sciendo

BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **23**, No 2
Sofia • 2023 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2023-0012

Tunnel Parsing with Ambiguous Grammars

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

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

***Abstract*:** *The article proposes an addition to the tunnel parsing algorithm that enables it to parse grammars having countable repetitions and configurations of grammar elements generating empty words without refactoring the grammar. The equivalency of trees built by the use of ambiguous grammar is discussed. The class of the ε-ambiguous grammars is defined as a subclass of the ambiguous grammars relative to these trees. The ε-deterministic grammars are then defined as a subclass of the ε-ambiguous grammars. A technique for linearly parsing on the basis of non-left recursive ε-deterministic grammars with the tunnel parsing algorithm is shown.*

***Keywords*:** *Parsing algorithm; tunnel parsing; ambiguous grammars; ambiguity; determinism.*

1. Introduction

The development of grammars is a slow and difficult process, especially for large grammars. Tools [1] can be used, by the developer, to simplify this task that automatically calculate different grammar parameters (for example to detect if a grammar is deterministic) and visualize different grammar-related information (recursion types, rules connectivity graphs, possible conflicts between terminal symbols and so on). The more expressive the power of the meta syntax describing the grammar, the easier it is for the developer to describe the target language. Of practical interest are the context-free grammars and especially the deterministic context-free grammars (a grammar from which a deterministic pushdown automaton can be made using one token of look-ahead), because many domain-specific languages and structured data are described with them.

After the grammar is created, its use follows, which is often the creation of a parser. Again, tools [1, 2, 3] can be used to reduce the parser development time. Whether the parser will be developed manually or generated automatically, a parsing algorithm must be used. Different parsing algorithms have different properties, mostly depending on the grammars they can use.

The parsing result consists of a status (the input data belongs or does not belong to the language) and, if necessary, a syntax tree – a data structure that contains syntax information about the successfully parsed input. In a syntax tree, the information from

the recognized tokens (created from the input data) is placed in the tree leaves. The information about the grammar rules that are used during the parsing is placed in the other tree nodes.

If two different grammars describe the same language, they are **equivalent grammars** [4]. This kind of equivalence is defined as a weak equivalence in [5]. A strong equivalence requires in addition to the weak equivalence that the two grammars assign for each input two syntax trees that may be considered structurally similar [5]. The structural similarity should be formulated by the intended application, as a possible formulation is that two trees are structurally similar if their corresponding representations as condensed skeleton trees are equal [5]. A condensed skeleton tree is one in which the rule names (taken from the parser grammar) in each tree node are dropped and the nodes that are the only child of their parents and have only one child are removed. However, this formulation of strong equivalency is relevant only for unambiguous grammars [5].

As the weak equivalence between arbitrary grammars (that two grammars generate the same sentences) is undecidable, the structural equivalence is decidable for any context-free grammar where two such grammars are structurally equivalent if they generate the same sentences and assign similar parse trees to each of them [6]. These trees differ only in the labeling of the nodes. Additionally, a grammar is said to be structurally ambiguous if it generates the same sentence twice with the same tree (except for a relabeling of nodes) [6]. The "relabeling" has a similar meaning as the formal definition in [4] for one-to-one tree correspondence, where two trees correspond to each other when there is a length preserving homomorphism (single value mapping) between the sets made of rules names.

Similarly to the condensed skeleton trees, Tomita [7] uses the term "packet node" to cope with the local ambiguities that occur in the syntax forest (all possible parse trees). A packet node is made of several nodes in the tree that represent local ambiguities and is threaded as a single node by the subsequent algorithms. The technique of the merging of the nodes into a packet node is called local ambiguity packing.

For a definition of ambiguity, we will reuse the one in [8] where grammar is said to be ambiguous when it admits more than one distinct derivation tree for the same input. In relation to the ambiguity, from the perspective of the syntax tree derivation (the sequence of steps performed by the parser to accept the input [9]), in [10] is noted that the different valid derivations for a sentence in a given language are not equivalent to each other. In terms of the generation of strings (from left to right) in a language, a grammar is ambiguous when it can generate the same string in more than one way. The presence of ambiguity in an arbitrary context-free grammar is undecidable [11].

We will hereby extend the equivalency of trees that cover specifically the trees built on the basis of ambiguous grammar.

We will denote the **empty word** (a word with zero characters) with $\boldsymbol{\varepsilon}$. There can be different structures in a syntax tree [12], but we will refer to all of them as nodes.

Any node that has no content (called "label" in [4]) made of/from tokens in itself or in its sub-nodes, we will call an $\boldsymbol{\varepsilon}$**-node**. The contemporary grammar forms (like

the Augmented Backus-Naur Form (ABNF) [13], which is used in this article) contain different components (like rules, elements, etc.).

Each component that can be a base for the creation of an ε-node (In an original tree – all of the input characters are placed in the tree, and only those [12]) will be called an ***ε*-component**. Each ε-component generates ε, and if this is a rule then it will be called an ***ε*-rule**. In some grammar forms, the ε-rules are the only ε-components that generate ε. When using grammars with ε-components there might be more than one ε-node possible to be built in a given tree location for some valid input data (when some ε-component generates ε in more than one way). An optimal ε-node can be built when specific criteria for optimality are chosen. There can be different criteria for optimality, for example, each node requires as little as possible runtime memory.

In an ***ε*-condensed syntax tree,** all ε-nodes satisfy the following optimality criteria:

1. Each ε-node in the tree has as few sub-nodes as allowed, by the grammar elements it reflects.

2. If more than one grammar concatenation in a given alternation can be the base for the creation of the same amount of sub-nodes then the nodes created on the basis of the earliest defined concatenation are created.

3. The non-ε-nodes are ordered before the ε-nodes when all of the nodes are created one after another in the same level on the basis of the same grammar element (as a consequence of this ordering a combinatorial explosion is prevented).

If two different syntax trees built by the use of the same grammar and the same input become exactly the same after they are made ε-condensed then they are ***ε*-equivalent**.

A grammar that is ambiguous only because the possibilities to build syntax trees are ε-equivalent to each other per any valid input data, we call an ε-ambiguous grammar. Conversely, when parsing some valid input data based on an ε-ambiguous grammar then all possible resulting syntax trees are ε-equivalent. If a grammar is non-deterministic (with one token of look-ahead) only because it is ε-ambiguous then we call it an ***ε*-deterministic grammar**.

The article places the ε-ambiguous grammars as a subclass of the ambiguous grammars. This means that every ε-ambiguous grammar is ambiguous, but not the other way around. In a similar way, every ε-deterministic grammar is ε-ambiguous, but not the other way around.

If a parsing algorithm cannot use grammars having ε-rules it is possible to change the grammar by refactoring it. A better solution is to use a parsing algorithm that can parse a grammar having ε-rules without refactoring it. While it is possible to convert a grammar having ε-rules into one without them, that is unacceptable due to the loss of clarity [14].

This article proposes an addition of the Tunnel Parsing Algorithm (TP Algorithm) [15]; (operational in the parsing machines [16] generated by Tunnel Grammar Studio [1]) that enables it to parse on the basis of ε-deterministic grammars having countable repetitions [15] without refactoring the grammars and to output one of all ε-equivalent trees (concrete or abstract with a different level of abstraction [12]).

