The Construction of Syntax Trees Using External Data for Partially Formalized Text Documents

Kirill Chuvilin
Institute of Computing for Physics and Technology, Protvino, Russia
Moscow Institute of Physics and Technology (State University), Moscow, Russia
kirill@chuvilin.pro

***Abstract*—This article investigates the possibility of logical structure (abstract syntax tree) automatic construction for text documents, the format of which is not fully defined by standards or other rules common to all the documents. In contrast to the syntax described by formal grammars, in such cases there is no way to build the parser automatically. Text files in LaTeX format are the typical examples of such formatted documents with not completely formalized syntax markup. They are used as the resources for the implementation of the algorithms developed in this work. The relevance of LaTeX document analysis is due to the fact that many scientific publishings and conferences use LaTeX typesetting system, and this gives rise to important applied task of automation for categorization, correction, comparison, statistics collection, rendering for WEB, etc. The parsing of documents in LaTeX format requires additional information about styles: symbols, commands and environments. A method to describe them in JSON format is proposed in this work. It allows to specify not only the information necessary to pars, but also meta information that facilitates further data mining. And it is really necessary, for example, for correct comparison of documents, which arises in the solution of the automatic correction problem. This approach is used for the first time. The developed algorithms for constructing a syntax tree of a document in LaTeX format, that use such information as an external parameter are described. The results are successfully applied in the tasks of comparison, auto-correction and categorization of scientific papers. The implementation of the developed algorithms is available as a set of libraries released under the LGPLv3. The key features of the proposed approach are: flexibility (within the framework of the problem) and simplicity of parameter descriptions. The proposed approach allows to solve the problem of parsing documents in LaTeX format. But it is required to form the base of style element descriptions for widespread practical use of the developed algorithms.**

I. Introduction

A. Methods for the description and analysis of formatted text documents

Text files are broadly applicable in the modern information technologies: data storage and transmission (XML, JSON), data viewing (HTML, CSS, Markdown, BBCode, Textile), data processing (C/C++, Python, JavaScript, C# and many other programming languages). The contents of such files is structured with a special *markup*.

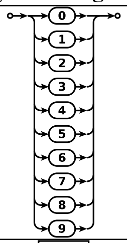
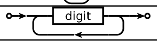
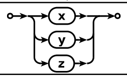
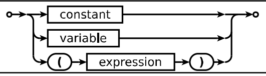
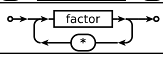
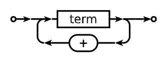
Files of each type are described by the own way of markup. It is necessary to know the rules of such markup to understand what information is contained in files. Usually these rules are called the *format* or the (computer) *language*. Examples are the programming languages, markup languages, specification languages, etc. The format is responsible for the way of data pieces are arranged and shaped and for the information of text document source elements.

But the syntax (or *grammar*) of language must be somehow described too. It often allows relatively large amount of possible structures that nevertheless consist of repeating units. Because of this specificity it is possible to describe structure of some blocks via other using a valid recursion. This approach is the basis for formal systems of syntax determination such as Backus — Naur Form (BNF) [1], extended Backus — Naur Form (EBNF) [2], the syntax diagrams [3]. The BNF and the EBNF describe grammar using text structures, the syntax diagrams are a visual interpretation of the EBNF.

The Table I is an example of grammar, with which you can build arithmetic expressions. There is a set of terminal (final) symbols: numbers, variables, operations, braces. The other structures are defined through them and through each other: digit, constant, variable, factor, term, expression.

TABLE I. Example of grammar: simple arithmetic expressions

Terminal symbols: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, x, y, z, (, ), *, +`

Symbol	EBNF	Syntax diagram
`digit`	`"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"`	
`constant`	`digit , {digit}`	
`variable`	`"x" | "y" | "z"`	
`factor`	`constant | variable | "(", expression, ")"`	
`term`	`factor | term, "*" , factor`	
`expression`	`term | expression, "+" , term`	

Formal grammars are described for the most popular markup and programming languages. Examples of such descriptions:

- C++ (ISO/IEC 14882:1998(E)): http://www.externsoft.ch/download/cpp-iso.html,

- C# 1.0/2.0/3.0/4.0: http://www.externsoft.ch/download/csharp.html,
- ECMAScript (JavaScript): antlr3.org/grammar/1153976512034/ecmascriptA3.g,
- JSON: http://rfc7159.net/rfc7159,
- XML: https://www.w3.org/TR/REC-xml/#sec-notation,
- HTML 5: https://gist.github.com/tkqubo/2842772.

