Design and Implementation of the Sweble Wikitext Parser: Unlocking the Structured Data of Wikipedia

Hannes Dohrn
Friedrich-Alexander-University
Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
+49 9131 85 27621
hannes.dohrn@cs.fau.de

Dirk Riehle
Friedrich-Alexander-University
Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
+49 9131 85 27621
dirk@riehle.org

ABSTRACT

The heart of each wiki, including Wikipedia, is its content. Most machine processing starts and ends with this content. At present, such processing is limited, because most wiki engines today cannot provide a complete and precise representation of the wiki's content. They can only generate HTML. The main reason is the lack of well-defined parsers that can handle the complexity of modern wiki markup. This applies to MediaWiki, the software running Wikipedia, and most other wiki engines.

This paper shows why it has been so difficult to develop comprehensive parsers for wiki markup. It presents the design and implementation of a parser for Wikitext, the wiki markup language of MediaWiki. We use parsing expression grammars where most parsers used no grammars or grammars poorly suited to the task. Using this parser it is possible to directly and precisely query the structured data within wikis, including Wikipedia.

The parser is available as open source from `http://sweble.org`.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*; D.3.4 [**Programming Languages**]: Processors—*Parsing*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*Parsing*; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Design, Languages

Keywords

Wiki, wikipedia, wiki parser, parsing expression grammar, PEG, abstract syntax tree, AST, WYSIWYG, Sweble.

1. INTRODUCTION

The content of wikis is being described using specialized languages, commonly called wiki markup languages. Despite their innocent name, these languages can be fairly complex and include complex visual layout mechanisms as well as full-fledged programming language features like variables, loops, function calls, and recursion.

Most wiki markup languages grew organically and without any master plan. As a consequence, there is no well-defined grammar, no precise syntax and semantics specification. The language is defined by its parser implementation, and this parser implementation is closely tied to the rest of the wiki engine. As we have argued before, such lack of language precision and component decoupling hinders evolution of wiki software [12]. If the wiki content had a well-specified representation that is fully machine processable, we would not only be able to access this content better but also improve and extend the ways in which it can be processed.

Such improved machine processing is particularly desirable for MediaWiki, the wiki engine powering Wikipedia. Wikipedia contains a wealth of data in the form of tables and specialized function calls ("templates") like infoboxes. State of the art of extracting and analyzing Wikipedia data is to use regular expressions and imprecise parsing to retrieve the data. Precise parsing of the Wikitext, the wiki markup language in which Wikipedia content is written, would solve this problem. Not only would it allow for precise querying, it would also allow us to define wiki content using a well-defined object model on which further tools could operate. Such a model can simplify programmatic manipulation of content greatly. And this in turn benefits WYSIWYG (what you see is what you get) editors and other tools, which require a precise machine-processable data structure.

This paper presents such a parser, the Sweble Wikitext parser and its main data structure, the resulting abstract syntax tree (AST). The contributions of this paper are the following:

- An analysis of why it has been so difficult in the past to develop proper wiki markup parsers;
- The description of the design and implementation of a complete parser for MediaWiki's Wikitext;
- Lessons learned in the design and implementation of the parser so that other wiki engines can benefit.

While there are many community projects with the goal to implement a parser that is able to convert wiki markup into an intermediate abstract syntax tree, none have succeeded so far. We believe that this is due to their choice of grammar, namely the well-known LALR(1) and LL(k) grammars. Instead we base our parser on a parsing expression grammar.

The remainder of the paper is organized as follows: Section 2 discusses related work and prior attempts to base a parser on a well-defined grammar. In section 3 the inner workings of the original MediaWiki parser are explained. Afterwards, we summarize the challenges we identified to implementing a parser based on a well-defined grammar. Section 4 discusses our implementation of the Sweble Wikitext parser. We first set out the requirements of such a parser, then discuss the overall design and the abstract syntax tree it generates. Finally, we give specific details about the implementation. Section 5 discusses the limitations of our approach and section 6 presents our conclusion.

2. PRIOR AND RELATED WORK

2.1 Related and prior work

The need for a common syntax and semantics definition for wikis was recognized by the WikiCreole community effort to define such a syntax [2, 3]. These efforts were born out of the recognition that wiki content was not interchangeable. Völkel and Oren present a first attempt at defining a wiki interchange format, aimed at forcing wiki content into a structured data model [26]. Their declared goal is to add semantics (in the sense of the Semantic Web) to wiki content and thereby open it up for interchange and processing by Semantic Web processing tools.

The Semantic Web and W3C technologies play a crucial role in properly capturing and representing wiki content beyond the wiki markup found in today's generation of wiki engines. Schaffert argues that wikis need to be "semantified" by capturing their content in W3C compatible technologies, most notably RDF triples so that their content can be handled and processed properly [16]. The ultimate goal is to have the wiki content represent an ontology of the underlying domain of interest. Völkel et al. apply the idea of semantic wikis to Wikipedia, suggesting that with appropriate support, Wikipedia could be providing an ontology of everything [25]. The DBpedia project is a community effort to extract structured information from Wikipedia to build such an ontology [4].

All these attempts rely on or suggest a clean syntax and semantics definition of the underlying wiki markup without ever providing such a definition. The only exception that we are aware of is our own work in which we provide a grammar-based syntax specification of the WikiCreole wiki markup [12]. In this work, we not only provide a syntax specification [11] but also an external representation format using XML as well as matching processing tools and specifications [13].

2.2 Prior parser attempts

There have been numerous attempts at implementing a new parser specifically for MediaWiki, and some of these have taken a clean approach with a grammar-based syntax specification [21]. As we argue later in this article, the chosen meta-languages fail to cope with the complexity of Wikitext. Our own choice of a PEG-based syntax specification overcomes these problems.

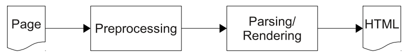


Figure 1: MediaWiki processing pipeline.

