BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **22**, No 2
Sofia • 2022 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2022-0021

Tunnel Parsing with the Token's Lexeme

Nikolay Handzhiyski[1,2], *Elena Somova*[1]

[1]*University of Plovdiv "Paisii Hilendarski", 24 Tzar Assen Str., 4000 Plovdiv, Bulgaria*
[2]*ExperaSoft UG (haftungsbeschraenkt), 10 Goldasse Str., Offenburg 77652, Germany*
E-mails: nikolay.handzhiyski@experasoft.com eledel@uni-plovdiv.bg

Abstract: *The article describes a string recognition approach, engraved in the parsers generated by Tunnel Grammar Studio that use the tunnel parsing algorithm, of how a lexer and a parser can operate on the input during its recognition. Proposed is an addition of the augmented Backus-Naur form syntax that enables the formal language to be expressed with a parser grammar and optionally with an additional lexer grammar. The tokens outputted from the lexer are matched to the phrases in the parser grammar by their name and optionally by their lexeme, case sensitively or insensitively.*

Keywords: *Parsing algorithm, tunnel parsing, lexeme matching, advanced grammar, phrase state machine.*

1. Introduction

To be able to work with data the software systems have to perform the process of the data recognition. This process is performed by a Parsing Machine (PM) – an abstract machine that includes all data recognition modules. During the execution of the PM, various subprocesses work with the data (a stream of bits) to be recognized [1]. The PM has the following modules:

- **Supplier** – supplies the input bits (for example by reading them from the file system of a computer);
- **Scanner** – decodes the input bits into characters (assigned Unicode codepoints [2]);
- **Lexer** – performs the lexical analysis, as during this process the characters are grouped into lexemes (as each lexeme is a part of a token) based on a formal grammar (for short a grammar) called later a **lexer grammar**;
- **Parser** – performs the syntax analysis (parsing [3, 4]). This process checks if the sequence of tokens received from the lexer belongs to the data language that is described by another given grammar – the **parser grammar**. The output of this subprocess is a sequence of Syntax Structure Construction Commands (SSCC);
- **Builder** – a module that explicitly builds syntax structure from the SSCC or uses the SSCC directly to perform the specific to the PM task.

The data is “flowing” from a supplier, through a scanner, a possible lexer, a parser and finally to a builder, all in a unique path (this is a singly linked list communication model described in [1]). The grammars consist of formally defined rules (the “laws” of the language [5]), later called only rules, and are often described with a meta syntax such as Augmented Backus-Naur Form (ABNF) [6] (used in this article) or EBNF [7, 8].

The phrasing “a grammar rule accepts/recognizes an input” later means that by the use of the rule the parser (or the lexer) accepts the input (recognizes it as valid according to the rule). The traditional way of working of the lexer, to work while at least one lexer grammar rule accepts the input data, has a drawback: the input characters that are not recognized by any lexer grammar rule cannot be used by the parser, because they are not valid according to the lexer grammar and respectively according to the lexer. In turn, the traditional way the parser works has a drawback because only the token name is used for recognition and the lexeme is just baggage that may not even be available in the token.

The main goal of the article is to propose an appropriate connection between the lexer and the parser, so that the unrecognized data from the lexer can then be used by the parser. The recognized tokens from the lexer (made of a name and a lexeme) must be transmitted entirely to the parser. The parser must be able to parse not only based on the name of the token but also on its lexeme. The way the parsing is performed must not significantly impair the speed of the parsing and must allow the usage of the lexemes with a selectable sensitivity to the capitalization of the characters in it. When the parser’s input contains all of the characters found in the input stream that enables it to send them subsequently to the builder module (for error message reporting and syntax structure building). This is particularly important when the different PM modules run on different dedicated threads of execution (on different processes and/or hardware), because the modules can operate without sharing common data structures.

To achieve this goal, the domain of the tokens created from the input characters must be divided into two types, and for each of them different formal rules to apply. One token type must only be created by the lexer, when a lexer grammar rule in the lexer grammar accepts the current input characters, and to consist of at least one character. The article will describe how the characters in this token type can be used for parsing based on a new type of grammar, case-sensitivity or insensitively. The other type of tokens must be created by the scanner or by the lexer (when no rule accepts the current input characters) and to consist of a single character. The formal definition of this token will allow its use to recognize character ranges from the module that receives the token.

Section 2 presents the related work in the process of recognizing a string of characters and briefly explains the Tunnel Parsing (TP) algorithm that uses the recognition approach presented in this article. Section 3 addresses the problem, to which the article is dedicated. Section 4 shows a solution to the problem. Section 5 gives the application of the proposed recognition approach in the TP algorithm. Section 6 focuses on the contributions of the article and its possible future development.

2. Related work

The traditional problem with the lexers is that the input that is not recognized by any lexer grammar rule cannot be used by the parser, because they are not valid (considered an error) according to the lexer grammar and respectively according to the lexer.

To overcome this problem an additional rule named *X* can be added to the lexer grammar, which recognizes all possible input characters that are no longer recognized by the existing grammar rules. Accordingly, *X* has the lowest priority over the other rules in order to be applied last. Thus, the parser will receive tokens with a name *X*, and will be able to process them according to the parsing grammar when none of the other lexer grammar rules recognize the current input characters. If a token named *X* is not expected then when it is received by the parser it will be processed as an error.

This article discusses a similar approach to solving the problem. The difference is that rule *X* is not required (which needs attention from the grammar developer) and this functionality is part of the formal definition of the lexer. We will call a lexer that works in this way a **Continuous Lexer** (CL).

CL works without detecting errors based on the lexer grammar, because all input characters in one way or another will be grouped into lexemes and will form tokens. The *X* rule is sometimes used to skip input characters directly [9], as some call them "meaningless characters" [10]. When this is done, the parser does not receive the missing characters, and therefore cannot perform logic based on them. Some systems process the lexemes to calculate the location of individual tokens in a text file (text line and column numbers, called **locator**). If this is done in the lexer [10, 11] then the omitted characters that the lexer does not send to the parser do not interfere with this calculation. However, this assumes that the lexer and the parser share their runtime data structures so that the parser can use the locator (calculated by the lexer) when needed, for example, to display error messages with it.

The lexer must not calculate the locators if the modules are considered separate and do not share common data structures. In this case, the calculation is best done by the builder. To do that, the builder must receive all of the lexemes to count accurately. Additionally, the symbol tables must not be created by the lexer at the time of the discovery of the lexeme [11], but by the builder upon receiving the tokens. The approach of gathering additional information in symbol tables is called "bookkeeping" in [4]. When each token is sent to the parser then the length of the token matters for technical reasons, for example limited memory. In some cases, the token can be assumed to be always short [11]. However, a multi-line comment in a given programming language source file, in the form of a lexeme, can be in thousands or more characters, therefore, special attention should be paid to this situation. It is possible that there are nested comments, which further suggests even longer lexemes. Some lexers have special processing of the nested comments [10, 12], because the nested comments are not regular in nature and cannot be recognized by a finite state machine [13], which is usually used by the lexer.

