Nikolay Handzhiyski
Elena Somova

TUNNEL PARSING WITH COUNTED REPETITIONS

Abstract *This article describes a new and efficient algorithm for parsing (called tunnel parsing) that parses from left to right on the basis of context-free grammar without left recursion nor rules that recognize empty words. The algorithm is mostly applicable for domain-specific languages. In the article, particular attention is paid to the parsing of grammar element repetitions. As a result of the parsing, a statically typed concrete syntax tree is built from top to bottom, that accurately reflects the grammar. The parsing is not done through a recursion, but through an iteration. The tunnel parsing algorithm uses the grammars directly without a prior refactoring and is with a linear time complexity for deterministic context-free grammars.*

Keywords parsing, syntax analysis, parser generator, concrete syntax tree

Citation Computer Science 21(4) 2020: 441–462

Copyright © 2020 Author(s). This is an open access publication, which can be used, distributed and reproduced in any medium according to the Creative Commons CC-BY 4.0 License.

1. Introduction

Many software systems process data that is formatted on the basis of some formal language. The most commonly used grammars to describe formal languages are as follows: **regular grammar** – the recognition based on such grammar is done by a **finite state machine** (**deterministic** or **nondeterministic** depending on the grammar); **context-free grammar** [28] – the recognition is done by a **nondeterministic pushdown automata**; and **deterministic context-free grammar** – where the recognition is done by a **deterministic pushdown automata**.

In order to understand the meaning of the data for a given language, a recognition process must be performed – **parsing** [2]. The recognition is performed by the use of the grammar **rules**. The main goals of the parsing are as follows:

- Check that a given string of characters (for short **string**) belongs to a given language.
- Build a syntax tree (a data structure containing a syntactic information) for the string.

As a successful parsing result, two types of syntax trees can be generated – abstract and concrete. An **abstract syntax tree** may not contain all of the grammar rules that are used during the parsing nor some of the recognized characters that are implied from the context (for example, the parentheses around mathematical expressions [1]). In contrast, a **concrete syntax tree** contains all of the used rules and recognized characters during the parsing. In a source code, a syntax tree can be represented in two ways: **statically typed syntax tree** – with different data types for each rule and grammatical element; and **dynamically typed syntax tree** – the rules and the elements are represented by a common data type. During runtime, for a dynamically typed tree, many dynamic checks must be performed to distinguish the real object represented by the common data type. This article covers statically typed concrete syntax trees because they do not require additional checks for the data types of the objects stored in the tree at runtime.

Grammars often use special rules for expressing an **empty string** [2] (a sequence of zero characters). An empty string is denoted as ϵ, and the grammar rule that recognizes ϵ, will be called **ϵ-rule**. The article describes an effective algorithm for the parsing of strings that contain countable repetitions and the building of the relevant statically typed concrete syntax trees, by the use of context-free grammars without left recursion and ϵ-rules [33].

Section 2 provides an overview of the parsing process as well as common approaches of its implementation. Section 3 introduces some basic concepts and parsing problems that are relevant to the article. Section 4 describes the tunnel parsing algorithm. Section 5 contains an example parsing with various changes to the internal state of the parser as part of the **parsing machine** (PM) – an object that performs all of the recognition steps of the input string such as lexing, parsing, and the eventual build of a syntax tree. The section also contains information for the runtime speed

performance and memory usage of the algorithm. Section 6 describes the future development of the algorithm and some of its other features that are not covered by the article.

2. Overview

The extraction of a meaning from a string by the software systems is done through a process with the following steps:

1. If the input data is encoded by any character encoding standards such as ASCII, UTF-8, or UTF-16, they are decoded into characters.
2. The characters from the previous step are grouped into tokens (optionally) by a lexical grammar describing the syntax of tokens as formed by characters. The step ends with a result – a string of tokens.
3. The parsing is performed with the string of tokens as an input by using a parsing grammar that describes the syntax of the language as formed by tokens.
4. A syntax tree is generated (optionally).
5. The process ends successfully or with an error found in the input (a string that does not belong to the language).

The characters are often grouped into tokens, by the use of a regular grammar, which is converted to a nondeterministic finite automaton that can be used directly for the recognition of the tokens or to be converted to a deterministic finite automaton [25] and then be used. This conversion is often done in practice [26] by the use of the Brzozowski algorithm [7] to create minimal deterministic final automata. During the tokens recognition from the automaton, the longest possible match is often taken for each character group, which is then converted into a token for further processing. If a lexical analysis is not performed, then each character becomes a token [33].

When it is not necessary to create a syntax tree (as a part of the parsing result), the grammar can be modified by a process called refactoring [21] to remove the ϵ-rules or the left recursion in order to make the parsing possible by certain algorithms, to reduce the amount of the used memory, or to reduce the recognition time. If a detailed syntax tree is required (for the translation from one language into another, compilation, decompilation, or a certain analysis of the input data), any change in the grammar by the parsing process in order to obtain certain properties and to become suitable for parsing affects the resulting tree. In this case, the parsing process must not change the grammar. There are two main **syntax tree building algorithms**:

- **top-down** – the first used derivation [9] is the left-most one: the first created node of the tree is the root, then the left-most subnode in depth; each right subnode is created after its left sibling, as the last created node is the right-most one;
- **bottom-up** – the first used is the right-most derivation: first, the deepest nodes of the tree are created, then they are grouped into their parent nodes; the last created node is the root.

The two main **parse strategies** are obtained by combining the parse direction from left to right with the syntax tree building direction. The first strategy, **from left to right with the left-most derivation** (*LL*), enables each grammar rule to be directly implemented as a function in the target programming language and the thread-dedicated stack (call stack) to be used to recurse into the rules [18, 23]. The *LL* parsing makes it easy to add events directly to the grammar (for example, functions to be called while the parser passes through specific places in the grammar). The intuitive working method (as a standard software program) of the *LL* parsing makes it a more appropriate strategy to use. These types of parsers can be developed manually and generated automatically [4, 18, 33].

In the second strategy (from left to right with the right-most derivation – *LR*), a list of syntax tree nodes is maintained during the parsing. The two main possible operations are: a) moving the last nodes from the list as sub nodes in a new node that takes their place at the end of the list (called a reduce operation), and b) shifting of a symbol from the input string to the end of the list as a new single node (called a shift operation [3, 15]).

The necessary symbols for making the decision to move the PM from one step to another will be called **look-ahead** symbols. When the parsing is done on the basis of a deterministic context-free grammar, one look-ahead symbol (at most) is required from the PM to progress. For some context-free grammars, the number of look-ahead symbols might be greater than one.

