# A PARSING MACHINE ARCHITECTURE ENCAPSULATING DIFFERENT PARSING APPROACHES

*Nikolay Handzhiyski, Elena Somova**

University of Plovdiv "Paisii Hilendarski", Plovdiv
Bulgaria

* Corresponding Author: eledel@uni-plovdiv.bg

**Abstract:**In the parsing theory, seemingly different parsing machines exist, due to the use of different terms that have similar meanings. This article addresses the diversity in the parsing terminology by proposing definitions for various objects and processes in a parsing machine. Based on the defined objects and processes, a common architecture of a parsing machine is proposed that is applicable in practice when using different parsing approaches. The modules of the parsing machine and their close relations are shown. The proposed parsing machine architecture is to a large extent used in the parsers generated by Tunnel Grammar Studio.

**Key words:** Lexing, Parsing, Token, Lexeme, Syntax Analysis.

## 1. INTRODUCTION

To be able to work on data the electronic systems have to perform a common process: the data recognition. This process is performed by a *parsing machine* (PM) – an abstract machine that includes all of the data recognition operations, and is confusingly called a parser by many authors. During the execution of the PM, various subprocesses (subtranslators [1]) perform operations on the data (usually in the form of a string of characters) that have to be recognized. The lexical analysis [1] is performed by the first subprocess, which we will call a *lexer* (sometimes called lexical translator). It converts the input characters into *tokens* according to a formal grammar (for short a grammar) that describes the structure of the tokens as formed by characters. The syntax analysis (*parsing* [1, 2]) is performed by the second subprocess, which we will call a *parser*. It checks whether the sequence of tokens (received from the lexer) belong to the data language, which is described by another given grammar – the parsing grammar. The output of this subprocess is a sequence of Syntax Structure Construction Commands (SSCC). The third subprocess outputs an explicitly built syntax structure from the SSCC or uses them directly to perform

the specific to the PM task. This architecture of a PM (that we will classify as "traditional") is often described in the literature, but with different terms for the same concepts.

A grammar consists of formally defined rules (for short only rules, according to Chomsky the "laws" of the language) that are described with a meta syntax such as ABNF [3] and EBNF, defined by Wirth and [4]. The rules are also called productions and their names are called *nonterminals*. Each rule accepts *terminal symbols*. To perform the data recognition, the subprocesses of the PM often do that internally as automata [5]. To avoid the confusion between terminal symbols and nonterminal symbols, some authors call the former symbols tokens [6].

The main goal of this article is to propose a common architecture of a PM that can encapsulate different parsing approaches as "sometimes it pays to revisit an old concept from a fresh angle" [7].

Section 2 provides an overview of the various terms in the parsing theory and their uses. Section 3 contains: a description of the problem that is the purpose of this article; gives a solution to the problem by proposing a list of clear definitions for existing terms, based not only on their initial definitions, but also on their established functionality; on the basis of them and several new terms, a common architecture of the PM is proposed, independent of the applied parsing algorithms, that enables different parsing approaches to be described with it. Conclusions and a perspective for further research in relation to this article are given in Section 5.

## 2. RELATED WORK

In some articles, a token is a synonym to a terminal [8] (same as a terminal symbol). Sometimes, a token is composed of a type and a value [9]. According to [10], a token has a type that symbolizes the class of words that can be described with the token, as each word is composed of one or more characters. In other articles, the elements defined in the parsing grammar are terminal symbols for the parser and syntactically structured symbols for the scanner, where the tokens that are created for a given production (rule) are in the same class, and each token in the class is an instance of the class [11]. For some [12], the lexical analyser partitions a stream of characters into groups called tokens. According to others [1], the lexer maps the input characters to a string of *lexical tokens* that are the terminal symbols defined in the parsing grammar. These lexical tokens (that are the elements at the logical level) could also contain semantic information.

