOPT_SSL_CIPHER_LIST	См. документацию по `libcurl`.
ssl_sessionid_cache	int	CURLOPT_SSL_SESSIONID_CACHE	Включить SSL session-ID кеш.
ssl_verifyhost	int	CURLOPT_SSL_VERIFYHOST	Проверять сертификат хоста.
ssl_verifypeer	int	CURLOPT_SSL_VERIFYPEER	Проверять сертификат пира.
sslcert	string	CURLOPT_SSLCERT	Имя файла с SSL-сертификатом.
sslcerttype	string	CURLOPT_SSLCERTTYPE	Тип SSL-сертификата.
sslengine	string	CURLOPT_SSLENGINE	См. документацию по `libcurl`.
sslengine_default	string	CURLOPT_SSLENGINE_DEFAULT	См. документацию по `libcurl`.
sslkey	string	CURLOPT_SSLKEY	Имя файла с SSL-ключом.
sslkeytype	string	CURLOPT_SSLKEYTYPE	Тип SSL-ключа.
sslversion	int	CURLOPT_SSLVERSION	Версия протокола SSL/TLS-соединения: `0` — по умолчанию `1` — `TLSv1 (TLS 1.x)`, `2` — `SSLv2`, `3` — `SSLv3`, `4` — `TLSv1_0`, `5` — `TLSv1_1`, `6` — `TLSv1_2`.
stderr	string	CURLOPT_STDERR	Имя файла, в который будет переадресован вывод из `stderr`.
timeout	int	CURLOPT_TIMEOUT	Таймаут с секундах.
timeout_ms	int	CURLOPT_TIMEOUT_MS	Таймаут в миллисекундах.
unrestricted_auth	int	CURLOPT_UNRESTRICTED_AUTH	Повторно отсылать параметры HTTP-авторизации, если при редиректе сменилось имя сервера.
url	string	CURLOPT_URL	URL-адрес.
useragent	string	CURLOPT_USERAGENT	Заголовок `User-Agent`.

date (класс)

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

Диапазон возможных значений:
- от 01.01.1970 до 01.01.2038 года;
- от 00.00.0000 до 31.12.9999 года. ***[3.4.4]***

Не нужно забывать, 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 было времени «02:00, 31 марта 2002», а время «02:00, 27 октября 2002» было 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 формат UNIX, представляющий собой число секунд, прошедших с EPOCH.

Форматы Unix и ISO 8601 можно использовать в JavaScript и ряде других языков сценариев, работающих в браузере.

Parser полностью поддерживает работу с UNIX-форматом дат.

Конструкторы

create. Дата или время в стандартном формате для СУБД

```
^date::create[год]
^date::create[год-месяц]
^date::create[год-месяц-день]
^date::create[год-месяц-день часы]
^date::create[год-месяц-день часы:минуты]
^date::create[год-месяц-день часы:минуты:секунды]
^date::create[год-месяц-день часы:минуты:секунды.миллисекунды]
^date::create[часы:минуты]
^date::create[часы:минуты:секунды]
```

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**, **часы**, **минуты**, **секунды**, **миллисекунды** являются необязательными, если не заданы, подставляются первый день, нулевые час, минута, секунда или текущий день.

Замечание: значение **миллисекунды** *игнорируется.*

Удобно использовать этот конструктор для работы с датами, полученными из базы данных, ведь

из запроса получаются 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 />
}
```

Вниманию пользователей Oracle: чтобы получать дату и время в удобном формате, в строке соединения с сервером нужно указать формат даты и времени, рекомендованный в Приложении 3.

create. Дата в формате ISO 8601

^date::create[год-месяц-деньTчасов:минут:секунд+TZ]

Создает объект класса **date**, содержащий значение произвольной даты и времени с точностью до секунды из строки даты в формате ISO 8601.
Временная зона имеет один из следующих форматов: **+hh:mm**, или **+hhmm**, или **-hh:mm**, или **-hhmm**, или строковое значение **Z**, являющееся синонимом UTC.

Удобно использовать этот конструктор для работы с датами, полученными из внешних источников, например от JavaScript.

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 раньше текущей.

create. Относительная дата

^date::create(количество суток после EPOCH)

Конструктор с одним параметром предназначен для задания *относительных* значений дат и времени. 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+1)]
```

В примере получается дата на сутки (24 часа) позже.

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)
^date::create(year;month;day;hour;minute;second)[TZ]   [3.4.5]
```

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**, **TZ** являются необязательными. Если они не заданы, подставляются первый день, нулевые час, минута и секунда, а также текущий часовой пояс.

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.

now. Текущая дата

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

Конструктор создает объект класса **date**, содержащий значение текущей даты с точностью до секунды, с использованием системного времени сервера. Если указано, то прибавляется **смещение в сутках**, которое не обязательно должно быть целым числом.

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

Пример

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

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

today. Дата на начало текущего дня

```
^date::today[]
^date::today(смещение в днях)  [3.4.6]
```

Конструктор создает объект класса **date**, содержащий значение текущей даты на 00:00:00, с использованием системного времени сервера. Если указано, то добавляется **смещение в днях**, которое обязательно должно быть целым числом.

Пример

```
$today[^date::today[]]
^today.sql-string[]
```

unix-timestamp. Дата и время в Unix-формате

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

Конструктор создает объект класса **date**, содержащий значение, которое соответствует переданному числовому значению в формате UNIX.

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)
$date.weekyear год, к которому принадлежит неделя (согласно стандарту ISO 8601)
$date.yearday день года (0 — 1 января, 1 — 2 января, ...)
$date.daylightsaving 1 — летнее время, 0 — стандартное время
$date.TZ часовой пояс; содержит значение, оно было задано этой дате

Значения полей **year, month, day, hour, minute, second** можно менять.

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

int, double. Преобразование даты в число

```
^date.int[]
^date.double[]
```

Методы возвращают числовое значение объекта класса **date**, равное числу суток с EPOCH (01.01.1970 00:00:00, UTC) до даты, заданной в объекте.

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

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

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

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

iso-string. Вывод даты в виде строки в формате ISO 8601

```
^date.iso-string[]
^date.iso-string[ $.colon(true|false) $.ms(false|false) $.z(false|true)
]       [3.4.5]
```

Метод преобразует дату к строке в формате ISO 8601 (например, 2002-04-29T12:00:00+03:00). Этот метод полезен, если нужно сохранить информацию о часовом поясе.

Можно задать хеш опций.

- **$.colon(true|false)** — исключать двоеточие из временной зоны (2002-04-29T12:00:00+0300). По умолчанию — не исключать.
- **$.ms(false|true)** — добавлять миллисекунды, всегда .000 (2002-04-29T12:00:00.000+03:00). По умолчанию — не добавлять.
- **$.z(false|true)** — записывать временную зону UTC в виде 00:00 (2002-04-29T09:00:00+00:00). По умолчанию записывается Z (2002-04-29T09:00:00Z).

last-day. Получение последнего дня месяца

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

Метод возвращает последний день месяца.

Пример

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

Возвратит: **29**.

roll. Сдвиг даты

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

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

Также можно узнать дату или время, соответствующие хранящимся в объекте класса **date** в другом часовом поясе, задав системное имя нового часового пояса. Список имен см. в документации к операционной системе, 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 получается 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-Анджелес;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 в вид, стандартный для СУБД

```
^date.sql-string[]
^date.sql-string[datetime|date|time]        [3.4.2]
```

При вызове без параметров или с параметром **datetime** метод преобразует дату в вид **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.

При вызове с параметром **date** возвращает только дату в формате **ГГГГ-ММ-ДД**, а при вызове с параметром **time** возвращает только время в формате **ЧЧ:ММ:СС**.

Пример

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

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

unix-timestamp. Преобразование даты и времени в Unix-формат

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

Метод преобразует дату и время в значение в формате UNIX.

Статические методы

calendar. Создание календаря на заданную неделю месяца

```
^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** так же, 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](2022;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	2021
03	04	05	06	07	08	09	01	2022
10	11	12	13	14	15	16	02	2022
17	18	19	20	21	22	23	03	2022
24	25	26	27	28	29	30	04	2022
31							05	2022

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)

Метод возвращает последний день месяца указанного года и месяца.

Пример

```
^date:last-day(2028;02)
```

Возвратит: **29**.

roll. Установка временной зоны по умолчанию

^date:roll[TZ][часовой пояс по умолчанию]

Метод устанавливает часовой пояс, который по умолчанию будет использован при работе с датами. Метод может быть использован, если часовая зона сервера отличается от желаемой часовой зоны сайта.

Пример
```
^date:roll[TZ;MSK+3]
```

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 платформы.
Объект класса **double** не может принимать значения NaN и Inf.

Методы

format. Вывод числа в заданном формате

^имя.format[форматная строка]

Метод выводит значение переменной в заданном формате (см. Приложение 2. «Форматные строки»).

Если не пользоваться **format** и выводить число просто так:
$имя
то это эквивалентно **^имя.format[%.15g]**. Такое форматирование автоматически выбирает наиболее подходящий вид: числа будут выведены в обычном формате, если это возможно, а очень малые (меньше 1e-05) или очень большие (больше 1e+15) числа — в экспоненциальном формате.

Примеры

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

Возвратит: **15.68**

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

Возвратит: **0x0123**

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

^имя.inc[] – Метод увеличивает значение переменной на 1 или **число**.
^имя.inc(число)

^имя.dec[] – Метод уменьшает значение переменной на 1 или **число**.
^имя.dec(число)

^имя.mul(число) – Метод умножает значение переменной на **число**.

^имя.div(число) – Метод делит значение переменной на **число**.

^имя.mod(число) – Метод 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)**.

int, double, bool. Преобразование объектов в числа или bool

^имя.int[] или **^имя.int(default)**

^имя.double[] или **^имя.double(default)**

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

Методы преобразуют значение переменной **$имя** в целое число, вещественное число либо логическое значение и возвращают это значение. При преобразовании вещественного числа к целому дробная часть отбрасывается.

Можно задать значение по умолчанию, которое будет получено, если преобразование невозможно, строка пуста или состоит только из white spaces (символов пробела, табуляции, 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, содержащие числа (значение 0 будет преобразовано в **false**, не 0 — в **true**), а также строки, содержащие значения **true** и **false** (без учета регистра). При применении метода **bool** к числам, любое не нулевое значение будет преобразовано в **true**, нулевое — в **false**.

Внимание: использование пустой строки в математических выражениях не является ошибкой, ее значение считается нулем.

Внимание: преобразование строки, не являющейся целым числом, в целое число является ошибкой (пример: строка «1.5» не является целым числом).

Примеры

```
$str[Штука]
^str.int(1024)
```

Выведет число **1024**, поскольку объект **str** нельзя преобразовать в целое число без исключения.

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

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

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

Статические методы

sql. Получение числа из базы данных

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

Метод 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(n) — Отбрасывание первых **n** записей выборки.

Если ответ SQL-сервера был пуст (0 записей), то в зависимости от значения параметра **default**

$.default не задан — будет выдано сообщение об ошибке;
$.default{код} — будет выполнен указанный **код**, и число, которое он возвратит, будет результатом метода;
$.default(выражение) — будет 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 w3c.org/cgi. Веб-сервер Apache задает значения ряда дополнительных переменных.

Статические поля

fields. Все переменные окружения

```
$env:fields
```

Такая конструкция возвращает хеш со всеми полями переменных окружения сервера.

Пример

```
^env:fields.foreach[field;value]{
      $field — $value
}[<br />]
```

Пример выведет на экран все переменные окружения сервера и соответствующие им значения:

```
SERVER_SOFTWARE — Apache/2.2.22 (Win32)
SCRIPT_NAME — /cgi-bin/parser3.cgi
PATH_INFO — /env.html
...
```

PARSER_VERSION. Получение версии Parser

```
$env:PARSER_VERSION
```

Такая конструкция возвращает полную версию Parser с указанием платформы.

Например...

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

Получение значения переменной окружения

```
$env:переменная_окружения
```

Возвращает значение указанной переменной окружения.
В режиме веб-сервера возвращается значение переменной окружения, связанной с текущим запросом. Если такая переменная отсутствует, возвращается значение соответствующей переменной процесса веб-сервера ***[3.5.0]***

Пример

```
$env:REMOTE_ADDR
```

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

Получение значения поля запроса

```
$env:HTTP_ПОЛЕ_ЗАПРОСА
$request:headers.ПОЛЕ_ЗАПРОСА [3.4.4]
```

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, начинающиеся с **HTTP_**, и знаки «-» в них заменены на «_».

Подробности — в документации используемого веб-сервера.

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 передаче 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 «Установка и настройка Parser»). 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

base64. Декодирование из Base64

^file::base64[закодированное]

^file::base64[text|binary;имя файла;закодированное;опции] *[3.4.1]*

Декодирует файл из Base64-представления. Для кодирования файла следует использовать **^файл.base64[].**

Можно задать хеш опций:

- **$.strict(true)** – будет выдаваться исключение при невозможности декодирования **всех** символов. Без указания данной опции файл будет создан из того, что было успешно декодировано. *[3.4.2]*
- **$.url-safe(false|true)** – использовать модифицированный алфавит, все символы которого не преобразовывались в %XX в URL (вместо «+» и «/» используются «-» и «_»). По умолчанию не использовать. *[3.4.6]*
- **$.pad(true|false)** – при кодировании были добавлены символы паддинга (=), по умолчанию. *[3.4.6]*
- **$.content-type[...]** – задать content-type создаваемого файла.

Подробная информация о Base64 доступна по ссылке: ietf.org/rfc/rfc2045.txt и 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/g7jAy8ysHnMdylnC0M3W1wAZ0JJvz2MY2OG+DIPcwcPS4uqUuyAPbbZjGuv0kwdzGXkbc+n1
/nnaeJkzQqCBwQYIDASQcm/MhSmdIkrE8HDKgAcYM2rE2M8Ig40gNw68GLJkxwEXJqoE96SkSwDe
wrCEQqCChZs4JSyMomFMh/8pjwLNizKgQ1BisaCgOjrm3ZCiTMXMfGo0apgnPa2CIDemo5CaOMNa
0BmFqxivQrSGGepxmKNeT5ZqdQqAQcyoUwFAVWskq9YLPqOAFRuWLJS7haKoXdtWK1wicucecXt0
6l6+RPxq1ZzvyWDChXf2/SZlcZjOk00bAxBZ8gHKlYlUXSzbNIhdq4d8xglBAWHDRQCL4SCFg/Hj
yJPfJT4Ew5zkyTNN3eUBunXjox48vw49cVp5KyUSgcbdelOCNsVCEOJbLPAh2oezcS6/+ZzipIfs
ykslvhguc9CFRYBXEEjEbjetN0R7oRXh323zjcGcEPT9F8V+RGB4yX1bGJj/hYdUZCIgghYoSASD
OYkGwIMTplEhhPZ9s9Jd0+V3xYMghIdBbkOQVx5y451nxS50kagAFCjeBByLEdZHoW01SpXFa5s9
YRsIQYoh4BS4fZVeWEdGkeRYOwkXRotovNjii1pFKZMW1FjFVo+2ZRnGllJoBo6RVIxZAQEA6Nnk
mUSwaZWbOW4RZ1RzAnClne5cYSYIPy1AWJh9EuYATGN456KEhUKZoY1ZXMZURd+ZBimeZc1RAAAo
YloFiuslwM+gMD4po0o0jirlGy5hNFuidIpxQbAbrYpFJhSwd5OsVrSnYEBheNCGmqGOgd+vQmjI
h7eOCvmhuFVo9oEA7EF7/wUBm+qVCWpqYBujhVCAC64e4IJYILlUmPWmG9SGgRaFOi6xy5oc1kvq
vf3+iJx0kHbg8HHKYtFOUmrg2KiVpsHL5rb/dktqLqY9Fmidxd6ZBY4eDLTFAOhQYVqjH1+48Mj9
LWayEJpZVTEWPXfgMhamggCvYmptXLPC3ALA8BQ4HrXzED0z9fMVMG/DxQHDUjq0EUk/sfQT9uIM

```
tWMYGxGw1SlHCicdH7A6RQTDTA3FHBqEh2oRByRblkbSfJTRwE+QhOzg/R6uEREicdHaWoQPccA+
dPCItJb/ZJ7F42ulXUQEVYfhqcz8am56FBPA9sEGGEyA0QQYbKC65aWVflL67UUw0LVpHXhOOua4
B1+4oVZ9wJ8V+gqvPAAHhP7IBxl8XUXyyyuPUjuBbDCB9FY0Xv33Dqa0gXF5Hwv++einr/767Lfv
/vvwxy///PTXj0YQADs=
]
$original[^file::base64[$encoded]]
$filespec[/p2.gif]
^original.save[binary;$filespec]
<img src="$filespec" />
```

Выведет...

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;...]
^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;...]
```

Конструктор **cgi** создает объект класса **file**, содержащий результат исполнения программы в соответствии со стандартом CGI.

Внимание: все пути в Parser 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.

Аргумент — строка или массив строк ***[3.5.0]*** или таблица с одним столбцом, содержащая аргументы.

Объект, созданный конструкторами **cgi** и **exec**, имеет дополнительные поля:
status — информация о статусе завершения 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**, который содержит 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).

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: в версиях с отключенным safe-mode переменным окружения можно задавать любые имена. ***[3.4.1]***

Внимание: при обработке HTTP POST-запроса, при помощи конструкции **$.stdin[$request:body]** *можно передать в стандартный поток ввода скрипта полученные POST-данные.*

Внимание: запускаемому скрипту также передаются все переменные окружения, которые были выставлены HTTP-сервером при запуске Parser.

Пример исполнения внешнего CGI-скрипта

```
$search[^file::cgi[search.cgi;$.QUERY_STRING[text=$form:q&page=$form:p]]]
```

Пример исполнения 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:

```
$args[^table::create{arg
длина
ширина}]
$script[^file::exec[script.pl;;$args]]
```

Пример скрипта для исполнения процесса в фоновом режиме

При необходимости исполнения длительного процесса его можно запустить в фоновом режиме с помощью промежуточного скрипта. При этом, чтобы скрипт завершился сразу, необходимо перенаправить **stdout** и **stderr** процесса:

```
#/bin/sh
sleep 60 >/dev/null 2>&1 &
```

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

create. Создание файла

```
^file::create[формат;имя;текст]
^file::create[формат;имя;текст;опции]        [3.4.0]
^file::create[строка;расширенные опции]      [3.4.2]
^file::create[файл;расширенные опции]        [3.4.2]
```

Создает объект класса **file** с указанными **именем** и содержимым.
При создании текстовых файлов производится нормализация символов переводов строк. *[3.4.2]*

Формат — формат представления создаваемого файла. До версии ***3.4.2*** поддерживался только текстовый (**text**) формат.
Опции — хеш, в котором можно указать **$.from-charset[**кодировка**]**, или **$.to-charset[**кодировка**]**, или **$.content-type[...]**. *[3.4.1] [3.4.5]*
Расширенные опции — хеш, в котором, помимо обычных опций, можно указать еще и **$.name[**имя файла**]**, **$.mode[**формат**]**

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

*Примечание: до версии **3.4.5** параметра с кодировкой, из которой нужно преобразовать данные (**from-charset**), не было, а параметр с кодировкой, в которую нужно преобразовать данные, назывался не **to-charset**, а просто **charset**.*

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 примерно 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>
```

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**(текстовым) или **binary**(двоичным). Различие между этими типами заключается в разных символах переноса строк. Для PC эти символы **0D 0A**. При использовании формата **text** при загрузке файла символ **0D** будет отброшен за ненадобностью, при записи методом **save** — будет добавлен.

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

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

опции загрузки — см. «Приложение 1. Пути к файлам и каталогам, 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**, содержащим все значения одноименных полей HTTP-ответа. Пришедшие **cookies** помещаются в поле **cookies** в виде таблицы со столбцами **name**, **value**, **expires**, **max-age**, **domain**, **path**, **HTTPonly** и **secure**. ***[3.4.2]***

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 с SQL-сервера

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

Загружает файл с SQL-сервера. Результатом выполнения запроса должна быть одна запись (при необходимости следует воспользоваться 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(n) — отбросить первые **n** записей выборки; ***[3.3.0]***
$.content-type[пользовательский тип] — задать пользовательский `content-type`;
$.name[имя] — задать имя файла.

Имя файла и его `content-type` будут 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}
```

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**.

Примечание: автоматическая нормализация переводов строк делается для текстовых файлов (mode=text), но не делается *для бинарных (mode=binary). Чтобы сделать нормализацию переводов строк для бинарных файлов, например тех, которые были получены из form, необходимо воспользоваться следующей конструкцией:*

```
$f[^file::create[$form:file;$.mode[text]]
$f.text
```

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** путем 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).

mode. Формат файла *[3.4.0]*
$файл.mode

Может иметь значение **text** или **binary**.

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

base64. Кодирование в Base64

```
^file.base64[]
^file.base64[опции]
```
[3.4.6]

Метод позволяет преобразовать файл в Base64-форму.

Чтобы преобразовать файл из Base64 к исходному виду, нужно воспользоваться конструкцией:
^file::base64[закодированное]

Можно задать хеш опций:
- **$.wrap(true|false)** – формировать результат с переносами строк (по умолчанию) или в одну строку;
- **$.url-safe(false|true)** – использовать модифицированный алфавит, все символы которого

не будут преобразовываться в в %XX в URL (вместо «+» и «/» используются «-» и «_»);
по умолчанию не использовать;
- **$.pad(true|false)** – добавлять символы паддинга (=), если кодируемая длина не кратна трем; по умолчанию добавлять.

Подробная информация о Base64 доступна по ссылкам: ietf.org/rfc/rfc2045.txt и wikipedia.org/wiki/Base64

Пример

```
$original[^file::load[binary;HTTP://www.parser.ru/i/p2.gif]]
<pre>^original.base64[]</pre>
```

выведет:

```
R0lGODlhCQAJAID/AMDAwAAAACH5BAEAAAAALAAAAAAJAAkAAAINhI8YqXwLQVyMJtscKgA7
```

crc32. Подсчет контрольной суммы файла

^file.crc32[]

Для файла будет подсчитана контрольная сумма (CRC32).
Метод выдает ее в виде целого числа.

md5. MD5-отпечаток файла

^file.md5[]

Для файла будет получен «отпечаток» размером 16 байт.
Метод выдает его представление в виде строки – байты представлены в шестнадцатеричном 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 по ссылке: ietf.org/rfc/rfc1321.txt

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.

Можно задать хеш опций:
- **$.charset[**кодировка**]** – кодировка для сохраняемого текстового файла.
- **$.append(false|true)** – если файл существует, то дописать файл в конец существующего файла; по умолчанию, если файл с указанным именем существует, то он будет перезаписан. ***[3.4.6]***

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[]
```

Метод 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[]')}
}
```

Статические методы

base64. Кодирование в Base64

```
^file:base64[имя файла]
```

Метод позволяет преобразовать файл с указанным именем в Base64-форму. Чтобы преобразовать файл к исходному виду, нужно воспользоваться конструкцией
`^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[]
```

При этом 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 первому нулевому символу и все символы перевода строк будут нормализованы.

basename. Имя файла без пути

```
^file:basename[filespec]
```

Из полного пути к файлу (**filespec**) метод получает имя файла с расширением имени, но без пути.