The parsing algorithms can be classified by the type of grammar that they can use. Of practical interest are those that use context-free grammars and particularly deterministic context-free grammars, as many programming languages and structured data are represented with them [11,31,34]. There are many linear algorithms for parsing of different grammar classes by the use of different parsing strategies. For example, the parsing expression grammars (PEG) [13] target the actual parser generation (not the enumeration of all possible strings that are targeted by the context-free grammars). A PEG is never ambiguous because of the way the parsing is performed: the first match found is used to continue the parsing, and the other alternatives are not explored. The PEG parsers that use memoization [22] run in linear time and are called packrat parsers [12]. The memoization can be used to accommodate the ambiguity and left recursion in polynomial time [14]. There are general parsing strategies that can produce all possible parse trees (a parse forest). A parse forest can be efficiently represented as a "shared-packed forest" [32] by a generalized LR parser (GLR). Such a forest can be pruned after the parsing has been completed [20]. The GLR parsers may operate without an explicit lexer by the use of disambiguation filters [6]. The generalized LL parser [27] runs in the worst case with a cubic time by maintaining multiple process threads [19] to facilitate full backtracking. The generalized parsers run in linear time for deterministic grammars.

The context-free grammars (including the deterministic ones) can be a basis for parsing by a nondeterministic pushdown automaton where, for each grammar rule,

a finite-state automaton is constructed. During parsing, these parsers use a link to a **current** automaton state – one of the generated automaton states for the grammar rules. The link changes its targeted state at the parsing steps depending on the symbols in the input string and the transitions from the current to the next automaton states. When a reference to another automaton is reached in the current automaton then the currently linked automaton state is added to a stack (called a **depth stack**). Then, the parsing continues with a new linked current state, which is the beginning of the referenced automaton. When the new automaton is completed, then the previous automaton state is popped from the depth stack and the parsing continues after it.

If a state is reached where it is not possible to continue the execution for a given input symbol during the parsing, then the following actions can be performed:

- When the parsing algorithm is LL, then the depth stack can be examined for all possible symbols that can be recognized in the place of the current wrong symbol. The found list of possible symbols is then displayed to the user, which is an intuitive solution for diagnosing input string errors and is more difficult to make for a parser that uses LR parsing.
- If the parsing machine only works with automata based on a deterministic grammar, then the parsing can be terminated immediately after the first error.
- If the parsing machine operates as a nondeterministic pushdown automaton, then the implementation must step back into the current automaton (possibly using the depth stack) and search for another path where it might recognize the input symbol. The maximum number of times for which it makes sense to go back is the maximum number of looking-ahead symbols, which is sufficient for the language recognition. There are algorithms [29] that can predict at the point of the error (by heuristics or randomly) which symbol(s) should be in the place of the erroneous one, to add or remove some of the input symbols and eventually to modify the depth stack in such a way that the parsing will continue after the error.

3. Problem

The purpose of this article is to present an efficient and iterative parsing algorithm (called tunnel parsing) that supports countable repetitions as defined by the ABNF [5] standard (or any other metasyntax that has the same expressive power, such as extended BNF [17], for example, which supports a maximum occurrences of an element in its syntax but not a minimum). The result of a successful parsing with the algorithm is a concrete syntax tree that mirrors the grammar structure without losing information that can be used for a direct translation from one language into another. As defined in this article, the grammars that are accepted by the algorithm are without left recursion and ϵ-rules. The built statically typed concrete syntax trees from Tunnel Grammar Studio (TGS) [33] that implements the algorithm can be processed

quickly without dynamic checks of the data types stored in the tree and are self-sufficient – the tree contains all of the information in itself without references to other external data structures.

In tunnel parsing, as much information as possible is organized in advance from the grammar (such as rule enter/exit, alternative enter/exit, element repetitions and omissions, etc.) for a fast parsing, which can also be used at runtime by more than one PM.

A context-free grammar is defined by a tuple (N, Σ, R, S), where set N contains all non-terminal symbols [9], set Σ contains all terminal symbols, $N \cap \Sigma = \emptyset$ (empty set), set R contains all rules, and S is the start symbol of the grammar, $S \in N$. The subsequent grammars will be described with the ABNF metasyntax, where the definitions 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 non-terminal symbol ($r \in N$): a grammar rule (for short a "rule") when it is on the left side of the sign `=` or a grammar reference (for short a "reference") to a rule when it is on the right side;
- `x y` – concatenated grammar elements (for short, "elements");
- `(z w)` – defines a grammar group (for short, a "group") of elements;
- `a / b` – defines an alternative (logical *or* for the elements);
- `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), $m \in \mathbb{N}$ is the maximum repetitions (if omitted, it is considered to be an infinity), and $n \leq m$.

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 a recursive entering into the referenced rules will be called reachable [16]. Reachable terminal symbols after an element are those that can be recognized after it without using the possible depth stacks to the rule where the element is located. In Figure 1 from the beginning of rule `main` the reachable terminal symbols are: a) `"5"` in the rule itself; and b) `"5"` in rule `sub` through its reference in rule `main`.

```
main = "5" "1" / 2*4sub
sub  = "5"
```

Figure 1. Linked grammar rules

To recognize repetitions of element A in ABNF syntax $n * mA$ where $n, m \in \mathbb{N}$, $n \leq m$, $m \geq 2$, n is the minimum and m is the maximum of repetitions (which will be

called a **countable repetition**), the tunnel parsing uses an additional stack called a **repetition stack** that contains the information about the number of times element A has repeated. The definitions of n and m in this way does not encompass repetitions as `0*1A`, `*1A`, `0*A`, `*A`, `1*1A` and `1*A`, which are recognized by the appropriate arrangement of the transitions connecting the states of the automaton used for the recognition. During the construction of automaton states and transitions for any kind of an element repetition, the template in Figure 2 is used (some of the transitions may be removed depending on the values of n and m), where the operations are as follows:

- *cpush*: pushing a repetition counter with a value of one in the repetition stack;
- *cpop*: pop one repetition counter from the top of the repetition stack;
- *ctop*: the repetition counter value in the top of the repetition stack;
- *cinc*: an incrementation by one of the repetition stack top (i.e., $ctop = ctop + 1$).