The presence of a formal grammar for a language allows you to not only get a standardized file format, but also automate the process of analyzing them.

In this work a text document is called *structured text document* if an abstract syntax tree can be build for it. The document analysis involves the construction of such a tree. The algorithm, which builds a syntax tree for a structured text document, is called *parser*.

It turns out that a formal grammar allows to automatically build a parser [4]. There are two approaches to use formal grammars in construction of syntax trees: the top-down parsing and the bottom-up parsing. While the top-down parsing process the formal grammar rules are applied beginning with the start symbol until the desired consistency is obtained. This approach is implemented by recursively descending parser and LL parser. While the bottom-up parsing process the expressions are restored to the starting symbol. Corresponding algorithms are LR parser and GLR parser.

Thus, the problem of constructing a syntax tree for the files described by the language with a formal grammar can be considered as solved.

B. *Problems with documents in the LaTeX format*

In general, the problem can be formulated as follows: there is no a formal grammar for the documents in the LaTeX format [5]. It means both the absence of strict standardization and the inability to build a parser automatically by known methods. The source of this problem are the following four facts.

Firstly, the signature of LaTeX commands and symbols isn't defined in general terms. The number of parameters and the methods for separating parameters for different commands may differ notably. In most of the commands the parameters are bounded by curly brackets. But in some cases there are optional parameters that could be in square brackets. Furthermore, it is possible to define a command in which parameters are separated, for example, by a dot or any other sign.

Secondly, the signature and the set of commands and symbols are both defined by the style files. LaTeX style files may contain formatting and design rules, overdetermined symbols, commands and environments. For example, there is a very common command `\author{author name}` to specify the author of the document. But some style files override it so that there is an optional parameter: `\author{author name}[name for headers]`.

Third, the signature and the set of available commands and symbols both are determined by context. For example, there is a common dashes symbol: `---`. It works regardless of the used language, but is displayed with the spacings not accepted in Russian typography. For correct spacings in the publication it's better to use the symbol `"---`, but it is not available if Russian language isn't activated. The sets of commands and symbols are also very different inside and outside of equations.

Fourthly, TeX does not imply that any syntax tree exists. LaTeX is the most popular package of macro extensions for TeX and TeX is a computer typesetting system developed by Donald Knuth [6], [7]. TeX provides tools for structuring and decoration of texts, but only LaTeX appends commands and environments that together can form an abstract syntax tree. LaTeX-documents accept "pure" TeX fragments, but such precedents are mostly exceptions and must be moved to the style file if the markup is qualitative. In this study such fragments of source code can be interpreted as a separate terminal nodes of the syntax tree.

Thus, the problem of constructing a syntax tree for the documents in the LaTeX format is solvable, but it requires a separate research and algorithms.

C. *The relevance of the LaTeX documents analysis*

Many scientific publishing houses and conferences use LaTeX to prepare publications. As a consequence there are a number of practical problems associated with the processing of documents in such a format both at the stage of documents preparation and for the intelligent processing of existing collections: the automation of correction, statistical analysis, data mining, format conversion.

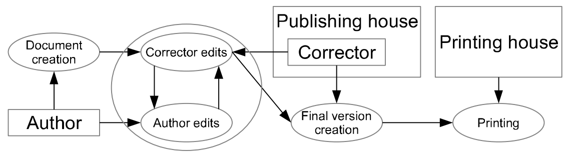


Fig. 1. Business process of the scientific article redaction

The very first task the author worked with that requires the parsing of LaTeX documents is the task of typographical error correction automation. Such errors are related to indentations, fonts, formulas, etc. At the current level of technology the corrections are made by editors manually. It spawns problems associated with the quality and processing time. The process of scientific articles correction is shown in the Fig. 1. It turned out that this problem can be solved by methods of machine learning, where the training set is formed by the syntax tree pairs of documents before and after the correction by professional editors [8].