Most of the alternative parsers we are aware of use either LALR(1) or LL(k) grammars to define Wikitext. Others are hand-written recursive-descent parsers. Two prominent examples are the parsers used by DBpedia [4] and the parser used by AboutUs [1]. Both parsers are actively in use and under development by their projects. While the DBpedia parser is a hand-written parser that produces an AST, the AboutUs parser is a PEG parser that directly renders HTML. To our knowledge, both parsers are not feature complete. Especially the AboutUs parser is promising in its ability to cope with Wikitext syntax, but does not produce an intermediate representation like an AST.

3. WIKITEXT AND MEDIAWIKI

The MediaWiki software grew over many years out of Clifford Adams' UseMod wiki [20], which was the first wiki software to be used by Wikipedia [24]. Though the first version of the software later called MediaWiki was a complete rewrite in PHP, it supported basically the same syntax as the UseMod wiki.

By now the MediaWiki parser has become a complex software. Unfortunately, it converts Wikitext directly into HTML, so no higher-level representation is ever generated by the parser. Moreover, the generated HTML can be invalid. The complexity of the software also prohibits the further development of the parser and maintenance has become difficult.

Many have tried to implement a parser for Wikitext to obtain a machine-accessible representation of the information inside an article. However, we don't know about a project that succeeded so far. The main reason we see for this is the complexity of the Wikitext grammar, which does not fall in any of the well-understood categories LALR(1) or LL(k).

In the following two sections, we will first give an outline of how the original MediaWiki parser works. Then we present a list of challenges we identified for writing a proper parser.

3.1 How the MediaWiki parser works

The MediaWiki parser processes an article in two stages. The first stage is called *preprocessing* and is mainly involved with the transclusion of templates and the expansion of the Wikitext. In the second stage the actual *parsing* of the now fully expanded Wikitext takes place [23]. The pipeline is shown in figure 1.

Transclusion is a word coined by Ted Nelson in his book "Literary Machines" [14] and in the context of Wikitext refers to the textual inclusion of another page, often called a *template*, into the page which is currently being rendered. The whole process works similar to macro expansion in a programming language like C. The term *expansion* describes the process of transclusion and the evaluation or substitution of so-called *parser functions*, *tag extensions* and *variables*. These three syntactic elements of Wikitext all basically trigger the evaluation of a function which is defined outside the content of the wiki. Usually these functions are either built into the original MediaWiki software or provided by extensions to the MediaWiki software. The result of such an evaluation is again Wikitext which is then embedded into the page that is currently being rendered. An example for a parser function and its evaluation is the substitution of the

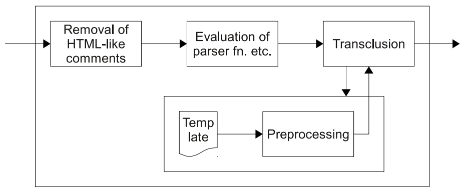


Figure 2: Preprocessing in the MediaWiki parser.

Wikitext "`{{lc:string}}`" with the text "*string*", where all characters in *string* are replaced by their lower-case variant. The process of preprocessing is illustrated in figure 2.

The parsing is realized as a sequence of distinct processing steps, each of which takes the current representation and applies a transformation to it (e.g. turns four dashes at the beginning of a line into an `<hr/>` tag). This way, the input is gradually morphed from the Wikitext representation into an HTML representation. Since no higher-level representation is built during this process, the Wikitext is directly transformed into HTML, the process is also described as "rendering" in figure 1. Table 1 gives a brief overview of some important elements of Wikitext.

Internal link (pointing to a page inside the same wiki)	`[[target page|Link text]]`
External link	`[http://example.com Link text]`
A horizontal line (<hr />)	`----`

Table with four cells on two rows	`{| class="wikitable"`
	` | Cell 1.1 || Cell 1.2`
	` |-`
	` | Cell 2.1 || Cell 2.2`
	` |}`

An itemization list	`* Item 1`
	`* Item 2`

Preformatted text (<pre>, mind the space at the beginning of each line)	`␣This text is rendered using`
	`␣a fixed font and spaces are`
	`␣preserved.`

Table 1: Some elements of MediaWiki's Wikitext [22].

The individual transformations are usually line-based. For example, although a table is a block-level element that can span many lines of Wikitext, its individual syntactic atoms are only recognized at the beginning of a line. Therefore, many processing steps start by breaking up the source by lines. The second step involves recognition of syntactic atoms (like table start "`{|`" or internal link start "`[[`"). Some syntactic elements (e.g. external links) are recognized as a whole using regular expressions while others (e.g. tables) are recognized step by step using algorithms which scan through the Wikitext or through individual lines of the Wikitext. Finally, once an element of Wikitext has been recognized, it is converted into its HTML counterpart. An incomplete list of transformations performed by the original MediaWiki parser is given in figure 3.

3.2 Challenges to parsing Wikitext

In our analysis of the original MediaWiki parser and the development of our own implementation of a parser for MediaWiki's Wikitext, we identified the following major challenges to parsing Wikitext with conventional parsers:

1. The original MediaWiki parser will never reject any input, which also translates to "every input is syntactically correct".

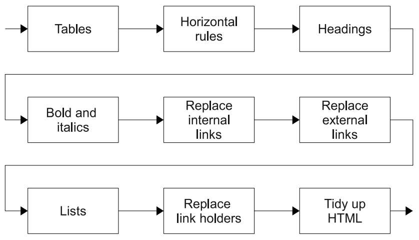


Figure 3: Coarse draft of the processing steps in the MediaWiki parsing stage.

2. Syntactic elements are recognized one after another in separate transformation steps, often not honoring the scope of other syntactic elements. This leads to wrong nesting in the generated HTML and allows the recognition of elements which, by a strict parser, would not have been recognized (see example 1).

Input Wikitext:

```
{|
 | before [[target|link
 | title]] after
 |}
```

Resulting HTML:

```
<table><tr>
  <td>
    before <a ... href="target"> link
  </td>
  <td>
    title </a> after
  </td>
</tr></table>
```

Example 1: A Wikitext link that is improperly spread across two table cells will be transformed to invalid HTML.

