Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded “Any” symbol

A.V. Goloveshkin, ORCID: 0000-0001-6947-0594 <alexeyvale@gmail.com>
Vorovich Institute for Mathematics, Mechanics and Computer Science, Southern Federal University, 8a, Milchakova st., Rostov-on-Don, 344090, Russia

Abstract. Tolerant parsing is a form of syntax analysis aimed at capturing the structure of certain points of interest presented in a source code. While these points should be well-described in a tolerant grammar of the language, other parts of the program are allowed to be described coarse-grained, thereby parser remains tolerant to the possible variations of the irrelevant area. Island grammars are one of the basic tolerant parsing techniques. “Islands” term is used as the relevant code alias, the irrelevant code is called “water”. Efforts required to write water rules are supposed to be as small as possible. Previously, we extended island grammars theory and introduced a novel formal concept of a simplified grammar based on the idea of eliminating water description by replacing it with a special “Any” symbol. To work with this concept, a standard LL(1) parsing algorithm was modified and LanD parser generator was developed. In the paper, “Any”-based modification is described for LR(1) parsing algorithm. In comparison with LL(1) tolerant grammars, LR(1) tolerant grammars are easier to develop and explore due to solid island rules. Supplementary “Any” processing techniques are introduced to make this symbol easier to use while staying in the boundaries of the given simplified grammar definition. Specific error recovery algorithms are presented both for LL and LR tolerant parsing. They allow one to further minimize the number and complexity of water rules and make tolerant grammars extendible. In the experiments section, results of a large-scale LL and LR tolerant parsers testing on the basis of 9 open-source project repositories are presented.

Keywords: tolerant parsing; robust parsing; lightweight parsing; partial parsing; island grammars; simplified grammar; LanD parser generator

For citation: Goloveshkin A.V. Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded “Any” symbol. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019, pp. 7-28. DOI: 10.15514/ISPRAS-2019-31(3)-1

1. Introduction
Tolerant parsing is a syntax analysis technique differing from the detailed whole-language (so-called baseline) parsing. The latter is performed by a full-featured compiler of a certain programming language to ensure the program satisfies the grammar and to prepare an internal program representation for some further steps. Tolerant parsing performs deep structural analysis only on certain parts of the program, passing other parts with minimal effort. It is achieved by generating the corresponding parser from a tolerant grammar, where these parts of interest are described in details and some minimal description of the irrelevant area is provided. From developer's perspective, tolerant parsing allows her to focus on the structure of the points valuable in the context of a current task, without worrying about irrelevant code variations. Among tolerant parsing use cases, the following ones are the most frequently mentioned:

- **Baseline grammar inaccessibility**: Full version of the language grammar can be inaccessible due to proprietary issues or manual baseline parser writing [1]. Besides, physical accessibility does not assume accessibility in terms of grammar comprehension. Baseline grammar usage requires intensive exploration to detect rules describing constructs of interest. Tolerant grammar structure and mapping between its entities and language constructs are transparent to the developer, as she writes it according to her own knowledge of the task and the language.

- **Language embedding**: Some program artifacts assume the usage of multiple languages in one source file. In this case, a parser for the relevant language must be tolerant to all the snippets written in other languages [2].

- **Domain-specific idioms**: In a certain project, some local domain-specific patterns can be applied [1, 3]. They represent a high-level abstraction layer which is not presented in the language syntax and obviously is out of scope of the whole-language parser. Nevertheless, tolerant parsers can be strictly focused on these patterns, ignoring the underlying structure. According to the island grammars tolerant parsing paradigm [1, 3], parts of the program that are well-described in the grammar are called islands, others are called water. Detailed grammar rules describing islands are named patterns, water is presented with as few liberal productions as possible.
However, sometimes it is required to describe some water parts in a fine-grained island-like style to avoid confusion with proper islands. Water parts mistaken for islands are called antipatterns. Island grammar development is always a finite iterative process consisting of in-the-wild parser testing and subsequent patterns and antipatterns refinement. Besides, some situations, when program entity can be treated as an island and as a water at the same time, are typically solved with generalized parsing algorithms [4, 5]. The author of the current paper is interested in tolerant parsing because of the long-term goal to develop a multi-language tool for concern-based markup of software projects. Talking about a program as a set of functionalities, so-called concerns, we may notice that many of them are implemented with pieces of code which are spread across solid program elements, such as classes or methods [6, 7]. These concerns are called vertical layers [8] or crossing concerns [9]. To work with this kind of concerns, it is vital to create and manipulate some meta-information about their location, this information should be sustainable with respect to code changes, so it cannot rely on text line and text column numbers. Abstract syntax tree is considered to be a more appropriate structure for this location binding, so, there must be a set structure for different languages, these parsers must build abstract syntax trees in one unified format. These trees should capture only the structure of program entities we plan to bind to, therefore, tolerant parsing is an option. It also should be easy to support new languages by developing additional grammars and generating tolerant parsers. Previously, to meet the requirements for parsers and trees, we developed a tolerant parser generator called LanD. It uses a modified LL(1) parsing algorithm which is theoretically and experimentally proved to be correct [10]. The contributions of this paper are: 1) a modified LR(1) parsing algorithm with incorporated notion of a special Any token allowing parser to match implicitly defined token sequences; 2) supplementary Any processing techniques for modified LL(1) and LR(1) parsing algorithms, filling the gap between the simplified grammar formal definition and real tolerant parsing use cases; 3) specific Any-based LL and LR error recovery mechanisms aimed at elimination of water rules and correct handling of possible ambiguities without parsing algorithm generalization; complexity analysis is also carried out; 4) lightweight LL(1) and LR(1) grammars for a broad range of languages, namely, for C#, Java, PascalABC.NET programming languages, Yacc and Lex specification formats, XML and Markdown markup languages; 5) an experimental evidence of the applicability of the generated tolerant parsers for large-scale software projects analysis. The remainder of the paper is organized as follows. In Section 2, main goals of the current research are listed. A brief overview of the previous author’s research, along with closest analogues analysis, is provided in Section 3. In Section 4, a modification of the standard LR(1) parsing algorithm aimed at Any symbol processing is introduced, Any implementation improvements and issues addressed are discussed. Novel Any-based error recovery algorithms are described. Section 5 includes a sufficient volume of experimental data obtained by applying generated tolerant parsers for C# and Java languages to a number of real-world software repositories. In Section 6, a brief summary of the contribution of the paper is provided along with future work outlining.