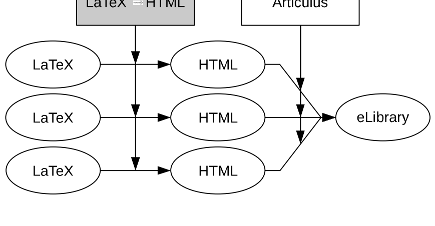


Fig. 2. Business process of the pushing articles to RISC

Published articles tend to land into one of citation systems. Russian Science Citation Index was created for Russian-language articles and is controlled by eLIBRARY. It is a national science citation bibliographic database that accumulates more than nine million publications of Russian authors as well as information about the citation of these publications from more than $6,000$ Russian journals [9]. The process of uploading items to the Russian Science Citation Index database is shown in the Fig. 2. At the moment one of the stages is the manually edition in "Articulus" system which accepts documents only in the HTML format. In this case, it is possible to convert LaTeX to HTML code qualitatively only by analyzing the syntax tree to allocate the logical blocks. Moreover, such an analysis would provide an opportunity to kick the manual processing of articles.

In fact, the format conversion problem arises not only in the context of Russian Science Citation Index. HTML is also handy for displaying articles on WEB-resources. It happens that publishers require the document in DOC or DOCX format, and materials, due to their scientific nature, are prepared only in LaTeX. XML is useful for the storage and processing of structured information. In each case it is required to allocate the correct logical blocks of documents, which can not be achieved without a qualitative analysis of the syntax structure, or, in other words, building a syntax tree.

The allocation of logical blocks and convenient their representation is also useful in text mining tasks, such as gathering of statistics on the authors and publishers, cataloging, sample building for the topic model.

Thus, the problem of constructing a syntax tree for the documents in the LaTeX format is relevant for the practical use.

D. Known implementations and their limitations

The most common way of LaTeX documents processing is the using of compilers: `latex`, `pdflatex`, `tex4ht`, etc. This approach means the sequential analysis of the source code according to the rules of the compiler. But this loses information about the overall structure of a document, since the TeX compiler does not imply that the syntax tree exists.

Some research related to the parsing of LaTeX documents was already done and there are several implementations: on Python — plasTeX [10], on Perl — LaTeX::Parser [11], on Java — SnuggleTeX [12]. LaTeX::Parser and SnuggleTeX deal only with the defined sets of macros so they can process only special documents. This is not acceptable for the problems stated in I-C since different publishers have different styles and significantly different macro sets. Due to the peculiar properties of the format analysis there is a need to external resources. plasTeX does it. But, at the same time, it totally ignores the logical sense of the syntax tree elements. For example, it is impossible to determine whether a mathematical symbol is an operator or a letter, and whether a command in the text is a marking tag or a state change. These properties are not only important for the correct analysis of the LaTeX document syntax structure, but also for the subsequent intelligent mining in the syntax tree. The most simple example: if you don't know what characters are letters or digits you can not group them in words or numbers respectively.

So, we have the following. The construction of syntax trees for LaTeX documents is necessary to solve a number of problems. This is the most fully illustrated in the research [8]. Existing projects can be divided into two types: the projects that implement compiling or conversion without building a syntax tree (which don't solve the discussed problem at all), and the projects that build the syntax tree without the possibility of data mining. Thus, there was a need for a new study, and this work focuses on satisfaction of this need.

II. LaTeX DOCUMENT STRUCTURE

This section describes how the documents in the format LaTeX are perceived in this paper. It is well established heuristics allowing to formalize the syntax tree notion for such documents.

It must be said separately that comments are allowed in the LaTeX source code. They are string parts starting with not escaped `%`. It is assumed that the comments are removed during the preprocessing to simplify the description of algorithms.

All the source code of a file consists of blocks, each block may be a symbol, a command or an environment.