3. The choice of some syntactic elements leads to severe ambiguities (e.g. bold and italics apostrophes). In order to remedy the situation, the original parser uses heuristics to guess what the user might have meant.

Input Wikitext:

```
l'''Encyclopédie''
but three apostrophes also means '''bold'''
```

Resulting HTML:

```
l'<i>Encyclopédie</i>
but three apostrophes also means <b>bold</b>
```

Example 2: An algorithm discovers all apostrophes on a single line and, using a heuristic, decides which are real apostrophes and which are formatting markup.

4. There are no unique prefixes in Wikitext which could be used to unambiguously identify certain syntactic elements. In Wikitext, an opening bracket is the prefix of a bracketed external link but can as well appear anywhere else. Therefore, whether a prefix is really a prefix or just text can only be decided if the whole element has been recognized.

5. As seen in example 1, the MediaWiki software generates wrongly nested HTML. Furthermore, the original parser does not enforce HTML-like semantics, which determine what constitutes valid nesting of different elements. Consequently, the HTML produced by the parser can be ambigious and it eventually depends on an HTML tidier or the browser's rendering engine how wrongly nested HTML elements will be rendered.

6. The individual processing steps often lead to unexpected and inconsistent behavior of the parser. For example, lists are recognized inside table cells. However, if the table itself appears inside a framed image, lists are not recognized. The behavior of MediaWiki requires productions of a grammar to take their context into account and check, whether the table itself appears inside a link.

7. Other unpredictable behavior often surfaces due to the complexity of the original parser.

4. THE SWEBLE WIKITEXT PARSER

In the following section we first lay out the requirements that we imposed on our implementation of a parser for MediaWiki's Wikitext. Following is a discussion about the design decisions we made to overcome the problems discussed in the previous section. We then give an overview of the AST data structure. Finally, we present specifics of our implementation.

4.1 Requirements for the parser

We identified the following requirements for our parser:

- Is grammar-driven and therefore provides a formal specification of Wikitext.
- Transforms Wikitext into a well-defined machine-readable structure, i.e. an abstract syntax tree (AST).
- Doesn't fail on any input, no matter how "syntactically incorrect" it is. In a best effort approach in recognizing syntactically wrong input, the parser continues to parse the rest of the input.
- The parsed representation can be converted back into Wikitext. This requirement makes it possible for a generated AST to be converted back into Wikitext which matches the original input exactly.
- ASTs are limited in their ability to represent the structure of the input as they are tree-like (e.g. when it comes to wrong nesting). Therefore, the conversion must preserve the meaning inherent to the input as best as possible.
- The parser should not resort to emitting syntax errors instead of trying to interpret the input exactly like the original parser does.

The Wikitext language is defined by the MediaWiki parser itself. As a consequence, every string that is accepted by MediaWiki's parser is syntactically valid Wikitext. And since MediaWiki's parser accepts every possible input, there is technically no incorrect Wikitext syntax.

Yet for individual elements of Wikitext, one still can say whether they are syntactically correct, given that the Wikitext is expected to describe for example a bracketed external link. Consider

`[http://example.com]`
vs.
`[ http://example.com]`.

Both strings are valid Wikitext, however, the second string will not be interpreted as a bracketed external link by the MediaWiki parser due to the whitespace in front of the URL. Therefore, the first string is an example for valid bracketed external link syntax, while the second string is not.

Accordingly, when one writes a parser for Wikitext, one has to meet two requirements: First, the parser must never reject any input and second, the parser has to recognize the same elements which the original MediaWiki parser recognizes. While the first requirement is relatively simple to achieve with a PEG, the second requirement is in some cases hard to maintain.

4.2 Parser design

Looking at the complexities of Wikitext syntax, as we did in section 3.2, it is clear that realizing a parser using a LALR(1) or LL(k) grammar is difficult. While these grammars are only a subset of context-free grammars, Wikitext requires global parser state and can therefore be considered a context-sensitive language. Writing a parser by hand is also possible, however, since there is no definitive grammar for Wikitext, writing a parser is an exploratory task. It takes many iterations and therefore fast prototyping to approximate the behavior of the original MediaWiki parser. Writing a parser by hand hinders fast prototyping.

Using the original MediaWiki parser as starting point is also an option, however, that would violate the first requirement for a grammar-driven parser. Also the MediaWiki parser is a very complex software by now and hard to evolve and maintain.

In light of these problems, we decided to use a so-called *parsing expression grammar* (PEG) as means to describe the grammar of Wikitext.

Parsing expression grammars

Strong points of PEGs are the availability of syntactic predicates and that they are scannerless. Further, PEGs unlike LR and LL parsing algorithms are closed under composition and therefore enable modularity. While GLR parsing algorithms recognize all context free grammars, they return a forest of parse trees for ambiguous grammars which is inefficient if one knows that only one tree is correct [8]. One key difference between LALR(1) or LL(k) and PEG grammars is that PEGs use prioritized choices ($e1/e2$) instead of alternatives ($e1|e2$). Prioritized choices are tested one after another (in the given order) and the first successful match is employed. This renders PEGs free of ambiguities.

Prioritized choices have their advantages as well as their disadvantages. As different authors [7, 15, 17] argue, the grammar will be free of ambiguities, but will it describe the language the author intended? This question stems from the problem of "prefix capture" [15]. Given the parsing expression `("["/"[[")[a-z]`, the input `[[target` will not be matched. The first choice of `("["/"[[")` matches, and the parser will go on and try to recognize `[a-z]`. This fails on the second "`[`" and the whole expression fails without reconsidering the second choice. While this is easy to fix in the given example by exchanging the choices, it is difficult to spot when several productions separate the captured prefix and the point where a production fails.

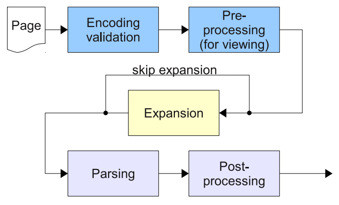


Figure 4: The processing pipeline in the Sweble Wikitext parser.