2. Problem statement

The first assumption of the current research is that the concept of Any, previously successfully embedded into a top-down parsing, can be embedded in a bottom-up parsing too, making tolerant grammars more expressive and easy-to-write. The second assumption is that ambiguities originated in islands and water similarity can be resolved not only by adding special antipatterns or by generalized algorithms usage, but also by a special recovery mechanism embedded in a deterministic parsing.

The key goals of the current research are:

1) to design an LR(1) parsing algorithm with built-in notion of a special Any grammar symbol that provides skipping the token sequences that are not explicitly described in the grammar;
2) to introduce into the LanD parser generator additional capabilities for correct Any processing in case Any usage does not fully satisfy simplified grammar formalization;
3) to design specific error recovery mechanisms for LL(1) and LR(1) tolerant parsing, aimed at handling ambiguities originating in water and island similarity;
4) to implement tolerant island grammars for a broad range of languages;
5) to evaluate parser’s applicability through the analysis of large-scale software projects written in C# and Java languages.

3. Related work

3.1 «Any» implementation

The concept of Any symbol is implemented in several parser generators. Historically, the first tool with embedded capability to match tokens from sets which are not directly specified in a grammar is the Coco/R recursive-descent parsers generator. According to the documentation [11, p. 14], developer can use a special symbol ANY, which denotes any token that is not an alternative to that ANY symbol in the current production. A set of admissible tokens for the position of a particular ANY is precomputed to make the situation when parser has to make a choice between ANY and some explicitly specified token unambiguously solvable in favor of the explicit option. As shown in [10], these precomputed sets are both incomplete due to the lack of nonterminal outer context analysis and excessively restrictive due to a single restriction applied to all the elements of the sequence corresponding to the iteration of ANY. As a result, there are grammars for which parsers generated by Coco/R do not parse programs valid from the developer’s point of view. For example, a parser generated by the grammar

\[ A = a \cdot b \cdot c | \{\textit{ANY}\} \cdot d. \]

is not capable to recognize the input string \texttt{bad$5$} (\(5\) denotes the end of the input, \{\textit{ANY}\} denotes zero or more ANY tokens). Similar Any implementation is built into a tool for lightweight LALR(1) parser development, called LightParse [12]. LightParse grammar is not directly used to generate a parser. Instead, it is transformed to the YACC-like format supported by the standard LALR(1) parser generator GPPG. In the transformed grammar, every entry of Any symbol is presented as a separate rule with single-element alternatives, by an alternative for each of the admissible terminal symbols. To ensure these rules are valid in terms of GPPG, LightParse imposes additional restrictions on Any usage. It only deepens drawbacks inherited from Coco/R. The most recent Any token implementation is introduced by the author of the current paper for LanD parser generator [10] aimed at LL(1) tolerant parsers generation by island grammars. In terms of the island grammars paradigm, Any symbol allows one not to specify the particular content of the water area, writing Any instead. Unlike ANY symbol in Coco/R, our Any corresponds to a sequence of zero or more tokens, not a single token. In its implementation, all the known shortcomings are eliminated. The decision about the current token’s admissibility at Any position is made dynamically at the parsing stage and restricts the set of admissible tokens no more than necessary to avoid ambiguities. LanD’s Any implementation does not assume the grammar translation to the form suitable for the standard parsing algorithm. Instead, the standard LL(1) algorithm is modified to integrate the notion of Any and make it possible to define admissible tokens by the content of a parsing stack. In the current paper, LanD parser generator is extended with the capability to generate LR(1) parsers with embedded Any support.
3.2 Formal definition of a simplified grammar

In [10], through the Any token, we formulate a formal concept of the simplified grammar. We denote by lhs(p) and rhs(p), respectively, the left and the right part of the production p. Notation x ∈ rhs(p) for x ∈ N ∪ T means that rhs(p) = α₁x₁α₂, where α₁ ∈ (N ∪ T)∗, α₂ ∈ (N ∪ T)∗. SYMBOLS(y) is used for the set of terminal symbols needed to compose all the ω: y ⇒ ω, y ∈ (N ∪ T)∗.

Definition 1. Let G = (N, T, P, S) be a context-free grammar, Any /∈ T. The grammar simplified with respect to G is a grammar Gₛ = (Nₛ, Tₛ, Pₛ, Sₛ) defined as follows:

1. Sₛ = S;
2. Pₛ = {p ∈ f(P) | lhs(p) = Sₛ ∨ 3p ∈ P; lhs(p) ∈ rhs(p)), where f: P → {p = A → α | A ∈ N, α ∈ (N ∪ T ∪ {Any})} is the mapping that satisfies the following criteria:
   a) 3P ⊆ P; P' = {p ∈ P | f(p) ≠ P}, P' ≠ Ø,
   b) ∀p ∈ P \ P', f(p) = p,
   c) ∀p ∈ P', ∃n ∈ N; p is representable in the form A → α₁γ₁β₁α₂γ₂β₂…αₙγₙβₙ and f(p) is representable in the form A → α₁Any₁β₁Any₂β₂…αₙAnyₙβₙ, where ∀i ∈ [1..n], αᵢγᵢβᵢ ∈ (N ∪ T)*, and ∀i ∈ [1..n], ∀j ∈ FOLLOW(A), SYMBOLS(γᵢ) ∩ FIRST(βᵢαᵢγᵢβᵢαᵢγᵢβᵢ…αₙγₙβₙα) = Ø;
3. Nₛ = {A ∈ N | 3p ∈ Pₛ; lhs(p) = A};
4. Tₛ = {α ∈ T | 3p ∈ Pₛ; α ∈ rhs(p)) ∪ {Any}.

Intuitively, Pₛ produces contents for the start symbol of Gₛ and productions for all the nonterminals which are reachable from the start symbol. The definition of the mapping f means that some of the strings generated by Gₛ contain substrings which can be replaced with Any, then we obtain strings generated by Gₛ. Symbol Any can be written instead of the parts denoted by γᵢ in production’s right-hand side, in case these parts satisfy the criterion 2c of the definition 1. Verification of this criterion is possible only when solving a direct problem: when the grammar Gₛ is created on the basis of some available G. In theory, G can correspond to the baseline language grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. Developer writes an initial approximation in the form of a simplified grammar, as well as be a more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. In practice, it is usually not available or does not exist, so direct problem is rarely considered. Writing an island grammar for a certain programming language is equivalent to solving an inverse problem. 