One possibility [14] is the lexer to be "context-aware". This means that the lexer outputs a set of token types, taking into account the current state of the parser, and if none of the currently acceptable tokens is recognized, an empty set is returned. This is used for the embedding of one language in another to avoid certain problems when mixing two separate pairs of lexer and parser. This approach makes the lexer as a subroutine of the parser [15]. A similar definition of context-aware is given in [4], where the lexer operates "directly", when it recognizes the set of token types according to the following input characters, and "indirectly" when checking whether the input characters are a specific token type.

The context-aware technique (called "lexical feedback" in [16] and "backdoor approach" in 17) prevents [16] the lexer and the parser to be in separate threads of execution [18]. It is possible for separate threads to be used, but it will probably be ineffective, due to the threads synchronization time. The synchronization takes time with the modern hardware and popular operating systems, but it is possible to imagine that due to the growing number of cores in the processors (the AMD Ryzen™ Threadripper™ 3990X has 64 cores and 128 threads), there may come a time when the synchronization will be with an acceptable speed in more scenarios. For example, [19] executes one lexical analyzer in different threads on different processors, as the speedup is linear (which can be for a variety of reasons, including the threads synchronization).

When the lexer and the parser are in different threads (potentially in different processes), special attention should be paid to when the tokens will be sent from the lexer to the parser. One possible implementation of this is the lexer to collect a certain number of tokens before sending them, but then the lexer can recognize more tokens than the parser would accept before finding an error. On the other hand, the parser will not do any work if the lexer fails to recognize the required number of tokens before sending them to the parser (in case of an error in the input data stream). This approach, known as buffering, is a standard way of modern software development, and has long been described [14]. There, the division of responsibilities for data recognition between the lexer and the parser, working in the form of augmented transition networks, is discussed.

Once the lexer converts the input characters into tokens, they will become the input data of the parser. The traditional way the parser works with tokens is to use them for recognition by automata [20], generated from the rules of the parser grammar. Going through an automaton transition, from one automaton state to another, is done with the use of the name of the tokens, and the lexeme is just a baggage that may not even be available in the token if a syntax tree is not generated. Another possibility is to use regular expressions of conditions for each automaton transition [21].

Different parser generators generate parsers for languages described differently. Sometimes the language is described by one grammar, which consists of the mixture of the lexer and the parser grammars, as the rules having capital letters (or start with a capital letter) are part of the lexer grammar, and the rules that have lowercase letters (or start with a lowercase letter) are part of the parsing grammar [22, 23]. Some parser generators require that everything that does not exist in the grammar as a terminal token, is a rule [10]. Another option for describing a language is to have no lexer grammar (but only a parser grammar) and therefore no lexical analysis [24]. The parsing of this kind of language is called “single-phase” parsing in [25]. There is also the claim that “scannerless parsing does not need to make any assumptions about the lexical syntax of a language and is therefore more generically applicable for language engineering” [26] and the extra ambiguity that occurs because of this, can be tackled by disambiguation filters. However, the term “scannerless” is hard to be accepted instead of “lexerless”, because the scanning is a primary machine operation [1, 27].

The regular expressions (with an initial idea in [20] based on [28], where are the first definitions of the two popular types of repetitions – star (from zero to infinity) [28], and plus (from one to infinity) [28]) are often used to describe lexical constructs instead of formal grammars. One of the first compilers of regular expressions to machine code is described in [29].

Some argue [30] that it is better to write a lexer by hand than to write regular expressions in the syntax of a lexer generator, because complex regular expressions are not easily understood by everyone. Others prefer BNF syntax over regular expressions [12]. When the lexer uses a deterministic finite state machine to recognize the lexemes [13], it can also be made minimal (which is often done in practice [31]) through Brzozowski’s algorithm [32]. This transformation has an exponential time (in the worst case) according to the number of initial automaton states. The resulting automaton is “much bigger” [12] than the original automaton. If an acyclic automaton is already available, it can be minimized with a linear algorithm [33] to occupy less memory and to be processed faster. There are other algorithms for minimization of automata [34]. The parser generators that make this transformation often resolve conflicts between rules according to the priority coming from their definitions’ order [35]. A conflict between rules exists when more than one rule can recognize a sequence of characters. It is also possible to create a lexer only when the rules have no conflicts [10].

It is a good idea to use an algorithm to inform the developer if there are lexical ambiguities in relation to the order of the lexemes in the tokens. The criterion for ambiguity in this case is for one token to be a prefix of another, which, if not taken care of during the development of the grammar, may lead to unexpected (wrong) behavior of the lexer [36].

2.1. Tunnel parsing

The TP algorithm is in use by the PMs generated by Tunnel Grammar Studio (TGS). The modules in these PMs are considered to be separated (operating in parallel in different threads) and are described in detail in [37]. The modules can be developed in different programming languages and work on different hardware. As defined so far the TP algorithm is applicable mainly to domain-specific languages [38].

All symbols that can be matched by the algorithm from a given automaton state directly or by recursively entering into the referenced rules are called **reachable**.

To create a PM that is based on the TP algorithm, the following five steps are performed [37].

1. Designing of automata – an automaton is created for each rule in the grammar.

2. Extraction of tunnels – for each start state, each state after a rule reference and each terminal state (an automaton state after a transition that matches a terminal symbol) of each automaton (all together called **key positions**), all transitions to the next reachable terminal states (or to the end of the rule) are collected into tunnels in a depth-first search manner. A **tunnel** is a group of operations for changing the internal state of the PM and the related commands for the syntax tree building.

3. Construction of routers – all reachable terminal states for all key positions in the automata are collected. They are stored sorted (by the value of the transition's terminal symbol that led to each terminal state) in a static read-only memory to speed up the search for a next state of the PM at runtime. The object that contains the sorted terminal states reachable from a given key position in the automata is called a **router** and each of its elements – a **path**.

4. Creation of a control layer – to control the execution of the PM, a set of objects is created, which are using the tunnels and the routers to form a control layer, with the specific functionality. Each control object can be in one of several control states. The control objects signify the information to "where" in the automata the PM has reached, and the control states – "which" operations must be performed.

5. Parsing – a direct parsing is performed, or a parser is generated to a source code for a target programming language that can be integrated in other software tools. In TGS there is a visual debugger that performs the parsing and builds a syntax tree directly, in forward and backward steps, for a given grammar and an input.