LaTeX symbol is a random set of consecutive characters. A symbol could be a terminal or contain parameters. Typical representatives of terminal symbols are numbers and letters. Dash symbol `''---` is also a terminal. An example of a symbol with a parameter is equation in `$$#1$$` notation. Here `#1` is the parameter meaning the equation body in this case.

The space symbol should be noted separately. It is equivalent to any number of contiguous space or tab characters and, perhaps, a newline character. If the set of consecutive whitespace characters have more than one newline characters, this set is equivalent to a set of two line breaks and is perceived as a symbol of the paragraph separator. It is well-known specificity of LaTeX publishing system.

LaTeX command is a character sequence of the form `\command_name pattern`. Required `command_name` part must be a sequence of letters, which can end with an asterisk. Optional `pattern` part has the same format as LaTeX symbol and can also contain parameters. An example of a command that is used to highlight text in bold: `\textbf#1`.

LaTeX environment is a sequence of characters of the form:

```
\begin{environment_name}begin_command_pattern
environment_body
\end{environment_name}end_command_pattern.
```

environment_name has the same format as `command_name`. It is generally accepted for the documents in the LaTeX format that all the input content is placed inside the `document` environment.

Regardless of its nature (symbol, command or environment), each block has a purpose (logical sense), which is called the *lexeme type* in this work. The allocated lexeme types are presented in the Table II.

Taking the logical sense of elements in account is important not only in the subsequent intellectual analysis of files, but also is used for parsing. It allows to define the parameter context

TABLE II. LaTeX LEXEME TYPES

Lexeme type	Comment
BINARY_OPERATOR	binary mathematical operator
BRACKETS	logical brackets
CELL_SEPARATOR	tabular cell separator
CHAR	single char
DIGIT	digit
DIRECTIVE	LaTeX directive
DISPLAY_EQUATION	separate equation
FILE_PATH	file system path
FLOATING_BOX	floating unit
HORIZONTAL_SKIP	horizontal interval
INLINE_EQUATION	inline equation
LABEL	label ID
LENGTH	linear dimension
LETTER	letter
LINE_BREAK	line break
LIST_ITEM	list item
LIST	list of items
NUMBER	sequence of digits
PARAGRAPH_SEPARATOR	paragraph spacer
PICTURE	picture
POST_OPERATOR	mathematical postoperator
PRE_OPERATOR	mathematical preoperator
RAW	not processing part of the source
SPACE	space or analog
SUBSCRIPT	subscript
SUPERSCRIPT	superscript
TABLE	tabular
TABULAR_PARAMETERS	tabular arguments
TAG	formatting tag
UNKNOWN	unknown element
VERTICAL_SKIP	vertical spacing
WORD	sequence of letters
WRAPPER	wrapper

of symbols and commands and limit the sorting options. This approach is applied for the first time in the bounds of such researches.

The context is also determined by the parser state. This is similar to the behavior of the TeX compiler depending on active modes. Supported modes are shown in the Table III.

TABLE III. LaTeX MODES

Mode	Comment
LIST	in a list of items
MATH	in a mathematical expression
PICTURE	in the description of an image
TABLE	in a tabular
TEXT	plain text (default)
VERTICAL	between paragraphs

Modes can be changed anywhere in the document individually or in groups. In addition, a group of local mode determination can be started. The modes will be restored to the values before the group after the group is ended.

III. Description of LaTeX style elements

As mentioned earlier, the sets of available LaTeX characters, commands and environments are defined by the style files. This specificity leads to the need of the command descriptions for the parser. The task of the information extraction from the style files source code is extremely nontrivial and is comparable, if not superior to, the complexity of the TeX compiler implementation. Therefore, in this work we propose to use externally generated descriptions of symbols, commands and environments, which can then be transferred as a parameter for the parser. The basic structures, using which the information about the style files is formed, are presented below.

Operation is an operation on the LaTeX state.

- `directive` is the action directive: `BEGIN` or `END`.
- `operand` is LaTeX mode or `GROUP` (group of local mode definitions).

An operation describes the process of modes change. `BEGIN` means the activation of a mode, `END` means deactivation.

Parameter is a symbol or a command parameter description.