Compliance with the criterion 2c is crucial for correct Any processing. At the same time, it is hard to maintain while solving an inverse problem. In this paper, additional Any processing mechanisms are offered. They allow grammar developers to weaken the control over the consistency with the formalization.

3.3 LL(1) parsing algorithm modification

In fig. 1, modified algorithms from [10] are rewritten in the form more suitable for further discussion. The delta between the standard algorithms and the modified ones is highlighted with grey. As shown in fig. 1a, when no action can be performed with a current token, parser tries to interpret this token as the beginning of a sequence corresponding to Any. FIRST* set, a modified version of a standard FIRST, is computed for the parsing stack content to get all the tokens that are explicitly allowed in the current place. Then she performs an iterative refinement in accordance with full-LL(1) parsing [13, pp. 247–251]. Set construction routine is shown in fig. 1c. A modification is needed to handle the consecutive Any problem defined in [10], this problem is explained in detail in Section 4.2.1 along with a more general solution. M denotes a parsing table, Stack denotes a symbol stack which stores not just the symbols that are expected to be matched, but nodes of the syntax tree being constructed. There are grounds for an analogy between the LL(1) parsing modification given and well-known error recovery algorithms: Any symbol looks similar to the error token denoting place in the grammar where recovered parsing can be resumed, FIRST* set seems like the set of synchronization tokens. Moreover, speaking in terms of the formal definition, a tolerant parser is built by a simplified grammar Gₛ, and a program from L(Gₛ) is actually needed to be parsed. In terms of Gₛ, this program is erroneous. However, here also lies a fundamental difference between Any processing and error recovery. Recovery is performed for a program which is incorrect regarding to a baseline grammar G. While success is not guaranteed, the main goal is to resume parsing at any cost, including the loss of some significant results of the previous analysis and skipping a significant part of the input stream, possibly containing some points of interest. The goal of Any processing is to translate a presumably valid L(G) program into the language L(Gₛ) by replacing some token sequences with Any. The premise that the program under consideration is correct with respect to G, in conjunction with the observance of the criterion 2c, makes input tokens skipping totally predictable. One can be sure that the parts of the input stream replaced with Any belongs to the water and can be discarded without loss of the land. Furthermore, predictable and correct replacement with Any is possible for a program that is incorrect with respect to G, in case incorrectness is located in a water area.

4. Algorithms and modifications

4.1 LR(1) parsing

Though the modified LL(1) parsing algorithm described in Section 3.3 is enough to create reliable tolerant parsers, describing a real programming language with LL(1) grammar is a challenge even when this grammar is supposed to be lightweight and tolerant. Constructs of interest, such as class members, usually have a common beginning up to a certain point, so they cannot be presented as solid alternatives for a single nonterminal symbol in LL(1). Instead, we have to write rule sequences in the style of taking the common factor out of the brackets and making a separate rule for a tail: entity = attribute* keyword* { (class_tail | member_tail) } member_tail = type name (method_tail | property_tail) method_tail = arguments Any (init? ';' | block) As a result, the grammar structure is not transparent enough for a newcomer because the connection between existing island rules written in such a distributed manner and particular language constructs is non-obvious. This LL(1) limitation can be overcome through switching to a more complex LR(1) parsing. A modification of the standard LR(1) algorithm is shown in fig. 2a, modified areas are highlighted with grey. Like in a standard case, two stacks exist to keep parser state. SymbolsStack keeps the current viable prefix [14, p. 256]. In fact, similar to LL(1) Stack, in our implementation, it keeps not just symbols but nodes for a tree to be built. StatesStack keeps the indices of the states parser passed through to obtain the current viable prefix. An element ACTIONS[s, t] of the ACTIONS table keeps the knowledge of what action should be performed by the parser if token t is met while s is the parser’s current state. There are two basic types of action in LR algorithm: Shift and Reduce, they are shown in fig. 2c. GOTO[s, X] contains the index of a state to which parser must go from s state after reducing some part of a viable prefix to X.
Головешкин А.В. Толерантный синтаксический анализ с использованием модифицированных алгоритмов LL(1) и LR(1) со встроенной обработкой символа «Any». Труды ИСП РАН, т. 31, вып. 3, 2019 г., стр. 7-28

Fig. 1. Modified LL algorithms: (a) LL(1) parsing algorithm, (b) “Any” processing algorithm, (c) FIRST set construction, (d) Auxiliary algorithms: alternative applying and FIRST set memoization

Fig. 2. Modified LR algorithms: (a) Modified LR(1) parsing algorithm, (b) “Any” processing algorithm, (c) Shift and reduce algorithms.

The essence of the parsing algorithm modification is similar to LL(1) case: tolerant parser is responsible not only for checking if the program can be derived from the start symbol, but also for translating it from a baseline language into a simplified one. In case an action for some actual combination of parser state and input token is undefined, parser tries to interpret the current token as the beginning of the subsequence of the program from $L(\mathcal{G})$ that corresponds to $\text{Any}$ in the corresponding program from $L(G_0)$. In case there is an action available for $\text{Any}$, parser calls SkipAny routine (fig. 2b), where firstly all the possible Reduce actions are performed and secondly $\text{Any}$ token is shifted. Note that we consider $\text{ACTIONS}$ table to be cleared from Shift/Reduce conflicts in favor of $\text{Shift}$ action. Also, there is no additional checking if $\text{Shift}$ action exists, because this existence follows from the standard $\text{ACTIONS}$ and $\text{GOTO}$ construction algorithm. Having moved $\text{Any}$ to a viable prefix, parser looks for the first token which is explicitly expected in $L(G_0)$ program and then continues parsing in the usual way.

