Парсер «Докувики»

В этом документе излагаются детали функционирования парсера «Докувики», которые могут понадобиться разработчикам для модификации поведения парсера или получения контроля над выходным документом, возможно, изменив сгенерированный HTML или реализовав другие форматы вывода.

Обзор

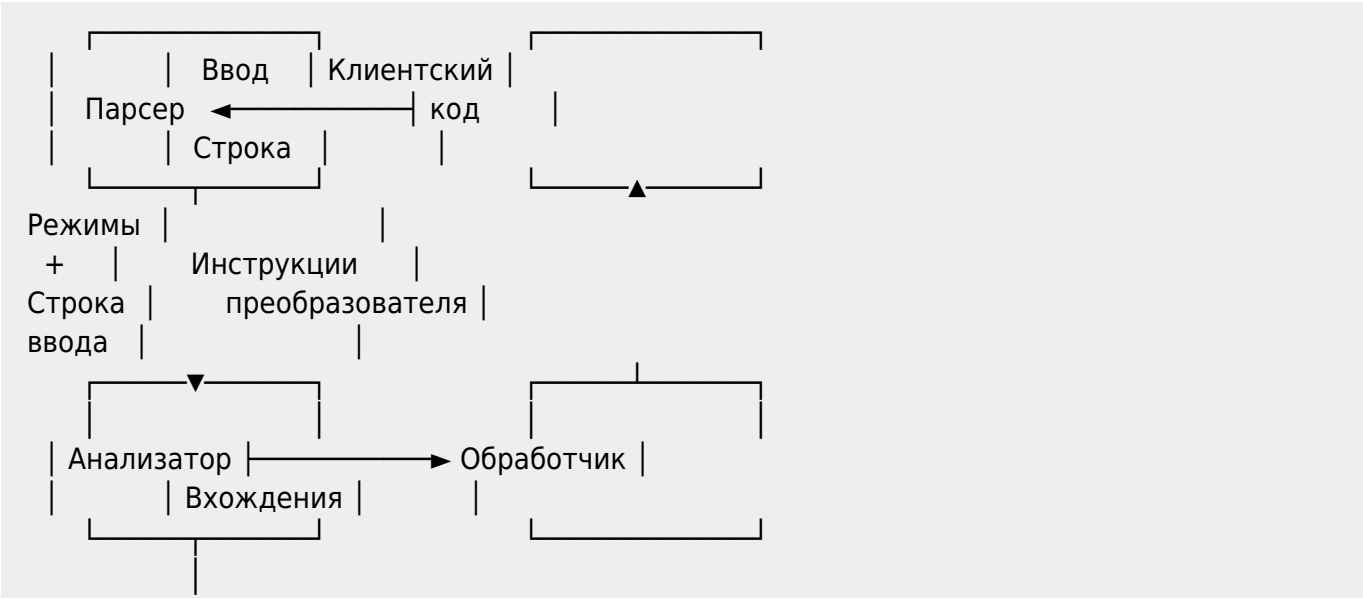
Парсер разбивает процесс трансформации исходного документа «Докувики» в финальный выходной документ (обычно XHTML) на дискретные стадии. Каждая стадия представлена одним или несколькими PHP-классами.

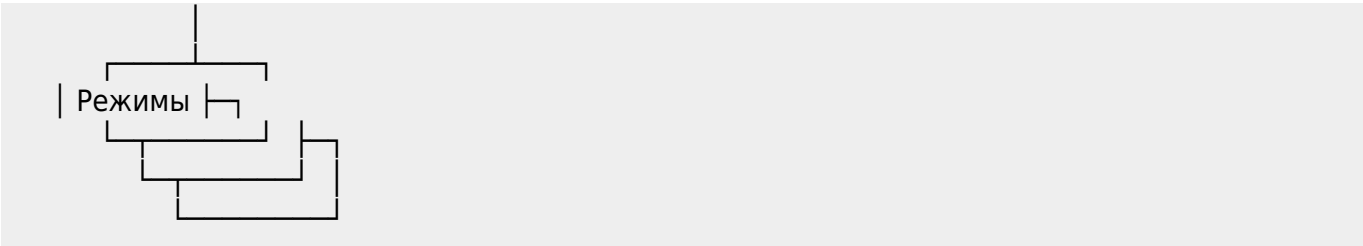
В общем рассмотрении этими элементами являются;

1. Lexer[1] (лексический анализатор): сканирует[2] исходный документ «Докувики» и выводит последовательность «вхождений» (токенов[3]), соответствующих синтаксической структуре документа.
2. Handler[4] (обработчик): получает вхождения от анализатора и преобразует их в последовательность «инструкций»[5]. Он описывает, как должет быть сформирован выходной документ, от начала до конца.
3. Parser[6] (парсер): «связывает» лексический анализатор с обработчиком, предоставляя синтаксические правила «Докувики», а также точку доступа к системе (метод `Parser::parse()`)
4. Renderer[7] (преобразователь): принимает инструкции от обработчика и «отрисовывает» готовый к выводу документ (например, в виде XHTML).

Механизма для соединения Handler (обработчика) с Renderer (логическим преобразователем) **не предусмотрено** — для этого требуется кодирование для каждого конкретного случая использования.

Схематическая диаграмма связей между компонентами;





«Клиентский код» (код, использующий парсер) вызывает парсер, передавая ему входную строку. В ответ ему возвращается перечень «инструкций» преобразователя, построенных обработчиком. Они могут быть использованы неким объектом, реализующим преобразователь.

Примечание: критическим моментом здесь является намерение позволить преобразователю быть настолько «тупым», насколько это возможно. От него *не* требуется осуществлять дальнейшую интерпретацию или модификацию переданных инструкций, но полностью сконцентрироваться на формировании выходных данных (например, XHTML) — в особенности, преобразователю не следует отслеживать состояния. Соблюдение этого принципа и, кроме того, составление преобразователя достаточно простым для реализации (сосредоточенной исключительно на том, что следует выводить), также сделает возможным преобразователю быть взаимозаменяемым (например, вывод PDF в качестве альтернативы XHTML). В то же самое время, выходные инструкции обработчика *направляются* для преобразования в XHTML и не всегда могут быть пригодынми для всех выходных форматов.

Лексический анализатор

Определяется в inc/Parsing/Lexer/Lexer.php

В самом общем смысле, он реализует инструмент для управления комплексными регулярными выражениями, где важным является состояние. Анализатор появился из Simple Test , но содержит три модификации (читай: «хака»):

- поддержка шаблонов просмотра назад и просмотра вперед[8)];
- поддержка изменения модификаторов шаблона в пределах шаблона;
- уведомление обработчика о начальном индексе байта в необработанном тексте, где был сопоставлен токен.

Короче говоря, лексер Simple Test действует как инструмент, упрощающий управление регулярными выражениями — вместо гигантских регулярных выражений вы пишете много маленьких/простых. Лексер заботится об их эффективном объединении, а затем предоставляет вам API обратного вызова в стиле SAX , чтобы вы могли писать код для ответа на соответствующие «события».

В целом Lexer состоит из трех основных классов;

- inc/Parsing/Lexer/ParallelRegex: позволяет строить регулярные выражения из нескольких отдельных шаблонов, каждый шаблон связан с идентифицирующей «меткой», класс объединяет их в одно регулярное выражение с помощью подшаблонов. *При использовании Lexer вам не нужно беспокоиться об этом классе.*
- inc/Parsing/Lexer/StateStack: предоставляет простой конечный автомат, чтобы лексирование могло быть «контекстно-зависимым». *При использовании Lexer вам не нужно беспокоиться об этом классе.*

- inc/Parsing/Lexer/Lexer: предоставляет точку доступа для клиентского кода с помощью Lexer. Управляет несколькими экземплярами ParallelRegex, используя StateStack для применения правильного экземпляра ParallelRegex в зависимости от «контекста». При обнаружении «интересного текста» вызывает функции для предоставленного пользователем объекта (Handler).

Необходимость в состояниях

Синтаксис вики, используемый в «Докувики», содержит разметку, «внутри» которой применяются только определённые синтаксические правила. Самый очевидный пример — тэг <code/> , внутри которого синтаксис вики не будет распознаваться анализатором. Для других синтаксических конструкций, таких как списки или таблицы, следует позволять использовать *некоторую* разметку, но не всю, например, в списка можно использовать ссылки, но не таблицы.

Анализатор обеспечивает «осведомлённость о состояниях», позволяющую применять корректные синтаксические правила в зависимости от текущий позиции (контекста) в сканируемом тексте. Если он видит открывающий тэг <code>, он переключается в другое состояние, в пределах которого другие синтаксические правила не применяются (т. е. что-либо, что выглядит как синтаксис вики должно восприниматься как «простой» текст), до тех пор, пока не найдёт закрывающий тэг </code>.

Режимы анализатора

Термин *режим* обозначает особенное состояние лексического анализа [9)]. Код, использующий анализатор, регистрирует один или более шаблон регулярного выражения с особенным наименованием режима. Затем анализатор, сравнивая эти паттерны со сканируемым текстом, вызывает функции обработчика с тем же самым наименованием режима (если метод `mapHandler` не был использован для создания псевдонимов — см. ниже).

API анализатора

Краткое введение в лексический анализатор можно найти в Simple Test Lexer Notes. Здесь предлагается более подробное описание.

Ключевыми методами анализатора являются:

Конструктор

Принимает ссылку на объект Handler, имя начального режима, в котором должен запускаться Lexer, и (необязательно) логический флаг, указывающий, должно ли сопоставление с образцом учитывать регистр.