Another downside of PEG parsers is their performance and memory consumption. PEG parsers trade memory for performance but are still slower than the classic LALR(1) parser [8, 15].

An in-depth explanation how we designed the PEG grammar for the Sweble Wikitext parser is given in section 4.4.

Pipeline architecture

Our parser is composed of five stages: encoding validation, preprocessing, expansion, parsing and postprocessing. The pipeline is outlined in figure 4. The individual stages will be explained in the upcoming sections.

A fact not illustrated in figure 4 are the two different representations of a page used in the pipeline. One representation is the Wikitext as found in a raw page itself and after different processing stages. The other representation of a page is the AST. Where the different representations are used is shown in figure 5.

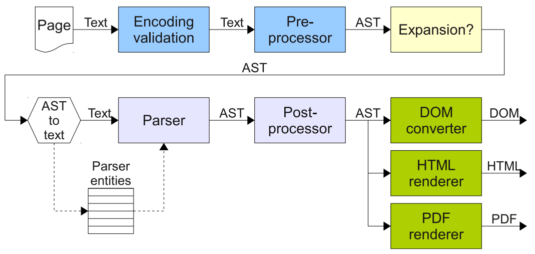


Figure 5: The different types of data flowing through the processing pipeline.

A problem that becomes visible in this illustration is the conversion into an AST before the expansion stage and the following conversion back into Wikitext before parsing starts. While the preprocessor already recognized some syntactic elements of Wikitext (e.g. comments, transclusions, etc.), the parser would have to repeat the recognition of these elements if they were just converted back into Wikitext after expansion took place.

To avoid this duplication of work and the increased complexity of the parser it entails, already recognized elements of Wikitext will not be converted back into their Wikitext equivalent but will be converted into *parser entities*. A parser entity is a reference into a table external to the Wikitext. The external table stores the AST representation of the recognized element, while the Wikitext only contains a reference into the table. When the parser hits a parser entity in its input, the parser entity is looked up in the table and the stored AST is inserted into the AST which the parser is generating.

Stage 1: Encoding validation

Not all possible characters are allowed in Wikitext. Also, to avoid the need for escaping the character used as prefix for parser entities should be unique. On the other hand, we don't want to loose information by stripping illegal characters from the input. Especially characters from the private use planes of the Unicode standard [19] might also be used by other applications to embed information into Wikitext. To guarantee that this information is not lost, the encoding validation step wraps illegal characters into parser entities. This way, characters that might be harmful to the processing pipeline, especially to the parsing stage, are safely locked away and can be restored after postprocessing.

Characters considered illegal or harmful in encoding validation are:

- non-characters and incomplete surrogates [19],
- control characters $[U{+}0000 - U{+}0020) \setminus \{U{+}0009, U{+}000A, U{+}000D\}$,
- character U+007F and
- characters from the private use planes [19].

Stage 2: Preprocessing

The stage of preprocessing prepares the Wikitext for expansion and involves the tasks of recognizing XML-like comments, redirect links, transclusion statements, tag extensions and conditional tags.

Redirect links are found at the beginning of a page and cause the wiki to redirect to another page when viewed. Tag extensions are XML elements that are treated similar to parser functions or parser variables. XML elements whose names are not know to the parser as tag extensions are treated as text. Finally, the conditional tags `<noinclude>`, `<includeonly>` and `<onlyinclude>` decide if a part of the Wikitext will be excluded from further processing depending on whether the page is preprocessed for direct viewing or if the page is preprocessed for transclusion into another page.

The preprocessing stage emits an AST which consists mostly of text nodes. The only other elements are either redirect links, transclusion nodes or the nodes of tag extensions.

Stage 3: Expansion

If one builds a wiki engine which not only parses but also renders pages from Wikitext, one has to insert one more stage to the pipeline called expansion. While the pipeline without the expansion stage is enough to recognize a single page, pages in a MediaWiki are often built using templates, magic words, parser functions and tag extensions [23]. In order to resolve these entities into Wikitext, expansion has to be performed after the preprocessing stage and before the Wikitext is parsed, as illustraged in figure 6. However, expansion is an optional stage. To use the AST in a WYSIWYG editor, one would leave out expansion to see the unexpanded transclusion statements and parser function calls in the original page.

Templates are pages that are transcluded into the Wikitext of the page that is being viewed. For example the Wikitext "`{{Featured article}}`" in a page will be replaced by the contents of the page "Template:Featured article" ("Template:" being an implicit namespace for transclusions). Pages used as templates can themselves transclude other pages which makes expansion a recursive process.

Once the expansion stage has finished, the entities recognized in the preprocessing stage have been (recursively)

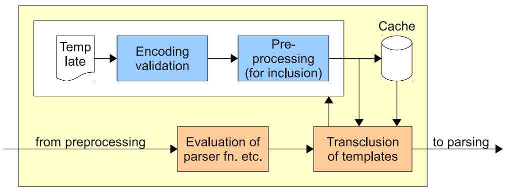

Template
Encoding validation
Pre-processing (for inclusion)
Cache
from preprocessing
Evaluation of parser fn. etc.
Transclusion of templates
to parsing

Figure 6: Illustration of the expansion process as implemented in our parser.

replaced by Wikitext. However, if the template for a transclusion could not be found or the Wikitext contains references to non-existing parser functions, the nodes for these elements will remain in the expanded AST and will not be replaced by Wikitext.

Stage 4: Parsing

Parsing is the stage in which formatting markup is recognized. It is also the stage in which most of the challenges to implementing a parser for Wikitext lie. However, before parsing starts, the AST that has been generated in the preprocessing stage and which was extended in the expansion stage must be converted back into Wikitext.

The conversion back into Wikitext is straight forward for text nodes. All other nodes are converted into parser entities. Once the AST is converted back into Wikitext, a PEG parser analyzes the text and generates an AST capturing the syntax and semantics of the wiki page. The details of the PEG parser are exposed in section 4.4.