In fig. 3, there is an LR(1) tolerant grammar for Java programming language written in the format supported by LanD parser generator. As it can be seen, island entities, such as enumerables, classes, methods, and fields, are clearly presented as solid rules. In comparison with a baseline Java
4.2 “Any” processing improvements

4.2.1 Consecutive “Any” problem

In fig. 1c, FIRST’ algorithm, which is the modified version of the standard FIRST, is presented. It is intended to solve the problem of consecutive Any described in [10]. The problem manifests itself when two or more Any tokens directly follow each other at the beginning of the sentence which can

derive be from the stack. In this case, the subsequent Any hides some stop tokens from the previous one. Consider the following grammar $G$:

$A = (a|b) \cdot B \cdot C; \quad B = d \quad | \quad C = (e|f)? \cdot C$

It can be simplified to the following $G_

\text{a}$:

$A = \text{Any} \cdot B \cdot C; \quad B = d \quad | \quad C = \text{Any} \cdot C$

The string $\text{abcs} \in L(G)$ is supposed to be successfully matched by the parser built for $L(G_\text{a})$, because the following derivation may be performed:

$A \Rightarrow \text{Any} \cdot B \cdot C \Rightarrow \text{Any} \cdot C \Rightarrow \text{Any} \cdot c$

Having met the token $\text{a}$, the tolerant parsing algorithm starts the first $\text{Any}$ processing. If the standard FIRST is used to find stop tokens, FIRST(Stack) set equals to $\{d, \text{Any}\}$, as a result, SkipAny skips all the input and returns an error. Taking into account that $\text{Any}$ is allowed to match an empty sequence, FIRST’ modification looks beyond the second $\text{Any}$ and, in general, beyond all the subsequent Any symbols in searching some explicitly specified tokens which may follow a sequence corresponding to these Any tokens. Stop token set found with FIRST’ (Stack) equals to $\{d, c\}$, thus the first $\text{Any}$ captures a and tokens and stops on $c$, the second one matches an empty sequence, and $\text{abcs}$ string is admitted to be correct.

This approach is proved to be enough to build working parsers for real programming languages, such as C#, Java or PascalABC.NET. It can also be implemented for LR(1) through ACTION and GOTO static analysis. However, on closer inspection it becomes clear that algorithms modified in this way work correct only for a subclass of simplified grammars, satisfying an additional constraint:

Definition 2. Let $G_\text{a} = (N_\text{a}, T_\text{a}, P_\text{a}, S_\text{a})$ be a grammar simplified with respect to a context-free grammar $G = (N, T, P, S)$. Enumerate as $\text{Any}_1, \text{Any}_2, ..., \text{Any}_k$ all the $\text{Any}$ entries from the right-hand sides of productions from $P$, which appeared as a result of replacement of the corresponding $y_1, y_2, ..., y_k$ subparts of the right-hand sides of productions from $P$ in compliance with Definition 1. Derivation $S_\text{a} \Rightarrow a\text{Any}_1\text{Any}_2...\text{Any}_k\text{Any}_l$, where $k, l, ..., t \in \{1..., n\}$, $\alpha_j, \beta_i \in (N_\text{a} \cup T_\text{a})^*$, $b \in T_\text{a}(\text{Any})$, is not acceptable in $G_\text{a}$ if $b \in \text{SYMBOLS}(y_1y_2...y_k)$. Informally speaking, the token which is a stop token for the last $\text{Any}$ in a sequence is not allowed to appear in the area corresponding to one of the preceding $\text{Any}$, otherwise it will cause premature completion of $\text{Any}$ processing. Let $G$ has a different structure:

$A = (a|b)c|d|e|f) + B \cdot C; \quad B = g \quad | \quad C = (h|i) + a$

It can be simplified to

$A = \text{Any} \cdot B \cdot C; \quad B = g \quad | \quad C = \text{Any} \cdot a$

Herein, both replacements with $\text{Any}$ are still satisfy the criterion 2c, but the restriction from Definition 2 is not satisfied, as a may follow the second $\text{Any}$, and at the same time it is a valid element of the area corresponding to the first one. As a result, while parsing $\text{abcs}$, the first $\text{Any}$ is matched with an empty token sequence because FIRST’ ([B, C]) equals to $\{a, g\}$, the second Any also cannot include a, so, valid input is not accepted.

In practice, the most common case of consecutive Any appearance does not break the restriction mentioned: in grammars we have developed, Any is often used as one of the possible variants for an element of a list, so, all the Any tokens in the derivation of such a list originate from a single Any entry in the grammar, therefore, derivation can be rewritten as $S_\text{a} \Rightarrow a\text{Any}_1\text{Any}_2...\text{Any}_k\text{Any}_l$, and the corresponding condition $b \in \text{SYMBOLS}(y_1y_2...y_k)$ is false in accordance with Definition 1. To cover the general case, we introduce a mechanism for passing an additional information at any processing stage. Any entry can be supplemented with two options: Except and Include. For each of them, a list of literals or token names can be passed as parameters. The concept of $\text{AnyExcept}$ initially appeared in LightParse parser generator [15], but there it was intended to compensate the lack of outer context analysis while constructing the set of admissible tokens. Our intention is different:

Fig. 3. Java LR(1) tolerant grammar
symbols specified for Except option are supposed to compensate the lack of information in consecutive Any problem: they are supposed to be explicitly specified tokens that may follow the area corresponding to Any in L(G). Include option allows one to approach this problem from a different angle, specifying tokens that shouldn’t be interpreted as stop tokens despite their appearance in FIRST(\text{Stack}). So, for the grammar above we can use one of the following simplified analogues:

\[
\begin{align*}
A &= \text{AnyExcept}(g,h,i) \ B \ C; \\
B &= g \ | \ C = \text{Any a} \\
C &= \text{Any a}
\end{align*}
\]

Having renamed stopTokens sets built in fig. 1b and 2b to stopTokensBasic, we transform stop token set construction for both LL and LR algorithms to

\[
\text{stopTokens} := \text{anyExceptSet.\text{Count} > 0} \\
\text{? anyExceptSet} \\
\text{stopTokensBasic.\text{Except}(\text{anyIncludeSet})};
\]

where anyExceptSet and anyIncludeSet denote sets of tokens passed as option parameters for Any currently being matched. For error recovery purposes discussed in Section 4.3, Any also supports Avoid option. Its arguments are tokens the presence of which in the Any-corresponding area signals about program incorrectness or wrong alternative choice. To take Avoid into account, while loop condition transforms to

\[
t \notin \text{stopTokens and } t \notin \text{anyAvoidSet and } t \neq \$.\]

In case token skipping is interrupted because current token equals to one of the Avoid arguments, this token passes to Error routine as a second argument.

Unlike in LL(1), there can be a situation in LR(1) when we do not know for sure what particular Any entry is being processed at the moment. This information is needed to access the corresponding options. To add support of Any options in LR(1), we introduce an additional type of LR(1) conflict called Any/Any conflict. It is reported when there is a state where multiple items have a dot before Any, and is needed to be resolved for successful parser generation.