Пример

```
^file:basename[/a/some.tar.gz]
```

выдаст: **some.tar.gz**

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**.

crc32. Подсчет контрольной суммы файла

```
^file:crc32[имя файла]
```

Для файла с указанным именем будет подсчитана контрольная сумма (CRC32). Метод выдает ее в виде целого числа.

delete. Удаление файла с диска

```
^file:delete[путь]
^file:delete[путь;опции]
```
[3.4.3]

Метод удаляет указанный файл.
Путь — путь к файлу

Если после удаления в каталоге больше ничего не осталось — каталог тоже удаляется (если это возможно).

Можно задать хеш **опций**:
$.keep-empty-dirs(true) — не удалять пустые каталоги, если таковые остались после удаления файла.
$.exception(false) — не выдавать исключение при невозможности удаления файла.

Пример
```
^file:delete[story.txt]
```

dirname. Путь к файлу

```
^file:dirname[filespec]
```

Для переданного файла или каталога (**filespec**) метод возвращает каталог, в котором он находится.

Пример
```
#имя файла
^file:dirname[/a/some.tar.gz]
#имя каталога...
^file:dirname[/a/b/]
```

Оба вызова выдадут:
/a

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 здесь, файл будет последовательно искаться в каталогах

- **`/i/раздел/`**
- **`/i/`**
- **`/`**

fullpath. Полное имя файла от корня веб-пространства

```
^file:fullpath[имя файла]
```

Метод принимает в качестве параметра ***`имя файла`*** и возвращает полное имя файла (абсолютный путь к файлу).

Пример

На странице `/document.html` создается ссылка 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 текущего запрашиваемого документа и ничего не знает про то, что на веб-сервере использован `mod_rewrite`.

Поэтому удобно заменить относительное имя полным:

```
$image[^image::measure[^file:fullpath[image.gif]]]
^image.html[]
```

Такая конструкция:

```
<img src="/image.gif" width="…" height="…" />
```

создаст код, содержащий абсолютный путь.

justext. Расширение имени файла

```
^file:justext[filespec]
```

Метод принимает в качестве параметра полный путь к файлу (**`filespec`**) и возвращает расширение имени файла без точки.

Пример

```
^file:justext[/a/some.tar.gz]
```

выдаст: **`gz`**

justname. Имя файла без расширения

```
^file:justname[filespec]
```

Метод принимает в качестве параметра полный путь к файлу и возвращает имя файла без пути и расширения.

Пример

```
^file:justname[/a/some.tar.gz]
```

выдаст: **some.tar**

list. Получение оглавления каталога

^file:list[путь]
^file:list[путь;фильтр]
^file:list[путь;опции] ***[3.4.3]***

Можно задать хеш **опций**:
- **$.filter[фильтр]** — строка с регулярным выражением или объект класса **regex**;
- **$.stat(true|false)** — **true** — заполнить столбцы **size**, **cdate**, **mdate** и **adate**.

Метод формирует таблицу (объект класса **table**) со столбцами **name**, **dir**, **size**, **cdate**, **mdate** и **adate** (до версии ***[3.4.3]*** возвращался только столбец **name**), содержащую файлы и каталоги по указанному пути, имена которых удовлетворяют шаблону, если он задан. Для каждой записи, являющейся каталогом, в результирующей таблице значение в столбце **dir** будет иметь значение **1**.

*Внимание: без указания опции **$.stat(true)** значения столбцов **size**, **cdate**, **mdate** и **adate** в результирующей таблице будут пусты.*

фильтр — строка с регулярным выражением (см. метод **match** класса **string**) или объект **regex** ***[3.4.0]***. Без указания фильтра будут выведены все найденные 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.

lock. Эксклюзивное выполнение кода

^file:lock[имя файла блокировки]{код}

Метод гарантирует, что **код** не будет выполняться параллельно в разных процессах, 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 следует помнить 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 блокировки всегда нужно думать об их взаимном сочетании, чтобы избежать ситуации «А ждет Б, Б ждет А», так называемого deadlock.

md5. MD5-отпечаток файла

^file:md5[имя файла]

Для файла с указанным именем будет получен «отпечаток» размером 16 байт. Метод выдает его представление в виде строки — байты представлены в шестнадцатеричном 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 по ссылке: ietf.org/rfc/rfc1321.txt

move. Перемещение или переименование файла

^file:move[старое имя файла;новое имя файла]
^file:move[старое имя файла;новое имя файла;опции] ***[3.4.3]***

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.

Можно задать хеш **опций**:
$.keep-empty-dirs(true) — не удалять пустые каталоги, если таковые остались после перемещения файла.

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.

form (класс)

Класс **form** предназначен для работы с полями (элементами) 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" />
```

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**.

Необходимо 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.name]]
	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>
```

Этот код 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**.

Статические поля

elements. Массивы всех полей формы

$form:elements

Поле возвращает хеш, содержащий массивы всех элементов формы или параметры, переданные через URL. Имена ключей хеша соответствуют названиям элементов формы, а значения ключей представляют собой массивы всех значений этих элементов.

fields. Все поля формы

$form:fields

Поле возвращает хеш со всеми элементами формы или параметрами, переданными через URL. Имена ключей хеша — это названия элементов формы, значениями ключей являются значения элементов 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
```

files. Получение множества файлов

$form:files

Поле возвращает редактируемый хеш со всеми файлами формы. Имена ключей — это названия файловых элементов 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: перед использованием хеша нужно проверить его определенность.

Пример

```
^if($form:files.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>
```

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` нужно написать:
```
$clicked[$form:imap]
^if(def $clicked){
    Посетитель нажал на ISMAP-ссылку:<br />
    x=$clicked.x<br />
    y=$clicked.y<br />
}
```

В файле `/test.html` нужно написать:
```
<a href="/go.html?a=b"><img src="map.png" ismap /></a>
```

Если открыть в браузере `/test.html` и кликнуть мышкой на картинке, это приведет к переходу по адресу:
`/go.html?a=b?10,30`

и отобразится:
```
    Посетитель нажал на ISMAP-ссылку:
    x=10
    y=30
```

qtail. Получение остатка строки запроса

`$form:qtail`

Поле 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`**.

tables. Получение множества значений поля

`$form:tables`

Поле возвращает редактируемый хеш со всеми элементами формы или параметрами, переданными через URL. Имена ключей хеша — это названия элементов 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: перед использованием таблицы нужно проверить ее определенность.

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.

hash (класс)

Класс предназначен для работы с хешами — ассоциативными массивами. Хеш запоминает порядок, в котором были добавлены элементы. 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. Хеш (ассоциативный массив)».

create. Создание пустого хеша и копирование хеша

```
^hash::create[]
^hash::create[существующий хеш, или хеш-файл, или пользовательский объект, или файл]
```

Если параметр не задан, будет создан пустой хеш.
Если указан **существующий хеш** или другой совместимый с хешем **объект**, 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*о* именно наполняем.

Если предполагается интенсивная работа по изменению содержимого хеша, но необходимо сохранить, скажем, исходные значения, то это можно сделать, например, так:

```
$pets[
      $.pet[Собака]
      $.food[Косточка]
      $.good[Ошейник]
]
$pets_copy[^hash::create[$pets]]
```

Замечание: поле **_default** *копируется.*

sql. Создание хеша на основе выборки из базы данных

```
^hash::sql{запрос}
^hash::sql{запрос}[$.limit(n) $.offset(n) $.distinct(true|false)
$.bind[variables hash] $.type[hash|string|table]]
```

Конструктор создает хеш, в котором имена ключей совпадают со значениями первого столбца выборки.

По умолчанию каждый элемент — тоже хеш, где имена столбцов используются в качестве ключей, а соответствующие им значения — это данные столбцов.
Если же запрос возвращает только один столбец, формируется хеш, где значения столбца формируют ключи хеша, и им ставится в соответствие логическое значение «**истина**».

Дополнительные параметры конструктора:

$.limit(n)	получить только **n** записей
$.offset(n)	отбросить первые **n** записей выборки
$.bind[hash]	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 нужно указать *два* столбца в SQL-запросе; **table** = значение каждого элемента — таблица со всеми колонками результата.

По умолчанию наличие в ключевом столбце одинаковых значений считается ошибкой. Если наличие дубликатов допустимо, следует задать опцию **$.distinct(true)**. В этом случае в качестве данных для каждого ключа используется первая встреченная строка, последующие строки с тем же ключом игнорируются без ошибки. А если задать **$.type[table]**, для каждого ключа формируется таблица со всеми записями, имеющими этот ключ.

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

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.

До версии ***3.4.4*** эта же запись могла быть использована для получения методов хеша. Начиная с версии ***3.4.4*** обращение к методам хеша возможно только при их вызове, **^my_hash.method[]**, причем методы имеют приоритет перед полями. Начиная с версии ***3.4.5*** **_default** воспринимается как ключ по умолчанию, только если его написать в коде на Parser.

Присваивание ключу значения добавит или обновит пару «**ключ / значение**» в хеше:
$my_hash.key[значение]

Для большей взаимозаменяемости таблиц и хешей поле **fields** хранит ссылку на сам хеш.

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

at. Доступ к элементу хеша по индексу

```
^хеш._at(число|-число)
^хеш._at[first|last]
^хеш.at(число|-число)    [3.4.4]
^хеш.at[first|last]      [3.4.4]
^хеш.at(число|-число)[key|value|hash]      [3.4.4]
^хеш.at[first|last;key|value|hash]         [3.4.4]
```

При добавлении элементов в хеш каждый из них получает свой индекс, начиная с 0. Метод позволяет получить доступ к элементу хеша по заданному индексу, т. е. **^хеш.at(0)** эквивалентен **^хеш.at[first]**. В случае отрицательного значения поиск элемента производится с конца, и **^хеш.at(-1)** эквивалентен **^хеш.at[last]**.

Второй параметр определяет возвращаемый результат: *[3.4.4]*
value – вернется значение элемента (по умолчанию);
key – вернется ключ элемента;
hash – вернется хеш из одного элемента.

contains. Проверка существования ключа

```
^хеш.contains[ключ]
```

Метод возвращает значение **«истина»**, если в хеше содержится запись с указанным ключом и **«ложь»** в противном случае. С помощью **^hash.contains[_default]** можно проверить, задано ли в хеше значение по умолчанию *[3.4.5]*.

Пример

```
^if(^man.contains[birthday]){
	У посетителя определена дата рождения.
}
```

count. Количество ключей хеша

```
^хеш._count[]
^хеш.count[]	[3.4.4]
```

Метод возвращает количество ключей хеша.

Пример

```
$man[
	$.name[Вася]
	$.age[22]
	$.gender[m]
]
^man.count[]
```

Вернет: **3**.

В выражениях числовое значение хеша равно количеству ключей:

```
^if($man > 2){больше}
```

delete. Удаление пары «ключ / значение»

```
^хеш.delete[ключ]
^хеш.delete[]	[3.4.4]
```

Метод удаляет из хеша пару «**ключ / значение**». При вызове без параметра удаляются все поля хеша.

Пример

```
^man.delete[name]
```

Удалит ключ **name** и связанное с ним значение из хеша **man**.

foreach. Перебор элементов 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 не пустым 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 помощью оператора **break** или принудительно закончить текущую итерацию и перейти к следующей с помощью оператора **continue**.

Пример
```
$man[
	$.name[Вася]
	$.age[22]
	$.gender[m]
]
^man.foreach[key;value]{
	$key=$value
}[<br />]
```

Выведет на экран:
```
name=Вася
age=22
gender=m
```

keys. Список ключей хеша

```
^хеш._keys[]
^хеш._keys[имя столбца]
^хеш.keys[] [3.4.4]
^хеш.keys[имя столбца] [3.4.4]
```

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]
	$.gender[m]
]
$tab_keys[^man.keys[]]
^tab_keys.save[keys.txt]
```

Будет создан файл `keys.txt` с такой таблицей:
```
key
name
age
gender
```

rename. Переименовывание ключей хеша

```
^хеш.rename[старое название ключа;новое название ключа]
^хеш.rename[ $.старое_название_ключа[новое название ключа] … ]
```

Метод изменяет названия одного или нескольких существующих ключей хеша с сохранением порядка элементов.

reverse. Обратный порядок элементов

```
^хеш.reverse[]
```

Метод возвращает новый хеш, в котором элементы идут в порядке, обратном порядку добавления элементов в исходный хеш.

Пример

```
$man[
	$.name[Вася]
	$.age[22]
	$.gender[m]
]
$man[^man.reverse[]]
^man.foreach[key;value]{
	$key=$value
}[<br />]
```

Выведет на экран:

```
gender=m
age=22
name=Вася
```

select. Отбор элементов

```
^хеш.select[ключ;значение](критерий_отбора)
^хеш.select[ключ;значение](критерий_отбора)[опции]
```

Метод последовательно перебирает все элементы хеша, применяя к ним выражение **критерий_отбора**; строки, подпавшие под заданный **критерий** (логическое выражение было истинно), помещаются в результат, которым является результирующий хеш.

Можно задать хеш опций:

$.limit(максимум)	максимальное число элементов, которые можно отобрать;
$.reverse(false|true)	**true** = перебирать элементы в обратном порядке.

Пример

```
$men[
	$.Serge(26)
	$.Alex(20)
	$.Misha(29)
	$.Denis(30)
]

$thoseAbove20[^men.select[;age]($age > 20)[ $.limit(2) ]]
```

В **$thoseAbove20** попадут элементы **Serge** и **Misha**.

set. Установка значения по индексу

```
^хеш.set[first|last][значение]
^хеш.set(индекс)[значение]
```

Метод присваивает **значение** существующему элементу хеша по указанному порядковому индексу:

first – устанавливает значение первого инициализированного элемента массива;
last – устанавливает значение последнего инициализированного элемента массива;
индекс – порядковый индекс элемента, которому будет присвоено значение.

sort. Сортировка хеша

```
^хеш.sort[ключ;значение]{функция_сортировки_по_строке}
^хеш.sort[ключ;значение]{функция_сортировки_по_строке}[направление_сортировки]
^хеш.sort[ключ;значение](функция_сортировки_по_числу)
^хеш.sort[ключ;значение](функция_сортировки_по_числу)[направление_сортировки]
```

Метод осуществляет сортировку элементов в хеше по указанной функции.

Функция сортировки – произвольная функция, по текущему значению которой принимается решение о положении поля в отсортированном хеше. Значением функции может быть строка (значения сравниваются в лексикографическом порядке) или число (значения сравниваются как действительные числа).

Направление сортировки – параметр, задающий направление сортировки. Принимает значения:
desc – по убыванию;
asc – по возрастанию.
По умолчанию используется сортировка по возрастанию.

Пример

```
$men[^hash::create[
      $.Serge(26)
      $.Alex(20)
      $.Misha(29)
]]
^men.sort[name;]{$name}
^men.foreach[name;age]{
    $name: $age
}[<br />]
```

В результате записи хеша **$men** будут отсортированы по строке имени:

```
Alex: 20
Misha: 29
Serge: 26
```

А можно отсортировать записи хеша по числу прожитых лет по убыванию (**desc**), если изменить в примере вызов **sort** на такой:

```
^men.sort[;age]($age)[desc]
```

...получится...

```
Misha: 29
Serge: 26
Alex: 20
```

Работа с множествами

add. Сложение хешей

^хеш.add[хеш-слагаемое]

Метод добавляет к **хешу** другой **хеш-слагаемое**, при этом одноименные ключи хеша перезаписываются.

Замечание: поле **_default** *добавляется. Если оно было, то перезаписывается новым значением.*

Пример

```
$man[
    $.name[Вася]
    $.age(22)
    $.gender[m]
]
$woman[
    $.name[Маша]
    $.age(20)
    $.smile[да]
]
^man.add[$woman]
```

Новое содержание хеша **$man**:

```
$man[
    $.name[Маша]
    $.age(20)
    $.gender[m]
    $.smile[да]
]
```

intersection. Пересечение хешей

^хеш_a.intersection[хеш_b]
^хеш_a.intersection[хеш_b[; $.order[self|arg]]] ***[3.5.0]***

Метод выполняет пересечение двух хешей. Возвращает хеш, содержащий ключи, которые принадлежат как хешу **a**, так и хешу **b**, значения берутся из хеша **a**. Результат необходимо присваивать новому хешу. Опция **$.order** задаёт порядок элементов в результирующем хеше. По умолчанию (или если значение опции равно **self**), порядок соответствует порядку элементов хеша **a**. Если же опция установлена в **arg**, порядок соответствует хешу **b**.

Пример

```
$man[
      $.name[Вася]
      $.age[22]
      $.gender[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
}
```

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]
        $.gender[m]
]
$woman[
        $.name[Маша]
        $.age[20]
]
^man.sub[$woman]
```

В результате в хеше **$man** останется только один ключ **$man.gender** со значением **m**.

union. Объединение хешей

```
^хеш_a.union[хеш_b]
```

Метод выполняет объединение двух хешей. Возвращает хеш, содержащий все ключи хеша **a** и те ключи из хеша **b**, которых нет в **a**. Результат необходимо присваивать новому хешу.

Пример

```
$man[
        $.name[Вася]
        $.age[22]
        $.gender[m]
]
$woman[
        $.name[Маша]
        $.age[20]
        $.weight[50]
]
$union_hash[^man.union[$woman]]
```

Получится хеш **$union_hash**:

```
$union_hash[
        $.name[Вася]
```

```
        $.age[22]
        $.gender[m]
        $.weight[50]
]
```

hashfile (класс)

Класс предназначен для работы с хешами, хранящимися на диске. В отличие от класса **hash**, объекты данного класса считаются всегда определенными (**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**».

Для хранения данных **hashfile** используются два файла — с расширением `.dir` и `.pag`

Замечание: существует ограничение на длину строк ключа и значения, в сумме они не должны превышать 8000 байт.

Чтение и запись данных происходят 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 2 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 **hashfile** используются два файла — с расширениями `.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]
$hashfile.ключ[
      $.value[строка]
      ...необязательные модификаторы...
]
```

Сохраняет на диск ассоциацию между **ключом** и **строкой**.

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 которых 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

cleanup. Удаление устаревших записей

```
^hashfile.cleanup[]
```

Метод перебирает все пары и удаляет устаревшие.

Замечание: физического удаления из файла не происходит. Устаревшие пары лишь помечаются как удаленные, и последующие записи новых данных могут использовать освободившееся место.

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 ключом лишь помечается как удаленная, и последующая запись новых данных может использовать освободившееся место.

delete. Удаление файлов данных с диска

```
^hashfile.delete[]
```

Метод удаляет с диска файлы, в которых хранятся данные хеш-файла.

foreach. Перебор ключей хеша

```
^hashfile.foreach[ключ;значение]{тело}
^hashfile.foreach[ключ;значение]{тело}[разделитель]
^hashfile.foreach[ключ;значение]{тело}{разделитель}
```

Метод перебирает все ключи объекта и соответствующие им значения (порядок перебора соответствует порядку добавления элементов в хеш). Метод аналогичен **foreach** класса **hash**.

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

hash. Получение обычного хеша

```
^hashfile.hash[]
```

Метод выдает обычный хеш с данными **hashfile**.

release. Сохранение изменений и снятие блокировок

```
^hashfile.release[]
```

Метод сохраняет на диске все сделанные изменения и снимает блокировки с файлов. Таким образом, хеш-файл 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 (exif.org).

Для представления цветов используется схема RGB, в которой каждый оттенок цвета представлен тремя составляющими (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 отведено два разряда в указанной последовательности. Формула для вычисления цвета выглядит так:

```
(R*0x100+G)*0x100+B
```

Так, для белого цвета, у которого все составляющие имеют максимальное значение — **FF**, данная формула при подстановке дает:

```
(0xFF*0x100+0xFF)*0x100+0xFF = 0xFFFFFF
```

Конструкторы

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 размещаться

на белом фоне.

Пример
```
$square[^image::create(100;100;0x000000)]
```

Будет создан объект **square** класса **image** размером 100x100 с фоном черного цвета.

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 задействовать 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.

measure. Создание объекта на основе существующего графического файла

```
^image::measure[файл]
^image::measure[имя файла]
^image::measure[файл;опции] [3.4.6]
^image::measure[имя файла;опции] [3.4.6]
```

Создает объект класса **image**, измеряя размеры существующего графического файла или объекта класса **file** в поддерживаемом формате. Поддерживаются GIF, JPEG и PNG, а начиная с версии ***[3.4.6]*** дополнительно поддерживаются TIFF, BMP, WEBP и при указании опции **$.video(true)** — еще и MP4 (MOV).

Сама картинка не считывается, основное назначение метода — определение размеров и, например, последующий вызов для созданного объекта метода **html**.

Параметры конструктора:
Файл — объект класса **file**;
Имя файла — имя файла с путем.