Stage 5: Postprocessing

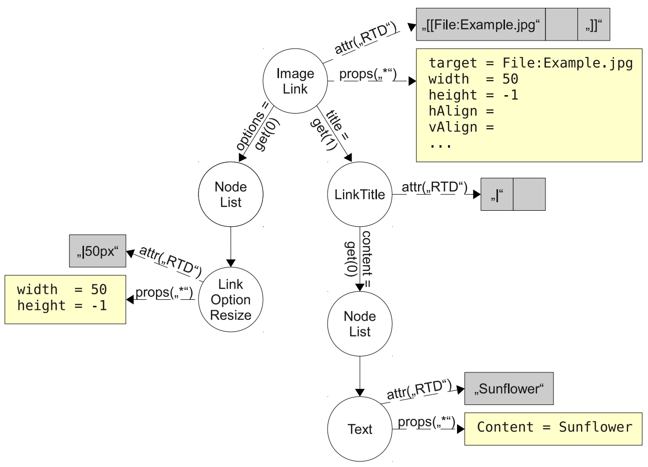
Although we believe that a PEG parser is well suited for most of the challenges found in Wikitext grammar, a single PEG parser does not solve all hard problems. As a consequence, we added one last stage after parsing: postprocessing. As explained in section 4.4, the parsing stage does not match XML opening and closing tags and it does not properly analyze apostrophes for bold and italics. However, the parsing stage does recognize the individual tags and the apostrophes. The task of postprocessing then is to match the tags to form whole elements and to analyze the apostrophes to decide, which of them are real apostrophes and which have to be interpreted as bold or italics markup. The assembly of paragraphs is also handled in postprocessing.

4.3 AST Design

An abstract syntax tree (or AST) is an n-ary tree that encodes the syntactic structure of a text (e.g. a Java program or Wikitext). Individual syntactic elements are represented by nodes and their children.

The AST we use in our parser implementation has three dimensions:

- Each node can have either a fixed number of children or a variable number of children. For example, the *Section* node always has two children: The section heading and the section body. The children are nodes of type *NodeList*. A node of type NodeList has a variable number of children and can contain, for example, the elements that make up the heading or the section body.
- Each node can have a fixed set of properties attached. For example, *StringContentNodes* like *Text* always provide the string property "`content`". A node of type Section always has an integer property called "`level`" that indicates the level of the heading.
- Finally, nodes can have arbitrary attributes attached to them. Attributes can be added and stripped any time and are used to associate temporary information with a node that is needed by the processing step currently working on the AST.



```
target = File:Example.jpg
width  = 50
height = -1
hAlign =
vAlign =
...
```

```
width  = 50
height = -1
```

```
Content = Sunflower
```

Figure 7: The AST representation of the image link "`[[File:Example.jpg|50px|Sunflower]]`".

To enable easy access to the structured data contained in the AST, we implemented XPath [27] queries on top of our AST implementation using the Apache Commons JXPath library [18]. Using a simple query like `//Section/title` one can extract all section titles from a page. Other possibilities include the extraction of, for example, the name of the capital of a country from the respective article. This can be done by retrieving the "capital" argument from the transclusion statement of a so-called "Country Infobox" template.

Storing round-trip information

Usually, when parsing a document, one stores only the semantically relevant portions of the document. These are either coded implicitly in the tree data structure or explicitly in the content and attributes of individual tree nodes. For example the brackets that indicate an internal link ("`[[` ··· `]]`") are nowhere to be found in the AST. The same goes for whitespace or XML comments if they are irrelevant to the rendering of the document.

However, all this information is vital when the resulting AST is to be used as underlying data structure for a visual editor while direct manipulation of the Wikitext shall remain an option. Or if the pages shall be stored as AST instead of storing the original Wikitext. In this case, whitespace and comments must not be discarded. Otherwise, once the Wikitext is restored from the AST , the complete layout of the original document would be destroyed to a point where it will pose a threat to user acceptance.

In order to preserve formatting information in the AST, a *RtData* (RountTrip Data) object is attached to each AST node as attribute (see figure 7). RtData objects are arrays with exactly $n + 1$ fields, where n is the number of children the respective node has. For example, the *ImageLink* node in figure 7 has two children (options and title) and its RtData (attach as attribute "RTD") has three fields. Each field of the RtData array is itself an array and can contain strings, characters or XML comments. In order to restore the original Wikitext of a node, one visits the RtData array and the children of that node in an interleaved manner. First the data from the first RtData field is rendered, then the first child of the node is rendered, then the second field of RtData and so on. This process repeats recursively for the children of each node. When finished, the Wikitext has been restored and matches the original Wikitext.

4.4 Parser implementation

The entire software including the parser is written in Java. The parser itself is generated from a PEG grammar using the Rats! parser generator [9]. The encoding validation is done using a lexer written in JFlex [10]. Postprocessing is performed using so-called Visitors. The different renderers are also realized as Visitors.

Visitors are instances of the Visitor software design pattern [6] and help to separate the AST data structure from the algorithms that operate on the data. The visitation of an AST is a depth-first traversal of the tree.

In the following sections, we use the grammar specification language used by R. Grimm in [8]. However, we omit implementation specific details like the data type of a production. Each production has the form:

```
Nonterminal = e ;
```

Where e is an expression composed from the operators in table 2.

Operator	Description
`' '`	Literal character
`" "`	Literal string
`[ ]`	Character class
`_`	Any character
`{ }`	Semantic action
(e)	Grouping
$e?$	Option
$e*$	Zero-or-more
$e+$	One-or-more
$\&e$	And-predicate
$!e$	Not-predicate
$id : e$	Binding
$e_1 \cdots e_n$	Sequence
$e_1 / \cdots / e_n$	Ordered choice

Table 2: The operators supported by Rats! [8].

Overview of the PEG grammar

Our PEG grammar consists of modules for individual syntactic elements like internal links or tables and one module, the content module, which represents the glue between the individual syntactic elements.