The tokens might not have just one type, but more than one, when more than one rule accepts a sequence of characters [10]. However, this approach allows the ambiguity that exists between the lexical rules in the lexer to spread into the parser. Then it is necessary for the PM to use parsing algorithms that accept the whole class of context-free grammars, such as GLR [13] or GLL [14]. A specificity of this approach is in the ambiguity that exists, when the different input characters can be grouped in different sequences of tokens. In some cases, when one possible sequence

of tokens is shorter than another, then an additional token type (null) has to be used, to make the number of tokens equal in length. However, this requires a change in the parsing grammar [10].

For some authors, a token is a pair of a name and an attribute value [15], where the name is an abstract symbol (representing a lexical unit) and is the input symbol for the parser. A lexeme is a sequence of characters that form the token and are accepted by a given pattern – practically a regular expression. Only the token's name is used by the parsing algorithm [15], as the attribute value is used after the parsing, for the specific task (translation, generations, etc.) and can indicate the lexeme and/or other important information (the text line number and the number of the character in this line where the token begins, for short a *locator*). A scanner might be executed before the lexical analysis and it does not group the characters into tokens, but removes the unnecessary parts of the input data (such as comments and white spaces). The lexical analysis subsequently performs the *tokenization* of the characters into lexemes based on the input it receives from the scanner [15]. In the implementation section, however, the lexer has the function "scan", and the token has a "tag", instead of a name [15]. Other authors [16], define a token as a tuple containing a terminal, a lexeme matching that terminal, and a locator. The last element eliminates the need to store this information in a symbol table that is shared between the lexer and the parser [15].

In [2], the term *symbol* is used as a synonym to a character and a letter, as well as a sentence and a word are used as synonyms to a *string*. In this sense an alphabet is the set of symbols/letters/characters and the sequences of them are a string/word/sentence.

For some, "scanner(e)" is the algorithm that checks if a string of characters belongs to the language defined by a regular expression "e" [17]. The regular expressions have unique names, called tokens, and the scanner is interchangeably called *lexical analyser*.

According to [2], the input of the lexical analyser is "a string of symbols from an alphabet of characters". "It is the job of the lexical analyser to group together certain terminal characters into single syntactic entities, called tokens". "A token is a string of terminal symbols, with which we associate a lexical structure consisting of a pair of the form (token type, data)", where the type has a value from a finite domain and the data is anything found to be relevant for the particular token. "The first component of a token is used by the syntactic analyser for parsing" and the second component is used by the later stages, after the parsing.

Some authors divide the lexer into two levels. The first level works in the traditional way (with a finite state machine) and outputs *universal lexemes*, that are lexical entities – the smallest part of the language [17]. These universal lexemes consist of a token, lexeme and the number of the characters the lexeme. The tokens, in this case, designate a class (all possible lexemes that can be recognized by a given lexical rule) and have a name according to the rule used to recognize the corresponding lexeme [17]. The second level performs on the result of the first level

and outputs *tokenized lexemes* that consist of the name of the rule used by the second level of the lexical analysis, together with all of the used universal lexemes from the first level and the total length of the lexemes as a number [17]. These tokenized lexemes are then the input of the parser. Such a division of the lexer, requires an additional processing and it takes some execution time: the first level of the processing is realized as deterministic finite automata, which is a standard part of a traditional lexer, but the latter is in the form of a non-conventional nondeterministic finite automata (NFA), where it's transitions are based on regular expressions with conditions. The execution of the NFA is with backtracking, and according to the authors, this does not "help the efficiency" [17].

From all said so far, it seems that the terminology changes through time, but still oscillates around the same concepts. It is difficult to pinpoint the origin of each term, and it is debatable to some extent what is the most correct meaning if each: the first defined, or the one "mostly used" in the literature. Tracking back the token term, leads to the final draft report of ALGOL 68 [18], where an extension is defined to be a comment between two symbols (as a symbol is the basic block of the language, made of one or more characters). Many symbols are classified as belonging to a group of tokens. Then in the revisited report a token is defined to be a symbol that can be preceded by pragments [19], that in turn are: a comment (that does not affect the program in any way); or a pragmat (that may affect the program). Additionally, the previous usage of a token to designate the class of symbols has been removed.