Можно задать хеш опций:

	По умолчанию	Описание
$.video(false|true)	false	Определять размеры у видеофайлов в формате MP4 (MOV);
$.exif(false|true)	false	Считывать EXIF-информацию из JPEG-файлов; до версии ***[3.4.6]*** EXIF-информация считывалась всегда;
$.xmp(false|true)	false	Считывать XMP-информацию из JPEG-файлов;
$.xmp-charset[кодировка]	UTF-8	Кодировка XMP-информации.

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(true) ]]
$exif[$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 />
}
```

Поля

$картинка.src	— имя файла
$картинка.width	— ширина
$картинка.height	— высота
$картинка.exif	— хеш с EXIF-информацией
$картинка.xmp	— строка с XMP-информацией (в формате XML)

Ключами **$картинка.exif** являются названия EXIF-тегов (см. спецификацию exiftool.org/TagNames/EXIF.html). Значения имеют тип **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 пикселях: $photo.width<br />
Высота изображения в пикселях: $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

gif. Кодирование объектов класса image в формат GIF

```
^картинка.gif[]
^картинка.gif[имя файла]
```

Метод используется для кодирования созданных Parser объектов класса **image** в формат GIF. **Имя файла** будет передано посетителю при указании этого файла в качестве значения **$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 x 100 пикселей.

html. Вывод изображения

```
^картинка.html[]
^картинка.html[хеш]
```

Метод создает следующий HTML-тег:

```
<img src="$image.src" width="$image.width" height="$image.height" border="0" />
```

В качестве параметра методу может быть передан хеш, содержащий дополнительные атрибуты изображения, например **alt** и **border**, задающие надпись, которая появляется при наведении курсора, и ширину рамки.

Замечание: атрибуты изображения можно переопределять.

Замечание: чтобы метод не выводил атрибут **border**, *ему необходимо передать параметр* **$.border[]** ***[3.4.1]***

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...**

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, — это точка с координатами (0:0).

Тип и ширина линий

```
$картинка.line-style[тип линии]
$картинка.line-width(толщина линии)
```

Перед вызовом любых методов рисования допустимо задавать тип и толщину используемых линий. **Тип линии** задается строкой, где пробелы означают отсутствие точек в линии, а любые другие символы — их наличие.

Пример

```
$картинка.line-style[***   ]
$картинка.line-width(2)
```

Для методов рисования будет использоваться пунктирная линия вида:
`***  ***  ***  ***  ***`
толщиной в два пикселя.

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 углами, измеряемыми 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 пикселей.

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 x 100 пикселей, внутри которого находится белый прямоугольник 90 x 20 пикселей, нарисованный по заданным координатам.

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 пикселей, нарисованной линией белого цвета с центром в точке (50;50).

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 расстановке значков на карте. В качестве параметров методу передаются:

- исходное **изображение**;
- координаты (**x1**;**y1**) верхнего левого угла копируемого фрагмента;
- **ширина** и **высота** копируемого фрагмента;
- координаты (**x2**;**y2**), по которым будет вставлен копируемый фрагмент;
- опционально — 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 цветов при этом уменьшается, и наоборот (по умолчанию эта величина 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 20 x 30 пикселей 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.

fill. Закрашивание одноцветной области изображения

```
^картинка.fill(x;y;цвет)
```

Метод используется для перекрашивания участка изображения одного цвета в переданный цвет. Участок закрашивания и заменяемый цвет определяются точкой 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 x 100 пикселей, перечеркнутый по диагонали белой линией. Нижняя половина квадрата — черная, а верхняя закрашена желтым цветом.

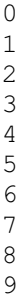
font. Загрузка файла шрифта для нанесения надписей на изображение

```
^картинка.font[набор_букв;имя_файла_шрифта.gif](ширина_пробела)
^картинка.font[набор_букв;имя_файла_шрифта.gif](ширина_пробела;ширина_символа)
^картинка.font[набор_букв;имя_файла_шрифта.gif][хеш с параметрами]   [3.4.0]
```

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 иметь формат GIF с прозрачным фоном и содержать изображение необходимого набора символов в приведенном ниже виде.

Пример файла `digits.gif` с изображением цифр:



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 пробела** в пикселях;
- **Ширина символа** – необязательный параметр.

Некоторые параметры могут быть переданы в третьем параметре в виде хеша:
$.space(0) – ширина пробела (по умолчанию равна ширине GIF со шрифтом);
$.width(x) – ширина символа для моноширинного шрифта (по умолчанию шрифт пропорциональный);
$.width(0) – включить моноширинный шрифт с автоматическим определением ширины символов (будет равна ширине GIF со шрифтом);
$.spacing(0) – межсимвольное расстояние (по умолчанию = 1).

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 будет определен шрифт для нанесения надписи, можно использовать метод **text** для нанесения надписей.

length. Получение длины надписи в пикселях

```
^картинка.length[текст надписи]
```

Метод вычисляет полную длину надписи в пикселях.

Пример

```
$square[^image::create(100;100;0x00FF00)]
^square.font[0123456789;digits.gif](0)
^square.length[128500]
```

В результате будет вычислена длина надписи **128500** в пикселях с учетом пробелов.

line. Рисование линии на изображении

```
^картинка.line(x0;y0;x1;y1;цвет)
```

Метод рисует на изображении линию заданного **цвета** от точки с координатами (**x0:y0**) до точки (**x1:y1**).

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 x 100 пикселей, перечеркнутый по диагонали белой линией.

pixel. Работа с точками изображения

```
^картинка.pixel(x;y)
```

Метод выдает цвет указанной точки изображения. Если координаты попадают за пределы изображения, выдает **-1**.

```
^картинка.pixel(x;y;цвет)
```

Задает **цвет** указанной точки.

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

```
$coordinates[^table::create{x y
0     0
50    100
100   0
}]

$square[^image::create(100;100;0x000000)]
^square.polybar(0x00FF00)[$coordinates]
$response:body[^square.gif[]]
```

В браузере будет выведен равнобедренный треугольник зеленого цвета на черном фоне. В таблице заданы координаты вершин треугольника.

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

```
$coordinates[^table::create{x y
0     0
50    100
100   0
}]

$square[^image::create(100;100;0x000000)]
^square.polygon(0x00FF00)[$coordinates]
$response:body[^square.gif[]]
```

В браузере будет выведен равнобедренный треугольник, нарисованный линией зеленого цвета на черном фоне. В таблице заданы координаты вершин треугольника.

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

```
$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)[$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 ломаной 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**.

rectangle. Рисование незакрашенных 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 x 100 пикселей, внутри которого находится прямоугольник 90 x 20 пикселей, нарисованный линией белого цвета по заданным координатам.

replace. Замена цвета в области, заданной таблицей координат

^картинка.replace(старый цвет;новый цвет)[таблица с координатами точек]
^картинка.replace(старый цвет;новый цвет) ***[3.4.1]***

Метод используется для замены одного цвета другим в области изображения, заданной с помощью таблицы координат. Если таблица с координатами не указана, то замена цвета производится во всем изображении.

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 вписанным в него розовым прямоугольником. Поскольку в методе **replace** задана замена только черного цвета розовым, белые линии не перекрасились.

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 пикселей. Сектор нарисован линией белого цвета.

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 имеет координаты (5;5).

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]
```

Метод преобразует в число переданную строку с IP-адресом. Метод аналогичен функции **inet_aton** MySQL-сервера и Perl.

Пример

```
^inet:aton[10.0.0.2]
```

— получаем число **167772162**.

hostname. Имя хоста

```
^inet:hostname[]
```

Метод возвращает текущее имя хоста (host name) — сетевой идентификатор узла, на котором выполняется программа.

ip2name. Определение домена по IP-адресу

```
^inet:ip2name[IP-адрес]
^inet:ip2name[IP-адрес;опции]
```

Метод возвращает доменное имя, соответствующее указанному IP-адресу. Переданная строка с IP-адресом будет преобразована в строку с доменным именем, соответствующим этому имени. Поддерживаются кириллические домены.

Поддерживаемые опции:

	Описание
`$.ipv[4/6/any]`	**4** — обрабатывать только IPv4-адреса (по умолчанию) **6** — обрабатывать IPv6-адреса **any** — обрабатывать любые адреса

Пример
`^inet:ip2name[91.197.112.64]` возвращает **test.артлебедев.рф**.

name2ip. Определение IP-адреса домена

```
^inet:name2ip[доменное.имя]
^inet:name2ip[доменное.имя;опции]
```

Метод возвращает IP-адрес для указанного доменного имени. Переданная строка с доменным именем будет преобразована в строку с IP-адресом, соответствующим этому имени. Поддерживаются кириллические домены.

Поддерживаемые опции:

	Описание
`$.ipv[4/6/any]`	**4** — обрабатывать только IPv4-адреса (по умолчанию); **6** — обрабатывать IPv6-адреса; **any** — обрабатывать любые адреса.
`$.table(true|false)`	Получить результат в виде строки с IP-адресом или в виде таблицы с колонками **ip** и **version**, которые содержат все IP-адреса, соответствующие имени, и их типы.

Простой пример
`^inet:name2ip[parser.ru]` возвращает **195.218.200.16**.

Пример
```
^inet:name2ip[test.артлебедев.рф; $.table(true)]]
^t.sort{$t.ip}
^t.menu{$t.ip $t.version
}}
```

Получаем:

```
91.197.112.64 4
91.197.112.65 4
::1 6
```

ntoa. Преобразование числа в строку с IP-адресом

```
^inet:ntoa(число)
```

Метод преобразует в число переданную 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 передаче 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 влияют 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.

json (класс)

Класс для работы с JSON (JavaScript Object Notation).

JSON — это альтернатива традиционным форматам (обычному тексту или XML), которые используются при обмене данными между сервером и клиентом. В отличие от XML и XML-совместимых языков, которые требуют синтаксического анализа, определения JSON могут быть просто включены в сценарии JavaScript. JSON также является текстовым форматом обмена данными и может довольно легко читаться людьми.

Статические методы

parse. Преобразование JSON-строки в хеш

^json:parse[JSON-строка;опции преобразования]

Метод преобразует JSON-строку в хеш.

Опции преобразования — хеш, в котором можно указать опции, рассмотренные ниже.

	По умолчанию	Описание
$.depth(число**)**	19	Максимальная глубина
$.double(true|false**)**	true	Преобразовывать вещес указано значение **false**
$.int(true|false**)**	true	Преобразовывать целые **false**, то числовые знач
$.distinct[first|last|all**]**	не определен	Способ обработки дубли **first** — будет оставлен **last** — будет оставлен г **all** — в результат попад со второго, получат суфо **по умолчанию** — в случ выдано исключение (exc
$.object[ссылка на метод**]**	не определен	Опция позволяет указати который будет вызывать объект (в том числе пол в результирующую струк
$.array[array|hash|ссылка на метод**]**	**$json:array**, значение которого по умолчанию **array**, можно поменять на **hash** ***[3.5.0]***	Опция позволяет указати который будет вызывать объект (в том числе пол в результирующую струк При задании значения **a** создается хеш с цифров
$.taint[язык преобразования**]**	не определен	Задает язык преобразов

Если JSON пришел из внешнего источника, то при его разборе необходимо обозначить доверие данным, например через **^taint[clean;$form:json]**.

Пример

```
@main[]
$json_string[{
      "a1":{"b": 1, "c": "abc", "d": "xyz"},
      "a2":{"b": 1.1, "b": 2.2, "b": 3.3, "d": {"da": 11, "db": 22}}
}]

$h[^json:parse[$json_string;
      $.double(false)
      $.distinct[all]
      $.object[$object_handler]
]]

@object_handler[key;value]
$result[^if($key eq "d"){object with key='$key' and ^eval($value)
fields}{$value}]
```

В результате разбора указанной JSON-строки хеш будет содержать:

```
$h[
      $.a1[
            $.b(1)
            $.c[abc]
```

```
            $.d[xyz]
      ]
      $.a2[
            $.b[1.1]
            $.b_2[2.2]
            $.b_3[3.3]
            $.d[object with key='d' and 2 fields]
      ]
]
```

string. Преобразование объекта Parser в JSON-строку

^json:string[*объект*;*опции преобразования*]

Метод преобразует системный или пользовательский объект в JSON-строку. По умолчанию объект пользовательского класса преобразуеытся в хеш.

Опции преобразования — хеш, в котором можно указать опции, рассмотренные ниже. ы

	По умолчанию	Описание
$.skip-unknown(true|false)	false	При указании значения **true** вместо exce в результирующую JSON-строку будут выд значения **null** при сериализации объекто отличными от **void**, **bool**, **string**, **int**, **d**
$.indent**(true\|false) **$.indent[строка]	false	При указании значения **true** будет включе форматирование результирующей JSON-с символами табуляции по глубине вложен Можно указать строковое значение, котор использоваться как префикс при формиро с отступами. ***[3.4.3]***
$.date[sql-string|gmt-string|iso-string|unix-timestamp**]**	sql-string	Опция определяет вид, в котором значени класса **date** будут попадать в результиру строку (см. одноименные методы объекта
$.void[null|string**]**	null	Опция определяет вид, в котором значени класса **void** будут попадать в результиру строку. По умолчанию выводится null, но в выдача в виде пустой строки. ***[3.4.4]***
$.table[object|array|compact**]**	object	Опция определяет вид, в котором значени класса **table** будут попадать в результиру строку. **object:** **[{"col1":"val11","col2":"val12",.** **{"col1":"val21","col2":"val22",..** **array: [["col1","col2",...] || nu** nameless-таблиц)**,["val11","val12",.** **compact: ["value11" || ["val11",**
$.array[compact|array|object**]** ***[3.5.0]***	compact	Опция определяет вид, в котором значени класса **array** будут попадать в результиру строку. **compact: ["v0","v2",...]**; **array: ["v0",null,"v2",...]**; **object: {"0":"v0","2":"v2",...}**.
$.file[text|base64|stat**]**	не определена	Опция определяет вид, в котором значени **file** будут попадать в результирующую JS Если опция не определена или ее значение в результирующую JSON-строку попадает и об объектах класса **file** (имя файла, разм `type`, `mode`), но их содержимое не попада
$.xdoc[параметры преобразования в текст**]**	не определена	Опция преобразования объекта класса **xdo** в строку ***[3.4.2]***
$.класс[ссылка на метод**]**	не определена	Любой класс (включая вышеупомянутые **da** и **file**) можно вывести, применяя пользо метод, который должен принимать три пар объект и опции вызова **^json:string[]** нужно для рекурсивного вывода пользова объектов). Поиск методов происходит во всех родите классах. ***[3.4.2]***
$._default[ссылка на метод**]**	не определена	Если опция определена, то метод будет вы вывода всех объектов пользовательских кл тех, вывод которых явно задан с помощью опции **$.класс[**ссылка на метод**]**). Мето принимать три параметра: ключ, объект и
$._default[название метода**]**	не определена	Если опция определена и метод с указанны у объекта пользовательского класса, мето для вывода объекта (кроме объектов тех к которых явно задан с помощью опции **$.к** на метод**]**). Метод должен принимать два ключ и опции вызова. ***[3.4.4]***

Пример

```
@main[]
$h[
	$.void[]
	$.bool(true)
	$.double(1/2)
	$.string[русские буквы]
	$.hash[
		$.e[ee]
	]
	$.date[^date::create(2006;08;18;06;09;00)]
	$.table[^table::create{c1	c2	c3^#0Av1	v2	v3^#0Av4	v5	v
6}]
	$.file[^file::create[text;zigi.txt;file-content]]
	$.img[^image::create(100;100;0)]
]
^json:string[$h;
	$.indent(true)
	$.table[array]
	$.file[base64]
	$.image[$image_handler]
]

@image_handler[key;value;params]
"custom value of image $key"
```

В результате выполнения будет выведено:

```
{
	"void":"",
	"bool":true,
	"double":0.5,
	"string":"русские буквы",
	"hash":{
		"e":"ee"
	},
	"date":"2006-08-18 06:09:00",
	"table":[
		["c1","c2","c3"],
		["v1","v2","v3"],
		["v4","v5","v6"]
	],
	"file":{
		"class":"file",
		"name":"zigi.txt",
		"size":12,
		"content-type":"text\/plain",
		"mode":"text",
		"base64":"ZmlsZS1jb250ZW50"
	},
	"img":"custom value of image img"
}
```

mail (класс)

Класс предназначен для работы с электронной почтой. Описание настройки Parser для работы этого класса см. п. «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**
- **print-debug** ***[3.4.0]***

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 письмо 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 будут переданы 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, предусмотрена возможность задать все поля заголовка сообщения, передав их в хеше, как у **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]
    $.format[uue|base64]
    $.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 формате uuencode (uue) до версии **3.4.0** и в формате base64, 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 отправлено сообщение для **petya@hotmail.ru** с содержимым «Как у тебя дела? У меня — изумительно!» в простом тексте и HTML. Сообщение будет содержать два изображения: `perfect_life1.jpg` — в виде аттачмента и `perfect_life2.jpg` — в теле письма.

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 разрядности для класса **double**.

Статические поля

Число пи

$math:PI — число π
$math:E — число e

Статические методы

abs, sign. Операции со знаком

Методы 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**

convert. Конвертирование из одной системы счисления в другую

^math:convert[число](исходная система счисления;целевая система счисления)
^math:convert[число|файл](исходная система счисления;целевая система счисления)[опции] ***[3.4.6]***
^math:convert[число|файл][алфавит](целевая система счисления)[опции] ***[3.4.6]***
^math:convert[число|файл](исходная система счисления)[алфавит][опции] ***[3.4.6]***
^math:convert[число|файл][алфавит][алфавит][опции] ***[3.4.6]***

Метод преобразует строчное представление числа (в том числе в виде двоичного файла) из одной

системы счисления в другую. Система счисления может быть задана алфавитом из как минимум двух символов, числом от 2 (эквивалентно алфавиту **01**) до 16 (эквивалентно алфавиту **0123456789ABCDEF**), числом 256 (эквивалентно алфавиту из всех ASCII-символов).

Можно задать хеш опций:
$.format[string|file] — формат результата, по умолчанию — строка.

Поддерживаются числа в диапазоне:
- 32 бита, до 0xFFFFFFFF;
- 64 бита, до 0xFFFFFFFFFFFFFFFF; ***[3.4.4]***
- произвольной разрядности. ***[3.4.6]***

Примеры

^math:convert[255](10;16) — получаем FF.

^math:convert[A](256;10) / ^math:convert[A](256;16) — получаем ASCII-код символа **A** в десятеричном и шестнадцатеричном представлении (65 / 41).

^math:convert[hello](256)[0123456789abcdefghijklmnopqrstuvwxyz] — получаем представление строки **hello** в Base36 (**5pzcszu7**).

^math:convert[5pzcszu7][0123456789abcdefghijklmnopqrstuvwxyz](256) — декодируем строку **hello** из ее Base36-представления.

crc32. Подсчет контрольной суммы строки

^math:crc32[строка]

Метод подсчитывает контрольную сумму (CRC32) для переданной строки. Затем выдает ее в виде целого числа.

crypt. Хеширование паролей

^math:crypt[password;salt]

Метод 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 его случайным образом.

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 усложняет подбор пароля даже при использовании специальных аппаратных ускорителей.

Внимание: обязательно нужно задавать случайное тело **`salt`**, либо это может сделать Parser, если указывать не тело **`salt`**, а только его начало.

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)	см. документацию к операционной системе, `man 5 crypt`, функция `crypt`	см. документацию к операционной системе, `man 5 crypt`, функция `crypt`

Внимание: в Parser, чтобы использовать в тексте символ **`$`**, *перед ним необходимо поставить символ* **`^`** *—* **`^$`**.

Примечание: веб-сервер Apache позволяет 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 по ссылке: ietf.org/rfc/rfc1321.txt

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.

digest. Криптографическое хеширование

^math:digest[алгоритм;строка или файл; $.format[hex|base64] $.hmac[ключевая строка]]
^math:digest[алгоритм;строка или файл; $.format[hex|base64|file] $.hmac[ключевая строка|ключевой файл]] ***[3.5.0]***

Метод объединяет возможность работы с разными алгоритмами криптографического хеширования. Хеширование применяется к переданной **строке** или **файлу**.

Поддерживаются следующие **алгоритмы**: **md5**, **sha1**, **sha256**, **sha512**. Результирующий хеш, в зависимости от опции **$.format**, преобразуется в шестнадцатеричное представление (по умолчанию) или строку в формате Base64 или выдается в виде бинарного файла ***[3.5.0]***.

Опция **$.hmac[ключевая строка|ключевой файл]** предназначена для проверки целостности переданных данных на основе секретного ключа и хеш-функций (HMAC),

exp, log, log10. Логарифмические функции

Методы вычисляют значения логарифмических функций от заданного **числа**.

^math:exp(число) – Экспонента **числа** по основанию **e**
^math:log(число) – Натуральный логарифм **числа**
^math:log10(число) – Десятичный логарифм **числа**

Примечание: логарифм по произвольному основанию base вычисляется как log(число)/log(base).

md5. MD5-отпечаток строки

^math:md5[строка]

Метод получает «отпечаток» размером 16 байт из переданной **строки**. Выдает его представление в виде строки – байты представлены в шестнадцатеричном 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 по ссылке: ietf.org/rfc/rfc1321.txt

pow. Возведение числа в степень

^math:pow(число;степень)

Метод возводит число в степень.

Пример
^math:pow(2;10) – получаем 1024 (2^{10}=1024).

random. Случайное число

^math:random(верхняя_граница)

Метод возвращает случайное число, попадающее в интервал от 0 до заданного числа, не включая заданное.

Примечание: в некоторых операционных системах метод выдает псевдослучайное число.

Пример
^math:random(1000)

Получим случайное число из диапазона от 0 до 999.

round, floor, ceiling. Округления

Методы возвращают округленное значение заданного числа класса **double**.

^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.

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**

sha1. Хеш строки по алгоритму SHA1

^math:sha1[строка]

Метод вычисляет хеш по алгоритму SHA1 для переданной **строки**.

sin, asin, cos, acos, tan, atan, atan2. Тригонометрические функции

^math:sin(радианы) – синус
^math:asin(число) – арксинус
^math:cos(радианы) – косинус
^math:acos(число) – арккосинус
^math:tan(радианы) – тангенс
^math:atan(число) – арктангенс
^math:atan2(число;число) – четырехквадрантный арктангенс ***[3.5.0]***

Методы вычисляют значения тригонометрических функций от заданного **числа**.

Пример
^math:cos(^math:radians(180)) – получаем **-1** (cos π = -1).

sqrt. Квадратный корень числа

^math:sqrt(число)

Метод вычисляет квадратный корень **числа**.

Пример
^math:sqrt(16) – получаем **4**.

Примечание. Корень n-й степени вычисляется как возведение в степень 1/n.

trunc, frac. Операции с целой/дробной частью числа

^math:trunc(число) – метод возвращает целую часть числа.

^math:frac(число) – метод возвращает дробную часть числа.

Пример
^math:trunc(85.506) – получаем **85**
^math:frac(85.506) – получаем **0.506**

uid64. 64-битный уникальный идентификатор

^math:uid64[]
^math:uid64[опции] ***[3.4.6]***

Метод выдает случайную строку вида:
BA39BAB6340BE370

Примечание: в некоторых операционных системах метод выдает псевдослучайную строку.

Можно задать хеш опций:
$.lower(false|true) – выдавать результат в нижнем регистре, по умолчанию – в верхнем.
См. **^math:uuid[]**.

uuid. Универсальный уникальный идентификатор версии 4

^math:uuid[]
^math:uuid[опции] ***[3.4.6]***

Метод выдает случайную строку вида:
22C0983C-E26E-4169-BD07-77ECE9405BA5

UUID (также известен как GUID) удобно использовать, когда трудно обеспечить или вообще нецелесообразно использовать сквозную нумерацию объектов. Например, при распределенных вычислениях. При работе с базами данных эффективнее использовать UUID версии 7.

Можно задать хеш опций:
- **$.lower(false|true)** — выдавать результат в нижнем регистре, по умолчанию — в верхнем;
- **$.solid(false|true)** — исключать из результата символы '**-**', по умолчанию — не исключать.

Пример
В разных филиалах компании собираются заказы, которые периодически отправляются в центральный офис. Для обеспечения уникальности идентификатора заказа используется **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: метод создает UUID версии 4, основыванный на случайных числах, а не на времени. 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 версии 4 доступна по ссылке: https://www.rfc-editor.org/rfc/rfc9562.html

uuid7. Универсальный уникальный идентификатор версии 7

^math:uuid7[]
^math:uuid7[опции]

Метод выдает случайную строку вида:
0189FC1E-44E6-7000-A014-BF0A34996F90