In addition to the possibilities to strictly define for a PM whether there is a lexer or not, a third option is possible: the recognition language to be described by two separate grammars, lexer and parser grammars [39], as the capitalization of the names of the rules in the grammars to be without special significance. Then, the functionality that the not recognized characters by the lexer's grammar are converted into tokens (as how a CL works, which is described in [1]), allows the lexer grammar to possibly be without rules. That will practically make the recognition of the input data for the given language only on the basis of the parsing grammar [39]. It effectively combines the previous two strict options, because the number of grammars becomes the choice of the language developer, not a strict definition of how the PM works.

3. Problem

The problem addressed in the article is to make an appropriate connection between the lexer and the parser, in such a way that the parser performs the parsing not only by using the tokens names, but also their lexemes, in a way that does not significantly change the speed of the parsing, but adds useful functionality.

The meta syntax ABNF (Internet standard #68) is widely used to describe the syntax of various popular Internet protocols, standards, and data structures [40, 41]. The terminal grammar elements in the ABNF standard are described by **terminal values**. By definition, each terminal value consists of characters, as the matching is defined as not sensitive to the capitalization of the individual characters in the value [7] (later called only **sensitivity**). Due to the need to make the meta syntax more expressive, it has been upgraded [42] with a richer syntax that enables the writing of grammars with explicitly selected sensitivity for each terminal value. The grammars described in the meta syntax of ABNF are context-free, although element repetitions and the selectable sensitivity give more expressive power in this direction than BNF [43] and the variants of EBNF [7, 8]. Some tools have a setting that is used during the generation of the PM, whether the parser will work sensitively or not regarding the characters found in the input [10].

Despite its great expressive power, the ABNF standard does not have a syntax that enables the writing of grammar rules that accept tokens based on their names and lexemes. In order for this functionality to be possible, an addition to the standard is needed. The addition (which is a better choice than creating a completely new meta syntax) shall be minimal so as not to create too many difficulties in its use.

4. Solution

We propose a solution by assuming that the PM is generated automatically, the architecture of the PM is according to [1], and the parsing algorithm is TP. The section describes the working with tokens in a way that is more limited in functionality than the recognition by regular expressions of conditions [21], but which in turn preserves the linear rate of parsing while offering sensitive and insensitive matching of the lexemes in the tokens. All relevant definitions from [1] are transferred into this article and are formally defined. The formal definitions of an advanced grammar follow with the note that a character might not be a Unicode code point, because the proposal in [1] that the alphabet of the parser is made of characters (that are Unicode code points) might not be accepted by a given parsing machine architecture.

4.1. Formal definitions

A module is classified as an **emitter**, when it emits tokens to another module that in turn is classified as a **receiver**. For two modules, one emitter and one receiver, the following is defined:

- Φ is the alphabet set of **characters** that the emitter uses internally.
- G is the **grammar** that is used by the emitter.
- Θ is the set of **rules names** in G. If $\nexists$ G then $\Theta = \emptyset$, where $\emptyset$ is an empty set.
- A **lexeme** is a sequence $(e_n)_{n=0}^{\infty}$, where $e_n \in \Phi$.
- An **attribute** is a tuple (l, v), where l is a label, v – a value, as their meanings and domains are defined separately per attribute.
- A **token** is a tuple (t, n, e, a), where $t \in$ {*t*-character, *t*-sequence, *t*-limit, *t*-eof} is the token’s type, n – the token’s name (its domain is denoted by the token type later), e – the token’s lexeme, and a – the token’s attributes set.
- A token with a *t*-character type is the tuple (*t*-character, φ, (φ), a), where $\varphi \in \Phi$.
- $|e|$ denotes the number of elements in e.

- A token with a *t*-sequence type is the tuple (*t*-sequence, *c*, *e*, *a*), where $c \in \Theta$, *e* is a lexeme, and $|e| > 0$. The unbounded length of *e* makes the *t*-sequence tokens elements of an unbounded set.
- A token with a *t*-limit type is the tuple (*t*-limit, *h*, *e*, *a*), where *h* is a sequence $(h_n)_{n=1}^{|\Theta|}$, $h_n \in \Theta$, and $|e| > 0$.
- A token with a *t*-eof type is the tuple (*t*-eof, null, (), *a*), where null means that no value is available.
- $\Phi \neq \emptyset$ is assumed from now on, because if $\Phi = \emptyset$ then the emitter would not be able to emit any of the hereby defined token types, except *t*-eof.

After the tokens are emitted from the emitter they will be received by the receiver that has its own grammar, defined as follows.

- An **advanced grammar** is the tuple $(C, N, \Sigma, \Omega, R, S)$, where:
    - *C* is a finite set of **categories**, where a category is a rule name that could be in the emitter's grammar as an actual rule name;
    - *N* is a non-empty and finite set of **rule names**;
    - Σ is a finite set of **characters**;
    - Ω is a finite set of **advanced symbols**, defined as follows:
        - Ω_c is a set of character symbols (called ***s*-character** for short);
        - Ω_p is a set of phrase symbols (called ***s*-phrase** for short);
        - Ω_e is a set of eof symbols (called ***s*-eof** for short);
        - $\Omega_c \cap \Omega_p = \emptyset$, $\Omega_c \cap \Omega_e = \emptyset$, and $\Omega_p \cap \Omega_e = \emptyset$;
        - $\Omega = \Omega_c \cup \Omega_p \cup \Omega_e$.
    - *R* is a non-empty and finite set of **all grammar rules**; the rules of a not advanced grammar use an alphabet of Σ, but the rules in *R* use an alphabet of Ω instead; the form of the rules is defined later;
    - *S* is a non-empty set of all rules that each of them could be a **starting rule**, $S \subseteq R$.
- A set $M = \{M_s, M_i\}$, where M_s indicates case-sensitive matching (binary equality operation), and M_i – case-insensitive matching.
- An *s*-character is a tuple (σ_f, σ_t, m), where $\sigma_f \in \Sigma$, $\sigma_t \in \Sigma$, $\sigma_f \leq \sigma_t$, and $m \in M$. The name of a *t*-character input token matches with the *s*-character's character range $[\sigma_f .. \sigma_t]$ sensitively when $m = M_s$ and insensitively when $m = M_i$.
- A **phrase** is a finite sequence $(e_n)_{n\in\mathbb{N}}$, where $e_n \in \Sigma$.
- An *s*-phrase is a tuple (*c*, *m*, *p*), where category $c \in C$, $m \in M$, and *p* is a phrase. During the parsing, the name of a *t*-sequence input token matches with the category of the *s*-phase. The *t*-sequence's lexeme matches (character by character) with the *s*-phrase's phrase sensitively when $m = M_s$ and insensitively when $m = M_i$. If $|p| = 0$ then only the *t*-sequence's name and the *s*-phrase's category match (the *t*-sequence's lexeme is accepted as is). If $|p| = 0$ then the element is called ***s*-any** else each *s*-phrase element that has $m = M_s$ is called an ***s*-sensitive** element and when $m = M_i$ – ***s*-insensitive**.