There is an interesting discussion on the topic that a token (as part of a sentence) is for the computer scientist, as is the symbol (as part of a word) for the practitioner of formal linguistics [20]. Additionally, the token has a general meaning for the computer scientist, because the tokens can have a structure of other tokens, that are part of a different language that itself is made of tokens (eventually letters).

It can be concluded that it is true for some of the concepts in the domain that "the more one reads, the more unclear it gets". There can be no formal definition of a parsing machine, when many of its components are a matter of interpretation. This is the purpose of this article, to clearly define a parsing machine and its related terms.

## 3. PROPOSED SOLUTION

This section defines the basic terms related to the parsing process. An attempt is made to give a brief and unambiguous definitions of the existing terms, based not only on their initial definitions, but also on their established functionality. On the basis of them and several new terms, a common architecture of the PM is proposed, independent of the applied parsing algorithms.

### 3.1. Description of the problem

The previous section highlights the problem that there is a wide variety of definitions of many terms and understandings of each, related to the parsing, that are sometimes used one instead of another. The only unambiguous claim is that there

are tokens and lexemes and they are related. Therefore, one of the purposes of this article is to propose clear definitions of aforementioned parsing terms. In addition, the different terms and the different relations between them give a rise to seemingly different PMs. The other goal of the article is to provide a common architecture that can combine the different approaches.

### 3.2. Definitions

The following definitions represent the view of the authors of how the different concepts should be called and what their functionality is:

1. *Character* – a Unicode codepoint [21]. The Unicode standard is old enough and our reasonable expectation is that the PMs (especially those generated by a parser generator) support it and are handling the invalidly encoded input bits properly;
2. *Lexeme* – a sequence of characters [22];
3. *Name* – an integer number representing a category from a particular set of categories;
4. *Attribute* – a tuple of: a name and a value, as their meaning is defined separately per attribute. A set of attributes (possibly empty) is called later only attributes;
5. *Token* – a tuple of: a type (defined later), a sequence of zero or more names, a lexeme (possibly of zero length), and attributes. When the name is only one, it will be referred to as the token's name;
6. *Module* – an entity that performs operations on an input to produce an output. It can be a coroutine [23], a subroutine or a thread of execution depending on the available resources and particular needs;
7. *Parsing Machine* – a set of modules that produces an output (that depends on the particular PM) from a given raw input (a stream of bits). This definition is similar to [24];
8. *Supplier* – these are the first modules in a PM that only transmit binary data to the next modules. There can be one or more of such modules. For example, a module that reads bits from a stream of bits stored in the file system of a computer;
9. *Scanner* – a module that scans characters from the input received from the last supply module. The term *scans* aligns to the usage of *scanning* from Turing, as well as with the card scanner in [23]. For each scanned character, the scanner outputs a token that has a *character* type (defined later) and eventually some additional information found to be relevant for the particular PM in the form of attributes. There must be exactly one scanner as this module is a "border" between the operations on bits and the operations on tokens. By defining the module in this way however, there can be no *scannerless* parsing, because the PM ultimately is defined to process the input bits in some way;
10. *Lexer* – a module that inputs tokens from a previous module and outputs the same (when none of its rules accepts the current input) or different tokens (according to its own grammar rules) to the next module. There can be zero (that effectively

means *lexerless* PM) or more lexers, each ordered one after another, but all after the scanner (similar to [17]);

11. *Parser* – a module that inputs tokens and outputs SSCC. This module: a) is a "border" between the lexer(s) and the optimizers(s) (defined later); b) performs the parsing per se - it verifies that the input tokens conform to the rules of its grammar; c) does not calculate the locator; d) does not emit the found errors in the input directly, but sends them in the SSCC stream; and e) it does not buffer any SSCC, but sends them directly to the next module;