Пример:

```
$handler  =  new MyHandler ( ) ;
```

```
$lexer  =  new dokuwiki\Lexer\Lexer ( $handler ,  'base' ,  true ) ;
```

Здесь указан начальный режим 'base'.

addEntryPattern / addExitPattern

addEntryPattern() в Lexer.php

Lexer.php

```
75.     public function addEntryPattern($pattern, $mode, $new_mode)
76.     {
77.         if (! isset($this->regexes[$mode])) {
78.             $this->regexes[$mode] = new
    ParallelRegex($this->case);
79.         }
80.         $this->regexes[$mode]->addPattern($pattern, $new_mode);
81.     }
```

addExitPattern() в Lexer.php

Lexer.php

```
89.     public function addExitPattern($pattern, $mode)
90.     {
91.         if (! isset($this->regexes[$mode])) {
92.             $this->regexes[$mode] = new
    ParallelRegex($this->case);
93.         }
94.         $this->regexes[$mode]->addPattern($pattern, "__exit");
95.     }
```

addEntryPattern() и addExitPattern() используются для регистрации шаблона для входа и выхода из определенного режима анализа. Например;

```
// arg0: регулярное выражение для сопоставления — обратите внимание, что нет
необходимости добавлять разделители начального/конечного шаблона
// arg1: имя режима, в котором может использоваться этот шаблон записи
// arg2: имя режима для ввода
$lexer -> addEntryPattern ( '<file>' , 'base' , 'file' ) ;

// arg0: регулярное выражение для сопоставления
// arg1: имя режима для выхода
$lexer -> addExitPattern ( '</file>' , 'file' ) ;
```

Код, приведённый выше, позволяет тэгу <file/> быть использованный при входе из базового в новый режим (`file`). Если в дальнейшем следует применить режимы, пока анализатор находится в режиме `file`, они должны быть зарегистрированы с режимом `file`.

Замечание: в паттернах не требуется использование ограничителей (разделителей начала и конца шаблона).

addPattern

`addEntryPattern()` в Lexer.php

Lexer.php

```
57.     public function addPattern($pattern, $mode = "accept")
58.     {
59.         if (! isset($this->regexes[$mode])) {
60.             $this->regexes[$mode] = new
    ParallelRegex($this->case);
61.         }
62.         $this->regexes[$mode]->addPattern($pattern);
63.     }
```

`addEntryPattern()` в ParallelRegex.php

ParallelRegex.php

```
53.     public function addPattern($pattern, $label = true)
54.     {
55.         $count = count($this->patterns);
56.         $this->patterns[$count] = $pattern;
57.         $this->labels[$count] = $label;
58.         $this->regex = null;
59.     }
```

addEntryPattern() используется, чтобы реагировать на дополнительные «вхождения» внутри существующего режима (без переходов). Он принимает паттерн и наименование режима, внутри которого должен использоваться.

Это наиболее наглядно видно из разбора парсером синтаксиса списков. Синтаксис списков выглядит в «Докувики» следующим образом;

```
До списка
  * Ненумерованный элемент списка
  * Ненумерованный элемент списка
  * Ненумерованный элемент списка
```

После списка

Использование addPattern() делает возможным сравнивать полный список, одновременно корректно захватывая каждый элемент списка;

```
// Сопоставляем открывающий элемент списка и меняем режим
$lexer -> addEntryPattern ( '\n {2,}[\*]' , 'base' , 'list' ) ;

// Сопоставить новые элементы списка, но остаться в режиме списка
$lexer -> addPattern ( '\n {2,}[\*]' , 'list' ) ;

// Если это перевод строки, который не соответствует указанному выше правилу addPattern, выходим из режима
$lexer -> addExitPattern ( '\n' , 'list' ) ;
```

addSpecialPattern

addSpecialPattern() в Lexer.php

Lexer.php

```
107.     public function addSpecialPattern($pattern, $mode, $special)
108.     {
109.         if (! isset($this->regexes[$mode])) {
110.             $this->regexes[$mode] = new
    ParallelRegex($this->case);
111.         }
112.         $this->regexes[$mode]->addPattern($pattern, "_$special");
113.     }
```

addSpecialPattern() используется для входа в новый режим только для совпадения, а затем сразу возвращается в «родительский» режим. Принимает шаблон, имя режима, в котором он может быть применен, и имя «временного» режима для входа для совпадения. Обычно это используется, если вы хотите заменить разметку wiki чем-то другим. Например, чтобы сопоставить смайлик, например :-), у вас может быть:

```
$lexer -> addSpecialPattern ( ':-)' , 'base' , 'smiley' ) ;
```

mapHandler

mapHandler() в Lexer.php

Lexer.php

```
121.     public function mapHandler($mode, $handler)
```

```
122.     {
123.         $this->mode_handlers[$mode] = $handler;
124.     }
```

mapHandler() позволяет сопоставить определенный именованный режим с методом с другим именем в Handler. Это может быть полезно, когда разный синтаксис должен обрабатываться одинаково, например, синтаксис DokuWiki для отключения другого синтаксиса внутри определенного текстового блока;

```
$lexer -> addEntryPattern ( '<nowiki>' , 'base' ,  'unformatted' ) ;
$lexer -> addEntryPattern ( '%%' , 'base' ,  'unformattedalt' ) ;
$lexer -> addExitPattern ( '</nowiki>' ,  'unformatted' ) ;
$lexer -> addExitPattern ( '%%' ,  'unformattedalt' ) ;

// Оба синтаксиса должны обрабатываться одинаково...
$lexer -> mapHandler ( 'unformattedalt' ,  'unformatted' ) ;
```

Подшаблоны не допускаются

Поскольку Lexer (анализатор) сам использует **подшаблоны** (внутри класса ParallelRegex), код, *использующий* анализатор, этого не может. Иногда это может пригодиться, но, по общему правилу, метод addPattern() может быть применён для решения проблем, когда обычно применяются подшаблоны. Его преимущество в том, что он делает регулярные выражения более простыми и, следовательно, более простыми в управлении.

Замечание: если вы используете в шаблоне круглые скобки, они будут *автоматически* пропущены анализатором.

Синтаксические ошибки и состояния

Для предотвращение «плохо форматируемой» (особенно при пропуске закрывающих тэгов) разметки, приводящей к тому, что Lexer (анализатор) входит в состояние (режим), который он никогда не покинет, может быть полезным использование паттерна просмотра вперёд для проверки наличия закрывающей разметки[10]. Например:

```
// Использование просмотра вперёд во входном шаблоне...
// Использовать предпросмотр в шаблоне записи...
$lexer -> addEntryPattern ( '<file>(?=.*</file>)' , 'base' , 'file' ) ;
$lexer -> addExitPattern ( '</file>' , 'file' ) ;
```

Шаблон входа проверяет, может ли он найти закрывающий </file> тег, прежде чем войти в состояние.

Handler (Обработчик)

Определено в inc/parser/handler.php и папке `inc/Parsing/Handler`

```
Handler
├─ AbstractRewriter.php
├─ Block.php
├─ CallWriter.php
├─ Nest.php
├─ Lists.php
├─ CallWriterInterface.php
├─ Preformatted.php
├─ Quote.php
├─ ReWriterInterface.php
└─ Table.php
```

Handler — это класс, предоставляющий методы, которые вызывает Lexer, когда он сопоставляет токены. Затем он «тонко настраивает» токены в последовательность инструкций, готовых для Renderer.

Обработчик в целом содержит следующие классы:

- Doku_Handler: все вызовы из Lexer производятся в этот класс. Для каждого режима, зарегистрированного в Lexer, будет соответствующий метод в Handler

`Doku_Handler` в /dokuwiki/inc/Extension/SyntaxPlugin.php

SyntaxPlugin.php

```
 6  use Doku_Handler;
57  * @see Doku_Handler_Block
75  * @param Doku_Handler $handler The Doku_Handler object
77  * @param   Doku_Handler $handler The Doku_Handler object
80  abstract public function handle($match, $state, $pos, Doku_Handler
$handler);
```

`Doku_Handler` в /dokuwiki/inc/Parsing/Parser.php

Parser.php

```
6 use Doku_Handler;
17  /** @var Doku_Handler */
32  * @param Doku_Handler $handler
34  public function __construct(Doku_Handler $handler)
```

`Doku_Handler` в /dokuwiki/inc/parser/handler.php

handler.php

```
16   * Class Doku_Handler
18   class Doku_Handler {
40   * Doku_Handler constructor.
795  $link[1] = Doku_Handler_Parse_Media($link[1]);
878  $p = Doku_Handler_Parse_Media($match);
1023 function Doku_Handler_Parse_Media($match) {
```

- CallWriter: обеспечивает слой между массивом инструкций (массив `Doku_Handler::$calls`) и методами Handler, *записывающими* эти инструкции. Пока идёт лексический анализ, он будет временно перемещён другими объектами, вроде `dokuwiki\Parsing\Handler\List`.
- List: отвечает за преобразование токенов списка в инструкции, пока выполняется лексический анализ
- Quote: отвечает за преобразование токенов `blockquote` (текст, начинающийся с одного или нескольких >) в инструкции, пока выполняется лексический анализ
- Table: отвечает за преобразование токенов таблицы в инструкции, пока выполняется лексический анализ
- Block: отвечает за вставку инструкций «p_open» и «p_close», при этом отслеживая инструкции «уровня блока», после завершения всего лексического анализа (т.е. выполняет цикл один раз по всему списку инструкций и вставляет больше инструкций)
- AbstractRewriter: расширено Preformattedи Nest...

- Nest: ...

- Preformatted: отвечает за преобразование предварительно отформатированных токенов (отступ в тексте dokuwiki) в инструкции, пока лексический анализ еще выполняется

Методы токенов обработчиков

Обработчик должен предоставлять методы, названные в соответствии с режимами, зарегистрированными в лексическом анализаторе (имея в виду mapHandler() метод лексического анализатора — см. выше).

Например, если вы зарегистрировали режим файла с помощью Lexer, например:

```
$lexer->addEntryPattern('<file>(?=.*</file>)','base','file');
$lexer->addExitPattern('</file>','file');
```

Обработчику понадобится такой метод:

```
class Doku_Handler {

    /**
    * @param string match содержит совпавший текст
    * @param int state - тип найденного соответствия (см. ниже)
    * @param int pos - индекс байта, где было найдено совпадение
    */
```

```
    public function file($match, $state, $pos) {
        return true;
    }
}
```

Примечание: метод Handler *должен* возвращать **true**, иначе Lexer немедленно остановится. Такое поведение может быть полезным при работе с другими типами проблем синтаксического анализа, но для парсера DokuWiki все методы Handler *всегда* будут возвращать **true**.

Аргументы, реализумые методом обработчика;

- `$match`: текст, который был обнаружен;
- `$state`: содержит константу, которая описывает как именно было найдено совпадение:
  1. DOKU_LEXER_ENTER: найден входной паттерн (см. Lexer::addEntryPattern);
  2. DOKU_LEXER_MATCHED: найден паттерн (см. Lexer::addPattern);
  3. DOKU_LEXER_UNMATCHED: внутри режима не было совпадений;
  4. DOKU_LEXER_EXIT: найден выходной паттерн (см. Lexer::addExitPattern);
  5. DOKU_LEXER_SPECIAL: найден специальный паттерн (см. Lexer::addSpecialPattern);
- `$pos`: это индекс байта (длина строки от начала), где было найдено *начало* вхождения. `$pos + strlen($match)` даёт индекс байта конца совпадения.