### 4.2.2 Nesting level checking

While writing a tolerant grammar, developer usually has to make an additional effort to determine what bracketed areas may appear in the particular water, and if they can influence Any processing. Intuitively, such areas are perceived as a whole, and when Any is written instead of some better-grained water description, it may be missed that bracketed areas exist in that water in a real program. These areas may contain something that also appears right after that Any and therefore should be treated as a stop token. For example, being interested in fields of a C# class, we must capture a, b, c, and d in the fragment

\[
\text{int a = 0, b = 1;}
\]

\[
\text{DateTime c = new DateTime(2019, 5, 29),}
\]

\[
\text{d = new DateTime(2019, 5, 31);}
\]

At the same time, we are not interested in initializers, so, the first intention is to describe field declaration with the rules

\[
\begin{align*}
\text{fields} &= \text{type name init? } (',,' \text{ name init?})* ';
\text{init} &= '=' \text{Any}
\end{align*}
\]

Unfortunately, these rules work only for the first declaration. The set \{',', ','\} is a stop token set for Any, and in the second declaration, comma separates not only fields but also arguments bordered with round brackets. Generally speaking, Any does not satisfy the formalization in this case. At the same time, simplicity is the crucial property of the tolerant grammar, and the way in which water is described above is more preferable than the following one:

\[
\text{fields} = \text{type name init? } (',,' \text{ name init?})* ';
\text{init} = '=' \text{Any}
\]

To return the first version of init rule to the boundaries of the simplified grammar definition, we add to the parsing algorithms a capability to take into account bracketed structures. A pair of brackets is described like

\[
\text{ROUND_BRACKETED} : \ \%left ' ( ' \Right ') '
\]

and nesting level is tracked by lexical analyzer. If several kinds of pairs are described, it is believed that any pair can be nested in any pair. When Any is processed, it is allowed to end only at the same depth at which it begins. To control this situation, SkipAny methods are modified uniformly both for LL and LR. Firstly, at the beginning of a skip process, an additional variable is initialized:

\[
\text{depth := Lexer.\text{CurrentDepth}();}
\]

Secondly, in-loop \text{Lexer.NextToken()} call is replaced with \text{Lexer.NextToken(depth)}.

Passing the initial nesting level to a lexer, we force it to read the input stream until the depth of the next token equals the depth of the first token in the sequence corresponding to Any. Thus, Any-corresponding area is allowed to include stop tokens in nested structures because these nested structures are invisible to the parsing algorithm. Third modification is an additional checking to prevent moving through the upper nesting level. In fig. 4, there are two cases allowed by the first two modifications. Token a is the beginning of Any area, and b is a stop token. Obviously, the way Any symbol is matched on the right breaks the semantic integrity of a bracketed area. We consider such Any usage to be a bad practice, so, if lexer returns a token denoting the end of some pair and rise to the level above the initial, and this token is not a stop token, parser reports an error which means that grammar should be refined.

\[
\text{Fig. 4. Possible "Any" matching supported by nesting level tracking.}
\]

### 4.3 Error recovery

#### 4.3.1 Algorithms

As noted in Section 1, in case water entities look similar to islands, developer has to refine patterns and to add some antipatterns to avoid false positives. For a deterministic parsing, the problem of water and island similarity may have unpleasant consequences not only when there is a full match between island pattern and water entity, but even if a water entity and an island have a number of common starting tokens. In this case, parser starts analyzing a water entity as an island, finds a mismatch and fails to proceed analysis. It is important to note that this parsing failure indicates not the incorrect L(G) program but misinterpretation of the program in terms of \(G\). Generalized parsing algorithms are able to process such a situation exploring both ways an entity can be interpreted in and rejecting the failed one. To get a similar benefit from our modified deterministic parsing while preserving mostly linear complexity, we add special Any-based error recovery routines in both LL(1) and LR(1) algorithms. These routines are shown in fig. 5.
In the modified parsing algorithms, two types of errors can occur. The first one happens when LL(1) parser cannot match the current token or apply some alternative and Any is not acceptable at the point, or when LR(1) parser has no shift or reduce action for the current token as well as for Any. The second type occurs when Any processing starts and no stop tokens are found till the end of the input or a token specified as Stop is met. Recovery initiated for the first type does not influence the algorithm linearity as parsing is resumed at the token where the error occurred. Acting for LR(1) algorithm, recovery is more complex and heuristic due to the nature of a bottom-up parsing. In this interpretation of the erroneous area derivation, Basing on the last popped symbol. Productions of the items added to this set are possible participants of the grammar extend the error recovery symbol's subtree to water. More precisely, the part of the input from the first token mistaken for an island can be revealed through the static grammar analysis. Given the grammar

Fig. 5. "Any"-based error recovery algorithms: (a) LL(1) algorithm, (b) LR(1) algorithm.

In the modified parsing algorithms, two types of errors can occur. The first one happens when LL(1) parser cannot match the current token or apply some alternative and Any is not acceptable at the point, or when LR(1) parser has no shift or reduce action for the current token as well as for Any. The second type occurs when Any processing starts and no stop tokens are found till the end of the input or a token specified as Stop is met. Recovery initiated for the first type does not influence the algorithm linearity as parsing is resumed at the token where the error occurred. Acting for LR(1) algorithm, recovery is more complex and heuristic due to the nature of a bottom-up parsing. In this interpretation of the erroneous area derivation, Basing on the last popped symbol. Productions of the items added to this set are possible participants of the grammar extend the error recovery symbol's subtree to water. More precisely, the part of the input from the first token mistaken for an island can be revealed through the static grammar analysis. Given the grammar

{\textstyle \text{Error}(\text{tokens}): \quad \begin{cases} \text{if } (\text{lexer}.\text{CurrentTokenIndex}) \# \text{RecoveredIn} \quad \text{then} \\ \quad \text{return } \text{ERROR_TOKEN}; \\ \quad \text{end}; \\ \quad \text{RecoveredIn} := \text{lexer}.\text{CurrentTokenIndex}(); \\ \text{currentNode} := \text{Stack.Pop}(); \\ \text{do} \\ \quad \text{if } (\text{currentNode}.\text{Parent} \# \text{null}) \quad \text{then} \\ \quad \quad \text{maxChildIndex} := \text{currentNode}.\text{Parent} . \text{Children}.\text{IndexOf}(\text{currentNode}); \\ \quad \quad \text{if } \text{maxChildIndex} < \text{currentNode}.\text{Parent} . \text{Children}.\text{Count} - 1 \quad \text{then} \\ \quad \quad \quad \text{isRecovered} := \text{false}(); \\ \quad \quad \quad \text{for } \text{i} \text{ from } \text{indexOfCurrent} + 1 \quad \text{to } \text{maxChildIndex} \quad \text{do} \\ \quad \quad \quad \text{Stack.Pop}(); \\ \text{end}; \\ \quad \text{end}; \\ \text{end}; \\ \text{else} \\ \quad \text{return } \text{ERROR_TOKEN}(); \\ \text{end}; 
\end{cases}