```
0189FC1E-44E6-7001-87F3-31344DA88C26
```

В отличие от полностью случайного UUID версии 4, эта функция формирует значения в соответствии со стандартом UUID версии 7. В нем первые 16 символов основаны на времени и порядковом номере, благодаря чему каждый следующий UUID больше предыдущего. Это снижает нагрузку на B-деревья (B-tree) при использовании UUID в качестве ключей в базах данных.

UUID (*также известен как GUID*) удобно использовать, когда трудно обеспечить или вообще нецелесообразно использовать сквозную нумерацию объектов. Например, при распределенных вычислениях.

Можно задать хеш опций:
- **`$.lower(false|true)`** — выдавать результат в нижнем регистре, по умолчанию — в верхнем;
- **`$.solid(false|true)`** — исключать из результата символы '-', по умолчанию — не исключать.

memcached (класс)

Класс предназначается для работы с серверами `memcached` и использует библиотеку `libmemcached`.

Пример
Небольшой класс, реализующий функциональность, аналогичную функциональности оператора **`cache`**, но хранящий скешированные результаты на сервере `memcached`:

```
@main[]
$m[^mcache::open[localhost]]
^m.cache[key2;10]{dt: $d[^date::now[]] ^d.sql-string[] ^sleep(3)}


@CLASS
mcache


@auto[]
$timeout(4) ^rem{ timeout, seconds }
$retry_on_timeout(false) ^rem{ retry cache lock attempts }


@open[connect-options]
$m[^memcached::open[$connect-options]]


@cache[key;expires;code][lock;i]
$result[$m.$key]
^if(!def $result){
      ^rem{ not cached yet }
      $lock[${key}-lock]
      ^while(!^m.add[$lock; $.value[$timeout] $.expires($timeout)]){
            ^rem{ another process got the lock, waiting ... }
            ^for[i](1;$timeout*5){
                  ^sleep(0.2)
                  $result[$m.$key]
                  ^if(def $result){^break[]}
            }
            ^if(def $result){
                  ^break[]
            }{
                  ^if(!$retry_on_timeout){
                        ^throw[$self.CLASS_NAME;Timeout while getting lock
for key '$key']
                  }
```

```
            }
      }
      ^if(!def $result){
            ^rem{ we got the lock, processing the code }
            ^try{
                  $result[$code]
                  $m.[$key][ $.value[$result] $.expires($expires) ]
            }{}{
                  ^m.delete[$lock]
            }
      }
}
```

Конструктор

open. Открытие

```
^memcached::open[параметры соединения]
^memcached::open[параметры соединения](время хранения записей по умолчанию,
в секундах)
```

Пример

```
$memcached[^memcached::open[server1:port1,server2]]
```

Пример

```
$memcached[^memcached::open[
      $.server[server1:port1]
      $.binary-protocol(true)
      $.connect-timeout(5)
]]
```

Чтение

```
$memcached.ключ
```

Возвращает строку, ассоциированную с **ключом**, если эта ассоциация не устарела.

Запись

```
$memcached.ключ[значение]
$memcached.ключ[
      $.value[значение]
      ...необязательный модификатор...
]
```

Сохраняет на сервер ассоциацию между **ключом** и **строкой**.

Необязательный модификатор:
$.expires(число секунд) — задает число секунд, на которое сохраняется пара «**ключ** / **строка**», **0** = навсегда.

Методы

add. Добавление записи

```
^memcached.add[ключ;строка]
```

Если на сервере уже есть запись с указанным ключом, то метод ничего не делает и возвращает **false**. Если такой записи нет — метод сохраняет ее на сервере и возвращает **true**.

Обычно для записи данных нужно применять конструкцию **$memcached.[$key][$value]**

clear. Удаление всех данных с сервера

```
^memcached.clear[]
^memcached.clear(секунды)
```

Метод инициирует удаление всех данных с сервера (серверов). При вызове без параметра инициируется сиюминутное удаление всех данных, при вызове с параметром удаление будет произведено по истечении указанного количества секунд.

Метод удаляет данные с серверов в том порядке, в котором они были указаны в параметрах соединения.

delete. Удаление записи

```
^memcached.delete[ключ]
```

Метод удаляет с сервера пару «**ключ / значение**».

mget. Получение множества значений

```
^memcached.mget[ключ1;ключ2;ключ3;...]
^memcached.mget[таблица_с_одним_столбцом_содержащим_ключи]
```

Метод принимает от сервера все актуальные записи с указанными **ключами** и возвращает их в виде хеша.

release. Закрытие соединения с сервером

```
^memcached.release[]
```

Метод закрывает соединение с сервером. Для продолжения работы не требуется производить его повторного открытия: любое обращение к его элементам автоматически откроет соединение.

Параметры соединения

Параметры соединения с серверами `memcached` могут быть заданы как в виде **строки**, так и в виде **хеша**.

Если параметры соединения заданы в виде строки, то они передаются функции **memcached_servers_parse** библиотеки `libmemcached` «как есть». Данная функция ожидает строку соединения в следующем формате:
server1:port1,server2,server3,server4:port4

Чуть подробнее прочитать о ее параметрах можно в документации библиотеки `libmemcached`.

Если параметры соединения указаны в виде хеша, то они обрабатываются более новой и универсальной функцией **memcached** (которая тем не менее может отсутствовать у библиотеки, установленной в системе). Ключами хеша с параметрами соединения могут быть любые опции,

доступные у установленной в конкретной системе библиотеки **libmemcached** (см. документацию). Имена опций нужно писать без префикса «--».

Список наиболее востребованных опций:

```
$.server[<servername>:<port>]
$.binary-protocol(true)
$.connect-timeout(N)
$.tcp-keepalive(true)
```

memory (класс)

Класс предназначен для работы с памятью Parser. Его использование поможет экономить память в скриптах.

Статические методы

auto-compact. Автоматическая сборка мусора

```
^memory:auto-compact(частота сборки)
```

Метод задает режим автоматической сборки так называемого мусора. Мусором считается память, более не используемая кодом, т. е. та, на которую в коде нет ссылок.

Параметр, целое число от 0 до 5, определяет частоту автоматической сборки мусора:

- **0** — автоматическая сборка мусора выключена (по умолчанию, для сборки мусора надо вызывать `^memory:compact[]`);
- **1** — частота сборки минимальна (быстрее, но больший расход памяти);

...

- **5** — частота сборки максимальна (медленнее, но расход памяти минимален).

Для любознательных: в Parser используется известный и хорошо зарекомендовавший себя консервативный сборщик мусора Boehm-Demers-Weiser, см. hpl.hp.com/personal/Hans_Boehm/gc/.

Частые сборки мусора замедляют скорость выполнения кода на десятки процентов.

compact. Сборка мусора

```
^memory:compact[]
```

Метод собирает мусор в памяти, освобождая ее для повторного использования кодом. Мусором считается память, более не используемая кодом, т. е. та, на которую в коде нет ссылок.

Например,

```
$table[^table::sql{SQL-запрос}]
$table[]
# освободит память, занимаемую результатом выполнения SQL-запроса
^memory:compact[]
```

Parser по умолчанию не собирает мусор автоматически, полагаясь в данном вопросе на кодера: следует поставить вызов **compact** в той точке (точках), где ожидается максимальная выгода, например перед XSL-преобразованием.

`$status:memory` поможет в отладке и поиске мест, наиболее выгодных для сборки мусора.

Важно: необходимо как можно более интенсивно использовать локальные переменные и обнулить глобальные, которые не понадобятся для дальнейшей работы кода. Это поможет **compact** *освободить больше памяти.*

Важно: не гарантируется, что будет освобождена абсолютно вся неиспользуемая память.

reflection (класс)

Класс предназначен для получения информации о классах и их методах.

Статические методы

base. Родительский класс объекта

```
^reflection:base[класс]
^reflection:base[объект]
```

Метод возвращает базовый класс объекта или класса (если он есть) либо **void**.

base_name. Имя родительского класса объекта

```
^reflection:base_name[класс]
^reflection:base_name[объект]
```

Метод возвращает имя базового класса объекта или класса (если он есть) либо пустую строку.

class. Класс объекта

```
^reflection:class[объект]
```

Метод возвращает класс объекта (аналогично **$объект.CLASS**).

class_alias. Создание псевдонима класса

```
^reflection:class_alias[имя класса;новое имя класса]
```

Метод создает псевдоним класса по переданному имени. После создания псевдонима к классу можно обращаться как по новому, так и по исходному имени.

class_by_name. Получение класса по имени

```
^reflection:class_by_name[имя класса]
```

Метод возвращает класс по переданному имени, в случае отсутствия класса с указанным именем выдается исключение.

class_name. Имя класса объекта

```
^reflection:class_name[объект]
```

Метод возвращает имя класса объекта (аналогично **$объект.CLASS_NAME**).

classes. Список классов

```
^reflection:classes[]
```

Метод 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**.

copy. Копирование объекта

```
^reflection:copy[объект-откуда;объект-куда]
```

Метод осуществляет копирование всех полей указанного объекта.

create. Создание объекта

```
^reflection:create[имя класса;имя конструктора]
^reflection:create[имя класса;имя конструктора;параметры;конструктора]
^reflection:create[ $.class[имя класса] $.constructor[имя конструктора]
;параметры;конструктора]      [3.4.5]
^reflection:create[ $.class[имя класса] $.constructor[имя конструктора]
$.arguments[ $.1[параметры] $.2[конструктора] ] ]      [3.4.5]
```

Метод создает объект указанного **класса**, вызывая конструктор с указанным **именем**. Использовать этот метод удобно, если необходимо создать объект класса, имя которого находится в переменной. При передаче параметров через хеш значения ключей игнорируются, параметры передаются в порядке следования в хеше.

Замечание: передать конструктору можно не более 100 параметров.

def. Проверка существования класса

```
^reflection:def[class;имя класса]
```

При передаче методу в качестве параметров значения **class** и **имени класса** метод проверяет существование класса с указанным именем и возвращает результат «истина / ложь».

delete. Удаление поля объекта

```
^reflection:delete[объект;имя поля]
^reflection:delete[класс;имя поля]
```

Метод удаляет **поле** с указанным именем у указанного **объекта** или **класса**. Метод аналогичен методу **^хеш.delete[ключ]**, но работает для объектов и классов.

Пример

```
@main[][a;h]
$a[^a::create[]]
^reflection:delete[$a;b]

$h[^hash::create[$x]]
^h.foreach[k;v]{$k='$v'}[, ]


@CLASS
a

@create[]
```

```
$a[1]
$b[2]
$c[3]
```

Вернет:
a='1', c='3'

dynamical. Тип вызова метода

```
^reflection:dynamical[]
^reflection:dynamical[класс]
^reflection:dynamical[объект]
```

При вызове без параметров метод 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.

field. Получение значения поля объекта

```
^reflection:field[объект;имя поля]
^reflection:field[класс;имя поля]
```

Метод возвращает поле объекта или класса.
Работает с пользовательскими и некоторыми системными классами.

Внимание: поиск полей по иерархии классов не производится.

fields. Список полей объекта

```
^reflection:fields[класс]
^reflection:fields[объект]
```

Для **класса** метод возвращает хеш со списком статических полей, для **объекта** — со списком динамических полей.

fields_reference. Ссылка на поля объекта

```
^reflection:fields_reference[объект]
```

Метод возвращает специальный ссылающийся хеш, непосредственно связанный с полями **объекта**. При добавлении, изменении или удалении элементов этого хеша такие же изменения произойдут с полями объекта, на который он ссылается, и наоборот, изменение полей объекта отражается в ссылающемся хеше. Ссылающийся хеш отличается от обычного еще и отсутствием **$_default**.

Замечание: использование метода **^reflection:fields_reference[$n]** *для получения списка полей объекта эффективнее, чем* **^reflection:fields[$n]** *и* **^hash::create[$n]**.

filename. Получение имени файла

```
^reflection:filename[класс, или объект, или метод]
```

Метод возвращает полный дисковый путь к файлу, в котором определен класс или метод. Для объекта возвращается путь к файлу, в котором определен его класс.

Замечание: в случае partial-классов возвращается путь к первому файлу, в котором определен класс.

is. Проверка типа

```
^reflection:is[имя элемента;тип]
^reflection:is[имя элемента;тип;контекст]
```

Метод возвращает результат «истина / ложь» в зависимости от того, относится ли элемент с указанным именем к заданному типу.

Метод расширяет функциональность оператора **is**, позволяя проверить, является ли параметр кодом. Для проверки того, является ли параметр кодом (передается в фигурных или круглых скобках), нужно указать в качестве **типа** специальное значение **code**. Для проверки того, является ли параметр ссылкой на метод, нужно указать в качестве **типа** специальное значение **method**.

По умолчанию контекстом является контекст вызова метода **is**. Если метод принимает неопределенное число параметров, в качестве контекста необходимо указать переменную, в которой они переданы.

Проверка типа параметра

```
@main[]
^method[string]
^method{code}
^method[$method]
^another-method[$method]

@method[param]
^if(^reflection:is[param;junction]){
    Param is ^if(^reflection:is[param;code]){code}{method reference}
}{
    Param is not code or method reference
}

@another-method[*params]
^if(^reflection:is[0;method;$params]){
    First param is method reference
}
```

method. Получение метода объекта

```
^reflection:method[объект;имя метода]
^reflection:method[класс;имя метода]
```

Метод возвращает метод **объекта** или **класса**. Может быть использован в пользовательских классах, где приоритет доступа к полям выше, чем к методам с тем же именем.

```
^reflection:method[метод]        [3.4.5]
^reflection:method[метод;объект]        [3.4.5]
```

Привязывает **метод** к вызывавшему его объекту или классу либо к переданному вторым параметром

объекту или классу. В Parser все методы привязаны к контексту исполнения (self), и таким образом можно поменять эту привязку.

Пример

```
@main[]
$a[^A::create[]]

# ^a.m[] - метод m не может использоваться напрямую, т. к. одноименное поле m имеет больший приоритет
# поэтому используем ^reflection:method[], чтобы добраться до метода m

$method[^reflection:method[$a;m]]
^method[]

$b[^B::create[]]

# подменяем self, чтобы вызвать метод m в контексте другого объекта, сохраняем результат в объекте b
$b.m[^reflection:method[$method;$b]]

# теперь в объекте b тоже есть метод m
^b.m[]

@CLASS
A

@create[]
$name[object of class A]
$m[object field]

@m[]
method of class A, called on $name

@CLASS
B

@create[]
$name[object of class B]
```

Выведет:

```
method of class A, called on object of class A
method of class A, called on object of class B
```

method_info. Информация о методе

^reflection:method_info[имя класса;имя метода]
^reflection:method_info[метод] ***[3.4.5]***

Метод возвращает хеш с параметрами либо указанного метода указанного класса, либо указанного метода.

Для методов системных классов возвращается хеш следующего вида:

```
$хеш[
    $.inherited[имя класса-предка, в котором метод был определен]
    $.min_params(минимально необходимое количество параметров метода)
```

```
$.max_params(максимально 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-предка, в котором метод был определен]
    $.overridden[имя класса-предка, в котором был определен перекрытый
метод]      [3.4.1]
    $.file[полный путь к файлу, в котором определен метод]  [3.4.1]
    $.max_params(максимально допустимое количество параметров метода)    [3.4.3]
    $.call_type[допустимый тип вызова метода: static, dynamic или any]  [3.4.3]
    $.extra_param[имя входной переменной, принимающей неограниченное число
параметров] [3.4.3]
    $.named_params[массив имен именованных параметров]      [3.5.0]
    $.0[имя первого параметра метода]
    $.1[имя второго параметра метода]
    ...
]
```

methods. Список методов класса

```
^reflection:methods[имя класса]
^reflection:methods[class name; $.reverse(true|false) ]      [3.4.5]
```

Метод возвращает хеш со всеми методами указанного **класса**. Ключами хеша являются имена методов, значениями — строки **native** (для системных классов) или **parser** (для классов, созданных пользователем).

Хеш отсортирован в порядке, обратном порядку добавления методов (последний добавленный метод будет первым). С помощью опции **$.reverse(false)** можно задать порядок элементов, соответствующий порядку добавления. **[3.4.5]**

mixin. Дополнение типа

```
^reflection:mixin[источник;опции]
```

Метод копирует в класс методы и поля другого класса.

Можно задать хеш опций.

- **$.to[получатель]** — Класс, в который будут копироваться методы и поля источника. По умолчанию — класс, из которого вызвали **mixin**.
- **$.name[имя]** — Копировать только метод или поле с указанным именем. По умолчанию — копируется все.
- **$.methods(true|false)** — Копировать ли методы класса-источника. По умолчанию — копировать.
- **$.fields(true|false)** — Копировать ли статические поля класса-источника. По умолчанию — копировать.
- **$.overwrite(false|true)** — Перезаписывать ли одноименные методы и поля класса-получателя. По умолчанию — не перезаписывать.

Пример

```
@CLASS
B

@auto[]
^reflection:mixin[$A:CLASS; $.fields(false) ]
```

При загрузке класса B скопирует в него методы класса A.

stack. Стек вызовов методов

```
^reflection:stack[опции]
```

Метод возвращает текущее состояние стека вызовов методов на Parser. Для каждого стекового кадра возвращается хеш, содержащий **self**, имя вызванного метода, имя файла и строку, в которой определен метод.

Можно задать хеш опций.

- **$.args(false|true)** — Дополнительно создавать хеш **args**, содержащий переданные методу параметры. По умолчанию — не создавать.
- **$.locals(false|true)** — Дополнительно создавать хеш **locals**, содержащий локальные переменные метода. По умолчанию — не создавать.
- **$.limit(n)** — Ограничить число возвращаемых стековых кадров. По умолчанию — возвращаются все.
- **$.offset(n)** — Возвращать стековые кадры, начиная с указанного. По умолчанию — возвращаются начиная с первого.

Пример

```
@example[value]
^json:string[^reflection:stack[ $.args(true) ]; $.indent(true) ]

@main[]
^example[some value]
```

Выведет:

```
{
	"1":{
		"self":{},
		"name":"example",
		"file":"filename.html",
		"line":1,
		"args":{
			"value":"some value"
		}
	},
	"2":{
		"self":{},
		"name":"main",
		"file":"filename.html",
		"line":4,
		"args":{}
	}
}
```

tainting. Преобразования строки

```
^reflection:tainting[строка]
^reflection:tainting[вид преобразования;строка]
```

Метод позволяет узнать, в каких преобразованиях нуждается **строка**. Результатом является строка, в которой каждому символу исходной строки соответствует символ с кодом преобразования. При указании **вида преобразования** с помощью **+** выделяются символы, подлежащие преобразованию

указанного вида. Кроме имени преобразования, можно указать значение **tainted** для показа неопределенно грязных символов и **optimized** для показа символов, нуждающихся в оптимизации при выводе.

Коды преобразований

clean	0
as-is	A
tainted	T
file-spec	F
uri	U
http-header	h
mail-header	m
sql	Q
js	J
json	S
parser-code	p
regex	R
xml	X
html	H
cookie	C

Пример

```
$s[clean ^taint[<tainted>] ^taint[uri;&] ^taint[json;"json"]]

^taint[as-is;$s]
^reflection:tainting[$s]
^reflection:tainting[tainted;$s]

Applied: $s
```

Выведет:

```
clean <tainted> & "json"
000000TTTTTTTT0U0SSSSSS
------+++++++++---------

Applied: clean &lt;tainted&gt; %26 \"json\"
```

uid. Уникальный идентификатор объекта

```
^reflection:uid[объект]
```

Метод возвращает уникальный идентификатор **объекта**.

regex (класс)

Класс предназначен для работы с *регулярными выражениями*, совместимыми с PCRE (Perl Compatible Regular Expressions). Частичный перевод описания PCRE приведен в «Приложение 4. Perl-совместимые регулярные выражения».

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 «Приложение 4. Perl-совместимые регулярные выражения».

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;
' — 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`) используется

класс **form**.

Часть информации о запросе доступна через переменные окружения, см. «Получение значения поля запроса».

Статические поля

argv. Аргументы командной строки

$request:argv

Хеш, содержащий аргументы командной строки с ключами 0, 1, 2 и т. д., которым может быть удобно пользоваться, если Parser задействуется в качестве интерпретатора скриптов (например, при запуске из cron).

$request:argv.0 — содержит имя обрабатываемого файла.

body. Получение текста запроса

$request:body

Получение текста POST-запроса.

Вариант использования: можно написать свой сервер XML-RPC (см. xmlrpc.com).

body-charset, post-charset. Получение кодировки пришедшего POST-запроса

$request:post-charset
$request:body-charset ***[3.4.4]***

Если в HTTP-заголовке content-type пришедшего POST-запроса содержится информация о кодировке, то ее 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-file, post-body. Тело содержимого запроса

$request:post-body
$request:body-file ***[3.4.4]***

Получение содержимого POST-запроса в виде файла.

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».

document-root. Корень веб-пространства

$request:document-root[/дисковый/путь/к/корню/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».

headers. Получение заголовков HTTP-запроса

$request:headers

Возвращает хеш с заголовками, с которыми был сделан HTTP-запрос (переменные окружения, начинающиеся с **HTTP_**).

Пример

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

Поля запроса имеют имена в верхнем регистре.

method. Получение метода HTTP-запроса

$request:method

Возвращает метод, которым был сделан HTTP-запрос (GET, POST или PUT).

path. Получение пути запроса

$request:path

Возвращает путь из URI, то есть декодированную часть URI, начинающуюся с символа / и заканчивающуюся перед символом ? (если он присутствует).

Пример

Предположим, пользователь запросил такую страницу:
`HTTP://www.mysite.ru/some%20news/articles.html?year=2000&month=05&day=27`

Тогда:

```
$request:path
```

вернет:

```
/some news/articles.html
```

query. Получение параметров 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) используется класс **form**.

Пример
Предположим, пользователь запросил такую страницу:
HTTP://www.mysite.ru/some%20news/articles.html?year=2000&month=05&day=27

Тогда:
$request:query

вернет:
year=2000&month=05&day=27

uri. Получение URI запроса

$request:uri

Возвращает URI запрошенного документа.

Пример
Предположим, пользователь запросил такую страницу:
HTTP://www.mysite.ru/some%20news/articles.html?year=2000&month=05&day=27

Тогда:
$request:uri

вернет:
/some%20news/articles.html?year=2000&month=05&day=27

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.

Примечания
Прежде чем будет задано или считано значение, имя поля преобразуется в верхний регистр. ***[3.4.4]***
При выдаче браузеру имя HTTP-заголовка приводится к формату, в котором первые буквы слов делаются заглавными, а остальные — строчными. Например, **CONTENT-TYPE** *будет преобразован в* **Content-Type**). ***[3.4.0]***
При задании пустого значения поле удаляется. ***[3.4.4]***
При задании **$response:status** *значения меньше 100 это значение будет возвращено в виде кода выхода процесса Parser.* ***[3.4.5]***

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 заголовку expires значения «завтра»

```
$response:expires[^date::now(+1)]
```

body. Задание нового тела ответа

$response:body[DATA]

Замещает все тело ответа значением **DATA**.

DATA — строка, файл или хеш параметров.

Ключи хеша параметров:
file — имя файла на диске (в этом случае Parser поддерживает докачку файлов);
name — имя файла, которое нужно передать посетителю;
mdate — дата и время изменения файла, которые нужно 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 ответа результатом работы скрипта

```
$response:body[^file::cgi[script.cgi]]
```

Весь ответ будет заменен 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**.

charset. Задание кодировки ответа

$response:charset[кодировка]

Задает **кодировку** ответа. После обработки запроса результат переводится 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».

download. Задание нового тела ответа

```
$response:download[DATA]
```

Идентичен **$response:body**, но выставляет флаг, который браузер воспринимает как «Предложить посетителю сохранить файл на диске».

Имя файла передается браузеру в заголовке Content-Disposition. С версии 3.5.0 в него добавлено поле filename* для поддержки имен файлов в кодировке UTF-8.

Браузеры умеют отображать файлы некоторых типов прямо внутри своего окна (например DOC- и 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 или открыть файл.

headers. Заданные заголовки HTTP-ответа

```
$response:headers
```

Возвращает хеш со всеми заголовками HTTP-ответа, которые были заданы в коде на данный момент. Заголовки имеют имена в верхнем регистре. ***[3.4.4]***

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.

Статический метод

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 позволяет находить в скриптах участки кода, на которые затрачивается значительная доля процессорного времени или памяти.

Если Parser используется как модуль Apache, при попытке использовать класс **status** сообщение `class not found` будет выводиться до тех пор, пока в `httpd.conf` не будут добавлены строки:

```
<Location />
# Разрешает использовать встроенный класс status
ParserStatusAllowed
</Location>
```

...а Apache — не перезапущен.

Поля

memory. Информация о памяти под контролем сборщика мусора

$status:memory — хеш с информацией о распределении памяти (в килобайтах), находящейся под контролем сборщика мусора.

Поле	Значение	Детали
used	Объем занятой памяти	В это число не включен размер служебных данных самого сборщика мусора.
free	Объем свободной памяти	Свободная память скорее всего фрагментирована.
ever_allocated_since_compact	Объем выделенной памяти 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.
ever_allocated_since_start	Объем выделенной памяти 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
С помощью временного добавления вызовов

```
^musage[before XXX]
^musage[after XXX]
```

до и после интересующего участка кода представленного ниже 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[]
```

формируется журнал, который зафиксирует потребление ресурсов.

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.

log-filename. Путь к журналу ошибок

$status:log-filename

В случае возникновения необработанной ошибки при исполнении скрипта Parser делает запись в журнал ошибок parser3.log, включающую дату и время, запрошенную страницу, место возникновения ошибки и сообщение об ошибке. Поле возвращает дисковый путь к журналу ошибок, по умолчанию он расположен в том же каталоге, где и сам интерпретатор Parser. Если у Parser нет возможности сделать запись в данный файл, об ошибке будет сообщено в стандартный поток ошибок.

mode. Режим работы

$status:mode

Возвращает режим, в котором работает Parser. Возможные значения: cgi, console, mail, httpd, apache, isapi.

pid. Идентификатор процесса

$status:pid

Идентификатор процесса (process) операционной системы, в котором работает Parser.

rusage. Информация о затраченных ресурсах

$status:rusage — хеш с информацией о ресурсах сервера, затраченных на данный момент системой на обработку Parser-скрипта. Точность измерения времени в микросекундах зависит от используемой операционной системы.

Ключ	Единица	Описание значения	Как уменьшить?
utime	секунды	Чистое время, затраченное текущим процессом (не включает время выполнения других задач)	Упростить манипуляции с данными внутри Parser (улучшить алгоритм, переложить часть действий на SQL-сервер).
stime	секунды	Время считывания системой пользовательских файлов, каталогов и библиотек	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 ненужны записи, отфильтровать как можно больше средствами самого SQL-сервера. Упростить SQL-запросы, воспользовавшись оператором **EXPLAIN**; для Oracle – **EXPLAIN PLAN** (см. документацию по серверу); для других SQL-серверов – см. их документацию.
		Точное системное время (позволяет оценить временные затраты на ожидание ответа от SQL-, HTTP-, SMTP-серверов) Сколько прошло с Epoch...	
tv_sec	секунды	...целых секунд;	
tv_usec	микросекунды (10E-6)	...еще прошло микросекунд (миллионных долей секунды)	

Рекомендуемый способ анализа
С помощью временного добавления в конец скрипта вызова

```
^rusage[total]
```

представленного ниже 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[]
```

формируется журнал, который зафиксирует потребление ресурсов.

Для более точного анализа следует добавить вызовы

```
^rusage[before XXX]
^rusage[after XXX]
```

до и после интересующего участка кода.

Примечание: для записи журнала не рекомендуется использовать веб-пространство.

Windows
Под ОС семейства Windows доступен ряд дополнительных значений:

Ключ	Единицы	Описание значения	Как уменьшить
ReadOperationCount **ReadTransferCount**	штук байт	Количество операций чтения с диска и суммарное количество считанных байтов	Уменьшить количество и размер файлов, необходимых для работы, 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 байтов	
OtherOperationCount **OtherTransferCount**	штука байт	Количество других операций с диском (не чтения / записи) и суммарное количество переданных байтов	
PeakPagefileUsage **QuotaPeakNonPagedPoolUsage** **QuotaPeakPagedPoolUsage**	байт	Максимальное количество памяти в файле подкачки (swap-файле)	См. комментарий к ключу **maxrss** выше.

tid. Идентификатор потока

$status:tid

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 в **double**. Если строка пуста, ее числовое «значение» в математических выражениях считается нулем.

Создание объекта класса **string**:
```
$str[Строка, которая содержится в объекте]
```

Для совместимости с пустым хешем пустая и пробельная строки допускают обращение к произвольным полям (**$str.key**) без сообщения об ошибке. ***[3.4.5]***
Ниже приведен пример кода, показывающего, когда такая возможность позволяет обойтись без дополнительных проверок.

```
^method[
   ^if($condition1){ $.option1[value1] }
   ^if($condition2){ $.option2[value2] }
]

@method[options]
^if(def $options.option1){ code }
```

Если оба условия будут ложны, то в качестве опций в метод будет передан не хеш, а строка, состоящая из пробельных символов. Тем не менее благодаря совместимости с пустым хешем код будет работать так, как задумано.

Статические методы

base64. Декодирование из Base64

```
^string:base64[закодированное]
^string:base64[закодированное;опции]
```
[3.4.2]

Замечание: именно метод, не конструктор!

Метод декодирует строку из Base64-представления. Для кодирования строки следует использовать **^строка.base64[]**

Можно задать хеш опций.

- **$.strict(true)** — Будет выдаваться исключение при невозможности декодирования *всех* символов. Без указания данной опции файл будет создан из того, что было успешно декодировано. ***[3.4.2]***
- **$.url-safe(false|true)** — Использовать модифицированный алфавит, все символы которого не преобразовывались в **%XX** в URL (вместо '+' и '/' используются '-' и '_'). По умолчанию не использовать. ***[3.4.6]***
- **$.pad(true|false)** — При кодировании были добавлены символы паддинга (=) по умолчанию. ***[3.4.6]***

Подробная информация о Base64 доступна по ссылкам: ietf.org/rfc/rfc2045.txt и wikipedia.org/wiki/Base64

Пример

```
$encoded[pyAxOTczLiDV7uT/8iDx6/P16Cwg9/LuIKvH5evl7fvlIPDz6uDi4Lsg7eDv6PHg6yDx4Oy
F]
$original[^string:base64[$encoded]]
$original
```

Выведет...

```
§ 1973. Ходят слухи, что «Зеленые рукава» написал сам...
```

idna. Декодирование из IDNA

```
^string:idna[закодированное]
```

Замечание: именно метод, не конструктор!

Метод декодирует строку из IDNA-представления (может потребоваться при работе с кириллическими доменами). Для кодирования строки следует использовать
^строка.idna[]

Подробная информация о IDNA доступна по ссылкам: tools.ietf.org/html/rfc3490

и wikipedia.org/wiki/IDN

Пример
```
$encoded[xn--e1afmkfd.xn--80akhbyknj4f]
$original[^string:idna[$encoded]]
$original
```

Выведет...
пример.испытание

js-unescape. Декодирование, аналогичное функции unescape в JavaScript

^string:js-unescape[закодированное]

Примечание: именно статический метод, не конструктор!

Метод выполняет преобразование строки, аналогичное методу **unescape**, описанному в ECMA-262. Для кодирования следует использовать
^string.js-escape[]

С помощью данного метода возможно декодировать строки, закодированные в браузере с помощью функции **escape**.

Подробная информация о ECMA-262 доступна по ссылке: ecma-international.org/publications/standards/Ecma-262.htm (B.2.2)

Примечание: метод также декодирует символы, закодированные в виде **\uXXXX**. ***[3.4.1]***

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%u
0432%u0433%u0434]
$original[^string:js-unescape[$escaped]]
$original
```

Выведет...
abcd `+-=~!@#%&*()_ []{}<>:'",./?абвгд

sql. Получение строки из базы данных

```
^string:sql{SQL-запрос}
^string:sql{SQL-запрос}[$.limit(1) $.offset(n) $.default{код} $.bind[variables
hash]]
```

Замечание: именно метод, не конструктор!

Метод 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(n) — отбросить первые **n** записей выборки;
$.bind[hash] — связанные переменные, см. «Работа с IN/OUT-переменными».

Если ответ SQL-сервера был пуст (0 записей), то будет:

$.default{код} ...выполнен указанный **код**, и строка, которую он возвратит, будет результатом выполнения метода;
$.default(выражение) ...вычислено указанное **выражение**, и оно будет результатом выполнения метода;
$.default[строка] ...возвращена указанная **строка**;
$.default не задан ...выдано сообщение об ошибке.

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.

unescape. Декодирование JavaScript- или URI-кодирования

```
^string:unescape[js|uri;закодированное]
^string:unescape[js|uri;закодированное;опции]
```

Примечание: именно статический метод, не конструктор!

С параметром **js** метод эквивалентен методу **js-unescape** и выполняет преобразование строки, аналогичное методу **unescape**, который описан в ECMA-262. Предусмотрена возможность декодировать строки, закодированные в браузере с помощью функции **escape**.

С параметром **uri** метод выполняет декодирование URI-кодированных (процентно кодированных) строк. Возможно декодировать, например, **$request:uri**.

Поддерживаемые опции:

	По умолчанию	Описание
$.charset[название кодировки]	не определен	После декодирования преобразовать результат из указанной кодировки

Методы

base64. Кодирование в Base64

```
^строка.base64[]
^строка.base64[опции]    [3.4.6]
```

Метод позволяет преобразовать строку в формат Base64. Для обратного преобразования строки из Base64 в исходный вид нужно воспользоваться **^string:base64[закодированное]**.

Можно задать хеш **опций**.

- **$.wrap(true|false)** — Формировать результат с переносами строк (по умолчанию) или в одну строку.
- **$.url-safe(false|true)** — Использовать модифицированный алфавит, все символы которого не будут преобразовываться в **%XX** в URL (вместо '+' и '/' используются '-' и '_'). По умолчанию — не использовать.
- **$.pad(true|false)** — Добавлять символы паддинга (=), если кодируемая длина не кратна 3. По умолчанию — добавлять.

Подробная информация о Base64 доступна по ссылкам: ietf.org/rfc/rfc2045.txt и wikipedia.org/wiki/Base64

Пример

```
$original[§ 1973. Ходят слухи, что «Зеленые рукава» написал сам…]
<pre>^original.base64[]</pre>
```

Выведет…

```
pyAxOTczLiDV7uT/8iDx6/P16Cwg9/LuIKvH5evl7fvlIPDz6uDi4Lsg7eDv6PHg6yDx4OyF
```

format. Вывод числа в заданном формате

```
^строка.format[форматная_строка]
```

Метод выводит значение переменной в заданном формате (см. «Приложение 2. Форматные строки преобразования числа в строку»). Выполняется автоматическое преобразование строки в число.

Пример

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

Возвратит: **15.68**

int, double, bool. Преобразование строки в число или bool

```
^строка.int[]
^строка.int(значение по умолчанию)
^строка.double[]
^строка.double(значение по умолчанию)
^строка.bool[]
^строка.bool(значение по умолчанию)
```

Методы преобразуют значение переменной **$строка** в целое вещественное число или **bool**-значение соответственно, а также возвращают это значение.

Можно задать **значение по умолчанию**, которое будет получено, если преобразование невозможно, строка пуста или состоит только из white spaces (символов пробела, табуляции, 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, содержащие числа (значение 0 будет преобразовано в **false**, не 0 — в **true**), а также строки, содержащие значения **true** и **false** (без учета регистра).

Внимание!
Использование пустой строки в математических выражениях не является ошибкой, ее значение считается нулем.
Преобразование строки, не являющейся целым числом, в целое число является ошибкой (например строка «1.5» не является целым числом).

Примеры использования

```
$str[123]
^str.int[]
```

Выведет число **123**, поскольку объект **str** можно преобразовать в класс **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**».

idna. Кодирование в IDNA

```
^строка.idna[]
```

Метод позволяет преобразовать строку в формат IDNA (может потребоваться для работы с кириллическими доменами). Для обратного преобразования строки из IDNA в исходный вид необходимо воспользоваться
^string:idna[закодированное]

Подробная информация о IDNA доступна по ссылкам: tools.ietf.org/html/rfc3490 и wikipedia.org/wiki/IDN

Пример

```
$original[пример.испытание]
<pre>^original.idna[]</pre>
```

Выведет...

```
xn--e1afmkfd.xn--80akhbyknj4f
```

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, необходимо воспользоваться
^string:js-unescape[закодированное]

Строки, закодированные данным методом, могут быть раскодированы в браузере с помощью функции **unescape**.

Подробная информация о ECMA-262 доступна по ссылке: ecma-international.org/publications/standards/Ecma-262.htm (B.2.1)

Пример

```
$value[abcd `+-=~!@#%&*()_ []{}<>:'",./?абвгд]
^value.js-escape[]
```

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
```

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. При вызове **^строка.left(-1)** возвращается вся

строка. ***[3.4.4]***

Пример

```
$str[О, сколько нам открытий чудных!…]
^str.left(10) ^str.right(10)
```

На экран будет выведено: **О, сколько чудных!**…

length. Длина строки

```
^строка.length[]
```

Метод возвращает длину **строки**.

Пример

```
$str[О, сколько нам открытий чудных!…]
^str.length[]
```

Вернет: **32**

match. Поиск подстроки по шаблону

```
^строка.match[шаблон]
^строка.match[шаблон][опции поиска]
```

Метод 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;
' — 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, которые заключены в круглые скобки, **n** — номер открывающей круглой скобки.

Замечание 1: значения столбцов **prematch**, **match**, **postmatch** *вычисляются, только если указана опция* **'**.

Замечание 2: значения столбцов **1**, **2**,..., **n** *вычисляются лишь в случае, если в шаблоне указаны круглые скобки.*

Замечание 3: если в шаблоне нужно использовать круглые скобки, но не требуется запоминания заключенного 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: Бывает, что даже самому разработчику через какое-то время трудно разобраться в своем регулярном выражении. Настоятельно советуем задавать комментарии к частям сложного регулярного выражения. Для этого нужно указать опцию **x**, *которая включает 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]
^строка.match[шаблон][опции поиска]{замена}{возвращается, если не было найдено совпадений} [3.4.1]
```

Метод 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.**

mid. Подстрока с заданной позиции

```
^строка.mid(P;N)
^строка.mid(P)
```

Метод 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**

pos. Получение позиции подстроки

```
^строка.pos[подстрока]
^строка.pos[подстрока](позиция начала поиска)
```
[3.3.0]

Метод возвращает число **int** — позицию первого символа **подстроки** в строке (начиная с нуля), или −1, если подстрока не найдена. Если задана **позиция начала поиска**, то поиск подстроки будет начинаться с указанной **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]
^строка.replace[что;на что]
```
[3.4.2]

Метод эффективно заменяет подстроки в строке в соответствии с **таблицей подстановок**, работает существенно быстрее **match**.

Таблица подстановок — объект класса **table**, содержащий 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]

Метод 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 A достать данные, положить в SQL-сервер B.

Если оба SQL-сервера доступны с какой-то машины, то допустимо сделать так:

```
^connect[A]{
    $data[
#       код, наполняющий data данными из SQL-сервера A
    ]
    ^connect[B]{
         ^void:sql{insert into table x (x) values ('$data')}
    }
}
```

При этом **$data** в SQL-запросе **insert** будет правильно обработан по правилам SQL-диалекта сервера B.

Однако если оба SQL-сервера *не*доступны одновременно с какой-то машины, то допустимо сделать 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.

split. Разбиение строки

```
^строка.split[разделитель]
^строка.split[разделитель;опции разбиения]
^строка.split[разделитель;опции разбиения;имя столбца]
```

Метод 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);
a – сформировать массив из частей исходной строки. ***[3.5.0]***

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
```

trim. Отсечение букв с концов строки

```
^строка.trim[]
^строка.trim[откуда]
^строка.trim[откуда;набор]
^строка.trim[набор]      [3.4.4]
```

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
```

upper, lower. Преобразование регистра строки

```
^строка.upper[]
^строка.lower[]
```

Методы переводят **строку** в верхний или нижний регистр соответственно. Для их работы необходимо, чтобы был задан **$request:charset**.

Пример

```
$str[Москва]
^str.upper[]
```

Вернет: **МОСКВА.**

table (класс)

Класс предназначен для работы с таблицами строк.

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]
```

Конструктор создает объект класса **table**, используя **табличные данные**, которые определены в самом конструкторе.

Табличные данные – данные в формате tab-delimited, то есть поля записи 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 выступают их порядковые номера, начиная с нулевого.

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 и поиска»).

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 — Основные опции описаны в разделе «Приложение 1. Пути к файлам и каталогам, работа с HTTP-серверами», также доступны дополнительные опции, см. «Опции формата файла».

Параметр **nameless** используется так же, как и в конструкторе **table::create**.