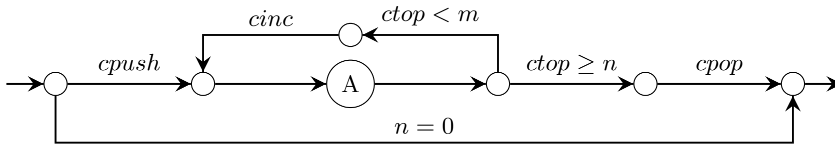

Figure 2. Template for building repeatable element automaton

The tunnel parsing is not implemented as a traditional recursive LL parser in order to avoid the drawbacks when performing a recursion with the use of the thread-dedicated stack. The algorithm is implemented as an iterative process to avoid the stack overflow of the thread-dedicated stack. To reduce the time of entering into functions in depth, all of the places in the grammar where there will be an in-depth search for a symbol are discovered at the parser generation time. For each such place, a control object is created (to be described later), that will guide the parsing machine at runtime in such a way that the search will be performed only once for each reachable terminal symbol. The tunnel parsing uses an **execution stack** that contains the information about the progress of the PM with a size that is proportional to the number of look-ahead symbols.

4. Tunnel parsing algorithm

The following steps must be performed to create and use a tunnel parsing machine: the design of automata, extraction of tunnels, construction of routers, preparation of segments, creation of a control layer, and parsing.

4.1. Design of automata

An automaton is created for each rule in the grammar (as in Figure 3) whose states will be called automaton states or only states. The transitions in these automata are of three types: a) recognizing a terminal symbol at which end there is a **terminal**

state; b) not recognizing a terminal symbol but an ϵ (i.e., no check for a terminal symbol is required to pass through them), as the transition label may indicate a certain operation on the internal state of the PM; and c) a reference to an automaton – to pass this transition, the referenced automaton must be successfully completed first. Hereafter, the "entering" and "exiting" of a rule or an alternative will mean the use of the respective transitions in the automaton built for the rule.

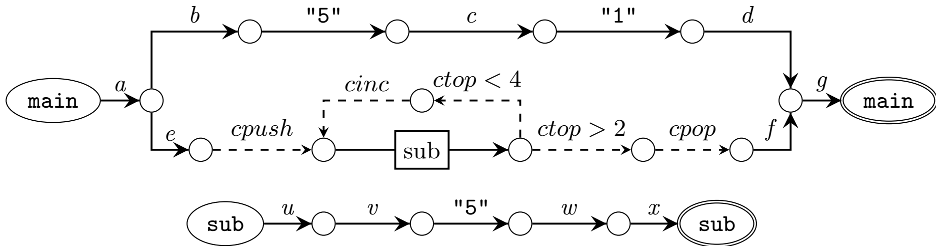

Figure 3. Automata generated from the grammar in Figure 1

In Figure 3, the dotted line transitions come from the repeatable element template in Figure 2 with transition $n = 0$ removed, as the minimum repetition is $n = 2$. The labels in the figure are as follows: a – entering into rule `main`; b and e – entering in the first and the second alternatives; c – next element; d and f – exiting from the first and the second alternatives with a success; g – exiting from rule `main` with success; u – entering in rule `sub`; v – entering into the first alternative of rule `sub`; w – exiting from the first alternative of rule `sub`; and x – exiting from rule `sub` with a success.

4.2. Extraction of tunnels

A **tunnel** is a group of operations for changing the internal state of a PM and the related operations for the syntax tree building. To enable a context-free grammar recognition, for each **forward tunnel**[1], there must be a **backward tunnel**[2]. For deterministic context-free grammars, the use of backward tunnels is not necessary.

For each rule start state, each state after a reference, and each terminal state of each automaton, all transitions to the next reachable terminal states are collected into tunnels in a depth-first search manner. In Figure 4, the dashed line shows the process of searching for and recording of the tunnels for terminal symbol `"5"`. Of all of the operations used from the template in Figure 2, only *cpush* is recorded into the tunnels. The rest of the operations are performed by the PM during the exiting of the referenced rule.The following definitions are going to 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 PM; T – the set that contains all of the tunnels $\tau \in \mathrm{T}$; $\tau = [e \mid o] - d + a$ – donates a tunnel, where the transitions that the tunnel uses are $e = \{e_1, e_2, ...\}, e_i \in E, i \in \mathbb{N}$;

[1]A tunnel that advances the parsing machine to a successful final state.
[2]A tunnel that will restore the PM as it was before the use of the forward tunnel.

the operations that change the depth stack are $o = \{o_1, o_2, ...\}, o_k \in O, k \in \mathbb{N}$; d – the number of counters that will first be removed from the repetition stack; a – the number of counters (each with a value of one) that will be added to the repetition stack; $\neg x$ – the reverse of x, where $x \in (E \cup O \cup cpush)$; $\downarrow r$ – entering into r; and $\uparrow r$ – exiting from r (after its successful recognition), where $r \in N$, $\downarrow r \in O$ and $\uparrow r \in O$.

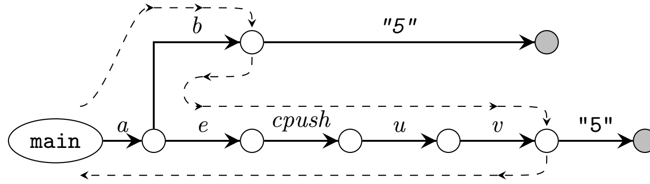

Figure 4. Search space for reachable terminal states

When $o = \emptyset$, the tunnel will be written as $[e] - d + a$, and when both d and a are zeroes, it will be written as $[e]$. For the grammar in Figure 1 with automata in Figure 3, the tunnels are as follows:

- $\tau_0 = [a, b \mid \downarrow main]$ – for an input symbol `"5"` from the beginning of rule `main`;
- $\tau_1 = [\neg b, e, u, v \mid \downarrow sub] + 1$ – if after τ_0 the parsing is unsuccessful, the PM, will attempt to recognize `"5"` at the beginning of rule `sub` by using this tunnel, whose terminal symbol is also reachable from the beginning of `main`;
- $\tau_7 = [\neg v, \neg u, \neg e, \neg a \mid \neg \downarrow sub, \neg \downarrow main] - 1$ – in case of an unsuccessful recognition after the second reachable terminal symbol `"5"` from the beginning of rule `main`, this tunnel will be used by the PM to change its internal state to the one before the execution of τ_0 and τ_1;
- $\tau_2 = [u, v \mid \downarrow sub]$ – a tunnel for entering into rule `sub` (directly or by the use of the reference to it from rule `main`) for input symbol `"5"`;
- $\tau_3 = [\neg v, \neg u \mid \neg \downarrow sub]$ – a tunnel that reverses the effect of $\tau 3$;
- $\tau_4 = [c]$ – a forward tunnel used after the recognition of `"5"` that will move the PM to the automaton state that is before the possible recognition of `"1"` in rule `main`;
- $\tau_6 = [\neg c]$ – a backward tunnel from element `"1"` to element `"5"` in rule `main`;
- $\tau_7 = [d, g \mid \uparrow main]$, $\tau_8 = [f, g \mid \uparrow main]$, $\tau_9 = [w, x \mid \uparrow sub]$ – tunnels for successful exits for rule `main` from its two alternatives as well as for rule `sub` from its single alternative; and
- $\tau_{10} = [\neg g, \neg d \mid \neg \uparrow main]$, $\tau_{11} = [\neg g, \neg f \mid \neg \uparrow main]$, $\tau_{12} = [\neg x, \neg w \mid \neg \uparrow sub]$ – tunnels for moving backwards into rule `main` in its two alternatives and into rule `sub` in its single alternative.