The following kinds of whitespace productions are defined:

```
Eof = !_ ;                              // EndOfFile
Eol = "\r\n" / "\r" / "\n" ;            // EndOfLine
Space = ' ' / '\t' / '\f' ;
Ws = Space / Eol ;                      // Whitespace
Tp = XmlComment ;                       // Transparent
TpSpace = Tp / Space ;
TpWs = Tp / Ws ;
```

A special kind of whitespace is found in the various *Tp* productions. Here not only whitespace in the literal sense is captured but also XML comments. Comments were not discarded by the preprocessor to allow a perfect reconstruction of the Wikitext from the AST. However, this makes comments nontransparent to the parser. If a comment appears inside a syntactic atom like the prefix of an internal link “`[<!- ··· ->[`”, the prefix will not be recognized by a parsing expression that expects only the literal string “`[[`”. However, making all rules aware of comments would clutter up the grammar (and also the AST). Therefore, the user of our parser is left with two choices: Strip comments after preprocessing and disrupt round-trip support or accept, that comments will interrupt syntactic atoms.

Separating prose from wiki markup

Most of Wikitext content is prose interleaved with markup tokens. To efficiently recognize the prose (or text) the *Text* production consumes all text until it hits a character that might constitute the prefix to a markup token:

```
Text =  !TextStopperPrefix _  ;
TextStopperPrefix =
  Eol / "=" / "|" / ":" / "[[" / "~~~" / ... ;
```

Consider the following two lines of Wikitext:

```
prefix[[target]]
http://example.com
```

Links can have prefixes whose form is determined by a regular expression that depends on the content language of a wiki. A similar situation is given for the embedded URL. In every wiki instance different sets of protocols can be allowed. The problem is that the prefix and the protocol are already consumed by the Text production for performance reasons. Stopping at every character to test the regular expression for link prefixes is expensive. As remedy we added special productions to the *InlineContent* production:

```
InlineContent =
  t:Text ':' p:UrlPath &{isUrl(t, p)}
      { /* makeUrl(t, p), crop(t) */ }
/ t:Text l:InternalLink
      { /* addPrefix(t, l), crop(t) */ }
/ Text
/ ':'
/ !InlineContentStopper InlineContentAtom
/ !InlineContentStopper TextStopperPrefix ;
```

The choices printed in bold font assure that prefixes are recognized and attributed to the respective link and that URLs are recognized.

Best effort parsing

As stated earlier the parser must never fail on any input. To assure this the syntactic elements subsumed under the *InlineContentAtom* production all have extra fall-back choices:

```
InlineContentAtom = Signature / InternalLink /
  ExternalLink / Ticks / MagicWord / ... ;

InternalLink =
    "[[" LinkTarget LinkOption* LinkTitle? "]]"
  / '[' ; // fall-back
```

By adding more elaborate fall-back choices which try to parse content atoms with minor syntactical errors one can inform the user about possible errors he might want to correct or even auto-correct these errors immediately. Otherwise, the smallest possible prefix has to be consumed. In the given example we must not consume “`[[`” in the fall-back choice because a second “`[`” could be the prefix to a bracketed external link.

Emulating MediaWiki's scopes

In Wikitext not every element can appear and be recognized in every scope. We describe three problems of scoping which we solved in our grammar:

- Consider the following Wikitext:

  ```
  [[target|... [http://example.com] ...]]

  [[target|... [[target]] ...]]

  [http://example.com ... [[target]] ...]
  ```

  On each line a link appears inside the scope of another link's title. However, every nesting has a different effect when rendered. When an external link is embedded in the title of an internal link, the external link (in bold font) is not recognized. When embedding an internal link in the title of another (otherwise syntactically correct) internal link, the enclosing internal link is not recognized (in bold font). Finally, if an internal link is embedded in the title of an external link, both are recognized.

  One solution would be to recognize all elements at first and later analyze the AST to discover wrong nesting. However, this leads to another problem: If, for example, the enclosing link is reinterpreted as plain text *after* discovering an internal link inside its title, other parts of the affected Wikitext may need reinterpretation as well:

  ```
  [[target|
  * item 1
  * item 2
  [[other link]] ]]
  ```

  Lists (in bold) are not recognized inside a link title. However, in this case the enclosing link will not be recognized once the other link (in red) is discovered inside its scope. Consequently, the list *has* to be recognized as list after all and must not be treated as plain text.

- Newline characters play a vital role in Wikitext. Consider the following examples:

  ```
  * Item 1 [[target|link
  title]] still item 1
  * Item 2 <span> ...
  </span> not item 2 any more
  ```

  Lists usually terminate at the end of a line. However, internal links can hide newline characters inside their scope. As a result, the content of the first item continues on the next line. Other syntactic elements of Wikitext cannot hide newline characters. The second list item stops inside the `<span>` element. The following line is not part of the second list item any more.

  Now consider the same situation but with a table as surrounding element:

  ```
  {| class="wikitable"
   | Cell 1.1 || Cell 1.2
   |-
   | Cell 2.1 [[target|link
     title]] || still cell 2.1
   |}
  ```

  Again the link contains a newline. However, unlike lists, inline table cells can "see" into the scope of links and can therefore discover otherwise hidden newline characters. Inline table cells are cells which are terminated with "`||`" and are followed by more cells on the same line. Accordingly, the inline cell separator "`||`" in the second row (in bold font) is not recognized as such and "Cell 2.1" is treated as non-inline cell.

- Finally consider the following situation:

  ```
  [[target|
  {|
   |
  * item 1
  * item 2
   |}
  ```

  Here an itemization list (in bold font) is found inside a table cell and the table is part of a link title. Usually, lists are allowed inside table cells. However, if the table is part of a link's title, in which lists are not recognized, then the list is also not recognized in the table's cells. It follows, that the scope of an element has to take the context into account in which the element appears.