The unadvanced context-free, context-sensitive and unrestricted grammars [5] are special cases of the advanced context-free, advanced context-sensitive and advanced unrestricted grammars respectively when $C = \Omega_\mathrm{p} = \Omega_\mathrm{e} = \emptyset$, $|S| = 1$, and each *s*-character has $\sigma_f = \sigma_t$ and $m = M_\mathrm{s}$. The different grammar types are advanced as follows:

- The **advanced context-free grammar** has its rules in the form: $r \rightarrow \alpha$, where $r \in N$, $\alpha \in (N \cup \Omega)^*$, and the asterisk represents the Kleene star operation. The previously used context-free grammars by the TP algorithm from now on change to advanced context-free grammars.
- The **advanced context-sensitive grammar** has its rules in the form: $\alpha r\beta \rightarrow \alpha q\beta$, where $\alpha,\beta \in (N \cup \Omega)^*$, $r \in N$, $q \in (N \cup \Omega)^+$, and the plus represents the Kleene plus operation.
- The **advanced unrestricted grammar** has its rules in the form: $\alpha \rightarrow \beta$, where $\alpha \in (N \cup \Omega)^+$, and $\beta \in (N \cup \Omega)^*$.

The following conclusions can be made:

- One advanced grammar generates a set of languages, one per starting rule, each with its own set of strings.
- If $\Theta \neq \emptyset$ and $\Theta \subseteq C$ then all of the received *t*-sequence tokens can be recognized by the receiver.
- If $\Theta \neq \emptyset$ and $\Theta \cap C = \emptyset$ then none of the received *t*-sequence tokens can be recognized by the receiver. Upon receiving them, the receiver will simply account them as an input error.
- If $\Theta \neq \emptyset$ and $\Theta - (\Theta \cap C) \neq \emptyset$ then some *t*-sequence tokens can be recognized, but not all.
- If $\Phi \subseteq \Sigma$ then all of the received characters (in the *t*-character tokens and in each lexeme inside the *t*-sequence tokens), can be recognized by the receiver.
- If $\Phi \cap \Sigma = \emptyset$ then only the *t*-eof tokens can be recognized by the receiver.
- If $\Phi - (\Phi \cap \Sigma) \neq \emptyset$ then some tokens can be recognized, but not all.
- A **perfect configuration** (the receiver expects exactly the tokens that could possibly be emitted from the emitter) is when $\Theta = C$ and $\Phi = \Sigma$.
- A **flawless configuration** (all tokens emitted from the emitter can be used for matching by the receiver; the receiver expects a wider range of tokens and/or characters) between the emitter and the receiver is when $(C - \Theta) \cup (\Sigma - \Phi) \neq \emptyset$. Such a configuration is assumed from now on.

The advanced grammar is a finite grammar with a finite set of s-phrases (not as powerful as a grammar with infinite terminal symbols), but still allows the matching of a token's lexeme with an explicitly chosen lexeme content (case-sensitively or insensitively) or a lexeme with any content.

In relation to the formal definitions some informal notes can be made:

- A receiver module should not use advanced grammars with $\Omega_\mathrm{p} \neq \emptyset$ when it directly receives its input from a scanner (the scanner has no grammar ($\Theta = \emptyset$ when $\nexists\, G$) and the expected categories in the receiver's grammar ($C \neq \emptyset$ when $\Omega_\mathrm{p} \neq \emptyset$) will never be matched, because no *t*-sequence token will ever be emitted from the scanner).

- If there is more then one lexer, one after another, then the lexers after the first might use advanced grammars, because the previous lexer might output *t*-sequence tokens.
- The same is valid for the parser grammar – it should not be advanced when there are no lexers before the parser module, but only a scanner.
- One module that uses an advanced grammar must know its previous module's grammar rules set and alphabet, to be able to handle the received tokens.

4.2. Advanced grammar meta syntax

The ABNF standard adequately describes the syntax for grammars that accept characters, and there is no need for a change in this direction. The addition to recognize *t*-sequence tokens is that a new ABNF element must be introduced, with which to be possible the writing of grammar elements that will match to this type of tokens. For the purpose of this article, the terminal values (defined in the ABNF standard) in a parsing grammar will be a string of *s*-character tokens (and according to the formal definitions match to the names of the tokens received by the receiver, when the tokens are of *t*-character type). The base ABNF standard is used to describe context-free grammars where the matching is insensitive by default. This means that $m = M_{\mathrm{i}}$ for each element in an ABNF grammar. The ABNF case-sensitive upgrade of the base standard applies to the matching of the *t*-character tokens, by allowing the grammar to have a matching type ($m \in M$) explicitly selected for each *s*-character element.

The matching of a *t*-sequence token must be possible only with its name, and optionally with its lexeme. The addition to the standard [6] shown in Fig. 1 is sufficiently expressive for this purpose (the case-sensitive string support from [42] is required). The addition also contains a way to describe an *s*-eof token with the syntax in rule *eof-val*.

```
element         =/ phrase-val / eof-val
phrase-val      = "{" rulename 0*1 phrase-content "}"
phrase-content  = "," (char-val / num-val / prose-val)
eof-val         = "{" %s"$EOF" "}"
```

Fig. 1. An advanced grammar meta syntax as an ABNF meta syntax addition

Rule *phrase-val* in Fig. 1 describes the syntax of one *s*-phrase grammar element. If a hypothetical language is described with lexer and parser grammars then the parser grammar can reference the names (the ABNF standard has the *rulename* rule) of the *t*-sequence tokens (received from the lexer) and possibly the lexemes in the tokens (with the syntax of rule *phrase-content* in Fig. 1).

```
word = 1*(%x41-5A / %x61-7A)
```

Fig. 2. Example of a lexer grammar

```
document     = {word} SP some-hello SP large-world "!"
some-hello   = {word, %i"Hello"}
large-world  = {word, %s"WORLD"}
```

Fig. 3. Example of a parser grammar

The lexer grammar in Fig. 2 and the parser grammar in Fig. 3 describe an exemplary language. The lexer grammar has a rule with a name *word*, which accepts one or more lowercase (ASCII codes in hexadecimal format from 61 to 7A inclusive) or uppercase (ASCII codes in hexadecimal format from 41 to 5A inclusive) latin characters. For each input sequence of *t*-characters that the lexer receives from the scanner one *t*-sequence token will be generated. The parsing grammar has a start rule *document*, which first accepts a *t*-sequence token named *word*, then a space (*s*-character advanced symbol that matches with a *t*-character token with name *" "* (rule *SP* is defined in ABNF [6] and recognizes exactly one space – hexadecimal format 20)), followed by an insensitively matched *t*-sequence token with a name *word* and lexeme *"Hello"*, then another space, followed by a sensitively matched lexeme *"WORLD"*, followed by an exclamation mark (that in order to be matched the lexer must emit a *t*-character token with a name *"!"*, as this will happen when the input character is not recognized by rule *word* in the lexer grammar, in the same way as the two spaces).