12. *Optimizer* – a module that inputs SSCC and operates on them before outputting the same or different SSCC. There might be zero or more optimizers. For example: a) if the PM algorithm uses backtracking, the optimizer might buffer the SSCC and only output them, after some sufficient number of SSCC are collected; or b) the module might change the SSCC to insert or delete syntax structure elements;

13. *Builder* – a module that inputs SSCC and its output (if any) is not explicitly defined, because it depends on the particular PM. The possible builder types are: *explicit* (that builds a syntax structure) and *implicit* (that uses the SSCC without building anything from them). The explicit builder has a syntax structure *architect* that itself builds the syntax structure and in the case where the structure is a syntax tree, it can be of two types: an *abstract architect* (builds an abstract syntax tree) and a *concrete architect* (builds a concrete syntax (parse) tree). The implicit builder has no architect, but it accepts a *visitor* (from the visitor design pattern) that will receive the SSCC for its specific task. This is the module that calculates the locator. The disambiguation filters [25] are used on the result of the explicit builder output.

### 3.3. Parsing machine architecture

Fig. 1 shows the common architecture of a PM with the participating modules according to the proposed definitions (with SSCC examples). No particular *communication model* between the modules is mandatory. The different possible models are (where one applies only when the more restrictive model does not):

- *singly linked list* – the data is only transmitted from the first supplier to the builder in a unique path. In such a case, each module might operate without the expectation of an interruption from any other module;
- *doubly linked list* – a module might send data only to its direct neighbours. This model can effectively be used for the implementation of the *lexical feedback* in [10] (called *backdoor approach* in [6]);
- *graph* – each module might produce data to any other module. There can be more than one supplier as well as more than one builder.

The connections between the modules might transport different data types, and it is not defined how the modules will agree for the data transmission (a module pushing its output to the next, a module pooling its input from the previous, or a mixture of these): a) bits – receivable by the suppliers; b) tokens – receivable by the

the lexers and the parsers; c) SSCC – receivable by the optimizers and the builders; d) flags/options – receivable by any module.