In the context of a deterministic tolerant parsing problem, recovery symbols have specific semantics. They represent decision points at which parser may choose the wrong alternative, try to match a water entity as an island, and provoke an error. Recovery itself means returning to a decision point through the grammar ancestors of the currently unmatched token or unparsed nonterminal and choosing the alternative that 

Recovery algorithms presented simplify the process of grammar extension and reduction. Recovery symbols are pre-computed at parser construction stage. Developer can disable recovery at all or specify particular nonterminals from this set which should be used for recovery, otherwise, all the elements of the set are taken into consideration when Error routine is called.

In the context of a deterministic tolerant parsing problem, recovery symbols have specific semantics. They represent decision points at which parser may choose the wrong alternative, try to match a water entity as an island, and provoke an error. Recovery itself means returning to a decision point through the grammar ancestors of the currently unmatched token or unparsed nonterminal and choosing the alternative that
As noted, errors happening on Any processing require limited backtracking. The particular increase in running time of the algorithm depends on number and length of backtracked sequences. From the prohibition of multiple recovery at the same token, it follows that there can be only one backtracking to a particular position, so, the worst case is when the following situation repeats sequentially for each token except the first one: Any processing starts on the token, fails by reaching the end of the input and backtracks to that token, then recovery starts, the token matches successfully with the help of the water alternative, and the next token becomes the token under consideration. In this scenario, a number of times the token is examined equals to its sequential number counting from one. For the i-th token, i − 1 examinations are occurred on Any skipping started at previous tokens and at the current one, and 1 examination is for some final match. As backtracking itself consists of a simple index reassignment, it does not increase this counter. It can be shown that this worst-case scenario takes place for inputs acS, aacS, aacS$, etc. and a parser generated by the following LL(1) grammar:

\[
S = \text{a Any b | Any S} \\
\text{CURVE_BRACKETED : } \%\text{left } ('\text{ bright }') \\
\text{ROUND_BRACKETED : } \%\text{left } ('\text{ bright }') \\
\text{SQUARE_BRACKETED : } \%\text{left } ('\text{! GENERAL_ATTRIBUTE_START }\text{ bright })' \\
\text{namespace = 'namespace' name } ('\text{ namespace_content }') \\
\text{entity = enum | class struct interface | method} \\
\text{ | field decl | property | water entity} \\
\text{ enum = common 'enum' name Any 'Any'} 'Any' ';'; \\
\text{class struct interface =} \\
\text{ common 'class' 'struct' 'interface'} name Any '{' 'entity*'} '{'; \\
\text{method = common type name arguments Any (init expression? ';') | block} \\
\text{field decl = common type field (',' field') 'I'}; \\
\text{field = name } ('\text{! Any}'); init_value? \\
\text{property =} \\
\text{ common type name (block (init_value ';') | init_expression 'I') water_entity =} \\
\text{AnyInclude ('delegate', 'operator', 'this') (block | ')') +} \\
\text{common = entity_attribute* modifier*} \\
\text{modifier = MODIFIER | 'extern' \\
\text{init expression = '->' Any} \\
\text{init value = '='} 'init part+ \\
\text{init part = Any | type} \\
\text{arguments = } ('\text{! Any}') \\
\text{block = } ('\text{! Any}')' \\
\]

Fig. 6. Fragment of the C# tolerant grammar.

The total number of token examinations equals to \( \frac{n^2}{2} + \frac{n}{2} \), it means that our algorithms are \( O(n^2) \) in the worst case. However, experiments show that the percentage of recoveries required backtracking is insignificant in comparison with the total number of recoveries and tokens: for example, in all the Java projects from subsection 5.2 taken together, there are 27393 files splitting at 2625589 tokens, while total number of recoveries is 32683 for LL(1) and 31861 for LR(1), and only 20 recoveries for each type of parsing are performed after on-Any error.

### 5. Experiments

To test the algorithms described in Section 4, tolerant grammars for the following programming languages, markup languages and specification formats are developed: C#, Java, PascalABC.NET, XML, Markdown, YACC, and Lex. All the sources are available on GitHub. For a large-scale testing, C# and Java are chosen as the languages complex enough and having a large number of well-known open-source repositories. For both languages, LL(1) and LR(1) tolerant parsers are generated with LanD parser generator on the basis of the corresponding tolerant grammars.

As tolerant parsers are created to capture particular islands, the purpose of the experiment is to evaluate precision and recall of this capturing. Stages of the experiment are the same for both languages. For each of the projects under consideration, tolerant parser is firstly applied to parse all the project files written in the corresponding language. By traversing syntax trees built, types and names of the islands are extracted in report files, per report for each island type. This extraction does not require some severe postprocessing: island type is actually a node type, and name is stored in one of this node’s children. Secondly, the same files are parsed by a baseline parser. Roslyn is used as a baseline parser for C#, and Java parser is generated with ANTLR from the full grammar of the language. Then information about program entities that are specified as islands for our tolerant parsers is extracted from trees built by these baseline parsers, so the second group of reports is obtained. At the third stage, two reports for the same type are compared in an automated way to eliminate the human factor. Matches are excluded, so only the information about entities found by one parser and not found by another one remains. It is then explored manually. For each of the languages, there is a table whose rows correspond to projects parsed and columns correspond to island types. There is also an additional “Total files” column allowing to estimate the scale of the project. In a table cell, there is a number of islands of the corresponding type found by our tolerant parser for the corresponding project. We have obtained that these numbers are the same for LL(1) and LR(1) parser, so we do not need two separate tables for a single language. In case tolerant parser finds less island entities than the baseline one, the number of entities missed is specified in parentheses with a minus sign. In addition to the tables, a detailed analysis of mismatches is provided.