4.3. Conflicts

As a result of defining the whole recognition process not only for terminal symbols but for advanced symbols, in addition to the well-known conflicts between terminal symbols (simultaneously reachable during execution), additional conflicts are formed in the parser grammar – conflicts between *s*-phrase elements, as well as conflicts between *s*-eof tokens. Possible conflicts are visualized with a Venn diagram in Fig. 4 and are as follows:

- two *s*-eof elements are in conflict with each other;
- two *s*-sensitive elements are in conflict when they have the same names and exactly the same phrase;
- two *s*-insensitive elements are in conflict when they have the same names and their phrases are the same if the capitalization is ignored;
- two *s*-any elements are in conflict when they have the same names;
- *s*-sensitive and *s*-insensitive elements are in conflict when they have the same names and the phrase in the *s*-sensitive element differs only in the capitalization of the characters from the phrase in the *s*-insensitive element;
- *s*-sensitive and *s*-any elements are in conflict when they have the same names;
- *s*-insensitive and *s*-any elements are in conflict when they have the same names;
- *s*-sensitive and *s*-insensitive and *s*-any elements are in conflict when they all have the same names, and the phrase in the *s*-sensitive element differs only in the capitalization of the letters from the phrase in the *s*-insensitive element.

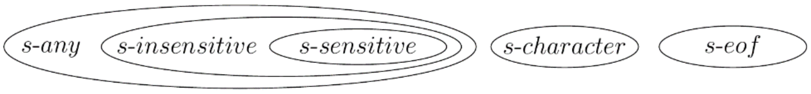

Fig. 4. Possible conflicts in an advanced grammar

Parser grammar with different conflicts is shown in Fig. 5. If a PM has a parser (using the grammar of Fig. 5) and a lexer (using the grammar of Fig. 2), then during parsing the following conflicts are possible:

- elements {*word*}, {*word, %i“json”*} and {*word, %s“JSON”*} are in conflict for a *t*-sequence token with a name *word* and a lexeme *“JSON”* (capital letters);
- elements {*word*} and {*word, %i“json”*} are in conflict for a *t*-sequence token with a name *word* and a lexeme *“JsOn”* (for the different capitalizations of the letters);
- the *s*-character elements (concatenations from 4 to 7 inclusive in Fig. 5) will never create a conflict during parsing, because the lexer will emit *t*-sequence tokens (with names *word*) rather than individual *t*-character tokens.

If there is no lexer (the grammar of Fig. 2 is not used for lexing or the lexer grammar is empty), then each input character will be sent directly from the scanner to the parser. The tokens will be of type *t*-character with names and lexemes that consist of the respective characters. Then the following conflicts are possible:

- elements *%i“json”*, *%s“JSON”*, *%x4A* (the hexadecimal code of capital letter *“J”*) and the character range *%x41-5A* (capital letters from *“A”* to *“Z”* inclusive) are in conflict for a capital letter *“J”*;
- the *s*-phrases (concatenations from 1 to 3 inclusive in Fig. 5) will never create a conflict during parsing, because there is no lexer that emits *t*-sequence tokens.

```
main  = {word} / {word, %i"json"} / {word, %s"JSON"}
          / %i"json" / %s"JSON" / %x4A / %x41-5A
```

Fig. 5. An advanced grammar with conflicts

4.4. Phrase state machine

A Phrase State Machine (PSM) is an abstract machine that can be in exactly one (current) state of a finite number of states at any given time. The information needed to change the current state, from one state to another, is stored in a transition. To be able a PSM to use a transition, to change its current state, an input is needed (a non-empty sequence of characters). PSM is a tuple $(\Phi, \Sigma, M, Q, \delta, F, q_0)$, where:

- Φ is the alphabet set of characters that the emitter uses internally (denoted previously);
- Σ is the alphabet of the advanced grammar used by the receiver (denoted previously);
- M is the matching set of the advanced grammar (denoted previously);
- Q is a non empty and finite set of **states**;
- δ is a set of **transitions** in the form: $\langle \sigma, q_s \rightarrow q_d, m \rangle$, where $\sigma \in \Sigma$ is a character that has to match exactly to the current input character, $q_s \in Q$ is the source state (the current state before the matching of σ), $q_d \in Q$ is the destination state (the current state after the matching of σ), and $m \in M$ is the reason for which the transition is created (by the use of a sensitive (then $m = M_s$) or an insensitive (then $m = M_i$) *s*-phrase, explained later);

- F is non empty set of **final states**, $F \subseteq Q$, in the form: $f(q) = [n, m]$, where $f(q) \in F$, $q \in Q$, n is a natural number (an index for the recognized lexeme), and $m \in M$;
- q_0 is the **starting state**, $q_0 \in Q$, and $f(q_0) \notin F$.

Invariants for the PSM are as follows:

a) there are no sequence of transitions that lead from one state to itself;

b) no two transitions have the same σ and q_s values;

c) if there are two transitions with the same q_s values, both have $m = M_i$, and their σ characters differ only in their character case variants then both of the transitions must have the same q_d;

d) every sequence of transitions from q_0 to another state is made of zero or more transitions that have $m = M_s$ followed by zero or more transitions that have $m = M_i$.

```
$i$ is the current input character
if $\nexists i$ then
    if $f(q_c) \in F$ then return $f(q_c)$
    else return $null$
$l \leftarrow$ LOWERCASE($i$), $u \leftarrow$ UPPERCASE($i$)
if $l = u$ then
    if $\neg$(find $L$ in $\delta$ where $q_s = q_c \wedge \sigma = l$) then return $null$
    $q_c \leftarrow L.q_d$
else
    if $\neg$(find $L$ in $\delta$ where $q_s = q_c \wedge \sigma = l$) then return $null$
    if $\neg$(find $U$ in $\delta$ where $q_s = q_c \wedge \sigma = u$) then return $null$
    if $L.m = M_i \wedge U.m = M_i$ then $q_c \leftarrow L.q_d$    ▷ invariant $c$ states that $L.q_d = U.q_d$
    else if $L.m = M_i$ then $q_c \leftarrow L.q_d$
    else if $U.m = M_i$ then $q_c \leftarrow U.q_d$
    else $q_c \leftarrow L.q_d$    ▷ prefer the lowercase transition when $L.m = M_s \wedge U.m = M_s$
▷ move after the current input character and repeat
```