В качестве более сложного примера, для поиска списков в парсере определено следующее;

```
    function connectTo($mode) {
        $this->Lexer->addEntryPattern('\n {2,}[\-\*]',$mode,'listblock');
        $this->Lexer->addEntryPattern('\n\t{1,}[\-\*]',$mode,'listblock');

        $this->Lexer->addPattern('\n {2,}[\-\*]','listblock');
        $this->Lexer->addPattern('\n\t{1,}[\-\*]','listblock');

    }

    function postConnect() {
        $this->Lexer->addExitPattern('\n','listblock');
    }
```

Метод `listblock` в обработчике (вызов просто `list` приводит к ошибке обработчика PHP, поскольку `list` зарезервировано в PHP) выглядит как:

```
    function listblock($match, $state, $pos) {

        switch ( $state ) {

            // Начало списка...
            case DOKU_LEXER_ENTER:
                // Создать List rewrite, пропуская текущий CallWriter
                $ReWriter = & new Doku_Handler_List($this->CallWriter);

                // Заменить текущий CallWriter на List rewriter
                // все поступающие вхождения (даже, если они не являются вхождениями
```

```
list)
                // теперь направляются в list
                $this->CallWriter = & $ReWriter;

                $this->__addCall('list_open', array($match), $pos);
            break;

            // Для конца списка
            case DOKU_LEXER_EXIT:
                $this->__addCall('list_close', array(), $pos);

                // Дать указание List rewriter об очистке
                $this->CallWriter->process();

                // Восстановить прежний CallWriter
                $ReWriter = & $this->CallWriter;
                $this->CallWriter = & $ReWriter->CallWriter;

            break;

            case DOKU_LEXER_MATCHED:
                $this->__addCall('list_item', array($match), $pos);
            break;

            case DOKU_LEXER_UNMATCHED:
                $this->__addCall('cdata', array($match), $pos);
            break;
        }
        return TRUE;
    }
```

Конвертирование вхождений

«Тонкая обработка» задействует вставку символа дроби «/», переименование или удаление вхождений, переданных анализатором.

Например, список вроде:

```
This is not a list
  * This is the opening list item
  * This is the second list item
  * This is the last list item
This is also not a list
```

в результате превратиться в последовательность вхождений вроде;

1. base: «This is not a list", DOKU_LEXER_UNMATCHED
2. listblock: «\n *", DOKU_LEXER_ENTER
3. listblock: « This is the opening list item", DOKU_LEXER_UNMATCHED
4. listblock: «\n *", DOKU_LEXER_MATCHED

5. listblock: « This is the second list item", DOKU_LEXER_UNMATCHED
6. listblock: «\n *", DOKU_LEXER_MATCHED
7. listblock: « This is the last list item", DOKU_LEXER_UNMATCHED
8. listblock: «\n", DOKU_LEXER_EXIT
9. base: «This is also not a list", DOKU_LEXER_UNMATCHED

Но чтобы быть использованными преобразователем, это может быть конвертировано в следующие инструкции:

1. p_open:
2. cdata: «This is not a list"
3. p_close:
4. listu_open:
5. listitem_open:
6. cdata: « This is the opening list item"
7. listitem_close:
8. listitem_open:
9. cdata: « This is the second list item"
10. listitem_close:
11. listitem_open:
12. cdata: « This is the last list item"
13. listitem_close:
14. list_close:
15. p_open:
16. cdata: «This is also not a list"
17. p_close:

В случае со списками, это требует помощи класса Doku_Handler_List, который принимает вхождения, заменяя их на корректные инструкции для Преобразователя.

Parser (Парсер)

Определено в /dokuwiki/inc/Parsing/Parser.php и /dokuwiki/inc/parser/parser.php.

dokuwiki \Parsing\Parser действует как интерфейс для внешнего кода и настраивает Lexer с помощью шаблонов и режимов, описывающих синтаксис DokuWiki.

Использование парсера обычно выглядит так:

```
// Создаем обработчик Handler
$handler = new Doku_Handler();

// Создаем парсер с обработчиком
$parser = new dokuwiki\Parsing\Parser($handler);

// Добавить требуемые режимы синтаксиса в парсер
$parser->addMode('footnote', new dokuwiki\Parsing\ParserMode\Footnote());
$parser->addMode('hr', new dokuwiki\Parsing\ParserMode\Hr());
$parser->addMode('unformatted', new
dokuwiki\Parsing\ParserMode\Unformatted());
```

```
# etc.

$doc = file_get_contents('wikipage.txt.');
$instructions = $parser->parse($doc);
```

Более подробные примеры приведены ниже.

В целом Parser также содержит классы, представляющие каждый доступный режим синтаксиса, базовым классом для всех них является dokuwiki\Parsing\ParserMode\AbstractMode. Поведение этих режимов лучше всего понять, рассмотрев примеры добавления синтаксиса далее в этом документе.

Причина представления режимов с помощью классов заключается в том, чтобы избежать повторных вызовов методов Lexer. Без них пришлось бы жестко кодировать каждое правило шаблона для каждого режима, в котором может быть сопоставлен шаблон, например, регистрация одного правила шаблона для синтаксиса ссылок CamelCase потребовала бы чего-то вроде:

```
$lexer->addSpecialPattern('\b[A-Z]+[a-z]+[A-Z][A-Za-z]*\b', 'base',
'camelcaselink');
$lexer->addSpecialPattern('\b[A-Z]+[a-z]+[A-Z][A-Za-z]*\b', 'footnote',
'camelcaselink');
$lexer->addSpecialPattern('\b[A-Z]+[a-z]+[A-Z][A-Za-z]*\b', 'table',
'camelcaselink');
$lexer->addSpecialPattern('\b[A-Z]+[a-z]+[A-Z][A-Za-z]*\b', 'listblock',
'camelcaselink');
$lexer->addSpecialPattern('\b[A-Z]+[a-z]+[A-Z][A-Za-z]*\b', 'strong',
'camelcaselink');
$lexer->addSpecialPattern('\b[A-Z]+[a-z]+[A-Z][A-Za-z]*\b', 'underline',
'camelcaselink');
// и т.д.
```

Каждый режим, которому разрешено содержать ссылки CamelCase, должен быть явно назван.

Вместо того, чтобы жестко кодировать это, вместо этого это реализовано с использованием одного класса, например:

```
namespace dokuwiki\Parsing\ParserMode;

class CamelCaseLink extends AbstractMode {

    public function connectTo($mode) {
        $this->Lexer->addSpecialPattern(
            '\b[A-Z]+[a-z]+[A-Z][A-Za-z]*\b', $mode, 'camelcaselink'
        );
    }
}
```

При настройке лексического анализатора парсер вызывает `connectTo()` метод объекта `dokuwiki\Parsing\ParserMode\CamelCaseLink` для каждого другого режима, который принимает синтаксис CamelCase (некоторым такой `<code />` синтаксис не нравится).

За счет усложнения понимания настройки лексического анализатора это позволяет сделать код более гибким при добавлении нового синтаксиса.

Формат данных инструкций

Плагин Parserarray — это экспортный рендерер, который показывает инструкции для текущей страницы. Он может помочь вам понять формат данных. Ниже показан пример сырого текста вики и соответствующий вывод парсера;

Исходный текст (содержит таблицу):

```
abc
| Row 0 Col 1    | Row 0 Col 2     | Row 0 Col 3        |
| Row 1 Col 1    | Row 1 Col 2     | Row 1 Col 3        |
def
```

После обработки будет возвращён следующий массив PHP (описан ниже):

```
Array(
    [0] => Array(
            [0] => document_start
            [1] => Array()
            [2] => 0
        )
    [1] => Array(
            [0] => p_open
            [1] => Array()
            [2] => 0
        )
    [2] => Array(
            [0] => cdata
            [1] => Array(
                    [0] =>

abc
                )
            [2] => 0
        )
    [3] => Array(
            [0] => p_close
            [1] => Array()
            [2] => 5
        )
    [4] => Array(
            [0] => table_open
            [1] => Array(
                    [0] => 3
                    [1] => 2
                )
            [2] => 5
```

```
    )
[5] => Array(
        [0] => tablerow_open
        [1] => Array()
        [2] => 5
    )
[6] => Array(
        [0] => tablecell_open
        [1] => Array(
                [0] => 1
                [1] => left
            )
        [2] => 5
    )
[7] => Array(
        [0] => cdata
        [1] => Array(
                [0] =>  Row 0 Col 1
            )
        [2] => 7
    )
[8] => Array(
        [0] => cdata
        [1] => Array(
                [0] =>
            )
        [2] => 19
    )
[9] => Array(
        [0] => tablecell_close
        [1] => Array()
        [2] => 23
    )
[10] => Array(
        [0] => tablecell_open
        [1] => Array(
                [0] => 1
                [1] => left
            )
        [2] => 23
    )
[11] => Array(
        [0] => cdata
        [1] => Array(
                [0] =>  Row 0 Col 2
            )
        [2] => 24
    )
[12] => Array(
        [0] => cdata
        [1] => Array(
```

```
                    [0] =>
                )
            [2] => 36
        )
    [13] => Array(
            [0] => tablecell_close
            [1] => Array()
            [2] => 41
        )
    [14] => Array(
            [0] => tablecell_open
            [1] => Array(
                    [0] => 1
                    [1] => left
                )
            [2] => 41
        )
    [15] => Array(
            [0] => cdata
            [1] => Array(
                    [0] =>  Row 0 Col 3
                )
            [2] => 42
        )
    [16] => Array(
            [0] => cdata
            [1] => Array(
                    [0] =>
                )
            [2] => 54
        )
    [17] => Array(
            [0] => tablecell_close
            [1] => Array()
            [2] => 62
        )
    [18] => Array(
            [0] => tablerow_close
            [1] => Array()
            [2] => 63
        )
    [19] => Array(
            [0] => tablerow_open
            [1] => Array()
            [2] => 63
        )
    [20] => Array(
            [0] => tablecell_open
            [1] => Array(
                    [0] => 1
                    [1] => left
```

```
            )
        [2] => 63
    )
[21] => Array(
        [0] => cdata
        [1] => Array(
                [0] =>  Row 1 Col 1
            )
        [2] => 65
    )
[22] => Array(
        [0] => cdata
        [1] => Array(
                [0] =>
            )
        [2] => 77
    )
[23] => Array(
        [0] => tablecell_close
        [1] => Array()
        [2] => 81
    )
[24] => Array(
        [0] => tablecell_open
        [1] => Array(
                [0] => 1
                [1] => left
            )
        [2] => 81
    )
[25] => Array(
        [0] => cdata
        [1] => Array(
                [0] =>  Row 1 Col 2
            )
        [2] => 82
    )
[26] => Array(
        [0] => cdata
        [1] => Array(
                [0] =>
            )
        [2] => 94
    )
[27] => Array(
        [0] => tablecell_close
        [1] => Array()
        [2] => 99
    )
[28] => Array(
        [0] => tablecell_open
```

```
            [1] => Array(
                    [0] => 1
                    [1] => left
                )
            [2] => 99
        )
    [29] => Array(
            [0] => cdata
            [1] => Array(
                    [0] =>  Row 1 Col 3
                )
            [2] => 100
        )
    [30] => Array(
            [0] => cdata
            [1] => Array(
                    [0] =>
                )
            [2] => 112
        )
    [31] => Array(
            [0] => tablecell_close
            [1] => Array()
            [2] => 120
        )
    [32] => Array(
            [0] => tablerow_close
            [1] => Array()
            [2] => 121
        )
    [33] => Array(
            [0] => table_close
            [1] => Array()
            [2] => 121
        )
    [34] => Array(
            [0] => p_open
            [1] => Array()
            [2] => 121
        )
    [35] => Array(
            [0] => cdata
            [1] => Array(
                    [0] => def

                )
            [2] => 122
        )
    [36] => Array(
            [0] => p_close
            [1] => Array()
```

```
            [2] => 122
        )
    [37] => Array(
            [0] => document_end
            [1] => Array()
            [2] => 122
        )
)
```