We solve these problems using a global context stack. Each context frame stores the *active scope* and a set of *sticking scopes*. Consider the following simplified set of productions for a link:

```
InlineContentStopper = ICStopperInternalLink / ... ;

ICStopperInternalLink =
  &{inScope(ILINK_TITLE)}("]]"/InternalLink) ;

InternalLink = "[[" &{accept(ILINK)} ... ;

stateful
LinkTitle =
  { enter(ILINK_TITLE); } '|' InlineContent* ;
```

The scope of the *LinkTitle* production is entered by pushing a new context onto the global stack (done by the `stateful` modifier) and calling to the method `enter`. If a certain content atom is allowed in the currently active scope is checked by the semantic predicate `&{accept(...)}`. The `accept` method tests the given atom against the active scope and the sticking scopes. Testing against the sticking scopes makes a production aware of all surrounding scopes, not only the immediate active scope.

Finally, the semantic predicate `&{inScope(...)}` tests whether the *InlineContent* production should fail for a certain input because a token was discovered that terminates the active scope. This also guarantees that an enclosing link fails if another link is found inside its title. The inner scope will terminate and return to the link production. However, the link production does only expect the terminating "`]]`", not another internal link and thus fails.

To ensure linear-time performance PEG parsers use memoization. However, the above set-up violates the functional nature of memoization as it is realized by the Rats! parser generator. To avoid unexpected behavior of the parser and not disable memoization altogether we added extra productions to make memoization aware of global state changes.

Analyzing apostrophes

The analysis of apostrophes is done by the postprocessor. The parser has already identified groups of apostrophes but did not match them to form bold or italics formatting nodes. We implemented a visitor that emulates the algorithm that is used in the original parser.

First the visitor is performing a search for all apostrophes in one line of Wikitext source. Performing this search on an AST means to traverse the tree until one finds a text node that contains a newline. However, newlines can also appear in the discarded whitespace of an internal link and in block-level elements like lists or tables. Since block-level elements only start at the beginning of a line, they always indicate a newline. Internal links on the other hand have an extra property "`hasNewline`", which is set to true if the parser encountered any newlines in whitespace.

Once all apostrophes on one line have been found and classified real apostrophes are converted to plain text while markup apostrophes are converted to special XML tags and their respective closing tags: `<@i>` and `<@b>`. These are then matched by the following postprocessing step, in which the scopes of XML elements are established.

Establishing the scope of XML elements

A pathological example, where it is hard to maintain the original representation as found in the Wikitext, is wrong nesting. Even if a parser recognizes wrong nesting, it could not directly represent this in the resulting tree structure. The MediaWiki parser recognizes elements even if they are wrongly nested. And since the MediaWiki parser only transforms Wikitext input into HTML, it can even represent wrong nesting in the target language, as illustrated in example 1.

It will be an HTML tidying step or the browser who will eventually correct the nesting and decide how to render the generated HTML. A corrected version of example 1's HTML will usually look like:

```
<table><tr>
  <td>
    before <a ... href="target">link</a>
  </td>
  <td>
    <a ... href="target">title </a> after
  </td>
</tr></table>
```

As one can see, the link has been closed in the first cell and been re-opened in the second cell.

This problem generally applies to all elements of Wikitext that can be nested. However, we propose to not tolerate wrong nesting in some cases, while in other cases we explicitly try to recognize the author's intention and convert wrong nesting into a valid tree structure. We further propose the following distinction between these two cases: For inline formatting elements with scope (bold and italics) as well as for allowed XML elements, we try to correct erroneous nesting. All other elements of Wikitext have strict nesting.

The consequences of wrong nesting of elements for which strict nesting rules apply were discussed in section *Emulating MediaWiki's scopes*. In those cases our parser also issues warnings to indicate, that strict nesting rules were violated and that the meaning of the Wikitext might not have been preserved exactly.

To fix wrong nesting, we implemented a two-staged process. First the parser recognizes opening and closing tags individually. Then, as second step in postprocessing, an algorithm will analyze where an opening tag should be closed and discards superfluous closing tags. The postprocessor will again issue warnings to indicate that auto-correction has taken place and that the user might want to review the corrections.

We also check for legal usage of empty-only XML elements. If in postprocessing we encounter an XML opening element, that can only appear as empty element (e.g. `<br/>`) but was written as non-empty opening tag (e.g. `<br>`), we automatically convert it into an empty XML element.

Finally, when correcting erroneous nesting of inline formatting elements, we propagate their effect into scopes. This is done for bold and italics but also for HTML formatting elements like `<u>`. Consider the following Wikitext:

```
...''italic [[target|still italic''...]]...
```

The opening apostrophes for italic formatting are opened in front of a link but closed inside the link's title. Since italics is considered a propagatable inline formatting element, the respective AST representation will be altered to correspond to the following Wikitext:

```
...''italic ''[[target|''still italic''...]]...
```

The italics formatting has been closed in front of the link, before entering the scope of the link title, and has been reopened after entering the scope of the link title, thus propagating the effect into the scope of the link title and at the same time restoring correct nesting. This way we account for the nesting correction done either by an HTML tidying step or the browser.

5. LIMITATIONS

Although we believe that we have solved all hard problems of parsing Wikitext, there are limitations to our approach. So far we have no means of automatically comparing the results our parser produces with the results of the original MediaWiki parser. Generation of HTML is a not only a question of parsing but also requires a complete (Media-)wiki engine that provides the parser functions, parser variables, tag extensions, templates, etc. In short, the whole context in which a page is rendered. Since we have not implemented such an engine yet, one would at least need a smart comparison algorithm that disregards differences in the output that do not stem from misinterpretation of the original Wikitext. As a consequence, we manually compare our results using a set of Wikipedia pages that span a variety of features (complicated tables, complex pages, random pages, most visited pages).

The complexity of the original parser makes it difficult to predict how a certain input will be interpreted. This of course also makes it difficult to judge whether a given alternative parser is complete, which also applies to our parser. Though our parser interprets most of the pages we have tested correctly, there are still pages, whose AST interpretation will not match with what one expects given the HTML rendered by the MediaWiki parser for the same page. Therefore, development of our parser is not finished. However, we don't expect major changes to be necessary to correct parsing errors that will be discovered in the future.