An LL(k) grammar can be a base for deterministically parsing an input by using at most k tokens of look ahead. This implies that a unique production (concatenation) can be chosen deterministically at any time. When a grammar is ambiguous, there is at least one place of non-determinism during parsing where there are at least two ways to complete the parsing successfully for at least one valid input. As a consequence, the ambiguous grammars (and the left recursive ones) are not LL(k) for any k. The left-recursive grammars are always non-deterministic, but not always ambiguous. The addition to the TP Algorithm in this article enables it to linearly parse on the base of non-left recursive ε-deterministic grammars that are not LL(k) for any k, because they are ε-ambiguous and in turn ambiguous. The TP Algorithm parses linearly LL(1) grammars as well [15]. For other non-left recursive grammars, it might exhibit an exponential time in the worst case depending on the particular grammar and input. The TP Algorithm can also parse some LL($k > 1$) grammars by using only $k - 1$ tokens with a simple optimization [15].

Section 2 discusses the state of the art of parsing algorithms for context-free grammars, mostly about the handling of ambiguous grammars. The problem being addressed in the article about the support of grammars having ε-rules (and other ε-components), is stated in Section 3. Section 4 shows the addition of the TP Algorithm that enables it to parse on the base of non-left recursive ε-deterministic grammars linearly. The required steps to create a parser that works with the TP Algorithm are listed. A demonstration of the TP Algorithm, based on an ε-deterministic grammar, with the various changes in the internal state of the parser, is shown. Section 5 lists the contributions of the article and its possible future development.

2. Related work

In linguistics, the context-free grammars are used to describe words and sentences in languages and in computer science to recursively describe data structures. A convenient way to describe context-free grammars is through a widely known meta syntax such as ABNF [13, 17].

The most popular parsing algorithms are the LL (Left to right, Left most derivation) and the LR (Left to Right, Right most derivation). Detailed examples for LL and LR parsing are shown in [18]. An algorithm to prove that a grammar is LL(k) for a fixed integer $k \geq 0$ in $O(n^{k+1})$ worst case time complexity, where n is the grammar size, is shown in [19].

It is decidable whether a context-free grammar is an LR(k) grammar for a given k, but it is undecidable whether there is such k for a given context-free grammar [20]. A large class of grammars that properly contains the LR(k) grammars is the class of the LR-regular grammars that can be parsed with two-phase parsing, linearly on the input's length [21]. If the grammar does not have hidden left recursion and it is not right recursive then the LR recognition can be improved [22]. However, article [22] gives little information about how this improvement can be applied by a parser as an extension to the recognizer.

Some ambiguous grammars (particularly the ones that contain dangling-else kind of ambiguity) can be used for parsing without backtracking if a predictive parser can be built from the grammar, as shown in [9], by specifying certain disambiguation rules. An LALR(1) parser can be constructed to solve a dangling ambiguity by preferring to attach the "else" to the nearest "unelsed" "then" [8].

One way to parse on the basis of a formal grammar is to do "all-paths" parsing with the help of the graph-structured stack of T o m i t a [23]. A similar strategy is discussed by Woods in [24], where the different Augmented Transition Networks (ATN), as part of the cascaded ATN, have their own configurations. These configurations form a tree that merges the common initial parts of the configurations. When some of the configurations become different from each other and later become the same configuration again then they should be merged (in order to reduce the number of combinations) [24].

If one is using regular expressions (regular grammar) instead of context-free grammar then all possible parse trees can be implicitly represented by a context-free grammar [25]. This grammar is created after the parsing with the use of the input. The different parse trees (represented as lists) can be generated by this grammar. The implicit representation (with a grammar) is needed, because the resulting parse trees can be an infinite number and cannot be written explicitly [25]. A practical implementation in Scheme for the algorithm in [25] is discussed in [26], where (in the context of the automatically generated lexical analyzers) it is noted that in case of an ambiguity, it is sufficient to build only one of the trees. This can be done by choosing a single case when more than one is possible [26].

Tomita's parsing algorithm is initially described to handle all context-free grammars including the ambiguous ones [7], but it is later discovered that it does not handle all non-cyclic context-free grammars having ε-rules properly. A modification of the handling of such grammars is shown in [27]. While the grammar having ε-rules can be used for recognition of an input by the modification in [27], the parsing (not only recognition but also outputting of trees [28]) by this algorithm is also needed [14]. Another modification of Tomita's algorithm that enables it to parse on the basis of grammars having ε-rules by introducing cyclic subgraphs in the original graph-structured stack is shown in [29].

Tomita's algorithm has a time complexity of $O(n^{p+1})$, where n is the input length and p is the length of the longest right-hand side, but the algorithm can be modified to recognize the input with time complexity of $O(n^3)$ for productions (rules) of arbitrary length, as shown in [30]. The algorithm has a space complexity of $O(n^2)$ in the worst cases [30]. A modification of the K i p p s [30] ancestor's table can be made in such a way that the parse tree can be extracted from the graph-structured stack in linear time (relative to the number of symbols) but by increasing the graph-structured stack space complexity to $O(n^3)$ [31].

The graph-structured stack can be used to store the different ambiguous cases during an LR parsing but is also usable for other than the LR parsing algorithms [32]. The ambiguities in the parse forest (all possible syntax trees) as a result of a table-driven Tomita's generalized LR-like parser can be resolved according to the defined order of the productions [33]. The parse forest can be seen as a generalization of a

parse tree [34]. To disambiguate a parse forest, one might wait for it to be built and then prune the forest with disambiguation filters designed as combinators [35].

The (generalized) LR algorithms cannot handle (do non-terminate) when built from grammars having hidden left recursion. This can be corrected with a change of the algorithm by doing a parse stack inspection upon reduction that can take a linear time to the length of the right-hand side [36]. However, this requires the (generalized) LR items to be changed, which can lead to fewer or more items, depending on the grammar. Instead of a graph-structured stack, it is possible to use dynamic programming to construct generalized LR parsers based on Earley's parsing algorithm [37].

The recognition algorithm of Earley [28] has an upper time bound of n^3 for an arbitrary context-free grammar (that does not need to be in a normal form), where n is the number of input symbols. The same algorithm has an upper time bound of n^2 for unambiguous grammars as well as for ambiguous grammars with bounded (not infinite) ambiguity (because the degree of ambiguity of the bounded ambiguity will be a constant multiplied by n^2). The number of distinct derivation trees one sentence has is its degree of ambiguity [28]. The standard Earley parser might not be able to recognize some inputs when the grammar has ε-rules, but it can be modified to gain this ability [38]. The usage of the shared packed parse forest of Tomita [39] instead of Earley's parse forest, is shown in [40]. By using memoization, a simple top-down backtracking parser can have the same time complexity as the Earley parsing algorithm [41].

To cope with the limits of the LL(k) parsers (when the used grammar is not LL(k) for any k, because it is ambiguous), predicates can be used to create a pred-LL(k) parser [42]. The predicates are used to perform some custom routine to determine how the parser has to continue the parsing. COCO/R in a similar way uses conflict-resolvers to deal with non-LL(1) grammars [43]. These resolvers are Boolean expressions that are part of the grammar and can access the next tokens or do some other kind of semantic checking in order to determine whether a particular production shall be accepted for the continuation of the parsing or rejected (in this case the parser will try the next production) [43].

Instead of pre-computing the possible look-ahead symbols needed at any position in the grammar rules, the ALL(*) algorithm does this dynamically during runtime, only for the finite collection of input sequences actually seen. To do this the parser launches sub-parsers that explore each production. The sub-parsers die off after their paths fail to match the remaining input. If one sub-parser survives then it has identified the unique production that has to be used. If more than one sub-parser reaches the same state, they coalesce, as this indicates that an ambiguity has been discovered. In this case, the production with the lowest production number is considered useful. The ALL(*) parsers memoize [44] the analysis results so they can be reused for subsequent production searches. This is done to avoid redundant computations and the exponential nature of non-deterministic sub-parsers at the cost of memory. The algorithm uses a graph-structured stack similar to the GLR algorithm. The ALL(*) parsing strategy has $O(n^4)$ worst-case time complexity and accepts context-free grammars that do not have direct, indirect, or hidden left

recursion [45]. The building of the deterministic finite automata, used for predicting the production, is incremental during runtime. This is done without a relation to the current call stack. Parsers that ignore the parser call stack for prediction are called Strong LL (SLL) parsers, and this is the first of two stages of the ALL(*) parsing algorithm. The second is an LL stage (that uses the call stack) and is used if the SLL prediction finds a conflict [45].