Верхний уровень массива — это просто список. Каждый из его дочерних элементов описывает возвратную функцию, которая будет запущена под преобразователем (см. описание Renderer ниже), также как и индекс байта исходного текста, где был найден особенный «элемент» синтаксиса вики.

Единственная инструкция

Рассмотрим единственный элемент, который представляет единственную инструкцию, из списка инструкций, приведённого выше:

```
    [35] => Array
        (
            [0] => cdata
            [1] => Array
                (
                    [0] => def

                )
            [2] => 122
        )
```

Первый элемент (индекс 0) — это имя метода или функции в Renderer, которую необходимо выполнить.

Второй элемент (индекс 1) сам по себе является массивом, каждый из элементов которого является аргументом для метода Renderer, который будет вызван.

В этом случае имеется один аргумент со значением `"def\n"`, поэтому вызов метода будет выглядеть так:

```
$Render->cdata("def\n");
```

Третий элемент (индекс 2) — это индекс байта первого символа, который «запустил» эту инструкцию в необработанном текстовом документе. Он должен быть таким же, как значение, возвращаемое функцией PHP strpos. Это можно использовать для извлечения разделов необработанного текста вики на основе позиций сгенерированных из него инструкций (пример ниже).

Примечание: Метод парсера `parse` дополняет необработанный текст вики предшествующим и последующим символом перевода строки, чтобы гарантировать корректный выход определенных состояний лексера, поэтому вам может потребоваться вычесть 1 из индекса

байта, чтобы получить правильное местоположение в исходном необработанном тексте вики. Парсер также нормализует переводы строк в соответствии со стилем Unix (т. е. все `\r\n` становятся `\n`), поэтому документ, который видит лексер, может быть меньше того, который вы ему фактически дали.

Пример массив инструкций страницы с описанием синтаксиса можно найти здесь.

Renderer (преобразователь)

Renderer (преобразователь) — это класс (или коллекция функций), определяемый вами. Его интерфейс описан в файле `inc/parser/renderer.php` и выглядит так:

```
<?php
class Doku_Renderer {

    // вырезка

    public function header($text, $level) {}

    public function section_open($level) {}

    public function section_close() {}

    public function cdata($text) {}

    public function p_open() {}

    public function p_close() {}

    public function linebreak() {}

    public function hr() {}

    // вырезка
}
```

Он используется для документирования Renderer, хотя его также можно расширить, если вы хотите написать Renderer, который захватывает только определенные вызовы.

Основной принцип того, как инструкции, возвращаемые парсером, используются против Renderer, аналогичен понятию SAX XML API - инструкции представляют собой список имен функций/методов и их аргументов. Проходя по списку инструкций, каждая инструкция может быть вызвана против Renderer (т. е. методы, предоставляемые Renderer, являются callbacks). Unlike the SAX API, where only a few, fairly general, callbacks are available (e.g. tag_start, tag_end, cdata etc.). В отличие от SAX API , где доступно только несколько, довольно общих, обратных вызовов (например, tag_start, tag_end, cdata и т. д.), Renderer определяет более явный API , где методы обычно соответствуют один к одному акту генерации вывода. В разделе Renderer, показанном выше, методы `p_open` и `p_close` будут использоваться для вывода тегов и в XHTML, соответственно, в то время как `header` функция принимает два аргумента — некоторый текст для отображения и «уровень» заголовка, поэтому вызов типа `header('Some`

`Title', 1)` будет выведен в XHTML типа `<h1>Some Title</h1>`.

Вызов рендерера с инструкциями

Клиентскому коду, использующему Parser, остается выполнить список инструкций для Renderer. Обычно это делается с помощью функции PHP call_user_func_array() function. Например;

```
// Получить список инструкций от парсера
$instructions = $parser->parse($rawDoc);

// Создаем рендерер
$renderer = new Doku_Renderer_xhtml();

// Проходим по инструкциям
foreach ($instructions as $instruction) {
    // Выполняем обратный вызов для Renderer
    call_user_func_array([$renderer, $instruction[0]], $instruction[1]);
}
```

Методы связи с рендерером

Ключевые методы Renderer для обработки различных типов ссылок:

- `function camelcaselink($link) {} // $link like "SomePage"`
  - Вероятно, это можно проигнорировать для проверки на спам — никто не должен иметь возможности ссылаться на сторонние сайты с таким синтаксисом.
- `function internallink($link, $title = null) {} // $link like "[[syntax]]"`
  - Хотя `$link` сам по себе является внутренним, `$title` может быть изображением, которое находится вне сайта, поэтому необходимо проверить
- `function externallink($link, $title = null) {}`
  - Оба изображения `$link` и `$title` (изображения) нуждаются в проверке
- `function interwikilink($link, $title = null, $wikiName, $wikiUri) {}`
  - Необходимость `$title` проверки изображений
- `function filelink($link, $title = null) {}`
  - Технически должны совпадать только действительные `file://` URL-адреса, но, вероятно, лучше все равно проверить, плюс `$title` может быть стороннее изображение
- `function windowssharelink($link, $title = null) {}`
  - Должен соответствовать только допустимым URL-адресам общих ресурсов Windows, но в любом случае проверять наличие `$title` изображений
- `function emaillink($address, $title = null) {}`
  - `$title` может быть изображение. Проверить почту тоже?$titleможет быть изображение. Проверить почту тоже?
- `function internalmedialink($src, $title = null, $align = null, $width = null, $height = null, $cache = null) {}`
  - Это не требует проверки — должно ссылаться только на локальные изображения.

        $title само по себе не может быть изображением
    - function externalmedialink($src, $title = null, $align = null, $width = null, $height = null, $cache = null) {}
        - $src нуждается в проверке

Особого внимания требуют методы, которые принимают $title аргумент, представляющий видимый текст ссылки, например;

```
<a href="https://www.example.com">This is the title</a>
```

Аргумент $title может иметь три возможных типа значений;

1. null: в вики-документе заголовок не указан.
2. string: в качестве заголовка использовалась простая текстовая строка
3. array (hash): в качестве заголовка использовано изображение.

Если это $title массив, он будет содержать ассоциативные значения, описывающие изображение;

```
$title = [
    // Может быть 'internalmedia' (локальное изображение) или 'externalmedia' (внешнее изображение)
    'type' => 'internalmedia',

    // URL-адрес изображения (может быть URL-адресом wiki или https://static.example.com/img.png)
    'src' => 'wiki:php-powered.png',

    // Для атрибута alt - строка или null
    'title' => 'Powered by PHP',

    // 'left', 'right', 'center' или null
    'align' => 'right',

    // Ширина в пикселях или null
    'width' => 50,

    // Высота в пикселях или null
    'height' => 75,

    // Кэшировать ли изображение (для внешних изображений)
    'cache' => false,
];
```

Примеры

Следующие примеры показывают общие задачи, которые будут решаться с помощью парсера.