4.3. Construction of routers

All reachable terminal states for all key positions (the start states, the states after each reference, and each terminal state) are collected from the constructed automata.

When generating a parser that works with the tunnel parsing algorithm [33], the collected data is stored sorted in static read-only memory to speed up the search for a next state of the PM at runtime. In Figure 4, the dark automaton states are the reachable terminal states from the start of the automaton generated for rule `main`.

The object that contains the information about the sorted states (by the value of the transition's terminal symbol that led to the respective terminal state) will be called a **router** and each of its elements a **path**. Thus, by having the tunnels and the routers before the start of the parsing, there is enough information on how the PM will progress.

The routers related definitions are as follows: U – the set of all routers in a PM; $\sigma \in \Sigma$ – terminal symbol; C – the set of all control states; a control state – $c \in C$; P – the set of all paths in a router; p – a path into a router as a pair of a terminal symbol and a control state (described later): $\sigma \to c$; $u = \langle P \mid c_\epsilon \rangle$ – a router where $u \in U$, $c_\epsilon \in C$, as c_ϵ will be used when the terminal symbol is not found in P.

The routers for the grammar in Figure 1 with the automata in Figure 3 are as follows:

- $u_0 = \langle \texttt{"5"} \to \mathrm{c}_7 \mid \rangle$ – with reachable terminal states from the beginning of rule `main`;
- $u_1 = \langle \texttt{"5"} \to \mathrm{c}_5 \mid \rangle$ – with reachable terminal states from the beginning of rule `sub`;
- $u_2 = \langle \mid \mathrm{c}_{11} \rangle$ – without reachable terminal states but with an exit path after `"1"` in rule `main`;
- $u_3 = \langle \texttt{"1"} \to \mathrm{c}_8 \mid \rangle$ – with reachable terminal states after `"5"` in rule `main`;
- $u_4 = \langle \mid \mathrm{c}_{13} \rangle$ – without reachable terminal states but with an exit path after `"5"` in rule `sub`.
- $u_5 = \langle \texttt{"5"} \to \mathrm{c}_9 \mid \rangle$ – with reachable terminal states from the repetition of the reference to rule `sub` in rule `main` before the minimum occurrences have been collected;
- $u_6 = \langle \texttt{"5"} \to \mathrm{c}_{10} \mid \rangle$ – with reachable terminal states from the repetition of the reference to rule `sub` in rule `main` after the minimum and before the maximum occurrences have been collected;
- $u_7 = \langle \mid \mathrm{c}_{12} \rangle$ – without reachable terminal states but with an exit path after the reference to rule `sub` in rule `main`.

4.4. Preparation of segments

A **segment** will be called an object that exists for each rule reference. It has a link to a router with the next reachable terminal states after the corresponding reference. If the corresponding reference has a countable repetition, then other routers containing the reachable terminal states from the beginning of the referenced rule are also linked to by the segment. For example, the grammar in Figure 1 has one segment, which uses router u_5 before the minimum occurrences are collected, router u_6 after the minimum but before the maximum occurrences are collected, and router u_7 with reachable

terminal states after the reference. To be able the PM to control the repetitions, the segment contains the minimum and the maximum numbers of repetitions defined for the reference in the grammar.

The **depth stack** in the tunnel parsing algorithm consists of segments. To enable the PM to progress backwards to its previous internal states, two additional stacks exist that are used to archive a portion of the depth stack and the repetition stack and will be called a **depth stack archive** and a **repetition stack archive**, respectively.

When a PM exits a rule (after its successful recognition) the removed element from the depth stack is not deleted but rather moved to the archive. To control the backwards progress distance, an integer counter is placed in each element of the execution stack to count how many elements are moved from the depth stack to the archive. When there is a backwards progress, the PM will restore the depth stack from its archive with as many items as the value of that counter. The repetition stack is used similarly. For example, upon exiting from the template in Figure 2 through the transition with the *cpop* label, a counter will be moved from the repetition stack to the repetition stack archive, and when the transition is used backwards, the repetition counter from the repetition stack archive will be moved back to the repetition stack.

4.5. Creation of control layer

To control the execution of the PM, a set of objects is created that use the tunnels and the routers to form a **control layer**. Each of these **control objects** can be in one of several **control states** (their number depends on the object type) that are used one after another depending on the input symbols. Each execution stack element uses one control state per an input symbol. At any given time, no more than the maximum look-ahead symbols plus one of stack elements are needed for the algorithm to operate. The PM performs the required operations based only on the top of the execution stack. After each execution of the operations defined by a control state the PM may pause, as this is one iterative step in practice. The control objects signify the information to "where" in the automata the PM has reached, and the control states – "which" operations must be performed. In this article, the following control objects and their states are presented:

- **c-origin** – created for each rule with a link to a router with all reachable terminal symbols from the beginning of the respective rule; the object has one state: "use";
- **c-terminal** – created for each terminal state that has one control state: "use"; it has a link to a router with all reachable terminal symbols after the respective terminal state;
- **c-token** – created for each terminal symbol (without a countable repetition) that can be found by a router search; there are two control states: a) "use" – the PM in this state moves with one input symbol forward, and b) "used" – after a subsequently unsuccessful recognition attempt, the PM in this c-state performs operations to restore its internal state to the one before the "use" c-state;