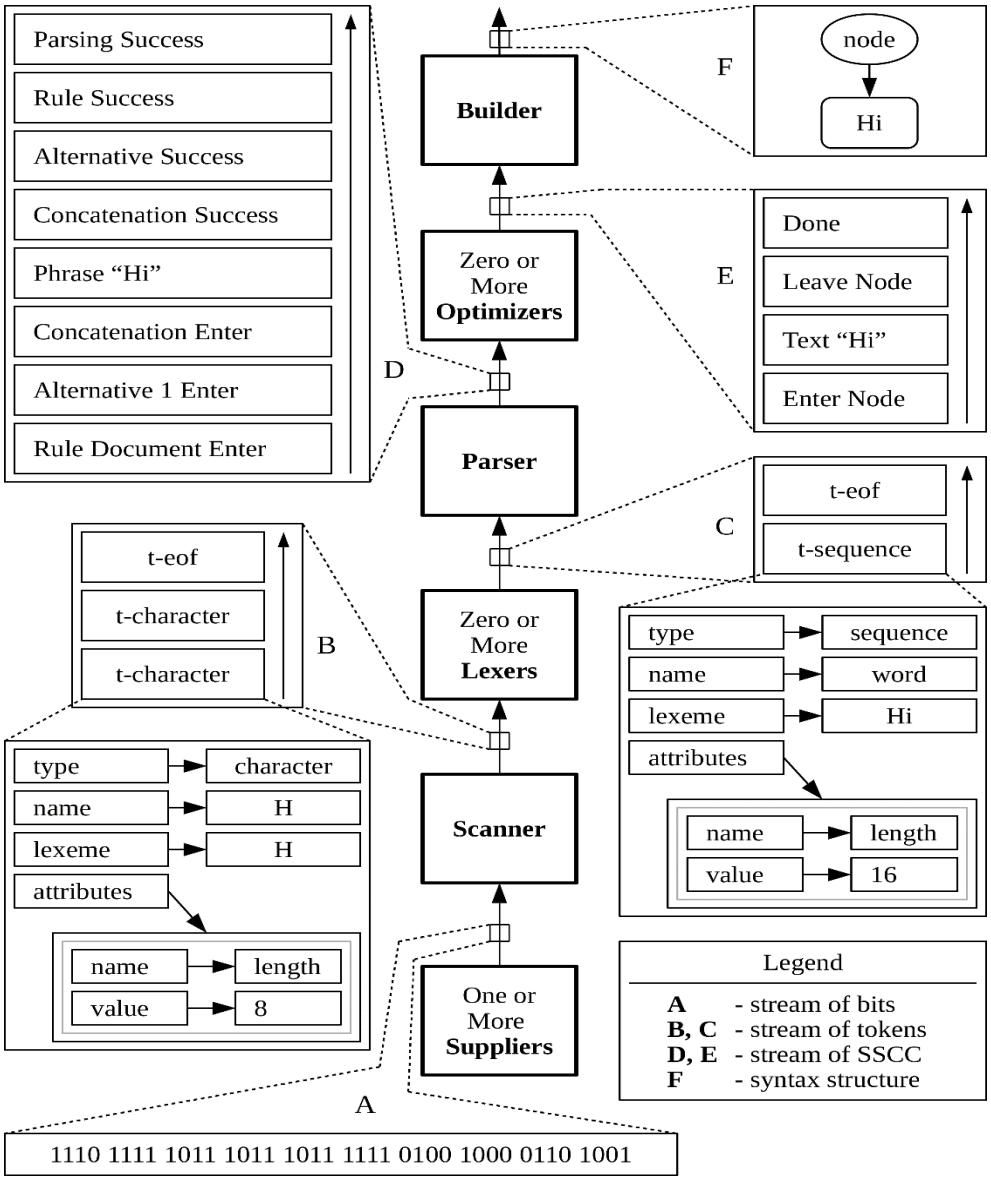


*Fig. 1. Common architecture of a parsing machine*

Other than the singly linked list communication models imply that some modules will have more then one input and/or output connection that in turn does imply that any sequence of PM items (bits/tokens/SSCC) can be transformed to any other sequence. That will make the PM Turing complete, when each of the

transported elements are from their respective finite sets. However, this article describes a PM that uses all of the token fields for a recognition. The unbounded length of the lexemes makes the set of tokens of not a finite length (set whose elements cannot be counted), that means that such a PM is at least Turing complete. From now on, a singly list model is assumed for simplicity.

The following token types are relevant to the described PM:

- *t-character* – a token that is created for a single character by the respective modules. The scanner is the first to create such a token. The lexers, after that, create tokens of this type, only when no rule in the particular lexer can be found to accept the current input. The token's name and its lexeme are having values like the character. This implies that the categories that the modules use to give names to the tokens they generate, include the whole set of characters (Unicode);
- *t-sequence* – a token that has a list of names from the rules in the lexer that accepted the particular input, and a lexeme that combines all of the accepted characters. When the lexer prioritizes the rules, only the name of the highest priority one will be used;
- *t-eof* – a token that is sent at the end of the input data. The token has no name and its lexeme has a length of zero. This token is often called a sentinel [15] and in practice there is a reserved value of zero [11, 26] (when the tokens are represented as integers) or null/nil (when the tokens are objects [27]);
- *t-limit* – a token that is sent only once by a module, when it cannot handle its input, because the number of potential characters that could be part of the eventually recognized t-sequence token are more than the maximum number that the lexer is willing to accept. This token type exists analogically to response code 431 (Request Header Fields Too Large) from the HyperText Transfer Protocol [28]. A hypothetical implementation might never send a t-limit token, when it is ready to accept any lexeme length, although this is not recommended.