Основной вызов

Чтобы вызвать парсер со *всеми* режимами, и обработать синтаксис документа «Докувики»:

```
require_once DOKU_INC . 'parser/parser.php';

// Создать парсер
$Parser = & new Doku_Parser();

// Добавить обработчик
$Parser->Handler = & new Doku_Handler();

// Загрузить все режимы
$Parser->addMode('listblock',new Doku_Parser_Mode_ListBlock());
$Parser->addMode('preformatted',new Doku_Parser_Mode_Preformatted());
$Parser->addMode('notoc',new Doku_Parser_Mode_NoToc());
$Parser->addMode('header',new Doku_Parser_Mode_Header());
$Parser->addMode('table',new Doku_Parser_Mode_Table());

$formats = array (
    'strong', 'emphasis', 'underline', 'monospace',
    'subscript', 'superscript', 'deleted',
);
foreach ( $formats as $format ) {
    $Parser->addMode($format,new Doku_Parser_Mode_Formatting($format));
}

$Parser->addMode('linebreak',new Doku_Parser_Mode_Linebreak());
$Parser->addMode('footnote',new Doku_Parser_Mode_Footnote());
$Parser->addMode('hr',new Doku_Parser_Mode_HR());

$Parser->addMode('unformatted',new Doku_Parser_Mode_Unformatted());
$Parser->addMode('php',new Doku_Parser_Mode_PHP());
$Parser->addMode('html',new Doku_Parser_Mode_HTML());
$Parser->addMode('code',new Doku_Parser_Mode_Code());
$Parser->addMode('file',new Doku_Parser_Mode_File());
$Parser->addMode('quote',new Doku_Parser_Mode_Quote());

// Здесь требуются данные. Функции  ''get*''остаются на ваше усмотрение
$Parser->addMode('acronym',new
Doku_Parser_Mode_Acronym(array_keys(getAcronyms())));
$Parser->addMode('wordblock',new
Doku_Parser_Mode_Wordblock(array_keys(getBadWords())));
$Parser->addMode('smiley',new
Doku_Parser_Mode_Smiley(array_keys(getSmileys())));
$Parser->addMode('entity',new
Doku_Parser_Mode_Entity(array_keys(getEntities())));

$Parser->addMode('multiplyentity',new Doku_Parser_Mode_MultiplyEntity());
$Parser->addMode('quotes',new Doku_Parser_Mode_Quotes());
```

```
$Parser->addMode('camelcaselink',new Doku_Parser_Mode_CamelCaseLink());
$Parser->addMode('internallink',new Doku_Parser_Mode_InternalLink());
$Parser->addMode('media',new Doku_Parser_Mode_Media());
$Parser->addMode('externallink',new Doku_Parser_Mode_ExternalLink());
$Parser->addMode('email',new Doku_Parser_Mode_Email());
$Parser->addMode('windowssharelink',new
Doku_Parser_Mode_WindowsShareLink());
$Parser->addMode('filelink',new Doku_Parser_Mode_FileLink());
$Parser->addMode('eol',new Doku_Parser_Mode_Eol());

// Загрузить исходный документ вики
$doc = file_get_contents(DOKU_DATA . 'wiki/syntax.txt');

// Получить список инструкций
$instructions = $Parser->parse($doc);

// Создать преобразователь
require_once DOKU_INC . 'parser/xhtml.php';
$Renderer = & new Doku_Renderer_XHTML();

# Здесь загрузите в преобразователь данные (например, типа смайлов)

// Проходимся по всем инструкциям
foreach ( $instructions as $instruction ) {

    // Выполняем обратный вызов через преобразователь
    call_user_func_array(array(&$Renderer,
$instruction[0]),$instruction[1]);
}

// Отображаем выходные данные
echo $Renderer->doc;
```

Выбор текста (для фрагментов)

Следующий код показывает, как выбрать фрагмент исходного текста, используя инструкции, полученные из парсера;

```
// Создаём парсер
$Parser = & new Doku_Parser();

// Добавляем обработчик
$Parser->Handler = & new Doku_Handler();

// Загружаем режим  header  для поиска заголовков
$Parser->addMode('header',new Doku_Parser_Mode_Header());

// Загружаем режимы, которые могут содержать разметку,
// которая может быть принята за заголовок
```

```
$Parser->addMode('listblock',new Doku_Parser_Mode_ListBlock());
$Parser->addMode('preformatted',new Doku_Parser_Mode_Preformatted());
$Parser->addMode('table',new Doku_Parser_Mode_Table());
$Parser->addMode('unformatted',new Doku_Parser_Mode_Unformatted());
$Parser->addMode('php',new Doku_Parser_Mode_PHP());
$Parser->addMode('html',new Doku_Parser_Mode_HTML());
$Parser->addMode('code',new Doku_Parser_Mode_Code());
$Parser->addMode('file',new Doku_Parser_Mode_File());
$Parser->addMode('quote',new Doku_Parser_Mode_Quote());
$Parser->addMode('footnote',new Doku_Parser_Mode_Footnote());
$Parser->addMode('internallink',new Doku_Parser_Mode_InternalLink());
$Parser->addMode('media',new Doku_Parser_Mode_Media());
$Parser->addMode('externallink',new Doku_Parser_Mode_ExternalLink());
$Parser->addMode('email',new Doku_Parser_Mode_Email());
$Parser->addMode('windowssharelink',new
Doku_Parser_Mode_WindowsShareLink());
$Parser->addMode('filelink',new Doku_Parser_Mode_FileLink());

// Загружаем исходный документ вики
$doc = file_get_contents(DOKU_DATA . 'wiki/syntax.txt');

// Получаем перечень инструкций
$instructions = $Parser->parse($doc);

// Используем эти переменные, чтобы узнать,
// находимся ли мы внутри необходимого фрагмента
$inSection = FALSE;
$startPos = 0;
$endPos = 0;

// Проходимся по всем инструкциям
foreach ( $instructions as $instruction ) {

    if ( !$inSection ) {

        // Ищем заголовки в списках
        if ( $instruction[0] == 'header' &&
                trim($instruction[1][0]) == 'Lists' ) {

            $startPos = $instruction[2];
            $inSection = TRUE;
        }
    } else {

        // Ищем конец фрагмента
        if ( $instruction[0] == 'section_close' ) {
            $endPos = $instruction[2];
            break;
        }
    }
}
```

```
// Нормализуем и разбиваем документ
$doc = "\n".str_replace("\r\n","\n",$doc)."\n";

// Получаем текст, идущий перед фрагментом, который нам необходим
$before = substr($doc, 0, $startPos);
$section = substr($doc, $startPos, ($endPos-$startPos));
$after = substr($doc, $endPos);
```

Управление входными файлами с данными в шаблонах

«Докувики» хранит части некоторых шаблонов во внешних файлах (например, смайлы). Поскольку парсинг и вывод документа являются отдельными стадиями, обрабатываемыми различными компонентами, при использовании данных также требуется дифференцированный подход.

Каждый подходящий режим принимает простой список элементов, который он собирает в список шаблонов для регистрации в анализаторе.

Например:

```
// Простой список вхождений смайлов...
$smileys = array(
    ':-)',
    ':-(',
    ';-)',
    // и т. д.
    );

// Создать режим
$SmileyMode = & new Doku_Parser_Mode_Smiley($smileys);

// Добавить режим в парсер
$Parser->addMode($SmileyMode);
```

Для парсера не имеет значения выходной формат смайлов.

Другие режимы, где применяется подобный подход, определяются классами;

- `Doku_Parser_Mode_Acronym` — для сокращений;
- `Doku_Parser_Mode_Wordblock` — для блоков специфических слов (например, ненормативной лексики);
- `Doku_Parser_Mode_Entity` — для типографических символов.

Конструктор каждого класса принимает в качестве параметра список указанных элементов.

На практике возникает необходимость в функциях для извлечения данных из конфигурационных файлов и размещение ассоциативных массивов в статической переменной, например:

```
function getSmileys() {

    static $smileys = NULL;

    if ( !$smileys ) {

        $smileys = array();

        $lines = file( DOKU_CONF . 'smileys.conf');

        foreach($lines as $line){

            // игнорировать комментарии
            $line = preg_replace('/#.*$/','',$line);

            $line = trim($line);

            if(empty($line)) continue;

            $smiley = preg_split('/\s+/',$line,2);

            // Собрать ассоциативный массив
            $smileys[$smiley[0]] = $smiley[1];
        }
    }

    return $smileys;
}
```

Эта функция может быть использована следующим образом:

```
// Загрузить шаблоны смайлов в режим
$SmileyMode = & new Doku_Parser_Mode_Smiley(array_keys(getSmileys()));
```

```
// Загрузить ассоциативный массив в Преобразователь
$Renderer->smileys = getSmileys();
```

Замечание: проверка ссылок, которые необходимо блокировать, обрабатывается другим способом, описанным ниже.

Проверка ссылок на спам

В идеале ссылки требуется проверять на спам *до* размещения документа (после редактирования).

> Этот пример следует использовать осторожно. Он создаёт полезные точки связывания, но по результатам тестирования является очень медленным — возможно проще использовать функцию, которая «закрывает глаза» на синтаксис, но ищет во всем документе ссылки, сверяя их с «чёрным списком». Между тем, этот пример может быть полезным как основа

> для построения «карты вики» или поиска «требуемых страниц» посредством проверки внутренних ссылок.

Это можно сделать, создав специальный преобразователь, который проверяет только относящиеся к ссылкам обратные вызовы и сверяет ULR с «чёрным списком».

Требуется функция для загрузки файла `spam.conf` и связывания его с единственным регулярным выражением:

> Недавно протестировал этот подход (единственное регулярное выражение) с использованием последней версии «чёрного списка» с blacklist.chongqed.org и получил ошибки о том, что окончательное регулярное выражение слишком велико. Возможно, следует разбить регулярное выражение на маленькие кусочки и возвращать их как массив.

```
function getSpamPattern() {
    static $spamPattern = NULL;

    if ( is_null($spamPattern) ) {

        $lines = @file(DOKU_CONF . 'spam.conf');

        if ( !$lines ) {

            $spamPattern = '';

        } else {

            $spamPattern = '#';
            $sep = '';

            foreach($lines as $line){

                $line = preg_replace('/#.*$/','',$line);

                // Игнорировать пустые строки
                $line = trim($line);
                if(empty($line)) continue;

                $spamPattern.= $sep.$line;

                $sep = '|';
            }

            $spamPattern .= '#si';
        }
    }

    return $spamPattern;
}
```

Теперь нам нужно расширить основной преобразователь ещё одним, который проверяет

только ссылки:

```
require_once DOKU_INC . 'parser/renderer.php';

class Doku_Renderer_SpamCheck extends Doku_Renderer {

    // Здесь должен быть код, выполняющий инструкции
    var $currentCall;

    // Массив инструкций, которые содержат спам
    var $spamFound = array();

    //  PCRE-шаблон для нахождения спама
    var $spamPattern = '#^$#';

    function internallink($link, $title = NULL) {
        $this->__checkTitle($title);
    }

    function externallink($link, $title = NULL) {
        $this->__checkLinkForSpam($link);
        $this->__checkTitle($title);
    }

    function interwikilink($link, $title = NULL) {
        $this->__checkTitle($title);
    }

    function filelink($link, $title = NULL) {
        $this->__checkLinkForSpam($link);
        $this->__checkTitle($title);
    }

    function windowssharelink($link, $title = NULL) {
        $this->__checkLinkForSpam($link);
        $this->__checkTitle($title);
    }

    function email($address, $title = NULL) {
        $this->__checkLinkForSpam($address);
        $this->__checkTitle($title);
    }

    function internalmedialink ($src) {
        $this->__checkLinkForSpam($src);
    }

    function externalmedialink($src) {
        $this->__checkLinkForSpam($src);
    }
```

```
    function __checkTitle($title) {
        if ( is_array($title) && isset($title['src'])) {
            $this->__checkLinkForSpam($title['src']);
        }
    }

    // Поиск по шаблону осуществляется здесь
    function __checkLinkForSpam($link) {
        if( preg_match($this->spamPattern,$link) ) {
            $spam = $this->currentCall;
            $spam[3] = $link;
            $this->spamFound[] = $spam;
        }
    }
}
```

Обратите внимание на строку `$spam[3] = $link;` в методе `__checkLinkForSpam`. Она вставляет дополнительный элемент в список найденных спамовых инструкций, позволяя легко определить, какие URL были «плохими».