- **c-list** – created when there is a countable repetition of an element; the object has four states: “use” – the PM in this state prepares to parse a repetition (by pushing one in the repetition stack), moves forward to the next input symbol, and changes its state to “repetition” – used to recognize each subsequent repetition of the symbol (then, the top of the repetition stack is incremented by one), “back” – when a symbol different than the expected one appears, the PM moves one input symbol back (then, the top of the repetition stack is decremented by one), and “used” – the PM performs operations to restore its internal state to the one before the repetitions began (one repetition counter is removed from the top of the repetition stack);
- **c-epsilon** – created when there is a path with ϵ transitions from a terminal state or an automaton state at the end of a rule reference transition to the end of the automaton; it has one control state exist: “use”;
- **c-back-start** – created to be used after all reachable (from the start of a rule) terminal symbols are iterated and none led to the successful recognition of the input; there is one control state: “use”;
- **c-back-token** – created to be used after the recognition of a terminal symbol that is not part of a countable repetition; there is one control state: “use”;
- **c-back-list** – created analogously to c-back-token and used after the passing from a reference with a countable repetition to another reachable element where the parsing has failed and the PM goes back; in this step, one counter is restored from the repetition stack archive to the repetition stack;
- **c-back-minimum** – created for each rule when there is at least one reference to it with $minimum > 1$ occurrences; the PM in this c-state will decrement the repetition counter used and replace the top of the execution stack with c-restore (described later);
- **c-back-middle** – created for each rule when there is at least one reference to it with $maximum > 1$ and $maximum > minimum$ occurrences; this control state is the same as c-back-minimum, as additionally after the counter is decremented, the PM will search for terminal states after the respective reference;
- **c-unwind** – a global control object for the entire PM with one state – “use”, which is placed on top of the execution stack after the use of c-epsilon; the PM in this c-state removes one element from the depth stack and adds it to the archive depth stack as well as increases the exit counter by one; if the element after which the rule is exited has a countable repetition, then one repetition counter moves to the archive repetition stack from the repetition stack;
- **c-restore** – a global control object for the entire PM with one state – “use”, which restores one or more depth stack elements from the depth stack archive and decreases the exit counter by one; if the restored segment has a countable repetition, one item from the repetition stack archive is restored to the repetition stack; the object remains on top of the execution stack until the exit counter reaches zero.

In Table 1, the control objects for the grammar in Figure 1 are written along with their relationships. Routers u_5 and u_6 are copies of router u_1 together with the respective control states c_9 and c_{10} instead of c_5. That makes the lists of control objects: (c_9, c_{17}) – used on a repetition attempt of the reference to rule `sub` in rule `main`, **before** the minimum occurrences are collected and (c_{10}, c_{18}) – used on a repetition attempt of the reference to rule `sub` in rule `main`, **after** the minimum occurrences are collected. Both lists are based on list (c_5, c_{15}) that would be used if the start rule was `sub` with first input symbol `"5"`.

Table 1
Control objects for the grammar in Figure 1

Type c-origin	
#	Router
0	u_0
1	u_1

Type c-terminal	
#	Router
2	u_2
3	u_3
4	u_4

Type c-token			
#	Next	c-terminal	Tunnel
5	c_{15}	c_4	τ_2
6	c_{14}	c_4	τ_1
7	c_6	c_3	τ_0
8	c_{16}	c_2	τ_4
9	c_{17}	c_4	τ_2
10	c_{18}	c_4	τ_2

Global c-objects	
#	Type
19	c-unwind
20	c-restore

Type c-epsilon		
#	Forward	Backward
11	τ_7	τ_{10}
12	τ_8	τ_{11}
13	τ_9	τ_{12}

Type c-back-*		
#	*	Tunnel
14	start	τ_7
15	start	τ_3
16	token	τ_6
17	minimum	τ_3
18	middle	τ_3

Note that there is a list (c_7, c_6, c_{14}) that is pointed to by router u_0. By the use of this list the PM will effectively iterate the reachable terminal symbols `"5"` from the start of rule `main`, which will be demonstrated later.

4.6. Parsing

A direct real-time parsing is performed by an interpreter, or a parser is generated to a source code for the target programming language that can be embedded in other software tools. TGS has a debugger with an integrated interpreter, which performs the parsing in real-time and visually builds a syntax tree in forward and backward steps for a given grammar and an input. TGS also generates parsers that operate on the base of the tunnel parsing algorithm. At runtime, such a parser builds a statically typed concrete syntax tree by default 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 is removed from the tunnels, an abstract dynamically typed syntax tree with a different level of abstraction can be built.

In tunnel parsing, the number of operations that the PM performs at each iterative step is independent from the number of input symbols. This enables the PM

to pause and resume its execution almost instantly. If this is not necessary, a good optimization of the algorithm [33] is the implementation to perform several iterative steps in a sequence before returning the control to the user of the PM.

5. Results

An example of a tunnel parsing algorithm runtime execution for the grammar in Figure 1 with automata in Figure 3, initial rule $S = \texttt{main}$, and three characters of input data (`"555"`) is presented in Table 2.

Table 2

Execution of a PM for the grammar in Figure 1

#	Input	Execution Stack	Depth	Repeat	Task
1	.555	$c_0 \vert use$	$\emptyset$	$\emptyset$	search in u_0 and found c_7
2	.555	$c_7 \vert use$	$\emptyset$	$\emptyset$	use of τ_0
3	.555	$c_7 \vert use$	$\emptyset$	$\emptyset$	rule enter
4	.555	$c_7 \vert use$	`main`	$\emptyset$	next token
5	5.55	$c_7 \vert use$	`main`	$\emptyset$	control state change
6	5.55	$c_7 \vert used$	`main`	$\emptyset$	control state addition
7	5.55	$c_7 \vert used, c_3 \vert use$	`main`	$\emptyset$	search in u_3 and not found
8	5.55	$c_7 \vert used, c_3 \vert use$	`main`	$\emptyset$	token back, c-state remove
9	.555	$c_7 \vert used$	`main`	$\emptyset$	next control state
10	.555	$c_6 \vert use$	`main`	$\emptyset$	use of τ_1
11	.555	$c_6 \vert use$	`main`	1	rule enter, next token
12	5.55	$c_6 \vert use$	`main,sub`	1	control state change
13	5.55	$c_6 \vert used$	`main,sub`	1	control state addition
14	5.55	$c_6 \vert used, c_4 \vert use$	`main,sub`	1	search in u_4 and found c_{13}
15	5.55	$c_6 \vert used, c_{13} \vert use$	`main,sub`	1	use of τ_9
16	5.55	$c_6 \vert used, c_{13} \vert use$	`main,sub`	1	control state change
17	5.55	$c_6 \vert used, c_{19} \vert use$	`main,sub`	1	rule exit
18	5.55	$c_6 \vert used, c_{19} \vert use$	`main`	1	search in u_5 and found c_9
19	5.55	$c_6 \vert used, c_9 \vert use$	`main`	1	use of τ_2
20	5.55	$c_6 \vert used, c_9 \vert use$	`main`	2	rule enter
21	5.55	$c_6 \vert used, c_9 \vert use$	`main,sub`	2	next token
22	55.5	$c_6 \vert used, c_9 \vert use$	`main,sub`	2	control state change
23	55.5	$c_6 \vert used, c_9 \vert used$	`main,sub`	2	control state addition
24-33: similar to rows 14 to 23 inclusive, with a search $[u_6 \rightarrow c_{10}]$ instead of $[u_5 \rightarrow c_9]$					
34-37: similar to rows 14 to 17 inclusive					
38	555.	..., $c_{10} \vert used, c_{19} \vert use$	`main`	3	search in u_6 and not found
39	555.	..., $c_{10} \vert used, c_{19} \vert use$	`main`	3	search in u_7 and found c_{12}
40	555.	..., $c_{10} \vert used, c_{12} \vert use$	`main`	3	repetition archive
41	555.	..., $c_{10} \vert used, c_{12} \vert use$	`main`	$\emptyset$	use of τ_8
42	555.	..., $c_{10} \vert used, c_{12} \vert use$	`main`	$\emptyset$	control state change
43	555.	..., $c_{10} \vert used, c_{19} \vert use$	`main`	$\emptyset$	rule exit
44	555.	..., $c_{10} \vert used, c_{19} \vert use$	$\emptyset$	$\emptyset$	success