### 3.4. Application of the architecture

The proposed architecture is applied to a large extent in Tunnel Grammar Studio (TGS) [29], in particular the single list communication model. The TGS generated parsers optionally store information about the number of bits in the stream, which are used by the scanner to recognize each individual character, in the form of an attribute. This allows the user of the syntax structure to know exactly where each syntax element is located in the stream. These generated parsers work with the tunnel parsing (TP) algorithm [30], where all of the defined token types in this article are in use. The handling of the infinite by definition lexemes is done by classifying each of them according to the expected lexemes, which are a finite number and each one with a finite length, because the grammars that TP uses are finite and context-free. The syntax of the grammars that TGS accepts is based on the ABNF meta syntax, with an extension that enables the matching of a lexeme, regardless of the characters in it (with other words, to ignore the lexeme altogether and use only the name of the

t-sequence token), as well as to match a lexeme case sensitively or insensitively. The defined character ranges in the ABNF standard are matched to the t-character tokens. The case sensitivity in [31] also applies to the t-character tokens not to the t-sequence tokens (that are handled by the extension).

The TGS generated parsers perform *linear multithreaded parsing*, by executing the different PM modules in dedicated threads of execution. The PM generation settings allow the thread that requests the parsing to be used for parsing (in this case the PM is a subroutine), or the PM to have from one to three dedicated threads for the execution of different module groups as follows:

- One supplier (that reads the bits from the input stream), a scanner (that decodes the bits into Unicode characters and then forms the t-character tokens) and a lexer (which exists if there is a non-empty lexical grammar to group the tokens received from the scanner into tokens for the parser) are optionally in their own thread. The purpose of the separate thread is to enable the PM to read the bits from the input stream and to convert them to tokens without this to stall the other PM modules (this occurs in a traditional PM that has its modules as subroutines, not threads);
- A parser module that executes the TP algorithm, together with an optimizer (if such is generated) are optionally in their own thread. The optimizer collects a certain number of SSCC before sending them to the builder. The benefit of the dedicated thread is when the parsing algorithm takes a significant amount of time to process the tokens. This can be expected when the algorithm moves backwards for nondeterministic grammars;
- A builder module might also be in a separate thread. That is useful when the construction of the syntax tree or the use of SSCC directly by a visitor takes a considerable time. If the visitor directly generates data during runtime (for example, a compiler), then the generation could be done directly on the basis of the received SSCC without the building of an explicit syntax tree. A beneficial side effect, when a syntax tree is not generated, is that there is no memory to be released after the PM completes. This saves not only memory space, but improves the execution time. Then in this thread, in addition to the receiving of the SSCC, the compiler's output will be stored in streams. This should not force the other PM modules to wait. It is better, in this case, for the other modules to execute in parallel.

The advantages of the proposed new PM architecture and the related concepts are:

- The PM architecture is with a clear separation of responsibilities between the modules and it is based on the Unicode standard;
- The PM might run linearly with different threads for the different modules (compared to the traditionally used subroutines);
- Many different PMs can be expressed with this common PM architecture, that is at least Turing complete, because of the infinite tokens (and for this reason is more capable than the traditional PMs);

• When the characters are transferred from the lexer through the parser (inside tokens) to the builder (inside SSCC) the different modules do not need to use any shared structures. That enables the modules to be developed in different programming languages (or even run on different hardware).

## 4. CONCLUSION

The main contributions to the study are the following:

• New unified concepts such as: PM, supplier, optimizer, SSCC, architect, the three PM communication models (singly linked list, doubly linked list, graph), t-character token, t-sequence token, t-limit token, implicit builder, explicit builder;

• Different interpretation of concepts such as: character, token, parser, scanner, lexer, builder;

• A PM that accepts input of bits (not characters nor symbols) and includes all processes up to the generation of the syntax structure;

• A clear separation of responsibilities between a scanner and a lexer, as their definitions are often blurred.

Additionally, in the study, the following are proposed:

• A new PM module, called supplier, to perform operations on the input, before the forming of the characters (file reading, internet transfer, etc.);

• The alphabet of the parser's grammar to be infinite in nature, by defining that the token's lexeme is also used by the parser for parsing, not only the token's name;

• A new PM module, called optimizer, to perform operations on SSCCs, where they represent the statically typed concrete syntax structure build information;

• The builder and the parser modules to be fully separated from each other – not to share common structures;

• The builder module to be responsible for the locator.