Even if it were possible to generate all possible syntax trees for a given ambiguous sentence, these trees would not give much information to the user [46]. The algorithm in [46] parses with $O(n^3)$ time according to the number of symbols tested. However, this recognition algorithm and its modification, which shows the ambiguity of the parsed sentence are defined to work with grammars in Chomsky Normal Form (CNF; grammars without ε-rules) [46]. For a grammar translated to CNF in [30], it is noted that it would take too much time, if it is at all possible, to receive some useful information from the derivation trees based on it.

Chart parsing (also called tabular parsing) is a form of dynamic programming that can be used for parsing on the basis of ambiguous grammars. It is related to memoization and its relation to Earley's, Cocke-Younger-Kasami (CYK), and the LR parsing are explored in [47]. There are chart-parsing examples in [48].

Another algorithm that has explicit call stack management is the Generalized LL (GLL) algorithm [49]. The GLL parsing algorithm is with worst-case $O(n^3)$ time and space complexities [50]. A more efficient graph-structured stack can be used than the original GLL algorithm, to improve its runtime, but it still remains with $O(n^3)$ time and space worst-case bounds [51]. Earley's algorithm, which also runs in $O(n^3)$ time in the worst case, has $O(n^2)$ upper bounded space requirements, whereas the CYK algorithm always has $O(n^2)$ space requirements [28].

The standard GLL algorithm can be upgraded to support EBNF-like parsing of a supplied BNF grammar by a grammar factorization prior to the parser generation [52]. The goal is to shorten the grammar size, so the algorithm can run faster and requires less memory than when a BNF grammar is used.

3. Problem

The parsing algorithms often have problems with the grammars having ε-rules. Additional complications might arise when there are ε-components other than ε-rules. If a given parsing algorithm cannot use grammars having ε-rules (or other ε-components), it is possible to change the grammar by refactoring it. However, a side effect of the refactoring process is that the grammar will not look as designed and that the syntax trees will have a structure that is not initially intended by the developer. Subsequent refinement of the grammar, even minor changes to it, might necessitate additional refactoring that might significantly change the resulting syntax trees, and increase the development time of all tools based on the grammar. A better solution is to use a parsing algorithm that can parse on the basis of a grammar having ε-rules (and other ε-components) without refactoring it.

We will describe an addition to the tunnel parsing algorithm that enables it to parse linearly non-left recursive ε-deterministic grammars having countable

repetitions without refactoring the grammars. We will show clear criteria of how to choose one of all possible syntax trees built by the use of ε-ambiguous context-free grammars.

The TP algorithm works with advanced context-free grammars [1], but for simplicity, we will use the well-known context-free grammar definition. A context-free grammar is the tuple (N, Σ, R, S), where set N contains all non-terminal symbols (rule names), set Σ contains all terminal symbols (alphabet), $N \cap \Sigma = \emptyset$ (empty set), set R contains all rules; S is the start symbol of the grammar and $S \in N$. The ABNF grammar notations have the following meanings (only those used in the article are listed):

- t defines a terminal value in ABNF, but for the purpose of this article defines a terminal symbol (an element of Σ). To simplify the algorithm description, each terminal symbol will consist of a single character;
- r defines a rule ($r \in N$) when it is on the left side of the sign "=" or a reference to a rule when it is on the right side;
- $x\ y$ is a concatenation of grammar elements (for short, "elements");
- ($z\ w$) defines a grammar group (for short, a "group") of elements;
- a / b defines an alternation (logical "or" for the concatenations);
- n*m A defines the repetitions of A, where $n \in \mathbb{N}$ is the minimum repetitions (if omitted, it is considered to be a zero), and $m \in \mathbb{N}$ is the maximum repetitions (if omitted, it is considered to be an infinity), $n \leq m$. When $n > 1 \vee m > 1 \wedge m \neq \infty$ the TP Algorithm uses an additional stack called a **repetition stack** that contains the number of repetitions of A during runtime. Details about the usage of the repetition stack are shown in [15].

The groups in an ABNF grammar can be seen as rules with a single implicit reference to them at the point of the definition. Therefore, everything written about the rules below will apply to the groups as well. Under a "reference", it will be understood as a reference to a rule in the ABNF syntax as well as the implicit reference to a group when it is seen as a rule.

All of the terminal symbols that can be recognized from the beginning of a rule directly or by recursively entering into the referenced rules (and possibly exiting from, when the rule generates ε) will be called **reachable** [53]. Reachable terminal symbols after an element are those that can be recognized after it without using the possible depth stacks (defined later) to the rule where the element is located.

4. Solution

This section describes how the TP Algorithm parses non-left recursive ε-deterministic grammars linearly, and outputs the commands for the building of a concrete or an abstract syntax tree (deterministically selected to be the smallest of all possible trees). The linearity of the TP Algorithm for LL(1) grammars extends to the ε-deterministic grammars, because the later defined objects and their relations do not add computations based on the length of the input, but properly guide the algorithm through the automata having multiple ε-paths (defined later).

The TP Algorithm should not be implemented as a depth-recursive LL parser but as an iterative process to avoid the possible overflowing of the thread dedicated stack. A good property of the iterative parsing is that it might pause after each iterative step. This is useful when not all of the input data is available at the start of the parsing. Once more data becomes available, the parser can continue. Another property of the iterative parsing is that the data (that the algorithm operates with) is available in instances of data structures (not stored in the thread-dedicated stack) and can be serialized and deserialized on demand. If a TP implementation cannot guarantee (during the entire parsing) that the use of the thread-dedicated stack (by the algorithm itself) is upper bounded by a constant (not dependent on the input data) then it does not conform to the algorithm.

A **segment** will be called an object that exists for each rule reference. The **depth stack** in the TP Algorithm consists of segments. There are more details about the segments in [15].

To create a parser working with the TP Algorithm [15] that parses linearly non-left recursive ε-deterministic grammars, the following steps must be performed: a) design of automata; b) finding the shortest ε-paths inside the automata; c) extraction of tunnels; d) construction of routers; and e) creation of a control layer.

Fig. 1 contains an ABNF grammar with two rules, where rule "main" has two references to rule "sub". The grammar is ε-deterministic and can be parsed linearly by the TP Algorithm, as a deterministic grammar.

```
main = 0*1(5 sub) / %s"a" *sub
sub  = 0*1 %s"b" / %s"c"
```

Fig. 1. Linked grammar rules

4.1. Design of automata

For each grammar rule, an automaton is constructed (as the one in Fig. 2), where the transitions are of two types:

a) recognizing a terminal symbol (a character enclosed in double quotes) at the end of which stands a **terminal state**;

b) not recognizing a terminal symbol, called an ***ε*-transition**. An ***ε*-path** is a sequence of at least one ε-transition.

A PM using the TP Algorithm can output an ε-condensed syntax tree from an ε-deterministic parser grammar when each ε-path has the minimum number of ε-transitions (satisfying ε-condensed optimality criterion No 1), and in case of an equal number of ε-transitions, the ε-transitions created from the earliest defined grammar concatenations are used (satisfying ε-condensed optimality criterion No 2). The ε-condensed optimality criterion No 3 is satisfied by the design of the TP Algorithm.