An alternative execution of the described is first to recognize `"5"` through the reference to rule `sub`. This is an alternation in a different order of the reachable

terminal symbols in the grammar. The alternation can be in any order when there are many duplicate reachable terminal symbols in a router. This is correct from the point of view of the defined grammar, but it is not completely intuitive to the user. If this is ignored, it is possible to profile the parsing of a large amount of data in order to determine which reachable terminal symbols have led to a successful recognition more often. Then, the order of the duplicate terminal symbols in the routers can be changed to speed up the parsing of a profile-like input.

Table 2 contains step-by-step changes on the internal state of the PM in each row. The content of the cells in column "Task" signifies the operation(s) performed by the PM to move from the current row to the next. The overall description of the events is as follows: a) the parsing starts with a search for a tunnel to use, from the start of rule `main`; b) tunnel c_7 is found and used; c) the next token is loaded and it is used to search for the next tunnel in router u_3 that has all reachable terminal states after `"5"` in rule `main`; d) no such tunnel is found, so the PM will try the next control state of c_6 which is c_7; e) the use of c_6 moves the PM into rule `sub` and one repetition of the reference is accounted for; f) because there is no next reachable terminal symbols after `"5"` in rule `sub`, the PM will exit the rule with the use of global control object c_{19}; g) because the repetition counter's value on the top of the repetition stack is **one** and it is less then the minimum required (**two**), the minimum repetition router u_5 for rule `sub` is used from the PM; h) tunnel c_9 is found and used; i) the parsing will continue until the next repetition of the reference – then, the PM uses router u_6 to search for a tunnel and finds c_{10}; j) the next time a repetition is attempted, it will fail because there are no more input tokens to use, and router u_7 will guide the PM after the reference; and k) the parsing completes successfully when the depth stack is empty and there are no more tokens to use.

As described in this article, the parsing can benefit from various optimizations, which are a consequence of how the algorithm operates:

- The recognition process can be divided into parts – lexical analysis, parsing, and syntax tree building, which can be run by different program threads so that a multi-threaded linear parsing is obtained as implemented in [33].
- The lexical analysis and the parsing can be moved to a separate library to be used by more than one runtime client with only the syntax tree being in the client program. The library and client program can be in different programming languages.
- The implementation of the tunnel parsing algorithm can be in the form of an online algorithm[3] as implemented in [33].

An experiment was made with the grammar in Figure 5, which contains 27 rules[4] and defines a language consisting of one word – a sequence of one or more lowercase

[3] An algorithm property to stop when there is not enough input data and to continue when new data is available.

[4] The three dots represent the missing rules from C to X that each recognizes one letter and references the next rule alphabetically.

letters from the English alphabet. The grammar has an ABNF syntax defined in [5] and upgraded by [8]. For the parser generators that do not accept it in this format, it is translated into the syntax of these parser generators. The purpose of the experiment is to compare the speed of different PMs when they are searching in depth and are choosing their next internal state from a large number of possible terminal symbols. The parser generators have the ability to move some computations described by this grammar in the lexical analysis. However, the purpose of the test is to measure the parsing speed, not the lexing speed – so, there is no extensive lexical analysis in the test.

```
document = 1*a
a = %s"a" / b
b = %s"b" / c
...
y = %s"y" / z
z = %s"z"
```

Figure 5. Performance test grammar

The results from the test are presented in Figure 6: in Figure 6a, the parsing throughput (in megabytes); in Figure 6b, the parsing plus the building of the syntax tree; and in Figure 6c, the memory used as the difference between the end and the start of the measured parsing period. All of the PMs are compiled by Microsoft® Visual Studio® 2015 Update 3, on C++ for a 64-bit processor, optimized for speed in release. The executables are executed in Microsoft® Windows® 10, a 64-bit operation system. The used hardware is Intel®Core™ i7-4790k @4GHz. The experimental input is 26 sequences of 1 million lowercase letters from the English alphabet. The first input (Depth 1) is with lowercase letters a, the second input (Depth 2) is with the letter b, and so on until z (Depth 26). Each value plotted in Figure 6 is the average of ten consecutive executions of the executable (containing the compiled PM) without filtering any values. This resembles a real work process when an external program starts a compiler or an interpreter (which have the PM) for many inputs in a row. The input data is preloaded into the operative memory, and only the recognition time (any processing of the input data until the end of the parsing) is measured.

In Figure 6, the labels are: TGS – v1.0.50 [33] (the postfix S means statically typed concrete syntax tree, and postfix D means dynamically typed concrete syntax tree), JavaCC and JJTree – v7.0 [10, 18, 30], and ANTLR – v4.8 [4, 23, 24]. Both the JavaCC and ANTLR generated parsers, built dynamically typed concrete syntax trees for this experiment.

The results show that TGS with the tunnel parsing algorithm is significantly superior in performance to the other PMs for this grammar. For example, when comparing TGS and JavaCC in a parse mode without a syntax tree generation

(see Figure 6a), the generated PM by the TGS parser parses especially faster when the depth is less then 15.