The proposed parsing machine architecture is inspired by PM generated by Tunnel Grammar Studio (TGS) [29] – a parser generator from ABNF grammars to program source code. The future goal of the authors is to present how the infinite lexemes in the tokens are used for parsing in the TGS generated parsing machines. It is also possible for the lexer to support ambiguity [32], as long as the parser supports ambiguous token streams.

An open question remains, what is the variety of the terminology in the literature related to the parsing process in languages other than English, when not in every language the terms have an unambiguous translation, and are used by different authors.

The cited definitions by the different authors are not intended to be exhaustive and none of the articles is deemed incorrect in its own context. Only those that are found to be directly related to this article are included.

## REFERENCES

[1] Deremer, F. L. *Practical translators for LR(K) languages*. Massachusetts Institute of Technology, 1969.

[2] Aho, A., J. Ullman. *The Theory of Parsing, Translation, and Compiling*. ISBN-10: 0139145567, Prentice-Hall, Inc., 1972.

[3] Crocker, D., P. Overell. *Augmented BNF for Syntax Specifications: ABNF*. 2008, Available at: https://tools.ietf.org/html/rfc5234 (visited on: 25.06.2021).

[4] *ISO/IEC 14977:1996(E) Information technology – Syntactic metalanguage – Extended BNF*, Available at: http://standards.iso.org/ittf/PubliclyAvailable Standards/ (visited on: 25.06.2021).

[5] Kleene, S. C. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies* (AM-34), Vol. 34, Princeton University Press, 2016, pp. 3-42, https://doi.org/10.1515/9781400882618-002.

[6] Johnson, S., M. Hill. Yacc: Yet Another Compiler-Compiler. *Unix Programmer's Manual*, Vol. 2, 1977, pp. 353-387.

[7] Posse, E., H. Vangheluwe. Parsing revisited: a transformation-based approach to parser generation. *Proc. of the 2007 Python Conference*, 2007, pp. 1-9.

[8] Parr, T., K. Fisher. LL(*): The Foundation of the ANTLR Parser Generator. *ACM SIGPLAN Notices,* Vol. 46, No. 6, 2011, pp. 425-436, https://doi.org/10.1145/ 1993316.1993548.

[9] Diekmann, L., L. Tratt. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. *34th European Conference on Object-Oriented Programming*, 2020, pp. 6:1–6:32, https://doi.org/10.4230/LIPIcs.ECOOP.2020.6.

[10] Aycock, J., R. Horspool. Schrödinger's Token. *Software: Practice and Experience*, Vol. 31, 2001, pp. 803-814, https://doi.org/10.1002/spe.390.

[11] Moessenboeck, H. *Coco/R - A Generator for Fast Compiler Front Ends*, Johannes Kepler Universitat Linz, 1990, https://doi.org/10.3929/ETHZ-A-000534270.

[12] Yang, W., et al. On the applicability of the longest-match rule in lexical analysis. *Computer Languages, Systems & Structures*, Vol. 28, No. 3, 2002, pp. 273-288, https://doi.org/10.1016/S0096-0551(02)00014-0.

[13] Tomita, M. Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. *The Springer International Series in Engineering and Computer Science*, ISBN: 978-0-89838-202-0, Vol. 8, Springer, 1985, https://doi.org/10.1007/ 978-1-4757-1885-0.

[14] Johnstone, A., E. Scott. Modelling GLL Parser Implementations. *Software Language Engineering*, Lecture Notes in Computer Science, Vol. 6563, Springer, 2011, pp. 42-61. https://doi.org/10.1007/978-3-642-19440-5_4.

[15] Aho, A., et al. *Compilers: Principles. Techniques, and Tools*, ISBN-10: 0321486811, Addison-Wesley, 2006.