In Fig. 2, the labels of the ε-transitions indicate certain operations on the internal state of the parser and the dotted transitions indicate operations related to the repetition stack [15], where *cpush* pushes one element into the stack with a value of one, *cinc* increases with one the top of the stack. *ctop* is the value on the top of the stack, and *cpop* removes one value from the stack. This stack contains the number of

repetitions already found (or in a process of finding) in the input for a particular grammar element.

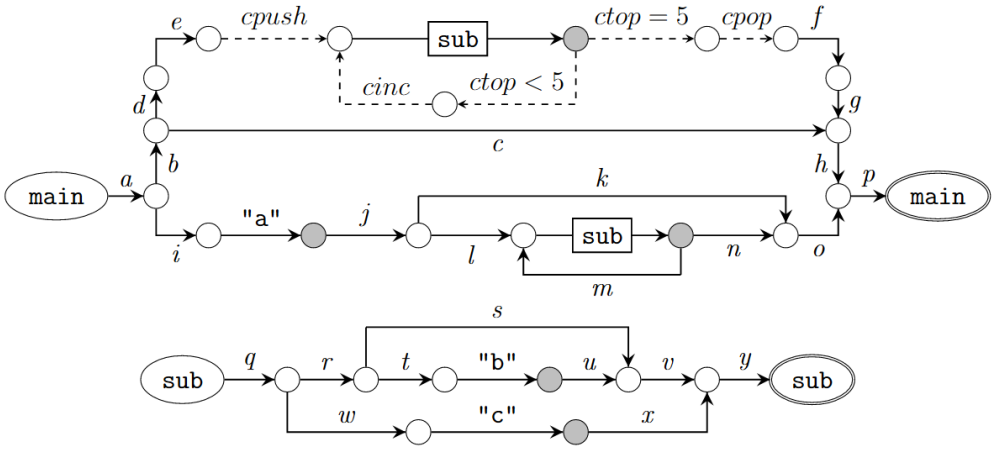

Fig. 2. Automata generated from the grammar in Fig. 1

The labels in Fig. 2 have the following meanings: *a*, *q* – entering in main and sub, respectively; *b*, *r* – entering in the first concatenation of main and sub, respectively; *c* – skipping the grammar group, called "later *G*"; *d* – entering in *G*; *e* – entering in the single concatenation of *G*; *f*, *g* – exiting from the single concatenation of *G* and *G* itself, respectively; *h*, *v* – successfully exiting from the first concatenation of main and sub, respectively; *i*, *w* – entering in the second concatenation of main and sub, respectively; *j* – next element; *o*, *x* – successfully exiting from the second concatenation of main and sub, respectively; *p*, *y* – exiting from main and sub, respectively with a success; *s*, *k* – skipping an element; *m* – repeating an element; and *l*, *n*, *t*, *u* – pure ε-transitions.

In the automata, there are two ε-paths from the beginning to the end of the main rule: 1) *abchp*; and 2) *abde* followed by five times *qrsvy* with an end of *fghp*. The first ε-path is the shorter one. After the recognition of *"a"*, there are an infinite number of ε-paths (due to the presence of a cycle with ε-transitions *mqrsvy*), as the shortest is *jkop*.

4.2. Finding the shortest ε-paths

The shortest ε-path from the start of each automaton to its final state is found. The shortest ε-path originating from a key state (defined later, other than a start state) is a concatenation of the shortest ε-paths (found for the transitions made on the basis of the respective grammar elements) till the end of the automaton. Later we say that the found ε-path is chosen. For the purpose of this article, these chosen shortest ε-paths have the lowest number of ε-transitions. These ε-paths can be chosen by another, more complex, criteria as the amount of time needed by the parser to use them and/or the amount of memory needed by the subsequently built syntax trees from them. Choosing the ε-paths will be the subject of another article. For the grammar in Fig. 1, the shortest ε-paths are as follows: a) *abchp* for rule main; b) qrsvy for rule sub.

4.3. Extraction of tunnels

A **tunnel** is a group of operations for changing the internal state of a parser and the related operations for the syntax tree building. To enable a context-free grammar-based recognition, for each **forward tunnel** (A tunnel that advances the parser to a successful termination), there must be a **backward tunnel** (A tunnel that will restore the parser as it was before the use of the forward tunnel).

For each rule start state, each state after a reference, and each terminal state of each automaton (all together called **key states**), all transitions to the next reachable terminal states (or the final state of the automaton from which the search began) are collected into tunnels in a depth-first like search manner. The darker states in Fig. 2 are key states. There is one collection per reachable terminal symbol, including all other terminal symbols in conflict with it, if any. Due to the possible presence of ε-components in the grammar, the collection might be able to reach the final state of some referenced automaton. In this case, the search explores the chosen ε-path transitions for the referenced automaton after all others and exits the automaton from there and only from there. This kind of collection ensures that when no token is recognized by the referenced automaton an optimal ε-node will be built. As a consequence of this for some nondeterministic grammars, the attempted parsing paths might not resemble an exact in-order traversal of the automaton states.

The following denotations will be used later on: E – the set of all transitions in the automata; O – the set of operations that change the depth stack of the parser, where $o = \{o_1, o_2,...\}$, $o_k \in O$, $k \in \mathbb{N}$; T – the set that contains all of the tunnels $\tau \in T$; $\tau = [e \mid o] - d + a$ – denotes a tunnel, where d is the number of counters that will first be removed from the repetition stack, a is the number of counters (each with a value of one) that will be added to the repetition stack, the transitions that the tunnel uses are $e = \{e_1, e_2,...\}$, $e_i \in E$, and $i \in \mathbb{N}$; $\neg x$ – the reverse of x; $\downarrow r$ – entering into r (when the recognition of r begins, by pushing into the depth stack the respective segment linking to r); and $\uparrow r$ – exiting from r (after the successful recognition of r, by popping a segment linking to r), where $r \in N$, $\downarrow r \in O$ and $\uparrow r \in O$. When d or a are zeroes they will be omitted.

For the grammar in Fig. 1 with automata in Fig. 2 the tunnels are:

$\tau_0 = [q, r, t \mid \downarrow\text{sub}]$, $\tau_1 = [a, i \mid \downarrow\text{main}]$,
$\tau_2 = [q, w \mid \downarrow\text{sub}]$, $\tau_3 = [\neg t, s, v, y \mid \uparrow\text{sub}]$,
$\tau_4 = [\neg y, \neg v, \neg u \mid \neg\uparrow\text{sub}]$, $\tau_5 = [\neg y, \neg x \mid \neg\uparrow\text{sub}]$,
$\tau_6 = [a, b, c, h, p \mid \downarrow\text{main}, \uparrow\text{main}]$, $\tau_7 = [q, r, s, v, y \mid \downarrow\text{sub}, \uparrow\text{sub}]$,
$\tau_8 = [\neg p, \neg h, \neg c, \neg b, \neg a \mid \neg\uparrow\text{main}, \neg\downarrow\text{main}]$,
$\tau_9 = [\neg y, \neg v, \neg s, \neg r, \neg q \mid \neg\uparrow\text{sub}, \neg\downarrow\text{sub}]$,
$\tau_{10} = [\neg p, \neg o, \neg k, \neg j \mid \neg\uparrow\text{main}]$, $\tau_{11} = [\neg i, b, c, h, p \mid \uparrow\text{main}]$,
$\tau_{12} = [\neg w, r, s, v, y \mid \uparrow\text{sub}]$, $\tau_{13} = [j, l, q, r, t \mid \downarrow\text{sub}]$,
$\tau_{14} = [j, l, q, w \mid \downarrow\text{sub}]$, $\tau_{15} = [j, k, o, p \mid \uparrow\text{main}]$,
$\tau_{16} = [\neg t, \neg r, \neg q, \neg l, k, o, p \mid \neg\downarrow\text{sub}, \uparrow\text{main}]$,
$\tau_{17} = [\neg w, \neg q, \neg l, k, o, p \mid \neg\downarrow\text{sub}, \uparrow\text{main}]$,
$\tau_{18} = [a, b, d, e, q, r, t \mid \downarrow\text{main}, \downarrow G, \downarrow\text{sub}] + 1$,
$\tau_{19} = [a, b, d, e, q, w \mid \downarrow\text{main}, \downarrow G, \downarrow\text{sub}] + 1$,
$\tau_{20} = [\neg t, \neg r, \neg q, \neg e, \neg d, c, h, p \mid \neg\downarrow\text{sub}, \neg\downarrow G, \uparrow\text{main}] - 1$,