#### 5.1 C# tolerant parsing

For C# programming language, five open-source projects from different domains are considered:

- **Roslyn** project includes C# and Visual Basic compiler sources and lots of test files capturing different complex and uncommon variants of a C# program;
- **PascalABC.NET** consists of the corresponding language compiler and IDE sources, it has a relatively long history reflected in the legacy code written by differently experienced contributors;
- **ASP.NET Core** refers to the web development domain: it is a cross-platform .NET-based web framework;
- **Entity Framework Core** is an object-relational mapper allowing one to work with a database using .NET objects;
- **Mono** is an open source third-party implementation of Microsoft’s .NET Framework including C# compiler, Common Language Runtime virtual machine, lots of core libraries and, again, a great number of test files.

---

1. https://github.com/antlr/grammars-v4/tree/master/java
2. https://github.com/alexeyvale/SYRCoSE-2019
3. https://github.com/antlr/grammars-v4/tree/master/java
Парсинг результатов представлен в Таблице 1, а фрагмент толерантного \(\text{LR}(1)\) грамматика приведен в фиг. 6. Отметим, что классы, структуры, а также элементы соответствуют одному сингулярному понятию, их общее количество представлено в "Классы" столбце. В данном разделе, метки относятся к директории проекта.

Для Roslyn источников, имеются 5 методов Roslyn и без LanD. Четыре из них являются локал\(^{9}\) метода (методы вложенные внутри других методов), эта особенность появилась в C# 7.0. В этом контексте методов важным является имеющийся факт, чтобы добавить их в поддержку рассмотрения грамматарной. Одно из обобщений грамматики посредством метода символа как альтернативы в попытке распознавания грамматику. Это хорошо заметно, Roslyn проект является без проекта в переопределение класса этого файла. Этот 5-й метод описывает файл с текстом грамматики в японском Shift-JIS кодировке. Название класса написано в японском языке, вызывает ошибку, которая не распознается LanD, потому что это интерпретируется как критический символ в грамматике. Мы пренебрегаем использованием национальных алфавитов для именования как редкое, но, если необходимо, ID токен может быть использован как поддержка.

Два свойства из разных файлов не найдены LanD, в обоих случаях это вызвано отсутствием выражения для выражения-исполнения предыдущего непосредственного. Эта зависимость определяется на внешнем условии компиляции символа и не заменяется наилучшим образом в случае изолированного файла. Анализ в следующем коде, IsWindows не распознаёт LanD, потому что это интерпретируется как часть выражения для Configuration:

```csharp
public static ExecutionConfiguration Configuration =>
#if DEBUG
    ExecutionConfiguration.Debug;
#elif RELEASE
    ExecutionConfiguration.Release;
#else
    #error Unsupported Configuration
#endif
```

В этом случае несоответствия можно частично обрабатывать с помощью AnyAvoid(MODIFIER) вместо Any в \text{init_expression} грамматической правила. Для примера выше, это обнаруживает проблема в отсутствии информации о \text{Configuration} свойстве, как это будет рассматриваться как вода, но защитит следующие объекты, начиная с того, что началось с ключевого слова.

Для PascalABC.NET и ASP.NET Core, все элементы, найденные Roslyn, также найдены LanD. Для Entity Framework Core, различие в числе и местоположении ключевых слов определяется сложностью в поддерживаемых свойств и циклического синтаксиса. Ограничение грамматики и вставка нужного в число и тип открытия и закрывания скобок, half

Таблица 1. Количество найденных элементов в C# проектах.

| Проект          | Общее число | Число | Общее число | Общее число | Число | Число |
|-----------------|-------------|-------|-------------|-------------|-------|-------|
| Roslyn          | 8759        | 482   | 23705       | 20265       | 23127 | 116312|
| PABC.NET        | 2802        | 359   | 5522        | 16739       | 12023 | 37027 |
| ASP.NET Core    | 7356        | 333   | 12604       | 10214       | 16301 | 44163 |
| EF Core         | 2997        | 101   | 7783        | 4687        | 16941 | 26421 |
| Mono            | 37224       | 4928  | (-1)        | 60187       | (-1)  | 1696958|
|                 |             |       | (-122)      | 1696895     | (-67) | 99167 |
|                 |             |       | (-2)        | (-36)       | (-670)| 309580|

Аналогично грамматике и усовершенствованной, мы не принимали во внимание, что есть некоторые ключевые слова в C# что недавно появляются и были включены в грамматические ключевые слова. Это означает, что они не могут быть настроены как \text{contextual} ключевые слова. Это не позволяет использовать [4] в C# (методы тела пропущены):

```csharp
namespace async
{
    partial class async
    {
      partial void partial();
    }
    partial class partial
    {
      // async method named ‘async’
      async Task<async> async() { ... };
      // method named ‘async’ returning an object of type ‘async’
      async async(async async) { ... };
    }
}
```

Полный текст метода ключевого слова зависит от сложного контекста анализа. В Roslyn источниках, есть специальный метод \text{ShouldAsyncBeTreatedAsModifier} проверяет ряд специфических условий, каждый из которых меняет частоту асинхронного метода, предопределенные типы, и \text{partial} ключевое слово. Кроме того, у 2 дополнительных токенов требуются для правильного решения. Именно в таком случае ключевые слова использовались как идентификаторы, которые должны быть практически неиспользуемыми. В наших опытах, такие случаи были только в небольших тестах, не в том реальном коде. Дополнительно, использование async или \text{partial} ключевые слова как публичные

4 src/Compilers/CSharp/Test/Emit/EndToEndTests.cs
5 src/Compilers/CSharp/Portable/Flow/Analysis/NullableWalker.cs
6 src/Compilers/Test/Resources/Core/Encoding/ijis.cs
7 test/EFCore.SqlServer.FunctionalTests/Query/SimpleQuerySqlServerTest/Where.cs
8 test/EFCore.TestModelBuilding/ModelBuilder/Other.cs
9 mcs/errors
10 mcs/tools/sqlmetal/src/Dblinq/Test
11 mcs/jay/skeleton.cs
12 mcs/test-async*.cs
13 mcs-tests/test-partial*.cs
identifiers one breaks general C# naming conventions14 which are usually used as a basis for particular code style rules being applied inside a developers team.

5.2 Java tolerant parsing

For Java programming language, the following projects are considered:

- **Java Development Kit** is a toolbox consisting of Java compiler, core libraries, and Java Runtime Environment;
- **Elasticsearch** is an engine for a full-text search;
- **Spring Framework** is a Java framework used to build applications for different subject domains;
- **RxJava** is a library for composing asynchronous and event-based programs.

Parsing results are presented in Table 2, and a tolerant LR(1) Java grammar is presented in fig. 3.