- `lexeme` is the lexeme type (logical sense), optional.
- `modes` are the modes where the parameter is defined.
- `operations` are the operations performed before the parameter.

Symbol is a LaTeX symbol description.

- `lexeme` is the lexeme type (logical sense).
- `modes` are the modes where the symbol is defined.
- `operations` are the operations performed after the symbol.
- `parameters` are the parameter descriptions.
- `pattern` is the LaTeX pattern.

Pattern describes the symbol signature, where `#parameter_index` defines the parameter position, and other characters correspond to the symbol characters in the document source. A JSON example of the inline equation symbol description:

```
{
    "lexeme": "INLINE_EQUATION",
    "modes": ["TEXT"],
    "operations": [{
        "directive": "END",
        "operand": "MATH"
    }],
    "parameters": [{
        "operations": [{
            "directive": "BEGIN",
            "operand": "MATH"
          }]
    }],
    "pattern": "$#1$"
}.
```

Command is a LaTeX command description.

- `lexeme` is the lexeme type (logical sense).
- `modes` are the modes where the command is defined.
- `operations` are the operations performed after the command.

- `parameters` are the parameter descriptions.
- `pattern` is the LaTeX pattern.
- `name` is the name of the command.

It differs from the symbol description only by the added name property. A JSON examle of the two author information commands with the same name but different parameters:

```
{
    "lexeme": "TAG",
    "modes": ["TEXT"],
    "parameters": [{ }, { }],
    "pattern": "[#1]#2",
    "name": "author"
},
{
    "lexeme": "TAG",
    "modes": ["TEXT"],
    "parameters": [{ }],
    "pattern": "#1",
    "name": "author"
}.
```

Environment is a LaTeX environment description.

- `lexeme` is the lexeme type (logical sense).
- `modes` are the modes where the environment is defined.
- `name` is the name of the environment.

A JSON example of the horizontal center alignment environment:

```
{
    "lexeme": "WRAPPER",
    "modes": ["TEXT"],
    "name": "center"
}.
```

The proposed way of describing the style elements is easy to understand: it does not require programming skills or deep knowledge of the formal language theory. The commands descriptions can be prepared even by LaTeX users having some experience in markup using this system. The knowledge of the valid syntax for symbols and commands and the understanding of element logical senses are enough. At the same time, this method is quite flexible, since it allows to describe not only the syntax structures, but also to define the meta information (LaTeX modes, lexeme types) to manage the context, and the set of the descriptions is not fixed and can be modified on the replacing or the adding of a style file.

IV. Syntax tree of LaTeX document

The relative position of LaTeX symbols, commands and environments forms a tree structure, the root of which is the `document` environment. The nodes of this structure are called *tokens*. During the parsing of a LaTeX document the source code elements can produce certain types of tokens, depending on the context and lexeme types. The token types are listed in the Table IV.

TABLE IV. Token types of LaTeX syntax tree

Token type	**Source exmaple**

LaTeX environment body	`\begin{tabular}{c|c}`
	`height & 1.2m`
	`\end{tabular}`

LaTeX command	`\includegraphics`
	`[width=10cm]`
	`{../figure.eps}`

LaTeX environment	`\begin{tabular}{c|c}`
	`height & 1.2m`
	`\end{tabular}`

Label	`\ref{equation1}`
Linear dimension	`\textwidth=10cm`
Number	`height 1.2\,m`
Paragraph separator	Paragraph □ New paragraph

Filesystem path	`\includegraphics`
	`[width=10cm]`
	`{../figure.eps}`

Space	`height□1.2\,m`
Symbol	`height 1.2 \, m`

Tabular parameters	`\begin{tabular}{c|c}`
	`height & 1.2m`
	`\end{tabular}`

Word	`height 1.2 \,m`
Raw char sequence	`\verb|complex source|`

The resulting tree is useful to convert LaTeX documents into other formats by interpretation of the individual tokens. And since each token has a lexeme type this structure is informative in addition for data mining.