τ_{21} = [¬w¬q, ¬e, ¬d, c, h, p | ¬↓sub, ¬↓G, ↑main] - 1,
τ_{22} = [f, g | ↑G], τ_{23} = [¬g, ¬f | ¬↑G],
τ_{24} = [h, p | ↑main], τ_{25} = [¬p, ¬h | ¬↑main],
τ_{26} = [n, o, p | ↑main], τ_{27} = [¬p, ¬o, ¬n | ¬↑main],
τ_{28} = [u, v, y | ↑sub], τ_{29} = [¬y, ¬v, ¬u | ¬↑sub],
τ_{30} = [x, y | ↑sub], and τ_{31} = [¬y, ¬x | ¬↑sub].

Note that the first pushed segment by the use of the tunnels starting from the start states of each automaton depends on how the rule is being used for parsing – the parsing starts from the rule or a subsequence repetition of a reference to that rule begins. The first case can be handled by pushing a null segment into the depth stack that signifies the start rule. However, this is an implementation choice. The second case is trivial, because it happens only when there is a subsequent repetition of a reference, and at that time, the last popped segment (from the end of the previous repetition) is the one that has to be pushed back.

Certainly, there are different optimizations that can be made. The pushing and then popping of the same segment by the use of t_6 could be optimized by an actual implementation, but in the article, it adds a presentation detail. Another possible optimization is the operations for popping from the depth stack to be replaced with a single number – how many elements to pop. The current denotations are chosen because we consider them easy to visually follow, even though they are abstract in nature.

4.4. Construction of routers

In order to speed up the search for the next state of the parser at runtime, before the start of the parsing, the information about the reachable terminal states is stored in static read-only memory, in objects called **routers**. Each segment has a link to a router with the next reachable terminal states after the segment's reference, and in case of a repetition, a link to one or two more routers (defined later) having the reachable terminal states from the start of the referenced rule is also used. The routers-related denotations are as follows: M is the set of all routers; $\sigma \in \Sigma$ is the terminal symbol; C is the set of all control states (described later); a control state $c \in C$; P is the set of all paths in a router; p is a path into a router as a pair of a terminal symbol and a control state: $\sigma \rightarrow c$; $\mu = \langle P | c_\varepsilon \rangle$ is a router, where $\mu \in M$, $c_\varepsilon \in C$, as c_ε (called the **escape *c*-state** for the router) will be used when the terminal symbol is not found in P.

For the grammar in Fig. 1 with automata in Fig. 2 the routers are:
μ_0 = 〈"a"→c_2, "b"→c_3, "c"→c_4 | c_0 〉 , μ_1 = 〈"b"→c_5, "c"→c_6 | c_1 〉 ,
μ_2 = 〈"b"→c_{11}, "c"→c_{12} | c_{24} 〉 , μ_3 = 〈 | c_{28} 〉 ,
μ_4 = 〈 | c_{29} 〉 , μ_5 = 〈"b"→c_7, "c"→c_8 | c_{39} 〉 ,
μ_6 = 〈 | c_{25} 〉 , μ_7 = 〈 | c_{26} 〉 ,
μ_8 = 〈"b"→c_9, "c"→c_{10} | 〉 , and μ_9 = 〈 | c_{27} 〉 .

There are three segments created from the grammar in Fig. 1, one per reference:

a) the segment for the first reference to sub in main, links router μ_5 for a repetition attempt (called later the segment's **minimum router**, because it is used

before the minimum number of repetitions are made), and μ_6 for a continuation after the reference (called later the segment's **next router**);

b) the segment for the second reference to sub in main, links router μ_8 for a repetition (called later the segment's **middle router**, because it is used after the minimum, but before the maximum, number of repetitions are made), and μ_9 for a continuation; and

c) the segment for group *G* links router μ_7 for a continuation.

4.5. Creation of control layer

On the base of the tunnels and the routers, a set of **control objects** is created. They are used during the execution of the parser. The control objects can be in one of several control states (*c*-states) that are different for each control object and prescribe the operations that have to be performed on the parser's internal state. The control objects signify "where" in the automata, the PM has reached, and the control states – "which" operations must be performed. The TP Algorithm uses an **execution stack** that contains the information about the progress of the parser. Each execution stack element links to one *c*-state. The top of the execution stack shows what is the next task that the parser must perform. "The parser in this *c*-state" later means that the top of the execution stack links to the particular *c*-state. The following objects are relevant for the presentation of the addition of the TP Algorithm that enables it to use grammars having ε-rules (and other ε-components) (to avoid duplication of content, some of the described objects in [15] are briefly summarized here).

- ***c*-origin** – an object created for each rule (but not for groups when seen as rules). It has a link to a router with all reachable terminal symbols from the beginning of the rule. The object has one control state – *use*. The parser in this *c*-state will perform a search in a linked router for the first input token and the result will replace the top of the execution stack. If it is known in advance which rules will be used as starting rules then it is possible to skip the creation of these *c*-origin objects for the non-starting rules.
- ***c*-terminal** – created for each terminal state with one *c*-state – *use*. The parser in this *c*-state will perform a search for the current input token in a linked router and the result will replace the top of the execution stack.
- ***c*-token** – created for each terminal symbol (when the counting of repetitions is not necessary – the maximum number of repetitions is one) that can be found by a router search. The *c*-object has two control states: a) *use* – the parser in this state moves with one input symbol forward; b) *used* – after a subsequently unsuccessful recognition attempt, the parser in this *c*-state performs operations to restore its internal state to the one before the *use* c-state. This includes the use of the backward tunnel of the escape *c*-state of the adjacent *c*-terminal's router, if any.
- ***c*-list** – similar to c-token (when the counting of repetitions is necessary – the maximum number of repetitions is more than one) [15]. If there are conflicts between terminal symbols, then the different c-token and *c*-list objects form a list. After all of the *c*-token and *c*-list objects in each list one *c*-back or *c*-passage object follows, both defined later. When a *c*-token or *c*-list *used* state is executed the next object will replace the top of the execution stack.