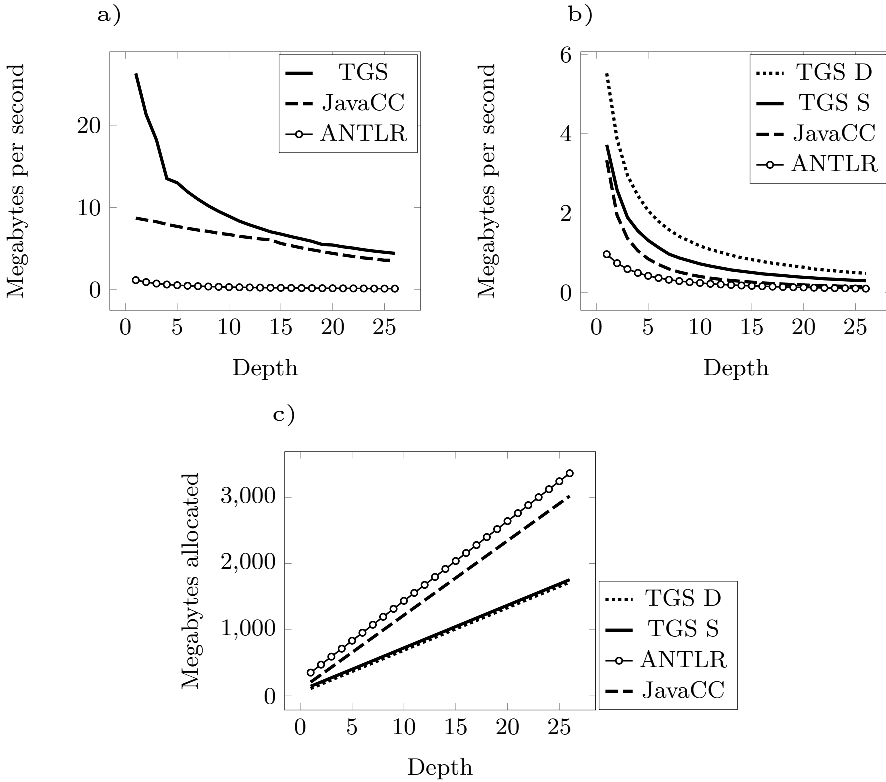

Figure 6. PMs comparison for the grammar in Figure 5: a) parsing; b) parsing plus tree generation; c) used memory

In Figure 6b, the fastest tree to be completed is the dynamically typed one from the PM generated by TGS, despite the fact that the used memory is nearly as much as that used for the statically typed tree, visible in Figure 6c. The concrete trees generated by TGS and JavaCC are next by the building speed, and the slowest built with the highest memory usage is made by ANTLR.

The final TGS syntax tree contains each input character in itself, which makes the tree generated by TGS self-sufficient (without any pointers to external data). For a contrast, the tree generated by JavaCC+JJTree contains pointers to the first and last token in each rule, which prevents the memory for the input string associated structures to be released after the tree has been generated. When the tree is self--sufficient, the tree builder and the parsing process itself can be in different programs and written in different programming languages, which is one of the goals of tunnel parsing.

For deterministic grammars, the tunnel parsing algorithm has a linear execution time relative to the number of input symbols because of the following:

- The PM progress is in iterative steps where each iterative step realizes the operations defined by the current execution stack top. As defined in Section 4.5, none of these operations is dependent on the input length but only on the current input symbol (for an eventual search in the routers).
- The control states that are placed on the top of the execution stack are never repeating because a) each control object has states that are used one after another and never loop, and b) the c-token control objects will have a next control object that is one of the c-back type control objects.

Additionally, the following observations can be made: a) the maximum number of control states executed for any deterministic grammar are linear to the input length but are not evenly distributed per token (notably, for the right recursive grammars); b) the stepping backwards into the input string and the again forward (Table 2, Rows 8 and 11) could be optimized (made "lazy", for example) from the implementation. This will effectively make the parsing for $LL(k > 1)$ possible with $k - 1$ tokens of look-ahead. The current implementation in TGS moves explicitly in the input because this optimization is possible only for some grammars (such as the grammar in Figure 1); however, this might change in the future.

In practice [33], not only the parsing but also the syntax tree construction, destruction, and its conversion (printing) to a string must be iterative to ensure a robust runtime. An automatic synchronization [35] of the previously generated trees and reflective printing [36] might also be desirable.

6. Conclusion

This article described an algorithm for the parsing of domain-specific languages as programming languages and data structures. The languages are defined by a context--free grammar without left recursion and ϵ-rules. As a result of the parsing, a concrete syntax tree can be built from top to bottom. In the article, particular attention has been paid to the processing of the repetitions of the grammar elements. The algorithm does not change the grammar prior to the parsing. For this reason, the resulting syntax tree reflects the grammar precisely. In the tunnel parsing all operations are performed iteratively to avoid the potential overflow of the thread-dedicated stack. As defined in the article, the algorithm has a linear execution time when operating on the basis of a deterministic context-free grammar (the most commonly used in practice) and with an exponential time in the worst case for some nondeterministic context-free grammar.

The use of tunnels speeds up the parsing because all of the necessary changes to the internal state of the PM are executed at once for each reachable terminal symbol (with the use of a tunnel) without a depth search in the automata by using the thread-dedicated stack. A PM based on the algorithm uses the control objects, their states,

the tunnels, and the routers to switch from one internal state to another. If there is no need to create a syntax tree, then there is no need to store its build information into the tunnels. This further reduces the amount of the generated code and speeds up the parsing.

In many programming languages, when a function is called, the memory is allocated on the thread-dedicated stack for all variables of the function that could be used. However, not all of them are actually used in every function call. When performing recursive calls of functions, a lot of memory on the stack might be reserved for the function variables (which will not actually be used). In tunnel parsing, the required data (arranged in stacks) is only allocated when it is necessary and deleted when it is no longer needed. This is important for the embedded microcontrollers, which often have a little operative memory.

The presented algorithm describes an execution of a PM through an iteration, as the depth relationships between the terminal symbols are accounted at the same time. The iterative execution avoids the problem of overflowing the thread-dedicated stack by replacing it with dynamic stacks, which are limited only by the total available operative memory. The algorithm also can parse some ambiguous grammars with a linear execution time by choosing one of the many possible ϵ transitions for a *c*-epsilon control object.