V. Parsing algorithms for LaTeX document elements

This section describes the algorithms developed during the research for parsing of LaTeX source individual fragments: pattern, symbol, command and environment. Cross recursive calls are allowed for all the algorithms. This is because of, generally speaking, there are no restrictions on the types and depth of nested tokens, and the proposed method of parsing is similar to the top-down recursive parser.

The algorithm 1 describes the parsing process of a LaTeX symbol or command pattern. It has two main purposes: to choose the applicable syntax and to obtain recursively the set of parameter tokens. If the algorithm returns TRUE, the proposed syntax is applicable, and $parameterTokens$ will list the obtained tokens for the parameters in the correct order. If the algorithm returns FALSE, the symbol or the command with the proposed signature cannot be used at the given position.

Algorithm 1 Parsing of a symbol or command pattern

```
Require:
    $W$ is the source string to parse,
    $pos$ is the current position in the source,
    $W_p$ is the LaTeX pattern,
    $pos_p = 0$ is the current position in the pattern,
```

$style$ is the description of the symbol of the command that owns the pattern,
$parameterTokens = []$ is the stack of the parameter tokens.
Ensure:
TRUE, if the source corresponds to the pattern;
FALSE otherwise.

```
while $pos_p$ not in the end of $W_p$ do
  if $W_p[pos_p]$ is a space then
    if cannot read a space from $W$ at $pos$ then
      return FALSE
    end if
    move $pos$ the the space end
    $pos_p = pos_p + 1$
  else if $W_p[pos_p] == \#$ then
    $pos_p = pos_p + 1$
    parameter index = $W_p[pos_p]$
    get the parameter properties from $style$
    if cannot read the parameter from $W$ at $pos$ then
      clear $parameterTokens$
      return FALSE
    end if
    push the readed parameter token in $parameterTokens$
    move $pos$ to the parameter end
    $pos_p = pos_p + 1$
  else
    if $W[pos]! = W_p[pos_p]$ then
      return FALSE
    end if
    $pos = pos + 1$
    $pos_p = pos_p + 1$
  end if
end while
return TRUE
```

The subtleties that are not included in the description of the algorithm: if a parameter has a special lexeme type corresponding, for example, to a table settings or a file system path, a special algorithm for parsing is used, which returns a token of the appropriate type.

The algorithm 2 describes how to parse a LaTeX symbol. The symbols with allowable patterns are selected according to the source code at the current position and the active LaTeX modes. All of them are iterated until the suitable one is found. If one of the allowable patterns proved to be applicable, the corresponding symbol token is generated, and the stack of the parameters, obtained during the pattern parsing process, forms the child tokens list. If none of the allowable patterns is applicable, a token symbol with an undefined description is returned. Thus, the algorithm always returns a positive result.

Algorithm 2 Parsing of a symbol

Require:
W is the source string to parse,
pos is the current position in the source.
Ensure: a symbol token.

```
backup the current state
get the descriptions of the symbols starting with $W[pos]$ for the current state
for all the obtained symbol descriptions do
  if $W$ staring from $pos$ corresponds to the symbol pattern then
    $t$ = token of the symbol with the current description
    child tokens of $t$= the parameter tokens stack obtained by the pattern parsing
    return $t$
  end if
  restore the backuped state
end for
return a symbol token with undefined description
```

The subtleties that are not included in the description of the algorithm: if the returned token has the lexeme type of a space, a paragraph separator, a letter or a digit, it is converted into a token of the appropriate type.

The algorithm 3 describes the process of a LaTeX command parsing. It is logically practically the same as the algorithm 2, except that the allowable commands are defined by name, and if the source code at the current position doesn't contain a command, the algorithm doesn't return a token.

Algorithm 3 Parsing of a command

Require:
W is the source string to parse,
pos is the current position in the source.
Ensure: a command token or nothing if there is no command at the current position.

```
if $W$ at $pos$ doesn't start with \command_name then
  exit
end if
backup the current state
get the command name
get the descriptions of the commands with the obtained name for the current state
for all the obtained command descriptions do
  if $W$ staring from $pos$ corresponds to the command pattern then
    $t$ = token of the command with the current description
    child tokens of $t$= the parameter tokens stack obtained by the pattern parsing
    return $t$
  end if
  restore the backuped state
end for
return a command token with undefined description
```

The algorithm 4 describes how to parse a LaTeX environment. If the source code at the current position does not contain an environment beginning, nothing is returned. Otherwise, it returns a token corresponding to the environment, the description of which is obtained by the name.