- ***c*-epsilon-origin** – created for each start rule that has an *ε*-path. It is used by the router of the *c*-origin object, created for the same rule when the first input symbol is not reachable from the beginning of the rule. It has two *c*-states: a) *use* – the parser will perform the steps to go through the ε-path and will signal that the end of the start rule has been reached; and b) *used* – the parser will return to its initial state and will signal that all possible recognition steps are explored.
- ***c*-epsilon-next** – created when there is an *ε*-path after a key state (other than a rule start) to the end of the automaton. It will be used when there are no reachable terminal symbols for the current input token from the key state. The *c*-object has one *c*-state – *use*. The parser will perform the steps to exit the rule and the top of the execution stack will be replaced by the *c*-unwind *c*-object (described below).
- ***c*-epsilon-fill** – an object that only links the tunnels used for the *ε*-fill operations (described below).
- ***c*-passage-origin** – after a terminal symbol is recognized from the beginning of a rule (which has an *ε*-path), the parser will use the found *c*-object to continue the parsing. If the parsing is subsequently unsuccessful then the parser will start to progress backward. In this case, after the previously used *c*-object(s), one *c*-passage-origin will stand at the top of the execution stack. The object has two *c*-states:

a) *use* – the parser will use a tunnel that will lead to the end of the *ε*-path of the current rule. This *c*-state will place the parser in the internal state it would be in if the *c*-epsilon-origin (created for the same rule) has been initially used. The parser then signals that the end of the start rule is reached. Reaching the end of the start rule does not imply that the entire input has been recognized;

b) *used* – the parser will use a tunnel to the beginning of the start rule, and will signal that the parsing has ended after all possible recognition steps are explored.

- ***c*-passage-minimum** – created at the end of a list of *c*-tokens/*c*-lists in a router, when the router has an escape c-state, a given reference's segment links the router as a minimum router, and the reference repeats at least two times. The object has one *c*-state: *use*. To get to the use of this *c*-state: a) the parser has to currently parse the referenced rule through that reference; b) the minimum number of repetitions must not yet be parsed; c) the parser progresses backward. In the situation thus created, the parser executes such a tunnel that the parser will become in the same internal state as if the entire *ε*-path of the referenced rule has been initially used (instead of the *c*-tokens/*c*-lists located before the *c*-passage-minimum). This will effectively end the current repetition of the reference. Then, the *ε*-path of the referenced rule is used for the remaining number of minimum repetitions for the reference, as this operation we call ***ε*-fill**. After the *ε*-fill, the segment's next router is used for a continuation of the parsing.
- ***c*-passage-next** – created at the end of each list made of *c*-token/*c*-list objects inside a router that in turn is linked by a *c*-terminal and has an escape *c*-state. The object has one *c*-state – *use*, when the parsing is not successful, after the last used *c*-token/*c*-list in the router of the *c*-terminal, a tunnel will be used that will place the parser in its internal state that it would be in if the list of *c*-tokens/*c*-lists has never been used, but instead of them, the *ε*-path after the *c*-terminal to the end of the rule

has been used. After the execution of this tunnel, the *c*-unwind (described later) will stand on the top of the execution stack.

- ***c*-back** – there are different c-states for progressing backward in [15].
- ***c*-unwind** – in its single c-state, the parser removes one segment from the depth stack. If the removed segment's minimum repetitions are not yet recognized, a repetition attempt is made by using the segment's minimum router. If the current token is not found in the router and the router has an escape c-state, its forward tunnel is used for an *ε*-fill. If the minimum number of repetitions is already made, but not the maximum number of them yet, a repetition is attempted with the segment's middle router. If this fails, no *ε*-fill is performed. If no repetition takes place then the segment's next router is used for a continuation. If the segment's next router does not have a path for the current token then its escape *c*-state's forward tunnel is used, if any, and a subsequent unwinding will be attempted. The removed segments are archived (not deleted) in case the parser progresses backward when they will be restored [15].
- ***c*-restore** – the parser restores one or more depth stack segments and adapts the other relevant data structures [15].

4.6. Parsing

The addition to the TP Algorithm, described in the previous sections, is operational in the parsing machines generated by Tunnel Grammar Studio (TGS) [1]. The tool can perform direct real-time parsing by a dynamically created interpreter and a supplied input, or a parsing machine can be generated to a source code for a target programming language that can be embedded in other software tools. The integrated interpreter in the tool is part of a parsing machine debugger, which also visually builds different syntax trees in forward and backward steps for a given grammar and an input. The generated source code parsing machine, when compiled and executed, can build a statically typed concrete syntax tree as instances of object-oriented classes because there is enough concrete information for the building from the used tunnels during the parsing. If some of this information from the tunnels is removed then an abstract dynamically typed syntax tree with a different level of abstraction can be built. Note that as a consequence of the definitions in this article, in an *ε*-deterministic grammar, the terminal symbols reachable from the start of an ε-rule referenced by a reference that repeats at least two times are not in conflict with themselves, during the parsing of the reference. When the parsing begins from rule main, the first c-object is c_{32}, and for rule sub is c_{33}.

Demonstration of the TP Algorithm based on the grammar in Fig. 1 with automata in Fig. 2 is shown in Table 2. The input data is of two characters: "bc". The used tunnels and routers are described in the previous sections. The control objects are described in Table 1.

In Table 2, column "Input" contains the parsed input (the dot is placed before the current input character), "Execution" – the *c*-objects and their *c*-states in the execution stack, "Depth" – the rules linked by the segments in the depth stack, "Repeat" – the repetition stack, and "Task" – the operation(s) performed by the parser to move from the current row to the next.

Table 1. Control objects of the parser for the automata in Fig. 2

Type *c*-origin	
No	Router
32	μ_0
33	μ_1

Type *c*-token			
No	Next	*c*-terminal	Tunnel
2	c_{13}	c_{34}	τ_1
3	c_{14}	c_{35}	τ_{18}
4	c_{15}	c_{36}	τ_{19}
5	c_{16}	c_{35}	τ_0
6	c_{17}	c_{36}	τ_2
7	c_{18}	c_{35}	τ_0
8	c_{19}	c_{36}	τ_2
9	c_{30}	c_{35}	τ_0
10	c_{31}	c_{36}	τ_2
11	c_{20}	c_{35}	τ_{13}
12	c_{21}	c_{36}	τ_{14}

Type *c*-passage-origin		
No	Forward	Backward
13	τ_{11}	τ_8
14	τ_{20}	τ_8
15	τ_{21}	τ_8
16	τ_3	τ_9
17	τ_{12}	τ_9

Type *c*-terminal	
No	Router
34	μ_2
35	μ_3
36	μ_4

Type *c*-passage-*		
No	*	Tunnel
18	minimum	τ_3
19	minimum	τ_{12}
20	next	τ_{16}
21	next	τ_{17}

Type c-epsilon-*			
No	*	Forward	Backward
0	origin	τ_6	τ_8
1	origin	τ_7	τ_9
24	next	τ_{15}	τ_{10}
25	next	τ_{22}	τ_{23}
26	next	τ_{24}	τ_{25}
27	next	τ_{26}	τ_{27}
28	next	τ_{28}	τ_{29}
29	next	τ_{30}	τ_{31}
39	fill	τ_7	τ_9

Global *c*-objects	
No	Type
37	*c*-unwind
38	*c*-restore

Type *c*-back	
No	Tunnel
30	τ_4
31	τ_5

The overall description of the parsing events is as follows: the parsing starts with a search for a tunnel to use, from the start of rule main; tunnel c_3 is found and used; the next token is loaded and it is used to search for the next tunnel in router μ_3 that has all reachable terminal states after “b” in rule sub; no such state is found, so the parser will use the tunnel τ_{28} till the end of the rule; the parser will unwind one element from the stack with the use of the global control object c_{37}; the attempt to repeat reference sub one more time succeeds, because in the minimum repetition router μ_5 there is a control state for the next input symbol “c”, with tunnel τ_2; after the second token is used, the parser will search for the reachable terminal states after “c”, that are in router μ_4; tunnel τ_{30} is used till the end of the rule, because no reachable state is found; there are no more input tokens, and instead of repeating the sub reference, an ε-fill is performed, that will use the shortest ε-path (τ_7) for rule sub three more times, to complete a total of five reference repetitions; there are no more elements to recognize in the concatenation of the current group *G*, so the group is exited by the use of tunnel τ_{22}; there are no more elements after group *G* to be

recognized, all input tokens are used, and the parsing completes successfully after the use of tunnel τ_{24}.