Наконец мы можем использовать преобразователь с проверкой на спам:

```
// Создать парсер
$Parser = & new Doku_Parser();

// Добавить обработчик
$Parser->Handler = & new Doku_Handler();

// Добавить режимы, которые могут содержать разметку,
// которая ошибочно будет принята за ссылку
$Parser->addMode('preformatted',new Doku_Parser_Mode_Preformatted());
$Parser->addMode('unformatted',new Doku_Parser_Mode_Unformatted());
$Parser->addMode('php',new Doku_Parser_Mode_PHP());
$Parser->addMode('html',new Doku_Parser_Mode_HTML());
$Parser->addMode('code',new Doku_Parser_Mode_Code());
$Parser->addMode('file',new Doku_Parser_Mode_File());
$Parser->addMode('quote',new Doku_Parser_Mode_Quote());

// Загружаем режим link...
$Parser->addMode('internallink',new Doku_Parser_Mode_InternalLink());
$Parser->addMode('media',new Doku_Parser_Mode_Media());
$Parser->addMode('externallink',new Doku_Parser_Mode_ExternalLink());
$Parser->addMode('email',new Doku_Parser_Mode_Email());
$Parser->addMode('windowssharelink',new
Doku_Parser_Mode_WindowsShareLink());
$Parser->addMode('filelink',new Doku_Parser_Mode_FileLink());

// Загружаем исходный документ вики
$doc = file_get_contents(DOKU_DATA . 'wiki/spam.txt');

// Получить список инструкций
```

```
$instructions = $Parser->parse($doc);

// Создать преобразователь
require_once DOKU_INC . 'parser/spamcheck.php';
$Renderer = & new Doku_Renderer_SpamCheck();

// Загрузить шаблон спама
$Renderer->spamPattern = getSpamPattern();

// Пройтись по всем инструкциям
foreach ( $instructions as $instruction ) {

    // Сохранить текущую инструкцию
    $Renderer->currentCall = $instruction;

    call_user_func_array(array(&$Renderer,
$instruction[0]),$instruction[1]);
}

// Что за спам был найден?
echo '<pre>';
print_r($Renderer->spamFound);
echo '</pre>';
```

Поскольку нам не нужны все режимы синтаксиса, проверка спама таким способом будет быстрее, чем обычный парсинг документа.

Добавление синтаксической конструкции

Предупреждение: приведённый ниже код ещё не испытан — это только пример.

Простая задача по модификации парсера: этот пример будет добавлять тэг-«закладку», который может быть использован для создания якоря в документе для создания ссылки на него.

Синтаксис для тэга будет таким:

```
BM{Моя закладка}
```

Строка «Моя закладка» является наименование закладки, а BM {} идентифицируется как сама закладка. В HTML эта конструкция будет соответствовать:

```
<a name="Моя закладка"></a>
```

Добавление этой синтаксической конструкции требует следующих шагов:

1. создать синтаксический режим парсера, для регистрации в лексическом анализаторе;
2. обновить код функции `Doku_Parser_Substition`, находящейся в конце файла `parser.php` и которая используется для быстрого получения списка режимов (используется в классах вроде`Doku_Parser_Mode_Table`);

3. обновить код обработчика, дополнив его методом, «ловящим» вхождения закладок;
4. обновление абстрактного класса преобразователя и какого-нибудь конкретного преобразователя.

Создание режима парсера подразумевает расширение класса Doku_Parser_Mode и перегрузкой метода connectTo:

```
class Doku_Parser_Mode_Bookmark extends Doku_Parser_Mode {

    function connectTo($mode) {
        // Разрешаются слова и пробелы
        $this->Lexer->addSpecialPattern('BM\{[\w ]+\}',$mode,'bookmark');
    }

}
```

Будет осуществляться поиск целой закладки с использованием единственного шаблона (извлечение имени закладки из остального синтаксиса будет осущевляться обработчиком). Используется метод addSpecialPattern анализатора, так что закладка присутствует в своём собственном состоянии.

Замечание: анализатор не требует ограничителей шаблона — он заботиться об этом за вас.

Поскольку ничто *внутри* закладки не должно рассматриваться как годная разметка вики, связывание с другими режимами, которые может принимать этот, отсутствует.

Следующая функция Doku_Parser_Substition в файле inc/parser/parser.php требует обновления, чтобы она возвращала в списке новый режим с наименованием bookmark;

```
function Doku_Parser_Substition() {
    $modes = array(
        'acronym','smiley','wordblock','entity','camelcaselink',
        'internallink','media','externallink','linebreak','email',
        'windowssharelink','filelink','notoc','multiplyentity',
        'quotes','bookmark',

    );
    return $modes;
}
```

Эта функция лишь помогает в регистрации режима с другими режимами, которые получают его (например, списки могут содержать этот режим — ваша ссылка может быть внутри списка).

Замечание: существует похожая функция, вроде Doku_Parser_Protected и Doku_Parser_Formatting, которые возвращают разные группы режимов. Группировка различных типов синтаксиса не является полностью совершенной, но всё равно остаётся полезной для экономии кода.

Описав синтаксис, мы должны добавить в обработчик новый метод, который сравнивает наименование режима (т. е.bookmark).

```
class Doku_Handler {

    // ...

    // $match - строка, которая сравнивается анализатором
    //     с регулярным выражением для закладок
    // $state идентифицирует тип совпадения (см. выше)
    // $pos - индекс байта первого символа совпадения в исходном документе
    function bookmark($match, $state, $pos) {

        // Технически не следует беспокоится о состоянии:
        // оно всегда будет DOKU_LEXER_SPECIAL, если
        // нет серьёзных багов
        switch ( $state ) {

            case DOKU_LEXER_SPECIAL:

                // Попытка извлечения наименования закладки
                if ( preg_match('/^BM\{(\w{1,})\}$/', $match, $nameMatch) )
{

                    $name = $nameMatch[1];

                    // arg0: наименование вызываемого метода преобразователя
                    // arg1: массив аргументов для метода преобразователя
                    // arg2: индекс байта
                    $this->__addCall('bookmark', array($name), $pos);

                // Если у закладки нет годного имени,
                // пропускаем не меняя как cdata
                } else {

                    $this->__addCall('cdata', array($match), $pos);

                }
            break;

        }

        // Должно вернуть TRUE или анализатор будет остановлен
        return TRUE;
    }

    // ...

}
```

Последний этап — обновление кода преобразователя (`renderer.php`) новой функцией и её реализация в XHTML преобразовании (`xhtml.php`):

```
class Doku_Renderer {
```

```
    // ...

    function bookmark($name) {}

    // ...

}
```

```
class Doku_Renderer_XHTML {

    // ...

    function bookmark($name) {
        $name = $this->__xmlEntities($name);

        // id is required in XHTML while name still supported in 1.0
        echo '<a class="bookmark" name="'.$name.'" id="'.$name.'"></a>';

    }

    // ...

}
```

См. скрипт `tests/parser_replacements.test.php` в качестве примера того, как можно использовать этот код.

Добавление синтаксиса форматирования (с состоянием)

Предупреждение: нижеприведённый код ещё не протестирован — это только пример.

Для того, чтобы показать расширенное использование анализатора, этот пример добавляет разметку, которая позволяет пользователям менять цвет обрамляемый текст на красный, жёлтый или зелёный.

Разметка будет выглядеть так:

```
<red>Это красный цвет</red>.
Это чёрный цвет
<yellow>Это жёлтый цвет</yellow>.
Это тоже чёрный цвет
<green>Это зелёный цвет</green>.
```

Шаги, необходимые для внедрения данной возможности, в сущности, являются такими же, как в предыдущем примере, начинаются с нового синтаксического режима, но добавляет некоторые детали, поскольку задействуются другие режимы:

```
class Doku_Parser_Mode_TextColors extends Doku_Parser_Mode {
```

```
    var $color;

    var $colors = array('red','green','blue');

    function Doku_Parser_Mode_TextColor($color) {

        // Предотвращает ошибки использования этого режима
        if ( !array_key_exists($color, $this->colors) ) {
            trigger_error('Invalid color '.$color, E_USER_WARNING);
        }

        $this->color = $color;

        // Этот режим принимает другие режимы:
        $this->allowedModes = array_merge (
            Doku_Parser_Formatting($color),
            Doku_Parser_Substition(),
            Doku_Parser_Disabled()
        );

    }

    // connectTo вызывается однократно для каждого режима, зарегистрированного
анализатором
    function connectTo($mode) {

        // Шаблон с просмотром вперёд проверяет наличие закрывающего тэга...
        $pattern = '<'.$this->color.'>(?=.*</'.$this->color.'>)';

        // arg0: шаблон сравнения при входе в режим;
        // arg1: другие режимы, может сравниваться этот шаблон;
        // arg2: наименование режима.
        $this->Lexer->addEntryPattern($pattern,$mode,$this->color);
    }

    // post connect вызывается однократно
    function postConnect() {

        // arg0: шаблон сравнения при выходе из режима;
        // arg1: наименование режима.
        $this->Lexer->addExitPattern('</'.$this->color.'>',$this->color);

    }

}
```

Некоторые особенности вышеприведённых классов:

1. В действительности представляют множество режимов, один для каждого цвета. Цвета следует выделять в отдельные режимы так, что, например, </green> не будет закрывающим тэгом для <red>.

2. Эти режимы могут содержать, например, <red>**Предупреждение**</red> для полужирного текста красного цвета. Это регистрируется в конструкторе класса назначением полученных наименований режимов свойству `allowedModes`.
3. Когда регистрируется входной шаблон, имеет смысл проверить существование выходного шаблона (с помощью просмотра вперёд). Это поможет в защите пользователей от них самих, когда они забудут добавить закрывающий тэг.
4. Входной шаблон требует регистрации для каждого режима, внутри которого тэги <color /> могут использоваться. Нам требуется толь один выходной шаблон, помещённый в метод `postConnect`, который исполняется однократно, после всех вызовов `connectTo` по всем вызванным режимам.