Algorithm 4 Parsing of an environment

Require:
W is the source string to parse,

pos is the current position in the source.
Ensure: an environment token or nothing if there is no environment at the current position.

```
1: if W at pos doesn't start with \begin{environment_name} then
2:   exit
3: end if
4: get the environment name
5: t = the environment token that corresponds to the name at the current state
6: move pos to the end of \begin{environment_name}
7: parse the pattern of the command with the name "environment_name"
8: store the corresponding token as the token of t begin command
9: while W at pos doesn't correspond to \end{environment_name} do
10:   parse a child token of t
11: end while
12: move pos to the end of \end{endenvironment_name}
13: parse the pattern of the command with the name "endenvironment_name"
14: store the corresponding token as the token of t end command
15: return t
```

The subtleties that are not included in the description of the algorithm: an incorrect LaTeX document may not contain any end command for an environment (in this case the end command will be automatically generated at the source end), and if there is no any description of the environment with the given name, an environment token with an undefined description is returned.

The algorithm 5 describes the general process of the next token extraction. Is not known a priori which type of token is at the current position of the source code. So the possible options are iterated: a space, an environment, a command, a symbol. A symbol token can always be returned, so each iteration consumes a nonzero fragment of the source code. Thus, the algorithm is realizable anyway.

Algorithm 5 Parsing of a LaTeX source code

Require:
W is the source string to parse,
$pos = 0$ is the current position in the source.
Ensure: $tokens$ is the sequence of the parsed tokens.

```
1: while pos not in the end of W do
2:   backup the current state
3:   if can parse a space from W at pos then
4:     push the space token to tokens
5:     move pos to the space end
6:     go to a new iteration
7:   end if
8:   restore the backuped state
9:   if can parse an environment from W at pos then
10:     push the environment token to tokens
11:     move pos to the environment end
12:     go to a new iteration
13:   end if
14:   restore the backuped state
15:   if can parse a command from W at pos then
16:     push the command token to tokens
17:     move pos to the environment end
18:     go to a new iteration
19:   end if
20:   restore the backuped state
21:   parse a symbol from W at pos
22:   push the symbol token to tokens
23: end while
```

The described algorithms in conjunction with the comments about the subtleties cover all the possible situations in the used LaTeX syntax interpretation.

VI. IMPLEMENTATION

The ideas described in this article where implemented by the author several times. The first time they have been successfully used for tasks of the document automatic correction [13] and the construction of article XML descriptions for the topic model. The implementation was done using Qt/C++, the source code was not publicly distributed.

Since 2016 the author launched a project for the implementation of LaTeX parser on JavaScript [14]. This is a set of libraries released under the LGPLv3 license. Their main goal is the transparent integration of LaTeX document analyzing tools for all the environments supporting JavaScript, including WEB.

The following libraries are available:

- `Latex.js` contains the basic definitions of the LaTeX structures such as lexemes, modes and operations.
- `LatexStyle.js` contains the definitions of the LaTeX style structures: symbols, commands, environments. It also provides tools for working with collections of such structures in JSON format.
- `LatexTree.js` contains the definitions of the LaTeX syntax tree structures: all the token types, the tree itself.
- `LatexParser.js` provides the parser class that receives style descriptions and allows build a syntax tree for a LaTeX document source.

The basic principle of the libraries is shown in the Fig. 3.

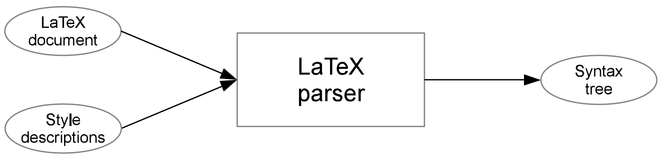


Fig. 3. Scheme of the parcing process

At the moment, the principles of all libraries are relevant to the algorithms given in this work. A project for the viewing of LaTeX documents in browsers with HTML is under development using this libraries.