Table 2. TP algorithm execution for the grammar in Fig. 1 with automata in Fig. 2

No	Input	Execution	Depth	Repeat	Task
1	.bc	c_{32}|use	∅	∅	search in μ_0 and found c_3
2	.bc	c_3|use	∅	∅	use of τ_{18}
3	.bc	c_3|use	∅	1	rule enter
4	.bc	c_3|use	main	1	group enter
5	.bc	c_3|use	main, *G*	1	rule enter
6	.bc	c_3|use	main, *G*, sub	1	next token
7	b.c	c_3|use	main, *G*, sub	1	control state change
8	b.c	c_3|used	main, *G*, sub	1	control state addition
9	b.c	c_3|used, c_{35}|use	main, *G*, sub	1	search in μ_3, not found
10	b.c	c_3|used, c_{35}|use	main, *G*, sub	1	control state change
11	b.c	c_3|used, c_{28}|use	main, *G*, sub	1	escape with τ_{28}
12	b.c	c_3|used, c_{28}|use	main, *G*, sub	1	rule success
13	b.c	c_3|used, c_{28}|use	main, *G*	1	control state change
14	b.c	c_3|used, c_{37}|use	main, *G*	1	search in μ_5, found c_8
15	b.c	c_3|used, c_{37}|use	main, *G*	1	counter increment
16	b.c	c_3|used, c_{37}|use	main, *G*	2	use of τ_2, rule enter
17	b.c	c_3|used, c_{37}|use	main, *G*, sub	2	control state change
18	b.c	c_3|used, c_8|use	main, *G*, sub	2	next token
19	bc.	c_3|used, c_8|use	main, *G*, sub	2	control state change
20	bc.	c_3|used, c_8|used	main, *G*, sub	2	control state addition
21	bc.	..., c_8|used, c_{36}|use	main, *G*, sub	2	search in μ_4, found c_{29}
22	bc.	..., c_8|used, c_{29}|use	main, *G*, sub	2	control state change
23	bc.	..., c_8|used, c_{29}|use	main, *G*, sub	2	use of τ_{30}
24	bc.	..., c_8|used, c_{29}|use	main, *G*, sub	2	rule success
25	bc.	..., c_8|used, c_{29}|use	main, *G*	2	control state change
26	bc.	..., c_8|used, c_{37}|use	main, *G*	2	search in μ_3, not found
27	bc.	..., c_8|used, c_{37}|use	main, *G*	2	ε-fill forward, counter remove
28	bc.	..., c_8|used, c_{37}|use	main, *G*	∅	search in μ_6, found c_{25}
29	bc.	..., c_8|used, c_{37}|use	main, *G*	∅	control state change
30	bc.	..., c_8|used, c_{25}|use	main, *G*	∅	use of τ_{22}, group success
31	bc.	..., c_8|used, c_{25}|use	main	∅	control state change
32	bc.	..., c_8|used, c_{37}|use	main	∅	search in μ_7, found c_{26}
33	bc.	..., c_8|used, c_{37}|use	main	∅	control state change
34	bc.	..., c_8|used, c_{26}|use	main	∅	use of τ_{24}
35	bc.	..., c_8|used, c_{26}|use	main	∅	rule success
36	bc.	..., c_8|used, c_{26}|use	∅	∅	control state change
37	bc.	..., c_8|used, c_{37}|use	∅	∅	success

5. Conclusions

The article describes and demonstrates an addition to the TP Algorithm [15] that enables it to parse on the basis of a grammar having ε-rules (and other ε-components). This addition complements the addition to the TP Algorithm shown in [54], where the accent is that the parser can parse not only using the tokens' names but also their lexemes, in a case-sensitive or case-insensitive manner. The TP Algorithm is mostly applicable for the parsing of domain-specific languages such as programming languages and structured data. The result of the parsing is a syntax tree that is built from top to bottom and accurately reflects the grammar because the algorithm does not change it prior to the parsing. When the parsing grammar is ε-ambiguous, the resultant syntax tree is one of all ε-equivalent trees. If the parsing grammar is ε-deterministic, the resultant syntax tree is ε-condensed when the described in the article ε-nodes optimality criteria are satisfied.

The main contributions of the article are:

- The ε-ambiguous grammars are defined as a subclass of the ambiguous grammars;
- The ε-deterministic grammars are defined as a subclass of the ε-ambiguous grammars;
- Enabling the TP Algorithm to parse linearly non-left recursive ε-deterministic grammars.

In a future work, we shall describe an addition to the TP Algorithm that enables it to parse on the basis of a grammar having left recursion. An algorithm can be derived, on the basis of this article, that verifies whether a given ABNF grammar is ε-deterministic. We shall show that in a future article as well.

***Contribution*:** Nikolay Handzhiyski has developed the concept (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 have performed substantial revisions, verified the definitions, and contributed to the final draft.

References

1. Tunnel Grammar Studio (Visited on 03.11.2022). **https://www.experasoft.com/products/tgs/**
2. JavaCC (Visited on 03.11.2022). **https://javacc.github.io/javacc**
3. ANTLR (Visited on 03.11.2022). **https://www.antlr.org/**
4. A h o, A. V., J. D. U l l m a n. Translations on a Context Free Grammar. – Information and Control, Vol. **19**, 1971, No 5, pp. 439-475,
5. R e g h i z z i, S. C., L. B r e v e g l i e r i, A. M o r z e n t i. Formal Languages and Compilation. Cham, Switzerland, Springer Nature Switzerland AG, 2019.
6. P a u l l, M. C., S. H. U n g e r. Structural Equivalence of Context-Free Grammars. – Computer and System Sciences, Vol. **2**, 1968, No 4, pp. 427-463.
7. T o m i t a, M. An Efficient Augmented-Context-Free Parsing Algorithm. – Computational Linguistics, Vol. **13**, 1987, No 1-2, pp. 31-46.