Пример загрузки таблицы с диска

```
$loaded_table[^table::load[/addresses.cfg]]
```

Пример создает объект класса **table**. Он содержит именованную таблицу, определенную в файле `addresses.cfg`, который находится в корневом каталоге 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(n) $.bind[variables hash]]
```

Конструктор создает объект класса **table**, который содержит 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(n) — отбросить первые **n** записей выборки;
$.bind[hash] — связанные переменные, см. «Работа с IN/OUT-переменными».

Пример

```
$sql_table[^table::sql{select * from news}]
```

В результате будет создан объект, содержащий все записи из таблицы **news**.

Примечание: всегда нужно указывать конкретный список необходимых полей. Использование «» крайне не рекомендуется, поскольку постороннему читателю (да и самому разработчику 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 создании таблицы, 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.

Примечание: если значением любой из вышеуказанных опций является символ **#**, *то отключается удаление из загружаемого файла строк, начинающихся с этого символа.* ***[3.4.1]***

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 обрамляются 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
table.join

и при поиске, см...
table.locate

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(true|false)	**true** = в обратном порядке.

Получение содержимого столбца

$таблица.поле

Возвращает содержимое столбца **поле** из текущей строки таблицы.

До версии 3.4.4 эта же запись могла быть использована для получения методов таблицы. Начиная с версии 3.4.4 обращение к методам таблицы возможно только при их вызове **^таблица.method[]**, причем методы имеют приоритет перед полями.

Пример
```
$таблица.name
```

Вернет значение, определенное в столбце **name** текущей строки таблицы.

Изменение содержимого столбца

$таблица.поле[новое значение]

Изменяет содержимое столбца **поле** текущей строки таблицы на заданную строку.

Пример
```
$таблица.name[Мыло]
```

Установит значение **Мыло** в столбец **name** текущей строки таблицы.

Получение содержимого текущей строки в виде хеша

$таблица.fields — содержимое текущей строки таблицы в виде хеша (для **nameless**-таблиц доступно начиная с версии ***3.4.0***).

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, которые называются 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

append. Добавление строки в таблицу

```
^таблица.append{табличные данные}
^таблица.append[табличные данные]    [3.4.0]
^таблица.append[хеш]    [3.4.4]
```

Метод добавляет строку в конец таблицы. Формат представления **данных** — tab-delimited или хеш. 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.append[
      $.name[Michael]
      $.pos[visitor]
]
^stuff.save[stuff.txt]
```

Пример добавит в таблицу **$stuff** новые строки и сохранит таблицу в файл `stuff.txt`.

array. Преобразование таблицы в массив

```
^таблица.array[]
^таблица.array[название колонки]
^таблица.array{код}
```

Метод возвращает массив, где каждый элемент соответствует одной строке таблицы. Переданный параметр определяет значение элемента массива:

- при вызове без параметра — хеш с названиями колонок в качестве ключей и соответствующими значениями колонок;
- при вызове с названием колонки — строка со значением указанной колонки;
- при вызове с кодом — результат выполнения переданного кода.

Пример

```
^таблица.array[]
^таблица.array[колонка1]
^таблица.array{ $таблица.колонка1 + $таблица.колонка1 }
```

Создаст следующие массивы:

```
[
  {"колонка1": "значение1", "колонка2": "значение2"},
  {"колонка1": "значение3", "колонка2": "значение4"},
  ...
]
[
  "значение1",
  "значение3",
  ...
]
[
  " значение1 + значение2 ",
  " значение3 + значение4 ",
  ...
```

```
]
```

cells. Получение значений столбцов текущей строки таблицы

```
^таблица.cells[]
^таблица.cells(лимит)
```

Метод возвращает массив из значений столбцов текущей строки таблицы. Опциональным параметром можно задать ограничение на количество возвращаемых значений столбцов.

columns. Получение структуры таблицы

```
^таблица.columns[]
^таблица.columns[имя столбца]
```

Метод создает именованную таблицу из одного столбца, содержащего названия столбцов исходной именованной таблицы. Имя столбца – **column** или переданное **имя столбца**.

Пример

```
$columns_table[^stuff.columns[]]
```

count. Количество строк в таблице

```
^таблица.count[]
^таблица.count[columns|cells|rows]  [3.4.2]
```

При вызове без параметров или с параметром **rows** метод выдает количество строк в таблице (**int**).
При вызове с параметром **columns** метод выдает количество столбцов в таблице (**int**).
При вызове с параметром **cells** метод выдает количество столбцов в текущей строке таблицы (**int**).

Пример

```
$goods[^table::create{pos     good  price
1     Монитор     1000
2     Системный блок    1500
3     Клавиатура  150
4     Колонки     100
}]
Столбцов: ^goods.count[columns]
Строк: ^goods.count[]
```

Выведет:

```
Столбцов: 3
Строк: 4
```

В выражениях числовое значение таблицы равно количеству строк, поэтому использовать метод **count** не требуется:

```
^if($goods > 2){больше}
```

csv-string. Преобразование в строку в формате CSV

```
^таблица.csv-string[]
^таблица.csv-string[опции]
^таблица.csv-string[nameless]
^таблица.csv-string[nameless;опции]
```

Метод выводит содержимое таблицы в виде строки в формате CSV.
Использование опции **nameless** выводит таблицу без имен столбцов.

Пример

```
$table[^table::create{object  action      subject
Маша  "мыла"      раму
Мама  мыла  Машу
}]
^table.csv-string[$.encloser["] $.separator[,]]
```

Выведет на экран:

```
"object","action","subject"
"Маша","""мыла""","раму"
"Мама","мыла","Машу"
```

delete. Удаление текущей строки

```
^таблица.delete[]
```

Метод удаляет текущую строку в таблице.

flip. Транспонирование таблицы

```
^таблица.flip[]
```

Метод 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

foreach. Последовательный перебор всех строк таблицы

```
^таблица.foreach[позиция;значение]{тело}
^таблица.foreach[позиция;значение]{тело}[разделитель]
^таблица.foreach[позиция;значение]{тело}{разделитель}
```

Метод перебирает все строки таблицы. Метод аналогичен методу **foreach** класса **hash**.

позиция — имя переменной, которая возвращает номер строки (отсчет начинается с 0);
значение — имя переменной, которая возвращает текущую строку;
тело — код, исполняемый для каждой строки;
разделитель — код, который вставляется перед каждым не пустым не первым телом.

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

Замечание: для уменьшения расхода памяти и ускорения в переменной **значение** *возвращается не отдельная строка, а вся таблица, у которой установлена текущая строка.*

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

Пример

```
$man[^table::create{name      value
name  Вася
age   22
gender      m
}]
^man.foreach[pos;row]{
      $pos $row.name=$row.value
}[<br />]
```

Выведет на экран:

```
0 name=Вася
1 age=22
2 gender=m
```

hash. Преобразование таблицы в хеш с заданными ключами

```
^таблица.hash[ключ]
^таблица.hash[ключ][опции]
^таблица.hash[ключ][столбец значений]
^таблица.hash[ключ][столбец значений][опции]
^таблица.hash[ключ]{код, формирующий значение}  [3.4.5]
^таблица.hash[ключ]{код, формирующий значение}[опции]  [3.4.5]
^таблица.hash[ключ][таблица со столбцами значений]
^таблица.hash[ключ][таблица со столбцами значений][опции]
```

Ключ может быть задан как:

- **[строка]**, которая содержит название столбца; значение этого столбца считается ключом;
- **{код}**, результат исполнения которого считается ключом;
- **(математическое выражение)**, результат вычисления которого считается ключом.

С опциями по умолчанию метод преобразует таблицу в хеш 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]	**hash** = значение каждого элемента — хеш (по умолчанию); **string** = значение каждого элемента — строка, при этом нужно указать **один** столбец значений; **table** = значение каждого элемента — таблица, при этом нельзя 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.
$.distinct[tables]	Создать хеш из таблиц, содержащих строки с ключом. Это устаревший ключ, который равносилен одновременному заданию **$.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. 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.

```
# это таблица с нашими товарами
$product_list[^table::create{id     name
1     хлеб
2     колбаса
3     масло
4     водка
}]

# это таблица с ценами на товары
$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
```

insert. Вставка строки в таблицу

```
^таблица.insert{табличные данные}
^таблица.insert[хеш]
```

Метод вставляет строку в таблицу на ту позицию, на которую указывает текущий указатель. Формат представления **табличных данных** — tab-delimited или хеш. Табличные данные должны иметь такую же структуру, как и таблица, в которую добавляются данные.

join. Объединение двух таблиц

```
^таблица1.join[таблица2]
^таблица1.join[таблица2;опции]
```

Метод добавляет в конец **таблицы1** записи из **таблицы2**. При этом берутся значения из тех столбцов **таблицы2**, имена которых совпадают со столбцами **таблицы1**, остальные столбцы будут содержать пустые строки. Также можно задать ряд опций, контролирующих добавление, см. «Опции копирования и поиска».

Пример

```
^stuff.join[$just_hired_people]
```

Все записи таблицы **$just_hired_people** будут добавлены в таблицу **$stuff**.

locate. Поиск в таблице

```
^таблица.locate[столбец;искомое_значение]
^таблица.locate[столбец;искомое_значение;опции]
^таблица.locate(логическое_выражение)
^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
```

Достаточно подставить такой поиск в пример:

```
^stuff.locate($stuff.status>1)
```

и будет найдена первая запись со значением статуса, превышающим 1.

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 не пустым не первым телом. Разделитель в квадратных скобках вычисляется один раз, в фигурных — многократно по ходу вызова.

Примечание: 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 помощью оператора **break** или принудительно закончить текущую итерацию и перейти к следующей с помощью оператора **continue**.

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 содержимое таблицы **$goods** в виде HTML-таблицы.

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[]
```

Метод позволяет сразу получить номер записи из таблицы в привычном виде, когда номер первой строки равен 1. Если в примере использовать **^men.line[]**, то нумерация будет идти от 1 до 3.

offset. Смещение указателя текущей строки

```
^таблица.offset(число)
^таблица.offset[cur|set](число)
```

Метод 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 строку.

Необязательный параметр:
cur — смещает указатель относительно текущей строки;
set — смещает указатель относительно первой строки.

Пример

```
^goods.offset(-1)
<table border="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**).

rename. Изменение названия столбца

```
^таблица.rename[старое название столбца;новое название столбца]
^таблица.rename[ $.старое_название_столбца[новое название столбца] … ]
```

Метод изменяет названия одного или нескольких столбцов таблицы.

Пример

```
^data.rename[ $.dt1[create_date] $.dt2[modify_date] ]
```

После выполнения кода колонка **dt1** будет называться **create_date**, а колонка **dt2** — **modify_date**.

save. Сохранение таблицы в файл

```
^таблица.save[путь]
^таблица.save[путь;опции]
^таблица.save[nameless;путь]
^таблица.save[nameless;путь;опции]
```

^таблица.save[append;путь] *[3.3.0]*
^таблица.save[append;путь;опции] *[3.3.0]*

Метод сохраняет таблицу в текстовый файл в формате tab-delimited.
Использование опции **nameless** сохраняет таблицу без имен столбцов.
При использовании опции **append** таблица сохраняется с именами столбцов только в том случае, если файла еще не существует.
Также доступны опции записи (см. «Опции формата файла»), позволяющие, например, сохранить файл в формате CSV 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/**.

select. Отбор записей

^таблица.select(критерий_отбора)
^таблица.select(критерий_отбора)[опции] *[3.4.1]*

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.

Можно задать хеш опций:

$.offset(количество строк)	перед копированием первой строки пропустить указанное **количество** подходящих под критерий строк таблицы
$.limit(максимум)	максимальное число строк, которые допустимо отобрать
$.reverse(false|true)	**true** = перебирать строки в обратном порядке

Пример

```
$men[^table::create{name      age
Serge 26
Alex  20
Misha 29
Denis 30
}]

$thoseAbove20[^men.select($men.age>20)[ $.limit(2) ]]
```

В **$thoseAbove20** попадут строки с **Serge** и **Misha**.

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 принимать

значения:
desc – по убыванию;
asc – по возрастанию.

По умолчанию используется сортировка по возрастанию.

Пример

```
$men[^table::create{name      age
Serge 26
Alex  20
Misha 29
}]
^men.sort{$men.name}
^men.menu{
    $men.name: $men.age
}[<br />]
```

В результате записи таблицы **$men** будут отсортированы по столбцу **name** (по строке имени):

```
Alex: 20
Misha: 29
Serge: 26
```

Можно отсортировать записи по столбцу **age** (по числу прожитых лет) по убыванию (**desc**), изменив в примере вызов **sort** на такой:

```
^men.sort($men.age)[desc]
```

Получится:

```
Misha: 29
Serge: 26
Alex: 20
```

void (класс)

Класс предназначен для работы с «пустыми» объектами. Он не имеет конструкторов, объекты этого класса создаются автоматически, например при обращении к несуществующей переменной.

У объекта класса **void** доступны все методы, присутствующие у объекта класса **string**, т. е. вызывать методы класса **string** можно без предварительной проверки определенности объекта. ***[3.4.1]***

Статический метод

sql. Запрос к БД, не возвращающий результата

```
^void:sql{SQL-запрос}
^void:sql{SQL-запрос}[$.bind[variables hash]]
```

Метод выполняет **SQL-запрос**, 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».

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 формате XML, запись в XML (w3.org/XML) и HTML, а также XSLT-трансформацию (w3.org/TR/xslt).

Работа с деревом производится в DOM-модели (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**, относительно которого указываются имена подключаемых файлов.

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**, относительно которого указываются имена подключаемых файлов.

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**, который состоит 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[]]
```

parser://метод/параметр. Чтение XML из произвольного источника

Parser может считать XML из произвольного источника. Везде, где можно считать XML, можно задать адрес документа в виде
`parser://метод/параметр`

Считывание документа по такому адресу приводит к 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 так же, как если бы файлы читались с диска. Скажем, если `parser://xsl_database/main.xsl` ссылается на `utils/common.xsl`, то 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, работа с HTTP-серверами»), что куда удобнее задания полного дискового пути, включающего путь к веб-пространству. По умолчанию равен пути к текущему обрабатываемому документу.

Внимание: символ «/» в 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]]
$Attr[^документ.createAttributeNS[namespaceURI;qualifiedName]]
$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 по ссылке: w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

Подробная спецификация DOM2 доступна по ссылке: w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

load. Загрузка XML с диска, HTTP-сервера или иного источника

```
^xdoc::load[имя файла]
```

Метод загружает XML-код из некоторого **файла** или адреса на HTTP-сервере и создает на его основе объект класса **xdoc**. Parser способен 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>
```

file. Преобразование документа в объект класса file

```
^документ.file[Параметры_преобразования_в_текст]
```

Метод преобразует документ в тип **file**. Возможно задание **параметров_преобразования** в текст. По умолчанию создается XML-представление документа с заголовком **<?xml ... ?>** (можно отключить вывод заголовка, задав соответствующий параметр).

Метод понимает опцию **$.file[имя файла]**, с помощью которой можно задать имя создаваемому объекту типа **file**. ***[3.4.2]***

Пример

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
строка1<br/>
строка2<br/>
</document>}]

$response:body[^document.file[]]
```

save. Сохранение документа в файл

```
^документ.save[путь]
^документ.save[путь;Параметры_преобразования_в_текст]
```

Метод 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]
```

string. Преобразование документа в строку

```
^документ.string[]
^документ.string[Параметры_преобразования_в_текст]
```

Метод 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**.

Пример

```
$document[^xdoc::create{<?xml version="1.0" encoding="windows-1251" ?>
<document>
строка1<br/>
строка2<br/>
</document>}]

^document.string[
    $.method[html]
]
```

transform. XSL-преобразование

```
^документ.transform[шаблон]
^документ.transform[шаблон][XSLT-параметры]
```

Метод 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"/>**?», следует обратить внимание на возможность задания базового пути: «Параметр создания нового документа: Базовый путь».

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 по ссылке: 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 определить эти префиксы в данном хеше.

Здесь:
- ключи — префиксы пространств имен;
- значения — их URI.

Пример добавления 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])]
```

Пример добавления одного префикса

```
$xdoc.search-namespaces.s[urn:special]
```

Параметры преобразования документа в текст

В ряде методов можно задать хеш **Параметры_преобразования_в_текст**.

Они идентичны атрибутам элемента **<xsl:output … />**.
Исключениями 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.

При создании объекта класса **file** можно задать параметр **media-type**, при задании нового тела ответа заголовок ответа **content-type** получит значение этого параметра.

Пример

```
^document.string[
      $.method[html]
      $.indent[no]
      $.omit-xml-declaration[yes]
      $.encoding[windows-1251]
#     $.charset[windows-1251]
]
```

[3.4.2] опция не может быть использована совместно с опцией **$.encoding[]**

Будет выдан документ в HTML-представлении без отступов и XML-декларации.

Выдача XHTML

Если необходимо выдать XHTML, следует использовать следующие 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"
>
```

Xmlns указывается 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.

Помимо этого, требуется задать следующие атрибуты элемента **<xsl:output ... />**:

```
<xsl:output
      doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
      doctype-system="DTD/xhtml1-strict.dtd"
/>
```

Внимание: атрибут **method** *не задается. 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
```

xnode (класс)

Класс предназначен для работы с древовидными структурами данных в паре с **xdoc**, поддерживает XPath-запросы (w3.org/TR/xpath).

Класс реализует DOM-интерфейсы Node и Element и их производные. Класс не создается напрямую, 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]]
^узел.setAttributeNS[namespaceURI;qualifiedName;value]
^узел.removeAttributeNS[namespaceURI;localName]
$Attr[^узел.getAttributeNodeNS[namespaceURI;localName]]
$Attr[^узел.setAttributeNodeNS[$newAttr]]
$NodeList[^узел.getElementsByTagNameNS[namespaceURI;localName]]
^if(^узел.hasAttribute[name]){…}
^if(^узел.hasAttributeNS[namespaceURI;localName]){…}
^if(^узел.hasAttributes[]){…}
```

В Parser:
- DOM-интерфейс — класс **hash** с ключами 0, 1, …;
- DOM-тип — класс **string**;
- DOM-тип boolean — логическое значение (0 = «ложь», 1 = «истина»).

Подробная спецификация DOM1 доступна по ссылке: w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

Подробная спецификация DOM2 доступна по ссылке: w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

select. XPath-поиск узлов

```
$NodeList[^узел.select[XPath-запрос]]
```

Метод 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 по ссылке: w3.org/TR/xpath

selectSingle. XPath-поиск одного узла

^узел.selectSingle[XPath-запрос]

Метод 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 />
```

selectString. Вычисление строчного XPath-запроса

^узел.selectString[XPath-запрос]

Метод 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)]
```

selectNumber. Вычисление числового XPath-запроса

^узел.selectNumber[XPath-запрос]

Метод выдает результат выполнения **XPath-запроса** в контексте **узла**, если это число. Если же это 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 />
```

selectBool. Вычисление логического XPath-запроса

^узел.selectBool[XPath-запрос]

Метод выдает результат выполнения **XPath-запроса** в контексте **узла**, если это логическое значение. Если же это 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
}
```

Поля

DOM

DOM1-интерфейс Node:

```
$узел.nodeName
$узел.nodeValue
$узел.nodeValue[новое значение]
^if($узел.nodeType == $xnode:ELEMENT_NODE){…}
$Node[$узел.parentNode]
$NodeList[$узел.childNodes]
$Node[$узел.firstChild]
$Node[$узел.lastChild]
$Node[$узел.previousSibling]
$Node[$узел.nextSibling]
$NamedNodeMap[$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
```

DOM1-интерфейс Notation:

```
$узел_типа_NOTATION.publicId
$узел_типа_NOTATION.systemId
```

В Parser:
- DOM-интерфейс — класс **hash** с ключами 0, 1, …;
- DOM-интерфейс — класс **hash**, где в качестве ключей выступают имена атрибутов;
- DOM-тип DOMString — класс **string**;
- DOM-тип boolean — логическое значение (0 = «ложь», 1 = «истина»).

Подробная спецификация DOM1 доступна по ссылке: w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

Подробная спецификация DOM2 доступна по ссылке: 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 />
}
```

Установка и настройка Parser 3

Проще всего установить Parser 3 в операционных системах, использующих пакеты Debian (например, Ubuntu). Достаточно выполнить следующую команду:
sudo apt-get install parser3-cgi
Если нужен драйвер MySQL (MariaDB), нужно дополнительно выполнить:
sudo apt-get install parser3-mysql

Если установлен Docker, можно скачать контейнер для запуска Parser 3 в режиме веб-сервера. Кроме того, есть программа для установки под Windows и архивы с комплектом установки, доступные в разделе www.parser.ru/download .

Parser 3 доступен в нескольких вариантах:
- CGI-скрипт (и интепретатор и веб-сервер);
- модуль веб-сервера apache (в пакетах Debian);
- ISAPI-расширение веб-сервера Microsoft Internet Information Server 8.0 или новее.

Дополнительно поставляются драйверы для различных SQL-серверов (сейчас доступны для MySQL, Postgres, Oracle, ODBC и SQLite).

Для локальной разработки рекомендуется использовать Parser 3 в режиме веб-сервера. Для этого достаточно выполнить две команды:
cd <корневая директория сайта>
cgi/parser3.cgi -p 8080
После этого сайт будет доступен в браузере по адресу localhost:8080.

Описание каталогов и файлов
`parser3.exe` или `parser3.cgi` — CGI-скрипт (и интерпретатор и веб-сервер);
`parser3isapi.dll` — ISAPI расширение веб-сервера IIS 8.0 или новее.
`auto.p` — конфигурационный файл;
`parser3.charsets/` — каталог с файлами таблиц кодировок:

`cp866.cfg`	— Cyrillic [CP866]
`koi8-r.cfg`	— Cyrillic [KOI8-R]
`koi8-u.cfg`	— Cyrillic [KOI8-U]
`windows-1250.cfg`	— Central European [windows-1250]
`windows-1251.cfg`	— Cyrillic [windows-1251]
`windows-1254.cfg`	— Turkish [windows-1254]
`windows-1257.cfg`	— Baltic [windows-1257]
`x-mac-cyrillic.cfg`	— Macintosh Cyrillic

Поскольку исходные коды являются открытыми, возможно собрать Parser самостоятельно (см. Получение исходных кодов) и написать свой SQL-драйвер.

Внимание: в целях безопасности версии скомпилированы так, что могут читать и исполнять только

файлы, принадлежащие тому же пользователю или группе пользователей, от имени которых работает сам Parser.

Как подключаются конфигурационные файлы

Для CGI-скрипта (`parser3.exe` или `parser3.cgi`):
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 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. файл `auto.p`).

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**.*

Здесь рекомендуется определить путь к классам созданного сайта:

```
$CLASS_PATH[/../classes]
```

А также строку соединения с SQL-сервером, используемым на созданном сайте (пример для ODBC):

```
$SQL.connect-string[odbc://DSN=www_mydomain_ru^;UID=user^;PWD=password]
```

Примечание: в коде следует использовать ее так:

```
^connect[$SQL.connect-string]{…}
```

Рекомендуется поместить сюда же определение метода **unhandled_exception**, который будет выводить сообщение о возможных проблемах на созданном сайте.

Внимание: конечно, конфигурационный файл можно не использовать, а конфигурационный метод — поместить в файл `auto.p`, размещенный 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 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;
- ограничение на размер загружаемых файлов *[3.4.5]*;
- ограничения на число итераций в циклах и глубину рекурсии *[3.4.5]*;
- сервер или программу отправки почты;
- SQL-драйверы 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».