VII. Conclusion

This article discusses the problem of parsing the structured text documents, the format of which is not fully defined by standards or other rules common to all the documents. The documents in the LaTeX format are chosen as an example of such files. It has been shown that their markup language has no any formal grammar, therefore it requires a separate approach, taking into account the external resources corresponding to loadable style files. Existing solutions do not use logical sense of syntax elements, therefore they can not be used for the intelligent text analysis.

An approach that allows to describe LaTeX style files with simple constructions was proposed. The developed parser algorithms working with such descriptions are listed. They are implemented as a set of JavaScript libraries distributed by the LGPLv3 license.

Thus, we can assume that the problem of LaTeX documents parsing is solved in the framework if syntax tree representation discussed in this work.

But it remains an important issue, which prevents the rapid implementation of the developed parser. There is a need to manually describe the style elements: LaTeX symbols, commands and environments. It is simple but time consuming process requiring typesetting skills of the performers. The continue of the research on this topic may be given to the automation of this process.

Acknowledgment

This work was supported by the RFBR grants: 16-37-60049, 16-07-01267.

The author is deeply indebted to Prof. Konstantin Vorontsov for the wise scientific supervision in the field of machine learning and the statement of this research original problem. Also I am grateful to Alexey Rozanov (the head of the publishing house “Fizmatkniga”) for the extensive proofreading experience with LaTeX.

References

[1] Saul Rosens, *Programming Systems and Languages*. McGraw Hill Computer Science Series. New York/NY: McGraw Hill, 1967. ISBN 0070537089.

[2] ISO/IEC 14977:1996. Information technology -- Syntactic metalanguage -- Extended BNF, Web: http://www.iso.org/iso/catalogue_detail?csnumber=26153.

[3] Syntax diagram — Wikipedia, Web: https://en.wikipedia.org/wiki/Syntax_diagram.

[4] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman, *Compilers: Principles, Techniques, and Tools* (2nd Edition). Prentice Hall, 2006.

[5] K. V. Chuvilin, “Parametric approach to the construction of syntax trees for partially formalized text documents”, *Machine Learning and Data Analysis*, vol. 2, issue 2, 2016.

[6] Donald Ervin Knuth, *The TeXbook*. Computers and Typesetting, A. Reading, MA: Addison-Wesley, 1984. ISBN 0-201-13448-9.

[7] Leslie Lamport, *LaTeX: A document preparation system: User’s guide and reference*. illustrations by Duane Bibby (2nd ed.). Reading, Mass: Addison-Wesley Professional, 1994. ISBN 0-201-52983-1.

[8] K. V. Chuvilin, “Machine Learning Approach to Automated Correction of LaTeX Documents”, *in Proceedings of the 18th FRUCT & ISPIT Conference, 18-22 April 2016*, Technopark of ITMO University, Saint-Petersburg, Russia. FRUCT Oy, Finland. ISSN 2305-7254, ISBN 978-952-68397-3-8, pp. 33–40. Web: http://fruct.org/publications/fruct18/files/Chu.pdf.

[9] eLIBRARY.RU — Rossijskij indeks nauchnogo citirovaniya [eLIBRARY.RU — Russian Science Citation Index]. Web: http://elibrary.ru/project_risc.asp.

[10] plasTeX A Python Framework for Processing LaTeX Documents. Web: http://plastex.sourceforge.net/plastex/index.html.

[11] Sven Heinicke LaTeX-Parser-0.01 - http://search.cpan.org. Web: http://search.cpan.org/~svenh/LaTeX-Parser-0.01/.

[12] SnuggleTeX - Overview & Features. Web: http://www2.ph.ed.ac.uk/snuggletex/documentation/overview-and-features.html.

[13] K. V. Chuvilin, *Automatic synthesis of correction rules for text documents in the LaTeX format*. PhD dissertation. Dorodnicyn Computing Centre of the Russian Academy of Sciences, Moscow, 2013. (in Russian)

[14] texnous latex-parser — Bitbucket. Web: https://bitbucket.org/texnous/latex-parser/.