Fig. 6. Insensitive phrase state machine execution

A PSM is deterministically processing its input one character at a time. After each change of its current state, the next input character, if any, becomes the current input character. The current state of a PSM is denoted as q_c and is changed on the basis of the PMS's input by the use of its transitions. The execution starts with $q_c = q_0$. The PSM has two different modes of execution, as follows:

- **sensitive** – executes as a traditional finite state machine. After all of the PSM's input characters are used, the result is $f(q_c)$ when $f(q_c) \in F$ and *null* otherwise;
- **insensitive** – executes similarly to the sensitive mode, but by taking into account the different case variants of the PSM's input characters. This execution is formally described with the pseudocode in Fig. 6, where the *uppercase* and *lowercase* functions return the respective case variants of the character (or the character itself, if it has only one variant). The notation *a.b* means that *b* is a member of *a*.

4.5. Phrase to lexeme matching

From a practical point of view, if one compares the characters in a token with the expected characters from the reachable *s*-phrase elements in a router, this will take time in the worst case $O(p*\max(k, n))$, where p is the number of *s*-phrase elements in the advanced grammar ($p = |\Omega_p|$) involved in the search, k is the length of the longest phrase in any *s*-phrase, n is the length of the lexeme that is used for the match, once it is recognized (which is a finite number at runtime, and might not be predictable in advance), and max is a function that returns the greater of two arguments.

The following is a description of an algorithm that allows the use of the lexeme e for time $O(1)$, as for each created *t*-sequence token, $|e|$ operations are required. This means that the run time is $O(1)$ for each individual character of the input data, or $O(n)$ operations in the worst case, with n being the number of input characters (if known). Because the lexeme is used as a functional part of the token (it is used for recognition) and it is unlimited in size, this changes the accepted definition that the input of the parser is made of elements from a finite set.

To have the functionality described up the following has to be done during the compilation of the PM:

- All *s*-phrase elements from the parser grammar are placed in two lists, based on the *s*-phrase matching types: a list with *s*-sensitive elements (***s*-list**), and a list with *s*-insensitive elements (***i*-list**). These lists could be sorted in any way.
- A PSM (as the one shown in Fig. 7) is created from these two lists, iteratively, as follows:
    - create the single starting state q_0;
    - merge (create PSM states and transitions) each *s*-list phrase in turn, one character at a time; maintain a current q_c state during the merging of each phrase, starting from $q_c = q_0$. For the current character $\sigma \in \Sigma$ in the current phrase, any state q_d, and transition $a = \langle\sigma, q_c \rightarrow q_d, M_s\rangle$: if $a \notin \delta$, then create a new state q_d and add the missing transition a to δ; this ensures that invariant b in the formal definitions of the PSM is maintained; continue the merging from the next state ($q_c = q_d$) for the next phrase's character. After all characters are merged and the current state is not a final state ($f(q_c) \notin F$) then q_c must be made a final state for the PSM, and a unique index i be set to it: add $f(q_c) = [M_s, i]$ to F;
    - merge all *i*-list phrases in turn, one character at a time, by starting from q_0, but simultaneously with a lower-case and upper-case variants for each phrase character, in the same way as for the *s*-list phrases, but use M_i instead of M_s; the new states that are created (or reused) in this step must be on at most one new transition sequence, as for example are the gray states in Fig. 7; this ensures that invariant d in the formal definitions of the PSM is maintained;
    - each unique index in a final state is called **phrase index**, and when the final state has $m=M_s$ then the phrase index will be called **sensitive phrase index**.

- Create a one based array of natural numbers (called later **filter array**). Iterate again the *s*-list elements and for each: a) execute the PSM insensitively with an input equal to the phrase of the *s*-list element, and b) push into the filter array the phrase index from the result of the PSM execution. For the grammar in Fig. 5, the PSM is in Fig. 7 and the filter array have one element at array index 1 with a value of 2.

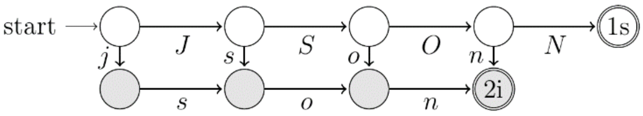

Fig. 7. Phrase state machine created from the grammar in Fig. 5

During the execution of the PM in each token of type *t-sequence* is added an attribute labeled “phrase index” and a value. This value is equal to the result of the execution of the built PSM sensitively with an input equal to the lexeme in the *t*-sequence token. The attribute can be added by the lexer creating the token or by the parser at the time the tokens are received by the lexer.

When the parser module is parsing, it uses the *t*-sequence token’s name and its associated phrase index value. Although this value is used, which belongs to a set, upper bounded by the number of unique *s*-phrase elements, the possible tokens are an infinite number, due to the infinite number of lexemes that can be inside. In other words, the parser accepts tokens by name and with an infinite lexeme length, recognizing only a finite number of them, as they are explicitly written in the parser grammar as *s*-phrases with their adjacent phrase, and tokens with any lexeme, when the phrase in an *s*-phrase element has zero characters. However, this is only a function of the parser, how to process the infinite tokens, and is not necessarily known to the lexer when it is creating the *t*-sequence tokens.

5. Integration into the Tunnel Parsing Algorithm (TP Algorithm)

To improve the TP Algorithm with the functionality described in this article, the routers used by the algorithm must be able to search not only for *t*-character tokens, but also for *t*-sequence and *t*-eof tokens. For this purpose, four new router types are added to the already existing single type of router – the one for *s*-characters (called *r*-character in this article). The types of routers are as follows:

- ***r*-character** – this router has only paths for the reachable *s*-character elements. For this reason it accepts a search only for *t*-character tokens that could eventually match with some *s*-character element;
- ***r*-sensitive** – this router has only paths for the reachable sensitive *s*-phrase elements. The sensitive phrase index of each *s*-phrase element is calculated by executing the PSM sensitively and it is used as a sorting criteria of the paths in the router; the router accepts searches for a *t*-sequence token that has a sensitive phrase index; if there is a reachable *s*-phrase element then the particular control state (of the PM’s control layer) in the found path is the search result;
- ***r*-insensitive** – this router has only paths for the reachable insensitive *s*-phrase elements. The phrase index of each element is calculated by executing the

PSM insensitively and it is used as a sorting criteria of the paths in the router; this router accepts a search for a *t*-sequence token by using its name and a phrase index; if the token used for the search has a sensitive phrase index then the index is first filtered through the filter array; the search in the *r*-insensitive router is performed after the search in the *r*-sensitive, if no control state is found there;

- ***r*-any** – this router has paths for all of the reachable *s*-phrase elements, sorted only by their name; the router accepts a search only for a token's name, without the use of the phrase index; the search in this router is performed last if no control state is found in the *r*-sensitive and *r*-insensitive routers;
- ***r*-eof** – this router accepts search only for *t*-eof tokens, because it has paths only for the reachable *s*-eof elements.