[16] Van Wyk, E. R., A. C. Schwerdfeger. Context-Aware Scanning for Parsing Extensible Languages. *Proc. of the 6th Int. Conf. on Generative programming and component engineering*, 2007, pp. 63-72, https://doi.org/10.1145/1289971.1289983.

[17] Rus, T., T. Halverson. A Language Independent Scanner Generator. *CiteSeer*, 1999, pp. 1-35.

[18] Mailloux, B. J., J. Peck, C. Koster. *Final draft report on the algorithmic language Algol 68*. 1968, https://doi.org/10.5555/1064072.1064073.

[19] Wijngaarden, A., et al. Revised Report on the Algorithmic Language Algol 68. *ACM SIGPLAN Notices*, Vol. 12, No. 5, 1977, pp. 1-70, https://doi.org/10.1145/ 954652.1781176.

[20] Grune, D., C. Jacobs. *Parsing Techniques: A Practical Guide*. ISBN: 978-0-387-20248-8, Springer-Verlag New York, 2008, https://doi.org/10.1007/978-0-387-68954-8.

[21] *Unicode Standard*, Available at: https://www.unicode.org/ (visited on: 25.06.2021).

[22] Bonami, O., et al. *The lexeme in descriptive and theoretical morphology*. ISBN-13 (15): 978-3-96110-111-5, 2018, https://doi.org/10.5281/zenodo.1402520.

[23] Conway, M. Design of a Separable Transition-Diagram Compiler. *Communications of the ACM*, Vol. 6, No. 7, 1963, pp. 396-408, https://doi.org/ 10.1145/366663.366704.

[24] Woods, W. Cascaded ATN Grammars. *American Journal of Computational Linguistics*, Vol. 6, No. 1, 1980, pp. 1-12.

[25] Macedo, J. N., J. Saraiva. Expressing disambiguation filters as combinators. *Proc. of the 35th Annual ACM Symposium on Applied Computing*, 2020 pp. 1348-1351, https://doi.org/10.1145/3341105.3374123.

[26] Lesk, M., E. Schmidt. Lex – a lexical analyzer generator. 1990, Available at: https://www.cs.utexas.edu/users/novak/lexpaper.htm (visited on: 25.06.2021).

[27] Kuhl, B., A.-T. Schreiner. Objects for Lexical Analysis. *ACM SIGPLAN Notices*, Vol. 37, No. 2, 2002, pp. 45-52, https://doi.org/10.1145/568600.568610.

[28] Nottingham, M., R. Fielding. *Additional HTTP Status Codes*. ISSN: 2070-1721, 2012, Available at: https://tools.ietf.org/html/rfc6585 (visited on: 25.06.2021).

[29] *Tunnel Grammar Studio*, Available at: https://www.experasoft.com/products/tgs/ (visited on: 25.06.2021).

[30] Handzhiyski, N., E. Somova. Tunnel Parsing with countable repetitions. journal *Computer Science*, ISSN 2300-7036, Vol. 21, No. 4, 2020, pp. 441-462, https://doi.org/10.7494/csci.2020.21.4.3753.

[31] Kyzivat, P. Case-Sensitive String Support in ABNF. *RFC 7405*, ISSN: 2070-1721, 2014, https://doi.org/10.17487/RFC7405.

[32] Quesada, L., F. Galiano, F. J. Cortijo. A Lexical Analysis Tool with Ambiguity Support. *ArXiv*, Vol. abs/1202.6583, 2012, pp. 1-5.

***Information about the authors:***

**Nikolay Handzhiyski** – Co-Founder and CEO of ExperaSoft UG (haftungsbeschränkt), Goldgasse 10, 77652 Offenburg, Germany, e-mail: nikolay.handzhiyski@experasoft.com; PhD student at the University of Plovdiv; Sole developer of Tunnel Grammar Studio

**Elena Somova** – Professor in Computer Science, Head of Computer Science Department at the University of Plovdiv "Paisii Hilendarski"; research areas: green technologies, sustainability, gamification of learning, e-learning, game-based learning

**Manuscript received on 28 June 2021**