```
$LIMITS[
#Максимальный размер POST-данных, по умолчанию — 10 МБ:
	$.post_max_size(10*0x400*0x400)
#Максимальный размер загружаемых в память файлов, по умолчанию — 512 МБ:
	$.max_file_size(512*0x400*0x400)
#Максимальное число итераций в циклах, по умолчанию — 20000:
	$.max_loop(20000)
#Максимальная глубина рекурсии, по умолчанию — 1000:
	$.max_recursion(1000)
#Время ожидания доступности файла для блокировки чтения или записи,
по умолчанию — 9,5 сек.:
	$.lock_wait_timeout(9.5)
]
```

Установка **max_file_size**, **max_loop**, **max_recursion** в нулевое значение означает «без ограничений».

Параметр отправки писем (см. ^mail:send[…])

Под Windows и UNIX 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 самостоятельно:

```
$MAIL[
	$.sendmail[/custom/mail/sending/program params]
]
```

В зависимости от системы по умолчанию используется команда

```
/usr/sbin/sendmail -t -i -f postmaster
```

или

```
/usr/lib/sendmail -t -i -f postmaster
```

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    libmariadb.so,libmysqlclient.so
odbc  /full/disk/path/parser3odbc.dll
pgsql /full/disk/path/parser3pgsql.dll    libpq.so
sqlite      /full/disk/path/parser3sqlite.dll   libsqlite3.so
oracle      /path/to/parser3oracle.dll    C:\Oracle\Ora81\BIN\oci.dll?PATH+=^;C
:\Oracle\Ora81\bin
}]
]
```

В колонке **client** таблицы **drivers** можно указывать сразу несколько клиентских библиотек, перечисляя их через запятую (будет использоваться первая доступная из перечисленных) ***[3.5.0]***. Кроме того, в драйвере Oracle допустимы параметры клиентской библиотеки, 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
7z    application/x-7z-compressed
...
zip   application/zip}]
```

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`.

При задании **$STRICT-VARS(true)** будет выдаваться исключение в случае попытки обращения к неинициализированным пременным. ***[3.4.2]***

При задании **$LOCALS(true)** все переменные всех методов всех классов будут считаться локальными. ***[3.4.6]***

Описание формата файла, описывающего кодировку

Данные представляются в формате tab-delimited со столбцами, указанными ниже.

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. Например, у буквы W он совпадает, а у буквы Я — нет.

unicode2 — Дополнительный Unicode-код символа, если имеется.

Установка Parser на веб-сервер как CGI

Для установки Parser необходимо внести изменения в основной конфигурационный файл веб-сервера, или, если доступ к нему отсутствует, необходима возможность использовать `.htaccess`-файлы.

По умолчанию, в установке Apache возможность использования файлов `.htaccess` отключена. Если она необходима, нужно разрешить ее использовать (по крайней мере, задавать `FileInfo`). Для этого в основном конфигурационном файле веб-сервера (обычно `httpd.conf`) в секцию `<virtualhost ...>` созданного сайта или вне ее — для всех сайтов нужно добавить директивы:

```
<Directory /путь/к/вашему/веб/пространству>
AllowOverride FileInfo
</Directory>
```

Файл с исполняемым кодом Parser (в текущей версии — `parser3.cgi`) надо переместить в каталог для CGI-скриптов (закачивать файл по ftp нужно в режиме binary, а не text). Ему нужно дать права на выполнение, которые можно уточнить у хостинг-провайдера (обычно необходимые права — 755).

Под UNIX

Нужно добавить в `.htaccess`-файл созданного сайта (или в `httpd.conf` в секцию `<virtualhost ...>`, или вне ее — для всех сайтов) блоки:

```
Action parser3-handler /cgi-bin/parser3.cgi
AddHandler parser3-handler html

# запрет на доступ к .p-файлам, в основном к auto.p
<Files ~ "\.p$">
    Order allow,deny
    Deny from all
</Files>
```

Под Windows

Нужно добавить в `.htaccess`-файл созданного сайта (или в `httpd.conf` в секцию `<virtualhost ...>`, или вне ее — для всех сайтов) блоки:

```
Action parser3-handler /cgi-bin/parser3.exe
AddHandler parser3-handler html

# запрет на доступ к .p-файлам, в основном к auto.p
<Files ~ "\.p$">
    Order allow,deny
    Deny from all
</Files>
```

Если расположение конфигурационного файла по умолчанию неприемлемо (см. «Установка и настройка Parser»), его допустимо задать 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, что 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 расположение журнала ошибок `parser3.log` по умолчанию неприемлемо, его допустимо задать 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 как модуля сервера

Для установки Parser необходимо внести изменения в основной конфигурационный файл веб-сервера, или, если доступ к нему отсутствует, необходима возможность использовать `.htaccess`-файлы.

По умолчанию, в установке Apache возможность использования файлов `.htaccess` отключена. Если она необходима, нужно разрешить ее использовать (по крайней мере, задавать FileInfo). Для этого в основном конфигурационном файле веб-сервера (обычно `httpd.conf`) в секцию `<virtualhost ...>` созданного сайта или вне ее — для всех сайтов нужно добавить директивы:

```
<Directory /путь/к/вашему/веб/пространству>
AllowOverride FileInfo
</Directory>
```

Под UNIX

Необходимо собрать Parser из исходных кодов, задав опцию `--with-apache` у скрипта `buildall`.

```
# динамическая загрузка модуля
LoadModule parser3_module /path/to/mod_parser3.so
```

Под Windows

Необходимо собрать Parser из исходных кодов, используя заранее подготовленные файлы проектов (`.sln`). Поместить файл с исполняемым кодом модуля Parser (в текущей версии — `mod_parser3.dll`) в произвольный каталог. Добавить в файл `httpd.conf` после имеющихся строк `LoadModule`:

```
# динамическая загрузка модуля
LoadModule parser3_module x:\path\to\mod_parser3.dll
```

Внимание: если это необходимо, сопутствующие `.dll`-файлы помещаются в тот же каталог.

Нужно добавить в `.htaccess`-файл созданного сайта (или в `httpd.conf` в секцию `<virtualhost ...>` или вне ее — для всех сайтов) следующие 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 8.0 или новее

Нужно поместить файлы с исполняемым кодом Parser в произвольный каталог. Если используется версия Parser с поддержкой XML, то в каталог, указанный в переменной окружения `PATH` (например, `C:\WinNT`), следует распаковать XML-библиотеки.

Затем назначить Parser обработчиком `.html`-страниц:
1) Запустить Management Console, кликнув правой кнопкой мыши на названии созданного веб-сервера, и выбрать `Properties`.
2) На вкладке `Home directory` и в разделе `Application settings` нажать на кнопку `Configuration...`
3) В появившемся окне нажать кнопку `Add`.
4) В поле `Executable` ввести полный путь к файлу `parser3.exe` или `parser3isapi.dll`.
5) В поле `Extension` ввести строку `.html`.
6) Включить опцию `Check that file exists`.
7) Нажать на кнопку `OK`.

Подобие mod_rewrite

Для веб-сервера IIS встроенного подобия Apache-модуля `mod_rewrite` (см. также 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 ошибки 404 (рекомендуется 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 подавляя ее запись в журнал ошибок.

Использование Parser в качестве веб-сервера

`/путь/к/parser3 -p [хост:]<порт>`

Данная команда запускает встроенный в Parser веб-сервер на указанном порту, корнем веб-пространства будет текущая директория (в которой пользователь находится в момент запуска).
В режиме веб-сервера все запросы обрабатываются методом **`main`** класса **`httpd`**, который добавлен в конфигурационный auto.p и в котором реализована логика работы веб-сервера — исходя из того, к какому адресу обратился пользователь (изображение, Parser-код, директория), веб-сервер либо самостоятельно обработает запрос (например вернет файл с изображением), либо передаст управление файлу с кодом на Parser, к которому сделан запрос. При запуске можно указать конкретный IP-адрес или имя, тогда Parser будет принимать соединения только на нем. Например, при запуске **`parser3 -p localhost:8000`** соединения будут приниматься только на локальном интерфейсе, закрытом от доступа извне. Встроенный веб-сервер не поддерживает шифрование keep-alive, на хостинге правильнее использовать его в комбинации с nginx. Настройки веб-сервера задаются в хеше **`$cfg`**:

`$.parser[(\.html^$)]` — регулярное выражение, содержащее расширения файлов с кодом на Parser;

`$.index[index.html]` — название индексного файла; при обращении к директории выдастся он; для упрощения кода предусмотрен один индексный файл, но несложно добавить еще;

`$.autoindex(true)` — отображение листинга файлов в директории при отсутствии индексного файла;

`$.404[$404]` — обработка ошибки 404; можно вызвать метод **`$.404[/404.html]`**, а можно передать управление определенному файлу;

**$.fix-trailing-slash(true)** — выдача редиректа с добавлением символа «/» при запросе директории без этого символа на конце;
**$.auth[$.url[^^/\.?admin/] $.login[admin] $.password[change me] $.realm[site administration]]** — запрос авторизации при доступе к разделам, подходящим под регулярное выражение
$.deny[(/\.ht[^^/]+|\.p|\.cfg)^$] — регулярное выражение, показывающее, к каким файлам доступ запрещен, например к файлам `.htaccess` и `auto.p`;
$.403[Permission denied] — обработка отказа в доступе; тоже может быть и методом, и файлом;
$.memory(64000) — вызов сборщика мусора, если при обработке ранее поступивших запросов было аллоцировано более 64 МБ памяти;
**$.log[/access.log]** — включение логирования приходящих запросов.

Класс веб-сервера достаточно прост, всего около сотни строк, поэтому можно либо напрямую редактировать логику его работы, либо воспользоваться заложенными возможностями по расширению — в корень веб-пространства можно поместить файл `httpd.p`, в котором переопределить методы класса **httpd**. В частности, можно переопределить методы **^config[]** (принимает аргументом конфигурацию по умолчанию и может изменить ее перед возвратом) и **^preprocess[]** (для обработки редиректов или замены адресов).

В конфигурационном методе, в хеше **$HTTPD**, можно задать режим работы веб-сервера и тайм-аут при обработке соединений:

- **$.mode[sequental]** — последовательная обработка запросов, по умолчанию; Parser работает в один поток, используя одно ядро процессора;
- **$.mode[parallel]** — параллельный режим, недоступный под Windows; на каждый запрос создается отдельный процесс; в случае необходимости используются все ядра, что дает максимальную производительность при обработке кода;
- **$.mode[threaded]** — многопоточный режим; на каждый запрос создается отдельный поток; за счет того, что все потоки используют один сборщик мусора, производительность немного ниже параллельного режима;
- **$.timeout(время)** — задает максимальное время ожидания в секундах; если в течение заданного времени от клиента не поступят данные, соединение будет разорвано; в многопоточном режиме заданный тайм-аут не работает, фактический тайм-аут определяется операционной системой.

Пример

```
$HTTPD[
	$.mode[parallel]
	$.timeout(8)
]
```

Задает параллельный режим и тайм-аут 8 секунд.

Использование Parser в качестве интерпретатора скриптов

```
/путь/к/parser3 [опции] файл_со_скриптом
x:\путь\к\parser3 [опции] 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.

Доступные опции:

-f /path/to/auto.p
Конфигурационный файл по умолчанию ищется в том же каталоге, где расположен интерпретатор Parser. С помощью этой опции можно указать другой путь к конфигурационному файлу.

-l /path/to/parser3.log
Об ошибках Parser делает записи в журнал ошибок parser3.log, который по умолчанию расположен в том же каталоге, где и сам интерпретатор Parser. С помощью этой опции можно изменить расположение журнала ошибок.

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)
```

Внимание: следует установить биты атрибута, разрешающие исполнение владельцу и группе. Команда:
`chmod ug+x файл`

Получение исходных кодов

Исходные коды Parser 3 можно скачать из раздела «Скачать» с сервера parser.ru или получить из CVS:
`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`**, будет загружена текущая разрабатываемая версия (HEAD). Для загрузки стабильной версии нужно запросить ветку `release_3_X_X` (например, `release_3_4_6`).

`Имя модуля`:
Имя основного модуля: `parser3`

Модуль, необходимый для сборки Parser 3 и SQL-драйверов под Windows: `win32`

Модуль с SQL-драйверами: `sql`

В нем доступны каталоги:

```
sql/mysql
sql/pgsql
sql/oracle
sql/odbc
sql/sqlite
```

Для сборки SQL-драйверов необходимо наличие исходных кодов Parser 3. Поскольку `.h`-файлы ищутся по относительным путям, структура каталогов должна быть следующей:

```
parser3project/ — директория, где находятся исходные коды;
  |
  |__parser3/ — исходные коды Parser;
  |
  |__sql/
  |      |__mysql/ — исходные коды драйвера MySQL;
  |      |__... — исходные коды других необходимых драйверов;
  |
  |__win32/ — каталог, необходимый для сборки Parser 3 под Windows.
```

Сборка под Linux и другие Unix-подобные системы

Для сборки Parser 3 под Linux и другие Unix-подобные системы необходимо использовать специально созданный скрипт `buildall`.

Т. е. в общем случае процесс скачивания исходных кодов и сборки Parser 3 будет выглядеть примерно так:

```
cd ~
mkdir parser3project
cd parser3project
wget HTTPs://www.parser.ru/off-line/download/src/parser-3.4.6.tar.gz
tar -xzf parser-3.4.6.tar.gz
mv parser-3.4.6 parser3
cd parser3
./buildall
```

Скрипт сборки поддерживает следующие параметры:
`--disable-safe-mode` — не проверять принадлежность открываемых Parser 3 файлов текущему пользователю;
`--without-xml` — собрать Parser 3 без поддержки XML;
`--with-mailreceive` — при запуске Parser 3 с ключом `-m` переданное на `stdin` письмо доступно в `$mail:receive`;
`--with-apache` — собирать модуль Apache (DSO, поддерживаются Apache 1.X и 2.X);
`--strip` — удалить отладочную информацию.

Сборка SQL-драйвера (на примере MySQL) будет выглядеть примерно так:

```
cd ~/parser3project
mkdir sql
cd sql
wget HTTPs://www.parser.ru/off-line/download/src/parser3mysql-10.8.tar.gz
tar -xzf parser3mysql-10.8.tar.gz
cd parser3mysql-10.8
./configure
make
```

Сборка под Windows

Для компиляции Parser 3 под Windows используется *Microsoft Visual Studio.NET (2003 или новее)* и заранее подготовленные нами файлы проектов (`.sln`). Все модули нужно распаковывать в один каталог, например `parser3project`.

Для сборки Parser 3 также необходимы каталоги:
`win32/tools`
`win32/gc`
`win32/pcre`
`win32/gnome/libxml2-x.x.x`
`win32/gnome/libxslt-x.x.x`

Для сборки SQL-драйверов необходим каталог:
`win32/sql`

Для сборки варианта Parser 3 без поддержки XML, в файле `parser3/src/include/pa_config_fixed.h` необходимо закомментировать директиву:
`#define XML`

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 указано относительно, то файл ищется по **CLASS_PATH** (если **CLASS_PATH** — таблица, то каталоги в ней перебираются снизу вверх).

Приложение 1. Пути к файлам и каталогам, работа с HTTP-серверами

Пример таблицы CLASS_PATH:

```
$CLASS_PATH[^table::create{path
/classes/common
/classes/specific
}]
```

Теперь по относительному пути `my/class.p` поиск файла будет происходить 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 вид **%Длина.ТочностьТип**

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 начало 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.

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.

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 — шестнадцатеричное целое число без знака; для вывода цифр, превышающих 9, используются буквы a, b, c, d, e, f;

X — шестнадцатеричное целое число без знака; для вывода цифр, превышающих 9, используются буквы A, B, C, D, E, F;

f — действительное число.

Приложение 3. Формат строки подключения оператора connect

Строка подключения обрабатывается драйвером базы данных для Parser 3.

Для MySQL

```
mysql://user:password@host[:port][,host[:port]]|[/unix/socket]/database?
      charset=значение& [значением может быть название кодировки для MySQL 4.1+]
      ClientCharset=кодировка&
      timeout=3&
      compress=0&
      named_pipe=1&
      autocommit=1&
      local_infile=0&    [3.4.2]
      multi_statements=0&      [3.3.0]
      config_file=~/.my.cnf&   [3.4.6]
      config_group=parser      [3.4.6]
```

Как правило, для подключения не нужно указывать дополнительные параметры:
`mysql://user:password@localhost/database`

Можно вместо `имени_хоста` и `номера_порта` передать путь к UNIX-сокету в квадратных скобках (UNIX-сокет — это некий магический набор символов (путь), который известен администратору MySQL. Через этот сокет может идти общение с сервером):
`mysql://user:password@[/unix/socket]/database`

`charset` — сразу после соединения выполняет команду «`SET NAMES 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 прочитать, как работает autocommit, в том числе какие команды вызывают COMMIT);

`local_infile` — если установлен в 1, то разрешается выполнение команды LOAD DATA [LOCAL] INFILE (подробности);

`multi_statements` — если установлен в 1, то текст SQL-запроса может содержать несколько инструкций, разделенных символом ';' (*символ ";" необходимо предварять символом "^"*);

`config_file` — использовать указанный файл с настройками (например, там может быть указан сертификат для безопасного соединения);

`config_group` — читать указанную группу настроек из файла с настройками.

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 использования версии MySQL 4.1 и выше, можно даже хранить в разных таблицах данные в разных кодировках, хотя мы считаем, что в этом случае лучше всего хранить данные в кодировке `UTF-8`.
В MySQL есть разные варианты этой кодировки, следует использовать `utf8mb4`.

Допустим, данные в 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`

Тогда сразу после соединения SQL-серверу будет выдана команда **`SET NAMES cp1251`**, и сервер сам будет перекодировать принимаемые данные из кодировки `cp1251` в кодировку, в которой данные хранятся у него в таблице, и обратно.

Внимание: в данном случае необходимо указать кодировку, в которой работает сайт. Данная опция выполняет команду MySQL, поэтому необходимо использовать названия кодировок MySQL-сервера, которые отличаются от названий кодировок Parser 3, определяемых в конфигурационном файле.

Пример: база в windows-1251, страницы в koi8-r, перекодирование драйвером (работает со всеми версиями MySQL-сервера)
В редких случаях бывает, что невозможно использовать функции перекодирования, предоставляемые MySQL-сервером. Тогда можно задействовать механизмы перекодирования драйвера, используя опцию `ClientCharset`.
Допустим, данные в 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`

Тогда отправляемые SQL-серверу данные будут перекодироваться драйвером из **`$request:charset`** (в данном примере `koi8-r`) в кодировку `windows-1251`, а принимаемые от SQL-сервера данные — обратно.

Внимание: в данном случае нужно указать кодировку, в которой данные хранятся в БД. В этой опции нужно указывать названия кодировок Parser 3, которые определяются в конфигурационном файле.

Пример: подключение к кластеру MySQL

```
mysql://user:password@node1,node2,node3/?timeout=1
```

Сперва произойдет попытка подключения к серверу `node1`. Если соединение не будет установлено в течение секунды (а обычно это занимает тысячные доли секунды), то произойдет переключение на сервер `node2`, если и с ним не получится установить соединение в течение секунды, произойдет переключение на сервер `node3`.

Пример: подключение к SphinxQL

```
mysql://@localhost:9306/?ClientCharset=utf-8
```

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 задается 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 3 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, разделенных символом ';' (*символ ";" необходимо предварять символом "^"*);

`ClientCharset` — по умолчанию драйвер перекодирует все отправляемые текстовые данные в UTF-8 и обратно (числа и BLOB не перекодируются), однако в некоторых случаях, если есть БД, содержащая данные в иной кодировке (что в применении к SQLite некорректно), используя данную опцию, можно задать кодировку, в которую драйвер будет перекодировать данные при общении с SQL-сервером.

В драйвер SQLite добавлена функция regexp(expr, string), реализованная на базе упрощённой библиотеки регулярных выражений. Параметр expr задаёт шаблон регулярного выражения, а string — строку, которую нужно проверить на соответствие этому шаблону. ***[3.5.0]***

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&
      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 3 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 3 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 путем добавления в него `TOP (limit+offset)`, для последнего — `FIRST (limit) SKIP (offset)`.

Рекомендуем сайт connectionstrings.com, где собраны строки соединения ко всевозможным базам данных.

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 формате ANSI SQL92 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 3 символ ";" в строке подключения к БД необходимо предварять символом "^".

Пример
Допустим, данные хранятся на MS-SQL-сервере в кодировке windows-1251, тогда 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&
      autocommit=1&      [3.3.0]
      standard_conforming_strings=0&      [3.4.3]
      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 с сервером выполняет команду `SET CLIENT_ENCODING=значение`;
`ClientCharset` — задает кодировку, в которой необходимо общаться с SQL-сервером, перекодированием занимается драйвер;

`autocommit` — по умолчанию Parser 3 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 сразу после соединения с сервером драйвер выполнит команду `SET DATESTYLE=значение`;

`standard_conforming_strings` — если установлен в 1, то отключается эскейпинг символа '\' для соответствия SQL-стандартам.

Пример: перекодирование средствами SQL-сервера (рекомендуется)
Допустим, данные в 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`

Тогда 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 данные хранятся у него в таблице, и обратно.

Внимание: в этом случае следует указать кодировку, в которой работает сайт. Эта опция выполняет команду Postgres, поэтому необходимо использовать названия кодировок Postgres-сервера, которые отличаются от названий кодировок Parser 3, определяемых в конфигурационном файле.

Пример: перекодирование драйвером (работает со всеми версиями Postgres-сервера)
В редких случаях бывает, что невозможно использовать функции перекодирования, предоставляемые Postgres-сервером. Тогда можно задействовать механизмы перекодирования драйвера, используя опцию `ClientCharset`.
Допустим, данные в 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`

Тогда данные, отправляемые SQL-серверу, 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 этом случае нужно указать кодировку, в которой данные хранятся в БД. В этой опции нужно указать названия кодировок Parser 3, которые определяются в конфигурационном файле.

Для Oracle

```
oracle://user:password@service?
      ClientCharset=кодировка&
      LowerCaseColumnNames=0&
      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 в ВЕРХНИЙ регистр. Parser 3 же по умолчанию преобразует их в нижний регистр. 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 возникновения 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` так, как указано выше.