Когда с классом режимами обработки покончено, новые режимы требуется добавить в в функцию `Doku_Parser_Formatting`:

```
function Doku_Parser_Formatting($remove = '') {
    $modes = array(
        'strong', 'emphasis', 'underline', 'monospace',
        'subscript', 'superscript', 'deleted',
        'red','yellow','green',
        );
    $key = array_search($remove, $modes);
    if ( is_int($key) ) {
        unset($modes[$key]);
    }

    return $modes;
}
```

Замечание: эта функция обрабатывается, чтобы снять режим для предотвращения включения режима форматирования в самого себя (так, например, нежелательно: <red>Срочное<red>и важное</red>сообщение</red>).

Далее обработчик должен быть обновлён методами для каждого цвета:

```
class Doku_Handler {

    // ...

    function red($match, $state, $pos) {
        // Метод  nestingTag  в обработчике предотвращает
        // многократное повторение одного и того же кода.
        // Он создаёт открывающий и закрывающий инструкции
        // для входных и выходных шаблонов,
        // пропуская остальные как  cdata.
        $this->__nestingTag($match, $state, $pos, 'red');
        return TRUE;
    }

    function yellow($match, $state, $pos) {
        $this->__nestingTag($match, $state, $pos, 'yellow');
        return TRUE;
```

```
    }

    function green($match, $state, $pos) {
        $this->__nestingTag($match, $state, $pos, 'green');
        return TRUE;
    }

    // ...

}
```

Наконец мы может обновить преобразователи:

```
class Doku_Renderer {

    // ...

    function red_open() {}
    function red_close() {}

    function yellow_open() {}
    function yellow_close() {}

    function green_open() {}
    function green_close() {}

    // ...

}
```

```
class Doku_Renderer_XHTML {

    // ...

    function red_open() {
        echo '<span class="red">';
    }
    function red_close() {
        echo '</span>';
    }

    function yellow_open() {
        echo '<span class="yellow">';
    }
    function yellow_close() {
        echo '</span>';
    }

    function green_open() {
        echo '<span class="green">';
    }
```

```
    function green_close() {
        echo '</span>';
    }

    // ...

}
```

См. скрипт `tests/parser_formatting.test.php` в качестве примера того, как можно использовать этот код.

Добавление блочных синтаксических конструкций

Предупреждение: приведённый ниже код ещё не тестировался — это только пример.

Развивая предыдущий пример, этот будет создавать новый тэг для разметки сообщений о том, что ещё предстоит сделать. Пример использования может выглядеть так:

```
===== Синтаксис цитирования в вики =====

Этот синтаксис позволяет

<todo>
Опишите синтаксис цитирования '>'
</todo>

Другой текст
```

Этот синтаксис позволяет искать страницы вики и находить вопросы, которые предстоит решить, выделяя их в документе бросающимся в глаза стилем.

Особенностью данного синтаксиса является то, что он должен отображаться в отдельном блоке документа (например, внутри

, так что он с помощью CSS может «плавать»). Это требует модификации класса `Doku_Handler_Block`, который пробегает по всем инструкциям, после того, как обработчиком найдены все вхождения, и заботиться о добавлении тэгов <p/>.

Режим парсера для этого синтаксиса может быть таким:

```
class Doku_Parser_Mode_Todo extends Doku_Parser_Mode {

    function Doku_Parser_Mode_Todo() {

        $this->allowedModes = array_merge (
            Doku_Parser_Formatting(),
            Doku_Parser_Substition(),
            Doku_Parser_Disabled()
```

```
        );

    }

    function connectTo($mode) {

        $pattern = '<todo>(?=.*</todo>)';
        $this->Lexer->addEntryPattern($pattern,$mode,'todo');
    }

    function postConnect() {
        $this->Lexer->addExitPattern('</todo>','todo');
    }

}
```

Затем этот режим добавляется в функцию `Doku_Parser_BlockContainers` в файле `parser.php`:

```
function Doku_Parser_BlockContainers() {
    $modes = array(
        'footnote', 'listblock', 'table','quote',
        // горизонтальные разрывы нарушают принцип, но они не могут использоваться в
таблицах / списках,
        // так что вставляем их сюда
        'hr',
        'todo',
    );
    return $modes;
}
```

Обновление класса `Doku_Handler`:

```
class Doku_Handler {

    // ...

    function todo($match, $state, $pos) {
        $this->__nestingTag($match, $state, $pos, 'todo');
        return TRUE;
    }

    // ...

}
```

Класс `Doku_Handler_Block` (см. файл `inc/parser/handler.php`) также нуждается в обновлении, чтобы регистрировать открывающие и закрывающие инструкции `todo`:

```
class Doku_Handler_Block {
```

```
    // ...

    var $blockOpen = array(
            'header',
            'listu_open','listo_open','listitem_open',
'table_open','tablerow_open','tablecell_open','tableheader_open',
            'quote_open',
            'section_open', // Needed to prevent p_open between header and
section_open
            'code','file','php','html','hr','preformatted',
            'todo_open',
        );

    var $blockClose = array(
            'header',
            'listu_close','listo_close','listitem_close',
'table_close','tablerow_close','tablecell_close','tableheader_close',
            'quote_close',
            'section_close', // Needed to prevent p_close after
section_close
            'code','file','php','html','hr','preformatted',
            'todo_close',
        );
```

Регистрация `todo_open` и `todo_close` в массивах `$blockOpen` и `$blockClose` сообщает классу `Doku_Handler_Block`, что любые предыдущие абзацы должны быть закрыты *до* входа в секцию `todo`, а новый абзац должен начинаться *после* секции `todo`. Внутри `todo` дополнительные абзацы не вставляются.

После этого должен быть обновлён код преобразователя:

```
class Doku_Renderer {

    // ...

    function todo_open() {}
    function todo_close() {}

    // ...

}
```

```
class Doku_Renderer_XHTML {

    // ...

    function todo_open() {
        echo '<div class="todo">';
    }
    function todo_close() {
```

```
        echo '</div>';
    }

    // ...

}
```

Сериализация инструкций преобразователя

Список выводимых обработчиком инструкций можно сериализировать, чтобы устранить повторную обработку исходного документа при каждом запросе, если содержание документа не менялось.

Самая простая реализация может быть такой:

```
$ID = DOKU_DATA . 'wiki/syntax.txt';
$cacheID = DOKU_CACHE . $ID.'.cache';

// Если кэш-файл отсутствует или утратил актуальность
// (исходный документ модифицирован), получить «свежий» список инструкций
if ( !file_exists($cacheID) || (filemtime($ID) > filemtime($cacheID)) ) {

    require_once DOKU_INC . 'parser/parser.php';

    // Создать парсер
    $Parser = & new Doku_Parser();

    // Добавить обработчик
    $Parser->Handler = & new Doku_Handler();

    // Загрузить все режимы
    $Parser->addMode('listblock',new Doku_Parser_Mode_ListBlock());
    $Parser->addMode('preformatted',new Doku_Parser_Mode_Preformatted());
    $Parser->addMode('notoc',new Doku_Parser_Mode_NoToc());
    $Parser->addMode('header',new Doku_Parser_Mode_Header());
    $Parser->addMode('table',new Doku_Parser_Mode_Table());

    // и т. д., и т. п.

    $instructions = $Parser->parse(file_get_contents($filename));

    // Сериализировать и кэшировать
    $sInstructions = serialize($instructions);

    if ($fh = @fopen($cacheID, 'a')) {

        if (fwrite($fh, $sInstructions) === FALSE) {
            die("Cannot write to file ($cacheID)");
        }
```

```
        fclose($fh);
    }

} else {
    // Загрузить и десериализировать
    $sInstructions = file_get_contents($cacheID);
    $instructions = unserialize($sInstructions);
}

$Renderer = & new Doku_Renderer_XHTML();

foreach ( $instructions as $instruction ) {
    call_user_func_array(
        array(&$Renderer, $instruction[0]),$instruction[1]
        );
}

echo $Renderer->doc;
```

Замечание: эта реализация не является полной. Что должно просходить, если кто-либо, например, модифицирует файл `smiley.conf`, добавив новый смайл? Это изменение должно порождать изменение кэша с обработкой нового смайла. Также необходимо позаботиться о блокировке файлов (или их переименовании).

Сериализация парсера

По аналогии с приведённым выше примером, возможна сериализация самого парсера до начала обработки. Поскольку установка режимов поддерживает довольно высокую перегрузку, этот пример может немного увеличить производительность. По неточной оценке, обработка страницы syntax в медленной системе занимает около 1,5-а секунд для завершения *без* сериализации и около 1,25-х секунды в версии парсера *с поддержкой* сериализации.

Если коротко, то сериализация может быть реализована таким способом:

```
$cacheId = DOKU_CACHE . 'parser.cache';

if ( !file_exists($cacheId) ) {

    // Создаём парсер
    $Parser = & new Doku_Parser();
    $Parser->Handler = & new Doku_Handler();

    // Загружаем все режимы
    $Parser->addMode('listblock',new Doku_Parser_Mode_ListBlock());
    $Parser->addMode('preformatted',new Doku_Parser_Mode_Preformatted());

    # и т. д., и т. п.
    // ВАЖНО: вызов  connectModes()
    $Parser->connectModes();
```

```
    // Сериализация
    $sParser = serialize($Parser);

    // Запись в файл
    if ($fh = @fopen($cacheID, 'a')) {

        if (fwrite($fh, $sParser) === FALSE) {
            die("Cannot write to file ($cacheID)");
        }

        fclose($fh);
    }

} else {
    // Загружаем сериализированную версию
    $sParser = file_get_contents($cacheID);
    $Parser = unserialize($sParser);
}

$Parser->parse($doc);
```

Некоторые замечания по реализации, не упомянутые выше:

- Для некоторых файлов вместо записи требуется блокировка, в противном случае в ответ на запрос может быть получен частично кэшированный файл, если он будет считываться, пока продолжается запись.
- Что следует делать, если обновляется один из файлов `*.conf`? Необходимо очистить кэш.
- Могут быть различные версии парсера (например, с проверкой на спам), так что используйте кэш-идентификаторы (cache IDs).