As explained earlier there are also cases where we deliberately chose not to reproduce the exact behavior of the original parser. When dealing with wrong nesting, we repair or discard wrongly nested elements. Though we tried to stay faithful to the behavior of the original parser together with an HTML tidying step or the browser's rendering engine, we do get different results in some cases. By issuing warnings whenever possible we inform the user that the original meaning of the Wikitext might not have been preserved exactly.

6. CONCLUSIONS

Accurate parsing into an intermediate format like an AST is a prerequisite to reliably access the data stored in wikis. In this paper we have presented an implementation scheme applicable to parsers for wiki markup languages similar to that of MediaWiki.

Still the development of such a parser remains an exploratory task for a parser as complex as the MediaWiki parser. As a consequence, such a replacement parser will not interpret the input exactly like the original parser. However, we have shown, that with the techniques presented in this paper it is possible to closely approximate the behavior of the original parser and still obtain a machine-accessible higher-level representation of a page's content. Where we cannot adhere to the behavior of the original parser we issue warnings. Yet the cases in which we diverge are rare corner cases.

Writing a parser like the one presented in this work enables a wiki software to re-engineer its wiki markup syntax. Though implementing a proper parser is no easy task, once a clean intermediate representation of a wiki is available, the content of the wiki can be translated into arbitrary other formats, including a re-engineered wiki markup, which can then be parsed by a simpler parser.

Future work will focus on the translation of the AST into a similar data structure that focuses on the semantic content and is detached from the syntactic idiosyncrasies of MediaWiki's markup. Analogous to HTML's Document Object Model, we call this concept *Wikitext Object Model* [5] (WOM).

The parser, its grammar and the visitors are available as open source from `http://sweble.org`.

7. ACKNOWLEDGEMENTS

We would like to thank Carsten Kolassa, Michel Salim and Ronald Veldema for their help and support.

8. REFERENCES

[1] AboutUs.org. *kiwi - Yet Another Peg WikiText Parser*, (accessed March 27, 2011). https://github.com/aboutus/kiwi/.

[2] C. Sauer, C. Smith (editors). *Wiki Creole*, 2007 (accessed March 23, 2011). http://wikicreole.org.

[3] C. Sauer, C. Smith (editors). *Wiki Creole 1.0*, 2007 (accessed March 23, 2011). http://wikicreole.org/wiki/Creole1.0.

[4] dbpedia.org. *DBpedia*, (accessed March 27, 2011). http://dbpedia.org/.

[5] H. Dohrn and D. Riehle. Wom: An object model for wikitext. Technical Report CS-2011-05, University of Erlangen, Dept. of Computer Science, July 2011.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1994.

[7] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.

[8] R. Grimm. Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 38–51, New York, NY, USA, 2006. ACM.

[9] R. Grimm. *Rats! — An Easily Extensible Parser Generator*, (accessed March 1, 2011). http://cs.nyu.edu/rgrimm/xtc/rats.html.

[10] JFlex.de. *JFlex – The Fast Scanner Generator for Java*, (accessed March 1, 2011). http://jflex.de/.

[11] M. Junghans, D. Riehle, R. Gurram, M. Kaiser, M. Lopes, and U. Yalcinalp. An ebnf grammar for wiki creole 1.0. *SIGWEB Newsl.*, 2007, December 2007.

[12] M. Junghans, D. Riehle, R. Gurram, M. Kaiser, M. Lopes, and U. Yalcinalp. A grammar for standardized wiki markup. In *Proceedings of the 4th International Symposium on Wikis*, WikiSym '08, pages 21:1–21:8, New York, NY, USA, 2008. ACM.

[13] M. Junghans, D. Riehle, and U. Yalcinalp. An xml interchange format for wiki creole 1.0. *SIGWEB Newsl.*, 2007, December 2007.

[14] T. Nelson. *Literary machines.* Mindful Press, Sausalito, 1981.

[15] R. R. Redziejowski. Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundam. Inf.*, 79:513–524, August 2007.

[16] S. Schaffert. IkeWiki: A Semantic Wiki for Collaborative Knowledge Management. In *WETICE '06: Proceedings of the 15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 388–396, Washington, DC, USA, 2006. IEEE Computer Society.

[17] S. Schmitz. Modular syntax demands verification. Technical report, LABORATOIRE I3S, UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS, 2006.

[18] The Apache Software Foundation. *Commons JXPath*, (accessed March 27, 2011). http://commons.apache.org/jxpath/.

[19] The Unicode Consortium. *The Unicode Standard, Version 5.0.0.* Addison-Wesley, Boston, 2007.

[20] Usemod.com. *UseModWiki*, (accessed Febuary 28, 2011). http://www.usemod.com/cgi-bin/wiki.pl.

[21] Various authors. *Alternative parsers for Mediawiki*, (accessed March 27, 2011). http://www.mediawiki.org/wiki/Alternative_parsers.

[22] Various authors. *Help:Formatting*, (accessed March 27, 2011). http://www.mediawiki.org/wiki/Help:Formatting.

[23] Various authors. *Markup spec*, (accessed March 27, 2011). http://www.mediawiki.org/wiki/Markup_spec.

[24] Various authors. *MediaWiki history*, (accessed March 27, 2011). http://www.mediawiki.org/wiki/MediaWiki_history.

[25] M. Völkel, M. Krötzsch, D. Vrandecic, H. Haller, and R. Studer. Semantic wikipedia. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 585–594, New York, NY, USA, 2006. ACM.

[26] M. Völkel and E. Oren. Towards a wiki interchange format (wif). In M. Völkel and S. Schaffert, editors, *Proceedings of the First Workshop on Semantic Wikis – From Wiki To Semantics*, 2006.

[27] W3C. *XML Path Language*, (accessed March 27, 2011). http://www.w3.org/TR/xpath/.