The number of searches in routers from a given position in the automata created from the parser grammar, for one *t*-sequence token can be from one to three (one search in *r*-insensitive, *r*-sensitive and *r*-any routers in the worst case), as this is a linear increase only for those places in the grammar where there is such a need and does not change the linearity of the TP Algorithm. The search for *t*-character and *t*-eof tokens is only in their respective routers: *r*-character and *r*-eof. For those languages that are described by only one parsing grammar that is not advanced, the search during the execution of the parsing machine will be only once per reachable advanced symbol, because there will be only *r*-character routers.

To search for a path in a router (based on a token) that accepts *s*-phrases (these are *r*-sensitive, *r*-insensitive, and *r*-any), the phrase value index is used (calculated from the token's lexeme), and there is no direct comparison of the characters in the lexeme with the phrases characters during the execution of the parsing algorithm. The size of the tokens has a constant influence on the execution time of the parser as a module of the PM, and should not increase the complexity of other parsing algorithms that use advanced grammars and PSM as defined here. If the lexer creates the tokens attributes with the phrase value index, then the length of the lexemes adds no overhead on the parser during their usage for the search of a path in a router, because the search uses the phrase value index directly, and PSM will not be used by the parser. However, this means that the lexer must know the set of *s*-phrase elements of the parser grammar in order to create and use the PSM.

When the tokens that a parser recognizes are described as in Section 4, it becomes possible:

- Only one grammar to describe a language, when the lexer grammar is considered empty and the scanner creates one *t*-character token for each character. For example, the grammar of the JavaScript Object Notation data exchange format [40] can be used directly for parsing from Tunnel Grammar Studio [39], which generates parsers to program source code from ABNF grammars that process the input data in the manner described in this article.
- The parser has access to the input characters. That enables the writing of character ranges, as defined in the ABNF standard, directly in the parsing grammar: for example, the declaration %×30-39 in a parser grammar, declares an *s*-character element that can match with the following tokens: (*t*-character, *"0"*, (*"0"*), ()), (*t*-character, *"1"*, (*"1"*), ()), ... , and (*t*-character, *"9"*, (*"9"*), ()). In a similar way the

declaration *%i“A”* declares a *s*-character element that can match with tokens: (*t*-character, *“a”*, (*“a”*), ()) and (*t*-character, *“A”*, (*“A”*), ()).

- The grammar developer may completely ignore the order of the rules in the parsing grammar, because their arrangement does not affect the language.
- Thanks to the way tokens are used, languages that are already described with one grammar can be used directly, and there is no need to extract the terminal symbols in another (lexer) grammar, which can be a very time consuming and error prone process. For large grammars, the developer may be forced to develop the parser by hand, rather than using a parser generator, because with a large number of changes, it becomes more likely for the language to be changed by the developer by mistake without this to be immediately evident.

6. Conclusion

This article has described an approach of how a PM processes an input. A minimal addition to the ABNF standard has been proposed that enables the use of the presented approach. It is shown that a formal language can be described with only a parser grammar and optionally (by the opinion of the grammar developer) with an additional lexer grammar. The tokens received by the parser can be matched not only by their name, but by their lexemes’ characters (case sensitively or insensitively) as well. This is done by having more than one type of token. The *t*-character token has a name made from a single character and is emitted by a scanner or by the lexer (because there is no full lexeme to be matched based on the lexer grammar). The *t*-sequence token has a name from the lexer grammar rule used to recognize the token’s lexeme. Its lexeme can be subsequently matched in the parser grammar by the addition to the ABNF meta syntax in this article. If a specific grammar formalism can be transformed to an advanced CFG then the TP algorithm can perform on the basis of this transformation, and by this support the source formalism indirectly.

The main contributions of the article are six.

- A formal description of an advanced grammar is made and is used for formal advancement of the three grammar types: the context-free, the context-sensitive, and the unrestricted grammars. The advanced grammar formalism, as defined in the article, might be used for an advancement of other grammar types as well.
- The advanced phrase symbol is presented as a part of an advanced grammar – a new type of symbol that matches with the token’s lexeme case sensitively or insensitively.
- Character range elements can be directly written into a parser’s grammar, because the lexer’s input character sequences that are not accepted by any lexer’s grammar rule are directly sent to the parser. This means that the lexer always (if its technical limits are not exceeded) emits tokens to the parser.
- Eight new conflict cases between different combinations of advanced symbols, inside an advanced grammar, are shown. They are an addition to the popular conflicts between terminal symbols that are characters.
- A new specially designed PSM is formally defined, its runtime pseudo code is given, and its iterative building algorithm on the base of an advanced grammar is

described in steps. Each *t*-sequence's lexeme is classified by the PSM (built for the particular advanced grammar), and is used with a $O(1)$ time during the *t*-sequence to phrase symbol matching (the same amount of time as a character to character matching). The classification by the PSM takes at most $O(k)$ time, where k is the lexeme's length. The classification is done only one time per *t*-sequence type of token, because the result of the classification is stored inside the token's attributes. That makes the total runtime overhead of the PSM classification $O(n)$ for an input of length n. If the grammar used by the TP algorithm is not advanced (there are no phrase symbols, but only character symbols), then there will be no overhead at all, because no classification will ever be performed.

- New routers for the Tunnel parsing algorithm are defined that enable it to use an advanced context-free grammar.

As a future extension of the presented approach we shall explore the possibility that after the token's lexeme is fully used the *t*-sequence token to disassemble in the module's input to *t*-character tokens, one per each lexeme character, and then they to be subsequently used for matching with the reachable character symbols. That means that the token to be divisible, instead of indivisible [44]. However, this kind of change shall be carefully examined, because it impacts the formally defined advanced grammars determinism.