A natural subsequent evolution of the current work is the extension of the presented algorithm with the ability to recognize context-free grammars that have ϵ-rules or left recursion.

Contribution

Nikolay Handzhiyski developed the theory by the supervision and encouragement of Elena Somova, based on his previously existing software implementation in Tunnel Grammar Studio [33]. Nikolay Handzhiyski performed the tests and (along with Elena Somova) verified the results. The authors discussed the theory and the results and contributed to the final article.

References

[1] Abstract Syntax Tree Metamodel. https://www.omg.org/spec/ASTM/. Accessed: 2020-04-14.

[2] Aho A.V., Lam M.S., Sethi R., Ullman J.D.: *Compilers. Principles, Techniques, and Tools (Second Edition)*, 2007.

[3] Aho A.V., Johnson S.C.: LR Parsing, *ACM Computing Surveys*, vol. 6(2), pp. 99–124, 1974. https://doi.org/10.1145/356628.356629

[4] ANother Tool for Language Recognition (ANTLR). https://www.antlr.org/. Accessed: 2020-04-12.

[5] Augmented BNF for Syntax Specifications: ABNF. https://tools.ietf.org/html/rfc5234. Accessed: 2020-04-14.

[6] Brand van den M.G.J., Scheerder J., Vinju J.J., Visser E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: *Proceedings of the 11th International Conference on Compiler Construction*. Springer-Verlag, 2002.

[7] Brzozowski J.A.: Canonical regular expressions and minimal state graphs for definite events. 1962.

[8] Case-Sensitive String Support in ABNF. https://tools.ietf.org/html/rfc7405. Accessed: 2020-04-14.

[9] Chomsky N.: On certain formal properties of grammars, *Information and Control*, vol. 2(2), pp. 137–167, 1959.

[10] Copeland T.: *Generating Parsers with JavaCC: An Easy-to-Use Guide tor Developers*, Centennial Books, 2nd ed., 2007.

[11] Extensible Markup Language (XML). https://www.w3.org/TR/xml/. Accessed: 2020-04-14.

[12] Ford B.: *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking.* Ph.D. thesis, Massachusetts Institute of Technology, 2002.

[13] Ford B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation, *ACM SIGPLAN Notices*, vol. 39, pp. 111–122, 2004. https://doi.org/10.1145/964001.964011.

[14] Frost R.A., Hafiz R.: A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time, *SIGPLAN Notices*, vol. 41(5), pp. 46–54, 2006. https://doi.org/10.1145/1149982.1149988.

[15] Grune D., Jacobs C.J.H.: *Parsing Techniques: A Practical Guide*, Ellis Horwood, 1990.

[16] Hopcroft J.E., Ullman J.D.: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, 1979.

[17] ISO/IEC 14977:1996(E) Information technology – Syntactic metalanguage – Extended BNF. http://standards.iso.org/ittf/PubliclyAvailableStandards/. Accessed: 2020-04-14.

[18] Java Compiler Compiler (JavaCC). https://javacc.org/. Accessed: 2020-04-12.

[19] Johnstone A., Scott E.: Modelling GLL Parser Implementations. In: Malloy B., Staab S., van den Brand M. (eds.), *Software Language Engineering. SLE 2010. Lecture Notes in Computer Science*, vol. 6563. Springer, Berlin, Heidelberg, pp. 42–61, 2011.

[20] Macedo J.N., Saraiva J.: Expressing Disambiguation Filters as Combinators. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing.* Association for Computing Machinery, 2020.

[21] Moore R.C.: Removing Left Recursion from Context-Free Grammars. In: *Proceedings of the 1st North American chapter of the Association for Computational Linguistics Conference*, 2000.

[22] Norvig P.: Techniques for automatic memoization with applications to context--free parsing, *Computational Linguistics*, vol. 17(1), pp. 91–98, 1991.

[23] Parr T.: *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers)*. Pragmatic Bookshelf, 2010.

[24] Parr T.: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd ed., 2013.

[25] Rabin M., Scott D.: Finite Automata and Their Decision Problems, *IBM Journal of Research and Development*, vol. 3(2), pp. 114–125, 1959.

[26] Saraiva J.: HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. In: *Proceedings of the ACM Workshop on Functional and Declarative Programming in Education*, University of Kiel Technical Report 0210. 2002.

[27] Scott E., Johnstone A.: GLL Parsing. In: *Electronic Notes in Theoretical Computer Science*, vol. 253(7), pp. 177–189, 2010.

[28] Sipser M.: *Introduction to the Theory of Computation*. Course Technology, 2nd ed., 2006.

[29] Spenke M., Mühlenbein H., Mevenkamp M., Mattern F., Beilken C.: A language independent error recovery method for LL(1) parsers, *Software: Practice and Experience*, vol. 14, 1984.

[30] Succi G., Wong R.W.: The application of JavaCC to develop a C/C++ preprocessor, *SIGAPP Applied Computing Review*, vol. 7(3), pp. 11–18, 1999.

[31] The JavaScript Object Notation (JSON) Data Interchange Format. https://tools.ietf.org/html/rfc7159. Accessed: 2020-04-14.

[32] Tomita M.: *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[33] Tunnel Grammar Studio. https://www.experasoft.com/products/tgs/. Accessed: 2020-04-14.

[34] Uniform Resource Identifier (URI): Generic Syntax. https://tools.ietf.org/html/rfc3986. Accessed: 2020-04-14.

[35] Zhu Z., Ko H., Zhang Y., Martins P., Saraiva J., Hu Z.: Unifying Parsing and Reflective Printing for Fully Disambiguated Grammars, *New Generation Computing*, vol. 38, pp. 423–476, 2020.

[36] Zhu Z., Zhang Y., Ko H.S., Martins P., Saraiva J., Hu Z.: Parsing and reflective printing, bidirectionally. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ACM, 2016.

Affiliations

Nikolay Handzhiyski
ExperaSoft UG (haftungsbeschraenkt), 10 Goldasse St., 77652 Offenburg, Germany;
University of Plovdiv "Paisii Hilendarski", 24 Tzar Assen St., 4000 Plovdiv, Bulgaria,
nikolay.handzhiyski@experasoft.com, ORCID ID: https://orcid.org/0000-0003-0681-6871

Elena Somova
University of Plovdiv "Paisii Hilendarski", 24 Tzar Assen St., 4000 Plovdiv, Bulgaria,
eledel@uni-plovdiv.bg, ORCID ID: https://orcid.org/0000-0003-3393-1058

Received: 15.04.2020
Revised: 17.07.2020
Accepted: 24.07.2020