8. A h o, A. V., S. C. J o h n s o n. LR Parsing. – ACM Computing Surveys, Vol. **6**, 1974, No 2, pp. 99-124.
9. A h o, A. V., S. C. J o h n s o n, J. D. U l l m a n. Deterministic Parsing of Ambiguous Grammars. – In: Proc. of 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'73), 1973, pp. 1-21.
10. K a s a m i, T. An Efficient Recognition and Syntax Analysis Algorithm for Context-Free Languages. – Technical Report, Air Force Cambridge Research Laboratory, Bedford, MA$, 1965.
11. C h o m s k y, N., M. P. S c h ü t z e n b e r g e r. The Algebraic Theory of Context-Free Languages. – In: P. Braffort, D. Hirschberg, Eds. Studies in Logic and the Foundations of Mathematics. Vol. **35**. Elsevier, 1963, pp. 118-161.
12. H a n d z h i y s k i, N., E. S o m o v a. The Expressive Power of the Statically Typed Concrete Syntax Trees. – In: CEUR Workshop Proceedings. Vol. **3061**. 2021, pp. 136-150.
13. C r o c k e r, D., P. O v e r e l l. Augmented BNF for Syntax Specifications: ABNF. Brandenburg Internet Working, 2008.
14. S h a b a n, M. A Hybrid GLR Algorithm for Parsing with Epsilon Grammars. OpenBU, USA, 1994.
15. H a n d z h i y s k i, N., E. S o m o v a. Tunnel Parsing with Counted Repetitions. – Computer Science, Vol. **21**, 2020, No 4, pp. 441-462.
16. H a n d z h i y s k i, N., E. S o m o v a. A Parsing Machine Architecture Encapsulating Different Parsing Approaches. – International Journal on Information Technologies and Security (IJITS), Vol. **13**, 2021, No 3, pp. 27-38.
17. K y z i v a t, P. Case-Sensitive String Support in ABNF. Internet Engineering Task Force (IETF), 2014.
18. H o l u b, A. Compiler Design in C. Prentice Hall, USA, 1990.
19. S i p p u, S., E. S o i s a l o n-S o i n i n e n. On the Complexity of LL(k) Testing. – Computer and System Sciences, Vol. **26**, 1983, No 2, pp. 244-268.
20. K n u t h, D. E. On the Translation of Languages from Left to Right. – Information and Control, Vol. **8**, 1965, No 6, pp. 607-639.
21. C o h e n, R., K. Č u l i k. LR-Regular Grammars – an Extension of LR(*k*) Grammar. – Computer and System Sciences, Vol. **7**, 1973, No 1, pp. 66-96.
22. A y c o c k, J., N. H o r s p o o l, J. J a n o u š e k, B. M e l i c h a r. Even Faster Generalized LR Parsing. – Acta Informatica, Vol. **37**, 2001, pp. 633-651.
23. T o m i t a, M. An Efficient Context-Free Parsing Algorithm for Natural Languages. – In: Proc. of 9th International Joint Conference on Artificial Intelligence, Vol. **2**, 1985, pp. 756-764.
24. W o o d s, W. A. Cascaded ATN Grammars. – Computer Linguistics, Vol. **6**, 1980, No 1, pp. 1-12.
25. D u b́ e, D., M. F e e l e y. Efficiently Building a Parse Tree from a Regular Expression. – Acta Informatica, Vol. **37**, 2000, No 2, pp. 121-144.
26. D u b́ e, D., A. K a d i r i. Automatic Construction of Parse Trees for Lexemes. – In: Proc. of Scheme and Functional Programming Workshop, 2006, pp. 51-62.
27. N o z o h o o r-F a r s h i, R. Handling of Ill-Designed Grammars in Tomita's Parsing Algorithm. – In: Proc. of 1st International Workshop on Parsing Technologies, 1989, pp. 182-192.
28. E a r l e y, J. An Efficient Context Free Parsing Algorithm. – Communication ACM, Vol. **13**, 1970, No 2, pp. 94-102.
29. N o z o h o o r-F a r s h i, R. GLR Parsing for ε-Grammers. – In: M. Tomita, Ed. Generalized LR Parsing. Boston, USA, Springer US, 1991, pp. 61-75.
30. K i p p s, J. R. Analysis of Tomita's Algorithm for General Context-Free Parsing. – In: Proc. of 1st International Workshop on Parsing Technologies, 1989, pp. 193-202.
31. T a n a k a, H., K. G. S u r e s h, K. Y a m a d a. A Family of Generalized LR Parsing Algorithms Using Ancestors Table. – IEICE Transactions on Information and Systems, Vol. **E77-D**, 1994, No 2, pp. 218-226.
32. T o m i t a, M. Graph-Structured Stack and Natural Language Parsing. – In: Proc. of 26th Annual Meeting on Association for Computational Linguistics, 1988, pp. 249-257.
33. K i p p s, J. R. Table Driven Approach to Fast Context-Free Parsing. Rand, USA, 1988.
34. N e d e r h o f, M.-J. Generalized Left-Corner Parsing. – In: Proc. of 6th Conference on European Chapter of the Association for Computational Linguistics, 1993, pp. 305-314.

35. M a c e d o, J. N., J. S a r a i v a. Expressing Disambiguation Filters as Combinators. – In: Proc. of 35th Annual ACM Symposium on Applied Computing, 2020, pp. 1348-1351.
36. N e d e r h o f, M.-J., J. J. S a r b o. Increasing the Applicability of LR Parsing. – In: Proc. of 3rd International Workshop on Parsing Technologies, 1993, pp. 187-202.
37. A l o n s o, M. A., D. C a b r e r o, M. V i l a r e s. Construction of Efficient Generalized LR Parsers. – In: D. Wood, S. Yu, Eds. Automata Implementation. Berlin, Heidelberg, Germany, Springer, 1998, pp. 7-24.
38. A y c o c k, J., R. N. H o r s p o o l. Practical Earley Parsing. – Computer Journal, Vol. **45**, 2002, No 6, pp. 620-630.
39. T o m i t a, M. Efficient Parsing for Natural Language. New York, USA, Springer, 1986.
40. S c o t t, E. SPPF-Style Parsing from Earley Recognisers. – Electronic Notes in Theoretical Computer Science, Vol. **203**, 2008, No 2, pp. 53-67.
41. N o r v i g, P. Techniques for Automatic Memoisation with Applications to Context-Free Parsing. – Computational Linguistics, Vol. **17**, 1991, No 1, pp. 91-98.
42. P a r r, T. J., R. W. Q u o n g. Adding Semantic and Syntactic Predicates to LL(*k*): Pred-LL(*k*). – In: P. A. Fritzson, Ed. Compiler Construction. Berlin, Heidelberg, Germany, Springer, 1994, pp. 263-277.
43. W ö ß, A., M. L ö b e r b a u e r, H. M ö s s e n b ö c k. LL(1) Conflict Resolution in a Recursive Descent Compiler Generator. – In: L. Böszörményi, P. Schojer, Eds. Modular Programming Languages. Berlin, Heidelberg, Germany, Springer, 2003, pp. 192-201.
44. M i c h i e, D. "Memo" Functions and Machine Learning. – Nature, Vol. **218**, 1968, pp. 19-22.
45. P a r r, T., S. H a r w e l l, K. F i s h e r. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. – ACM SIGPLAN Notices, Vol. **49**, 2014, No 10, pp. 579-598.
46. Y o u n g e r D. H. Recognition and Parsing of Context-Free Languages in Time n3. – Information and Control, Vol. **10**, 1967, No 2, pp. 189-208.
47. N e d e r h o f, M.-J., G. S a t t a. Tabular Parsing. – In: C. Martín-Vide, V. Mitrana, G. Păun, Eds. Formal Languages and Applications. Berlin, Heidelberg, Germany, Springer, 2004, pp. 529-549.
48. D o u g, A. Chart Parsing. CiteSeer, 2000, pp. 1-9.
49. S c o t t, E., A. J o h n s t o n e. GLL Parsing. Electronic Notes in Theoretical. – Computer Science, Vol. **253**, 2010, No 7, pp. 177-189.
50. S c o t t, E., A. J o h n s t o n e. GLL Parse-Tree Generation. – Science of Computer Programming, Vol. **78**, 2013, No 10, pp. 1828-1844.
51. A f r o o z e h, A., A. I z m a y l o v a. Faster, Practical GLL Parsing. – In: B. Franke, Ed. Compiler Construction. Berlin, Heidelberg, Germany, Springer, 2015, pp. 89-108.
52. S c o t t, E., A. J o h n s t o n e. Structuring the GLL Parsing Algorithm for Performance. – Science of Computer Programming, Vol. **125**, 2016, pp. 1-22.
53. G r u n e, D., C. J a c o b s. Parsing Techniques – A Practical Guide. New York, USA, Springer, 2008.
54. H a n d z h i y s k i, N., E. S o m o v a. Tunnel Parsing with the Token's Lexeme. – Cybernetics and Information Technologies, Vol. **22**, 2022, No 2, pp. 125-144.

Received: 16.11.2022; Second Version: 08.04.2023; Accepted: 18.04.2023