Тестирование

Тесты программных единиц обеспечивают использование «Simple Test for PHP». «Simple Test» является отличным инструментом для тестирования единиц php-кода. Особенно выделяются блестящая документация (см. simpletest.sourceforge.net и www.lastcraft.com/simple_test.php) и хорошо продуманый код, обеспечивающий «прозрачное» решение многих вопросов (вроде перехвата ошибок PHP и сообщения о них в результатах тестирования).

Для парсера «Докувики» тесты проводились по всем внедряемым синтаксическим конструкциям, и я *очень сильно* рекомендую написание новых тестов, если добавляется новый синтаксис.

Чтобы запустить тесты, вам следует модифицировать файл `tests/testconfig.php`, указав корректные директории «Simple Test» и «Докувики».

Некоторые заметки и рекомендации:

1. Повторно запускайте тесты каждый раз, когда вы меняете что-нибудь в парсере — проблемы немедленно выплывают на поверхность, экономя кучу времени.

2. Это только тесты для специфических ситуаций. Они не гарантируют отсутствие ошибок, если в этих специфических ситуациях работают корректно.
3. Если найдена ошибка, в процессе её устранения напишите тесты (даже лучше, *до* её устранения), чтобы предотвратить её повторное возникновение.

Ошибки и проблемы

Некоторые вопросы остаются за рамками подробного рассмотрения.

Важность порядка добавления режимов

Требуется выполение не столько «правил», сколько порядка, в котором добавляются режимы (парсер этого не проверяет). В особенности, режим `eol` должен быть загружен последним, т. к. он «съедает» «обёрточные» символы, что может нарушить корректную работу других режимов, вроде `list` или `table`.

В общем случае рекомендуется загружать режимы в порядке, описанном выше в первом примере.

> По моим наработкам, порядок важен, только если два и более режима имеют шаблоны, с которыми могут сравниваться одинаковые совокупности символов - в этом случае «выиграет» режим, имеющий низший порядковый номер. Синтаксический плагин может извлечь из этого выгоду, заменяя оригинальный обработчик, в качестве примера см. плагин «Code» — *ChrisS 2005-07-30*

Замены «блокиратора слов»

В оригинале функционирование «блокиратора слов» `wordblock` заключалось в сравнении URL ссылок с «чёрным списком». Сейчас этот режим используется для нахождения грубых слов. Для блокирования спамовых URL лучше использовать приведённый выше пример.

Рекомендация — файл `conf/wordblock.conf` следует переименовать в `conf/spam.conf`, содержащий «чёрный список» URL. Новый файл`conf/badwords.conf` будет содержать список цензурируемых грубых слов.

Слабые моменты

С точки зрения архитектуры, наихудшие части кода находятся в файле `inc/parser/handler.php`, преимущественно в «re-writing»-классах;

- `Doku_Handler_List` (inline re-writer)
- `Doku_Handler_Preformatted` (inline re-writer)
- `Doku_Handler_Quote` (inline re-writer)
- `Doku_Handler_Table` (inline re-writer)
- `Doku_Handler_Section` (post processing re-writer)
- `Doku_Handler_Block` (post processing re-writer)

- `Doku_Handler_Toc` (post processing re-writer)

«Inline re-writers» используются, пока обработчик получает вхождения от анализатора, в то время как «post processing re-writers», вызываются из `Doku_Handler::__finalize()` и выполняются однократно в отношении полного списка инструкций, созданных обработчиком.

Возможно лучше устранить `Doku_Handler_List`, `Doku_Handler_Quote` и `Doku_Handler_Table`, использовав взамен многострочные лексические режимы.

Также *возможно лучше* изменить `Doku_Handler_Section` и `Doku_Handler_Toc` в «inline re-writers«, срабатывающие на вхождения заголовков, принимаемых Обработчиком.

Самое «больное место» — это класс `Doku_Handler_Block`, отвечающий за вставку абзацев в инструкции. Имеет значение добавить в него больше абстракций для облегчения разработки, но в общем-то я не вижу каких-либо путей полного его устранения.

«Жадные» тэги

Рассмотрим следующий синтаксис вики:

```
Привет, <sup>Мир!

----

<sup>Пока,</sup> Мир…
```

Пользователь забыл закрыть первый тэг <sup>.

В результате получится:

Привет, <sup>Мир! —- ^{Пока,} Мир…

Первый тэг <sup> оказался слишком «жадным» в проверке своего входного шаблона.

Это применимо ко всем подобным режимам. Входные шаблоны проверяют наличие закрывающего тэга, но также они должны проверять, чтобы раньше не встретился второй открывающий тэг.

Сноски через список

В сущности, если сноска закрывается через несколько элементов списка, это вызывает эффект открывающей инструкции сноски без соответствующей закрывающей. Вот пример синтаксиса, вызывающего проблему:

```
*((A))
  *(( B
* C ))
```

Это будет происходить до тех пор, пока пользователи не поправят страницу. Решение —

разбить захват элементов списка в многострочные режимы (сейчас для списков есть только единственный режим `listblock`).

Захват «обёрточных» символов

Баг № 261.

Поскольку синтаксис заголовка, горизонтальной линии, списка, таблицы, цитаты и неформатируемого (выделяемого) текста полагается на «обёрточные» символы для разметки своих начала и окончания, им требуются регулярные выражения, которые поглощают «обёрточные» символы. Это означает, что пользователь должен добавлять «обёрточные» символы, если таблица находится сразу после списка, например:

```
До списка
  - Элемент списка
  - Элемент списка
| Ячейка  A  |  Ячейка  B  |
| Ячейка  C  |  Ячейка  D  |
После таблицы
```

Выдаёт:

До списка

1. Элемент списка
2. Элемент списка

| Ячейка A | Ячейка B |

Ячейка C	Ячейка D

После таблицы

Заметьте, что **первая строка** таблицы воспринимается как обычный текст.

Чтобы скорректировать это, синтаксис вики должен иметь дополнительную «обёртку» между списком и таблицей:

```
До списка
  - Элемент списка
  - Элемент списка

| Ячейка  A  |  Ячейка  B  |
| Ячейка  C  |  Ячейка  D  |
После таблицы
```

Что будет выглядеть следующим образом:

До списка

1. Элемент списка

2. Элемент списка

Ячейка A	Ячейка B
Ячейка C	Ячейка D

После таблицы

Без сканирования текста множества раз (некая разновидность «предварительных» операций, которые вставляют «обёртку»), едва ли можно найти простое решение.

Проблемы списков, таблиц и цитат

Для синтаксиса списков, таблиц и цитат есть вероятность, что использование внутри их другого синтаксиса «съест» несколько строк. Например, таблица вроде:

```
| Cell A | <sup>Cell B |
| Cell C | Cell D</sup> |
| Cell E | Cell F |
```

выдаёт:

Cell A	Cell B | | Cell C | Cell D
Cell E	Cell F

В идеале должно быть преобразовано так:

Cell A	<sup>Cell B
Cell C	Cell D</sup>
Cell E	Cell F

Т. е. открывающий тэг `<sup>` должен игнорироваться, если в текущей ячейке отсутствует закрывающий тэг.

Для устранения этого требуется поддержка многострочного режима внутри таблиц, списков и цитат.

Сноски и блоки

Внутри сносок блоки игнорируются, вместо этого используется эквивалент инструкции `<br/>`. Это связано с неудобным в разработке классом `Doku_Handler_Block`. Если внутри сноски используются таблица, список, цитата или горизонтальная линия, это *сработает* как абзац.

Устраняется модификацией класса `Doku_Handler_Block`, однако рекомендуется предварительно тщательно ознакомиться с его устройством.

Заголовки

Текущие заголовки могут находиться на той же строке, что и предшествующий текст. Это

вытекает из эффекта, рассмотренного выше в вопросе «Захват строк», и требует некоторой предварительной обработки для устранения. Например:

```
До заголовка есть
Некоторый текст == Заголовок ==
После заголовка
```

Если бы поведение было бы таким же, как в оригинальном парсере «Докувики», преобразование было бы таким:

До заголовка есть Некоторый текст == Заголовок == После заголовка

Но в результате будет:

До заголовка есть Некоторый текст

Заголовок

После заголовка

Конфликт блоков и списков

Существует проблема: если до списка находится пустая строка с двумя пробелами, всё это вместе будет интерпретироваться как блок:

```
* list item
* list item 2
```

Что ещё необходимо сделать

Вот некоторые вопросы, которые ещё предстоит решить…

Больше состояний для закрывающих инструкций

Для преобразования в иные форматы, нежели XHTML, может оказаться полезным добавление отождествления уровня для закрывающих инструкций списка, и т. д.

> Почему бы просто не «преобразовать» в XML и затем применить к нему некоторые парсеры XSLT/XML?

Подрежимы для таблиц, списков, цитат

Анализатор с множественными режимами для предотвращения случаев вложенности состояний друг в друга.

Обсуждение

Спасибо за перевод!

1)

Lexer относится к классу `Doku_Lexer` и содержится в файле `inc/parser/lexer.php`.

2)

Сканирование — чтение строки PHP от начала до конца.

3)

Термин «токен» в этом документе относится к совпадению регулярного выраждения, полученного лексическим анализатором, и соответствующему вызову метода обработчиком.

4)

Handler относится к классу `Doku_Handler` и содержится в файле `inc/parser/handler.php`.

5)

Последовательность инструкций содержится в массиве `$calls`, который является атрибутом обработчика. Предназначен для использования с call_user_func_array.

6)

Parser относится к классу `Doku_Parser` и содержится в файле `inc/parser/parser.php`.

7)

Renderer (от «to render» *в значении* превращать, преобразовывать) относится к абстрактному (implemented) классу `Doku_Renderer` - см. `inc/parser/renderer.php` и `inc/parser/xhtml.php`.

8)

Прим. переводчика: возможно, речь идёт о т.н. **незахватывающем поиске**, подробнее см. литературу по регулярным выражениям, например, Джеффри Фридла «Регулярные выражения: библиотека программиста. Второе издание.» — СПб.: Питер, 2003, или документацию по PHP — ru.php.net/manual/en/ref.pcre.php.

9)

Термины «состояние» и «режим» используются отчасти как взаимозаменяемые, когда здесь говориться об анализаторе

10)

Смысл «плохо форматируемый» не применим к парсеру «Докувики» — он разработан так, чтобы предотвращать случаи, когда пользователь забывает добавить закрывающий тэг некоторой разметки, полностью игнорируя эту разметку.