As it can be noted, there is the only difference between baseline and tolerant parsing results.

```
private final static int
CREATE_MASK = 1<<CREATE,
FIND_MASK = 1<<FIND,
NEW_MASK = 1<<NEW,
RELEASE_MASK = 1<<RELEASE,
ALL_MASK = CREATE_MASK|FIND_MASK|NEW_MASK|RELEASE_MASK;
```

Unlike all the other types of brackets considered in Section 4.2.2, angle brackets cannot be defined as a pair in the LanD grammar because they may appear in the program in different meanings, some of which assume they can be used separately from each other. However, in case they bracket type parameters, it is important to match these parameters as a whole to prevent inner commas from being interpreted as field separators. It is hard to resolve this problem correctly staying in the tolerant parsing boundaries and, actually, in the boundaries of a context-free parsing and lexing too [17]. To make a correct decision, an analysis of the context angle bracket appears at is needed. Recovery algorithm combined with Avoid-based error triggering helps to handle inputs like

```
private static final long ADD_WORKER = 0x0001L << (TC_SHIFT + 15);
```

by interpreting all the angle brackets as opening for a `type parameter` in fig. 3, triggering an error on `;` token which is forbidden in type parameters, and reinterpreting the outermost type parameter as `Any` from `init_part` water alternative. However, this processing cannot prevent the loss of some middle fields from the group of fields defined simultaneously.

Table 2. Number of entities found in Java projects

| Project   | Total files | Enums | Classes | Fields | Methods |
|-----------|-------------|-------|---------|--------|---------|
| JDK       | 7704        | 151   | 10590   | 46176 (-3) | 88709   |
| Elastic   | 10972       | 387   | 14914   | 36830  | 94722   |
| Spring    | 7063        | 100   | 12060   | 18402  | 61515   |
| RxJava    | 1654        | 36    | 2728    | 6258   | 19931   |

5.3 Summary

As experiments show, both C# and Java tolerant parsers using our modified LL(1) and LR(1) algorithms are viable and allow one to find almost all the islands that can be found with a baseline parser. Mismatches cannot be considered as a tolerant parsing disadvantage: the ones occurred in erroneous C# programs are not unexpected since our algorithms are designed to work with correct programs, while for the most part of the valid programs containing lost islands, possible grammar fix can be easily suggested due to grammar simplicity and extensibility. However, there is also a tiny group of valid programs for which it is impossible to catch the missing island without performing an additional context analysis. This problem is actually not a tolerant parsing problem but a context-free analysis problem in general.

6. Conclusion

In the present paper, several algorithms and algorithm modifications aimed at island-grammars-based deterministic tolerant parsing are proposed. LR(1) parsing algorithm modification is performed in accordance with the simplified grammar formal definition previously developed by the author of the paper. A special `Any` symbol is integrated into the algorithm to add a capability to match token sequences which are not explicitly described in the grammar. LR(1) tolerant grammars tend to be shorter and more comprehensible than their LL(1) analogues written for previously modified LL(1) algorithm. Additional restriction defining simplified grammars subclass for which LL(1) and LR(1) tolerant parsing algorithms are always able to correctly handle consecutive `Any` problem is revealed. `Any` processing mechanisms are introduced to expand correct consecutive `Any` processing to entire simplified grammars class. Nested bracketed structures tracking is implemented to give the grammar developer a possibility not to take into consideration the content of in-water bracketed areas while replacing water description with `Any`. Error recovery algorithms are proposed for LL(1) and LR(1) tolerant parsing. Unlike the standard error recovery, they are designed not to resume parsing for an incorrect program, but to find the area which was mistakenly interpreted as an island and reinterpret it as a water. Through the series of experiments with C# and Java parsers generated by tolerant grammars developed for LanD parser generator, modified LL(1) and LR(1) parsing algorithms are proved to be able to successfully analyze the source codes of industrial software products.

Though the current tolerant parsing implementation is enough to work on solution of the crosscutting concerns markup problem mentioned in Section 1, an improvement of parsing results for syntactically incorrect programs may broaden the markup tool application opportunities. We have an assumption that `Any`-based recovery responsibility area may be explicitly specified for a particular grammar, and outside of this area some other recovery algorithms aimed at parsing resumption for an incorrect program can be used. Thus, our tolerant parsers will be capable to capture constructs of interest in such a program, like baseline parser successfully does in Section 5.1, instead of totally failing or interpreting all of these constructs as a single water piece. Besides, as performance was not the key goal until the present, we were satisfied with the generally linear dependency between input length and running time of the algorithms. However, basing on the knowledge of LanD implementation details, we are sure that performance can be improved (not in terms of time complexity classes, but in terms of absolute values of the algorithm running time). So, the algorithms and structures optimization is the second possible direction for further work on tolerant parsing.

References / Список литературы

[1]. Moonen L. Generating robust parsers using island grammars. In Proc. of the Eighth Working Conference on Reverse Engineering (WCRE’01). IEEE Computer Society, 2001, pp. 13–22.

[2]. Aframesh, A., Bach, J.-C., van den Brand, M., Johnstone, A., Manders, M., Moreau, P.-E., Scott, E. Island grammar-based parsing using GLL and Tom. Software Language Engineering: 5th International Conference, Revised Selected Papers. Springer Berlin Heidelberg, 2013, pp. 224–243.

[3]. Moonen L. Lightweight impact analysis using island grammars. In Proc. of the 10th International Workshop on Program Comprehension (IWPC). IEEE Computer Society, 2002, pp. 219–228.

14 https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/capitalization-conventions
Головешкин А.В. Толерантный синтаксический анализ с использованием модифицированных алгоритмов LL(1) и LR(1) со встроенной обработкой символа «Any». Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 7-28.

27

Informatіon about author

Алексей Валерьевич ГОЛОВЕШКИН в 2015 году получил степень магистра по направлению «Фундаментальная информатика и информационные технологии» в Южном федеральном университете, Ростов-на-Дону, Россия. В настоящее время проводит исследования на базе данного университета, готовит диссертацию на соискание учёной степени кандидата технических наук. К сфере его научных интересов относятся компиляторы, языки программирования, программная инженерия.

Alexey Valeriyevich GOLOVESHKIN obtained the master’s degree in fundamental informatics and information technologies in 2015 at Southern Federal University, Rostov-on-Don, Russia. Currently he does research at Southern Federal University working on the PhD thesis. His current research interests include compilers, programming languages, and software engineering.