Пример
Допустим, данные в 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
```

Примечание. Существует специальная конструкция для записи больших строковых литералов. Oracle не умеет 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 «2000 букв» + «2000 букв», то также будет выдана ошибка «Слишком длинная сумма». Для хранения таких конструкций используется тип данных CLOB[Oracle] и OID[Postgres], а для того чтобы SQL-команды были максимально просты, при записи таких строк необходимо лишь добавить управляющий комментарий, который драйвер соответствующего SQL-сервера обработает нужным образом:

```
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.*

ClientCharset. Параметр подключения — кодировка общения с SQL-сервером

Параметр `ClientCharset` определяет кодировку, в которой необходимо общаться с SQL-сервером. Если параметр не указан, Parser 3 считает, что общение с SQL-сервером идет в кодировке **`$request:charset`**.

Список допустимых кодировок определяется в Конфигурационном файле.

Приложение 4. Perl-совместимые регулярные выражения

Подробная информация о Perl-совместимых регулярных выражениях (Perl Compatible Regular Expressions, PCRE) доступна в документации к Perl (см. perldoc.perl.org/perlre) в документации к использованной в Parser 3 библиотеке PCRE (см. pcre.org/current/doc/html/), а также большом количестве специальной литературы, содержащей, помимо всего остального, множество 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 собой, 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; предусматривают 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; совпадает с любым элементом из списка, заданного в квадратных скобках
|	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, указанных 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 допустимо использовать только следующие метасимволы:

****	escape-символ
^	инвертированный символьный класс, метасимвол обязательно должен быть первым символом в описании класса; совпадение будет происходить с любыми символами, не входящими в символьный класс
-	обозначение интервала символов
[...]	ограничитель символьного класса

Использование метасимвола «\»

Обратная косая черта предусматривает несколько вариантов использования. В случае если вслед за ним следует символ, не обозначающий букву алфавита, она 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. Если, например, необходимо найти символ «*****», то используется следующая запись в шаблоне «*****». В случае необходимости экранировать сам символ «****» используется запись «****».

Второй вариант использования этого метасимвола — описание 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 шестнадцатеричный код символа **hh**;
\ddd восьмеричный код символа **ddd.**

Третий вариант — определение специфических символьных классов:

\d любая десятичная цифра **[0-9]**;
\s пропуск, пробел, обычно **[\f\n\r\t]** (первый символ в квадратных скобках — пробел);
\w символ слова, обычно **[a-zA-Z0-9_]**;
\D \S \W отрицание **\d \s \w**.

Четвертый вариант — обозначение 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 целесообразно рассматривать как мнимые символы нулевого размера, расположенные на границе между реальными символами в точке, которая соответствует 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 самому разработчику, а в идеале — любому человеку, читающему написанный код. Имя может быть набрано русскими или латинскими буквами, главное — единообразие. Рекомендуем все же пользоваться английским. Слова в именах лучше использовать в единственном числе. Составные имена рекомендуется записывать в виде **column_color**. Глядя на такое имя, можно сразу понять, что оно означает.

Parser 3 чувствителен к регистру!

$Parser и **$parser** — разные переменные.

Есть определенные ограничения на использование в именах символов. Для Parser 3 имя всегда заканчивается перед:
- пробелом;
- табуляцией;
- переводом строки;

- символами **;] }) " < > # + * / % & | = ! ' , ?**;
- символом «**-**» минуса (в выражениях).

Код:
```
$var[значение_из_переменной]
$var>text
```

выдаст на экран:
```
значение_из_переменной>text
```

Т. е. символ «**>**» Parser 3 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
```

выдаст:
```
значение_из_переменной.text
```

Нельзя пользоваться в именах символами "**.**", "**:**", "**^**", поскольку они будут расцениваться как часть кода Parser 3, что приведет к ошибкам при обработке написанного 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 (на практике не встречаются), за исключением знака подчеркивания, который не используется Parser 3 и достаточно нагляден при использовании в именах.

Приложение 6. Как бороться с ошибками и разбираться в чужом коде

Для начала следует вдумчиво изучить сообщение об ошибке. В нем содержится имя файла, вызвавшего ошибку, и номер строки в нем. Здесь требуется внимательность при написании кода и справочник. Если номер строки не указан, следует проверить парность добавленных скобок, в сомнительных случаях комментируя строки символом **#**, чтобы быстрее локализовать ошибочный фрагмент кода. Всегда нужно помнить о том, что Parser 3 оперирует объектной моделью, и внимательно следить за тем, с объектом какого класса ведется работа. 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 преодолевается довольно быстро. Еще одна категория ошибок — ошибки в логике работы самого кода. Это уже сложнее, и придется запастись терпением. Необходимо давать грамотные имена переменным, методам, классам и комментировать код.

Если и в этом случае не удастся понять причины неверной работы — следует обратиться к справочнику. «*Если ничего не помогает — прочтите наконец инструкцию...*» Последняя стадия в поиске ошибок — разработчик близок к сумасшествию, пляшет вокруг компьютера с бубном, а код все равно не работает. Здесь остается только обратиться за помощью к тем, кто пока разбирается в Parser 3 чуть лучше. На сайте parser.ru есть форум — там ответят на любой вопрос.

Приложение 7. SQL-серверы, 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, рекомендуется пользоваться PL/SQL-оберткой (**begin** ...**; end;**), не забывая 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!
```

Индекс

- - -

- 54

- ! -

! 54
!| 54
!‖ 54
!= 54

- # -

56

- % -

% 54

- & -

& 54
&& 54

- * -

* 54

- . -

.csv * 196
.htaccess * 156
.log * 223

- / -

/ 54

- @ -

@GET_имя 49
@SET_имя[значение] 49

- \ -

\ 54

- _ -

_count 127
_default 124, 126
_keys 128

- | -

| 54
‖ 54

- ~ -

~ 54

- + -

+ 54

- < -

< 54
<= 54
<FORM ... 120
<IMG ... 138
<IMG ISMAP ... 122
<xsl:output ... 213
<xsl:param * 212

- = -

== 54

- > -

> 54
>= 54

- 4 -

404 225

- A -

abs 155
acos 160
Action * 223
adate 113
add 82, 131
AddHandler * 223

alt 138
and * 54
Apache 223, 224
Apache module 224
apache passwords * 156
append 82, 198
appendChild 215
apply-taint 71
arc 139
argv 175
array 41, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 198
  Класс 79
  Конструкторы 79, 80
  Методы 82, 83, 84, 85, 86, 87, 88
  Поля 81
asc 206
asin 160
as-is 65
at 82, 126
at service * 226
atan 160
ATTRIBUTE_NODE 218
attributes 217
auto 43, 78
auto.p 17, 22, 27, 32, 219, 220, 223, 224
auto-compact 165

– B –

background * 136
banner system * 123
bar 139
BASE 43
base * 209
base64 108, 114, 184, 186
basename 116
binary 115
bind variables * 239
body 175, 178
body * 153
bool 53, 105, 187
  Методы 105
border 138
bound variables * 239
brackets * 49

– C –

cache 60
calendar 102, 103
caller 46
caller.self 46
case 58
case * 193
catch * 72
cbr * 210
CDATA_SECTION_NODE 218
cdate 113
ceiling 159
cells 199
cgi 109, 223
CGI_* 106, 109
CGI_PARSER_CONFIG 223
CGI_PARSER_LOG 223
char 222
charset 77, 153, 175, 178, 230
CharsetDisable * 223
charsets 219, 221, 222
childNodes 217
circle 139
CLASS 42, 43, 46
class_alias 166
CLASS_PATH 229
cleanup 134
clear 179
ClientCharset 235
clone * 124, 194
cloneNode 215
columns 199
comment 56, 72
comment * 22, 63
COMMENT_NODE 218
compact 83, 165
compile 227
conf 219, 220, 221
connect 61, 230, 231, 232, 233, 234, 235
  Формат строки подключения 230, 231, 232, 233, 234, 235
console 89
  Класс 89
  Статическое поле 89
constructor * 42
contains 83, 127
content-type 153
content-type * 178
cookie 65, 89, 90
  Запись 90
  Класс 89
  Статические поля 90
  Чтение 89
copy 79, 116, 140
copy * 194

cos 160
count 199
count * 83, 127
counter * 119
cp * 116
crc32 115, 117, 156
create 77, 79, 97, 98, 99, 111, 124, 135, 142, 194, 208, 209
create table * 207
createAttribute 210
createCDATASection 210
createComment 210
createDocumentFragment 210
createElement 210
createElementNS 210
createEntityReference 210
createProcessingInstruction 210
createTextNode 210
cron * 226
crypt 156
cur 205
currency * 210
CVS 227

– – –

-d 54, 56

– D –

dashed * 139
date 97, 98, 99, 100, 101, 102, 103
  Класс 97
  Конструкторы 97, 98, 99
  Методы 100, 101, 102
  Поля 100
  Статические методы 102, 103
day 100, 101
daylightsaving 100
deadlock * 119, 133
dec 105
def 54, 55, 183, 194
default 106, 124, 126, 185
degree 158
delete 83, 117, 127, 134, 200
delete from * 207
desc 206
diagram * 178
digest * 115, 120, 158
digit 222
dir * 119
directory * 117, 223
dirname 117
div 105
DOCUMENT_FRAGMENT_NODE 218
DOCUMENT_NODE 218
DOCUMENT_ROOT * 176
DOCUMENT_TYPE_NODE 218
document-root 176
DocumentRoot * 176
DOM 37, 208, 210, 212, 214, 215
DOM1 210, 212, 215
DOM2 210, 215
domain 90
dotted * 139
double 53, 100, 104, 105, 106, 187
  Класс 104
  Методы 104, 105, 106
download 179
download * 178
draw * 135
drivers * 221
DSN 232

– E –

ELEMENT_NODE 218
elements 121
ellipse * 139, 144
encoding * 221
eng 102, 103
ENTITY_NODE 218
ENTITY_REFERENCE_NODE 218
env 65, 106, 107, 109
  Класс 106
  Получение версии Parser 107
  Получение значения поля запроса 107
  Статические поля 107
eq 54
equal * 54
error * 238
error.log * 223
error_log * 223
eval 57
eval comment * 56
ever_allocated_since_compact 180
ever_allocated_since_start 180
Excel * 196
exception 72, 76, 238
exec 109
EXIF * 135, 136, 137
exists * 56

exp 158
expires 90, 134
expires * 179
extension * 118

– – –

-f 54, 56

– F –

false 53
fields 90, 122, 126, 197
file 72, 107, 108, 109, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 153, 209, 211
  Класс 107
  Конструкторы 108, 109, 111, 112, 113
  Методы 114, 115, 116
  Поля 113
  Статические методы 116, 117, 118, 119, 120
file.access 72
file.missing 72
file::load 229
filename * 116
files 122
Files * 223
file-spec 65
fill 140
filled 139, 142
filled * 144
filter * 87, 129, 206
find 117
find * 189
firstChild 217
firstthat * 203
flip 200
floor 159
font 141
for 59, 84
foreach 84, 127, 135
foreach * 204
form 65, 100, 120, 121, 122, 123, 187
  Класс 120
  Статические поля 121, 122, 123
format 104, 187
format * 57
format specifiers * 229
frac 160
free 180
from 153
fullpath 118

– G –

ge 54
GET * 120
GET_имя 49
getAttribute 215
getAttributeNode 215
getElementById 210
getElementsByTagName 210, 215
getElementsByTagNameNS 215
getter * 49
GIF 136, 138
GIF * 136, 138
gmtime * 101
graph * 178
greater or equal * 54
greater then * 54
gt 54
GUID * 161

– H –

handled 72
has intersection * 132
hasAttribute 215
hasAttributeNS 215
hasAttributes 215
hasChildNodes 215
hash 41, 55, 124, 126, 127, 128, 129, 130, 131, 132, 135, 201
  Использование хеша вместо таблицы 126
  Класс 124
  Конструкторы 124
  Методы 126, 127, 128, 129, 130, 131, 132
  Поля 126
hash of bool * 124
hash of hash * 124
hashfile 133, 134, 135
  Запись 134
  Класс 133
  Конструктор 133
  Методы 134, 135
  Чтение 134
hashing passwords * 156
have method * 147
headers 176, 179
height 137
hexadecimal * 53
hex-digit 222
hostname 145

hour 100
htaccess * 156
html 65, 138, 153, 208, 211
HTTP * 77, 89, 111, 113, 120, 174, 177, 178, 195, 210, 229
HTTP://www.cbr.ru/scripts/XML_daily.asp 210
HTTP_ * 106, 107, 109
HTTP_USER_AGENT * 107
HTTP-header 65

– I –

if 53, 57
ifdef * 55
IIS 225
image 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145
Класс 135
Конструкторы 135, 136
Методы 138
Методы рисования 138, 139, 140, 141, 142, 143, 144, 145
Поля 137
image * 178
image.format 72
imap 122
img 138
importNode 210, 215
in 54, 55
IN * 239
IN/OUT * 239
inc 105
include 61
include * 46, 109
inetd * 89
insert 85, 203
insert into * 207
insertBefore 215
install 219, 220, 221, 222, 223, 224, 225, 226
Установка и настройка Parser 219, 220, 221, 222, 223, 224, 225, 226
int 53, 100, 104, 105, 106, 187
Класс 104
Методы 104, 105, 106
intersection 131
intersects 132
is 54, 55
ISMAP * 122

– J –

join 85, 203
JPEG * 136
JPG * 136
js 65
junction 147
Класс 147
justext 118
justname 118

– K –

keys 85
keys * 128

– L –

lastChild 217
last-day 101, 103
le 54
left 85, 188
legend * 140
length 142, 189
less or equal * 54
less then * 54
letter 222
limit 80, 106, 124, 185, 195
LIMITS 221
line 140, 142, 204
lineno 72
line-style 139
line-width 139
list 119
load 65, 111, 136, 195, 210
local 233
localtime * 101
locate 203
location 177
lock 119
log 158
log10 158
log-filename 181
loop * 59
lower 193
lowercase 222
ls * 119
lsplit * 192
lt 54

– M –

mail 152, 153, 221
Класс 152

mail 152, 153, 221
Статические методы 153
mail-header 65
MAIN 17, 27, 32, 42, 46, 78
make * 227
match 185, 189, 190
math 155, 156, 158, 159, 160, 161
Класс 155
Статические методы 155, 156, 158, 159, 160, 161
Статические поля 155
md5 115, 120, 158
mdate 113
measure 136
memory 165, 180
Класс 165
Методы 165
menu 204
message 153
method 176
method exists * 147
mid 86, 190
mime-type 113
MIME-TYPES 178, 221
minute 100
mod 105
mod_rewrite * 118, 225
mode 181
month 100, 101, 103
move 120
mul 105
multiply * 131
mv * 120
mysql 230

– N –

name 113, 217
name * 116, 118, 237
ne 54
news * 226
news:// * 89
nextSibling 217
NNTP * 89
no ext * 118
no path * 116
nodeName 217
nodeType 217
nodeValue 217
normalize 215
not * 54
not equal * 54
NOTATION_NODE 218
now 99, 100
NULL * 239
number * 100, 187, 204
number.format 72
number.zerodivision 72

– O –

odbc 232
offset 80, 106, 124, 185, 195, 204, 205
open 133
operator * 46, 77
optimized-html 65
or * 54
oracle 234
OUT * 239
ownerDocument 217

– P –

paint * 135
parentNode 217
parser.compile 72
parser.runtime 72
parser:// * 209
PARSER_VERSION 107
parser3.log 223
password * 156
path 90, 176, 229
path * 117
PCRE 235
Perl 235
pgsql 233
PI 155
pid 181
pixel 142
PL/SQL * 239
PNG * 136
polybar 142
polygon 143
polyline 143
pop 86
pos 191
POST * 120
PostgreSQL 233
postmatch 189
postprocess 78
pow 159
prematch 189

previousSibling 217
printf * 187
process 61
process id * 181
PROCESSING_INSTRUCTION_NODE 218
profile * 179, 180, 181
properties * 49
publicId 217
push 86

– Q –

qtail 123
query 177
query * 87, 129, 206
query tail * 123

– R –

radians 158
random 159
rectangle 144
reflection 166
refresh 177
regexp 235
regulary * 226
release 135
rem 63
remove 86
remove * 127, 134
removeAttribute 215
removeAttributeNode 215
removeChild 215
rename 129, 205
rename * 120
replace 144, 191
replace * 190
replaceChild 215
request 174, 175, 176, 177
  Класс 174
  Статические поля 175, 176, 177
request:charset 230
response 177, 178, 179
  Класс 177
  Статические методы 179
  Статические поля 177, 178, 179
result 46
reverse 87, 129
rewrite * 225
right 87, 188
roll 101
round 159
RPC 175
rsplit * 192
rus 102, 103
rusage 181

– S –

save 115, 191, 205, 211
schedule * 226
scientific * 53
script * 226
search-namespaces 213
second 100
sector 144
select 87, 129, 206, 215
selectBool 217
selectNumber 217
selectSingle 216
selectString 216
self 46
send 77, 153
server 225
session 90, 134
set 88, 130, 205
SET_имя[значение] 49
setAttribute 215
setAttributeNode 215
SetEnv * 223
setter * 49
sha1 159
shift * 205
sign 155
sin 160
size 113, 199
size * 142
smtp.connect 72
smtp.execute 72
sort 88, 130, 206
source 72
specified 217
split 192
sprintf * 187
sql 27, 61, 65, 77, 80, 97, 106, 112, 124, 185, 195, 207, 221
sql * 102
sql.connect 72
sql.execute 72
SQLite 231
sql-string 102, 116
sqrt 160

src 137
SSI * 109
stack 238
stat 113
Static fields and methods 43
status 177, 179, 180, 181, 183
  Класс 179
  Поля 180, 181, 183
string 37, 39, 52, 53, 55, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 208, 211
  Класс 183
  Методы 186, 187, 188, 189, 190, 191, 192, 193
  Статические методы 184, 185
sub 132
subject 153
substring * 190
switch 58
systemId 217

– T –

table 65, 194, 195, 196, 197, 198, 199, 200, 201, 203, 204, 205, 206
  Класс 194
  Конструкторы 194, 195
  Методы 198, 199, 200, 201, 203, 204, 205, 206
  Опции копирования и поиска 196
  Опции формата файла 196
  Получение содержимого столбца 197
  Получение содержимого текущей строки в виде хеша 197
table * 126
tables 123
tagName 217
taint 64, 65
tan 160
target 217
text 113, 115, 145, 153, 211
TEXT_NODE 218
thick * 139
thread id * 183
throw 72, 73
thumbnail * 136, 138
tid 183
time_t * 99, 102
to 153
transform 37, 212
trim 193
true 53, 124
trunc 160
try 72
type 72
TZ 100, 101

– U –

uid64 160
unhandled_exception 72, 74, 221
unicode 222
union 132
unix socket 230
unix-timestamp 99, 102
untaint 64, 70
upper 193
upsize * 191
uri 65, 177
USD * 210
USE 43, 64
used 180
USER-AGENT * 107
UTF-8 77
uuid 161
uuid7 161

– V –

void 207
  Класс 207
  Методы 207

– W –

web 225
web-server 225
week 100, 102
weekday 100
weekyear 100
while 59
white-space 222
width 137
word 222

– X –

xdoc 37, 208, 209, 210, 211, 212, 213
  parser://метод/параметр. Чтение XML из произвольного источника 209
  Класс 208
  Конструкторы 208, 209, 210
  Методы 210, 211, 212
  Параметр создания нового документа: Базовый путь 209
  Параметры преобразования документа в текст 213

xdoc 37, 208, 209, 210, 211, 212, 213
Поля 212, 213
XHTML 213
x-mailer 153
XML 37, 65, 72, 208, 211, 214
xml:base 209
XML-RPC 175
xnode 37, 214, 215, 216, 217, 218
Класс 214
Константы 218
Методы 215, 216, 217
Поля 217
xor * 54
XPath 37, 214, 215, 216, 217
XPath * 213
xsl:output ... 213
xsl:param * 212
XSLT 37

– Y –

year 100, 101
yearday 100

– Z –

Зеленые рукава * 184, 186
кодом 169
объект 124
опции формата 194
отпечаток * 115, 120, 158
память * 165, 179
свойства * 49
Статические поля и методы 43
статус * 179
точки изображения * 142