Contribution. Nikolay Handzhiyski developed the concept, the theoretical formalism (based on his previously existing software implementation in Tunnel Grammar Studio) and the initial draft under the thorough supervision, encouragement and critical feedback of Elena Somova. Both authors performed substantial revisions, verified the formal definitions, and contributed to the final draft.

References

1. Handzhiyski, N., E. Somova. A Parsing Machine Architecture Encapsulating Different Parsing Approaches. – International Journal on Information Technologies and Security (IJITS), Vol. **13**, 2021, No 3, pp. 27-38.
2. Unicode Standard.
https://www.unicode.org/
3. Deremer, F. L. Practical Translators for LR(K) Languages. Massachusetts Institute of Technology, USA, 1969.
4. Aho, A., J. Ullman. The Theory of Parsing, Translation, and Compiling. Prentice-Hall, USA, 1972.
5. Chomsky, N. Three Models for the Description of Language. – IRE Transactions on Information Theory, Vol. **2**, 1956, No 3, pp. 113-124.
6. D. Crocker, P. Overell, Eds. ABNF RFC 5234. Network Working Group, 2008.
7. Wirth, N. What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? – Communications of the ACM, Vol. **20**, 1977, No 11, pp. 822-823.
8. ISO/IEC 14977:1996(E) Information Technology – Syntactic Metalanguage – Extended BNF.
http://standards.iso.org/ittf/PubliclyAvailableStandards/
9. Dos Reis, A. J. Compiler Construction Using Java, JavaCC, and Yacc. Wiley-IEEE Computer Society Pr, 2011.
10. Moessenboeck, H. Coco/R – A Generator for Fast Compiler Front Ends. Zurich, ETH, 1990.
11. Aho, A. V., M. S. Lam, R. Sethi, J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing, Co., USA, 2006.
12. Szafron, D., R. Ng. LexAGen: An Interactive Incremental Scanner Generator. – Software: Practice and Experience, Vol. **20**, 1990, No 5, pp. 459-483.

13. Rabin, M. O., D. S. Scott. Finite Automata and Their Decision Problems. – IBM Journal of Research and Development, Vol. **3**, 1959, No 2, pp. 114-125.
14. Van Wyk, E. R., A. C. Schwerdfeger. Context-Aware Scanning for Parsing Extensible Languages. – In: Proc. of 6th International Conference on Generative Programming and Component Engineering, October 2007, pp. 63-72.
15. Aho, A. V., R. Sethi, J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing, Co., USA, 1986.
16. Aycock, J., R. Horspool. Schrödinger's Token. – Software: Practice and Experience, Vol. **31**, 2001, No 8, pp. 803-814.
17. Johnson, S. C. YACC: Yet Another Compiler-Compiler. CiteSeer, 2001.
18. Saltzer, J. H. Traffic Control in a Multiplexed Computer System. Massachusetts Institute of Technology, USA, 1966.
19. Barve, A., B. K. Joshi. Parallel Lexical Analysis of Multiple Files on Multi-Core Machines. – International Journal of Computer Applications, Vol. **96**, 2014, No 16, pp. 22-24.
20. Kleene, S. Representation of Events in Nerve Nets and Finite Automata. – Annals of Mathematics Studies, Vol. **34**, 1956, pp. 3-41.
21. Rus, T., T. Halverson. A Language Independent Scanner Generator. CiteSeer, 1999, pp. 1-35.
22. ANTLR.
https://www.antlr.org/
23. JavaCC.
https://javacc.github.io/javacc
24. Visser, E. Scannerless Generalized-LR Parsing. University of Amsterdam, Netherlands, 1997.
25. Afroozeh, A., A. Izmaylova. One Parser to Rule Them All. – In: ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), October 2015, Chicago, USA, pp. 151-170.
26. Van den Brand, M., J. Scheerder, J. Vinju, E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. – In: Proc. of 11th International Conference of Compiler Construction, April 2002, pp. 1-17.
27. Turing, A. On Computable Numbers, with an Application to the Entscheidungs Problem. – In: Proc. of the London Mathematical Society. Vol. **41**. 1937, pp. 230-265.
28. Kleene, S. Representation of Events in Nerve Nets and Finite Automata. US Air Forse, USA, 1951.
29. Thompson, K. Programming Techniques: Regular Expression Search Algorithm. – Communications of the ACM, Vol. **11**, 1968, No 6, pp. 419-422.
30. Kühl, B., A.-T. Schreiner. Objects for Lexical Analysis. – ACM SIGPLAN Notices, Vol. **37**, 2002, No 2, pp. 45-52.
31. Saraiva, J. HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. – In: Proc. of ACM Workshop on Functional and Declarative Programming in Education (FDPE/PLI'02), October 2002.
32. Brzozowski, J. A. Canonical Regular Expressions and Minimal State Graphs for Definite Events. – In: Proc. of Symposium on Mathematical Theory of Automata, MRI Symposia Series. Vol. **12**. 1963, pp. 529-561.
33. Revuz, D. Minimisation of Acyclic Deterministic Automata in Linear Time. – Theoretical Computer Science, Vol. **92**, 1992, No 1, pp. 181-189.
34. Berstel, J., L. Boasson, O. Carton, I. Fagnot. Minimization of Automata. – In: Automata: From Mathematics to Applications. European Mathematical Society, 2006.
35. Lesk, M., E. Schmidt. Lex – A Lexical Analyzer Generator. 1990.
36. Yang, W., C.-W. Tsay, J.-T. Chan. On the Applicability of the Longest-Match Rule in Lexical Analysis. – Computer Languages Systems & Structures, Vol. **28**, 2002, No 3, pp. 273-288.
37. Handzhiyski, N., E. Somova. Tunnel Parsing with Countable Repetitions. – Computer Science, Vol. **21**, 2020, No 4, pp. 441-462.
38. Van Deursen, A., P. Klint, J. Visser. Domain-Specific Languages. – ACM SIGPLAN Notices, Vol. **35**, 2000, No 6, pp. 26-36.
39. Tunnel Grammar Studio.
https://www.experasoft.com/products/ tgs/

40. T. Bray, Ed. The JavaScript Object Notation (JSON) Data Interchange Format. Internet Engineering Task Force (IETF), 2017.
41. T. Berners-Lee, R. Fielding, L. Masinter, Eds. Uniform Resource Identifier (URI): Generic Syntax. Network Working Group, 2005.
42. P. Kyzivat, Ed. Case-Sensitive String Support in ABNF. Internet Engineering Task Force (IETF), 2014.
43. Backus, J. W. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. – In: IFIP Congress, 1959, pp. 125-132.
44. Holub, A. Compiler Design in C. Prentice Hall, USA, 1990.

Received: 17.01.2022; Second Version: 30.03.2022; Accepted: 12.04.2022