Tolerant parsing with a special kind of «Any» symbol: the algorithm and practical application

A.V. Goloveshkin <alexeyvale@gmail.com>
S.S. Mikhalkovich <miks@sfedu.ru>
I.I. 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 the corresponding language grammar, other parts of the program are allowed to be not presented in the grammar or to be described coarse-grained, thereby parser remains tolerant to the possible inconsistencies in the irrelevant area. Island grammars are one of the basic tolerant parsing techniques. "Island" is used as the relevant code alias, while the irrelevant code is called "water". In the paper, a modified LL(1) parsing algorithm with built-in "Any" symbol processing is described. The "Any" symbol matches implicitly defined token sequences. The use of the algorithm for island grammars allows one to reduce irrelevant code description as well as to simplify patterns for relevant code matching. Our "Any" implementation is more accurate and less restrictive in comparison with the closest analogues implemented in Coco/R and LightParse parser generators. It also has potentially lower overhead than the "bounded seas" concept implemented in PetitParser. As shown in the experimental section, the tolerant parser generated by the C# island grammar is proven to be applicable for large-scale software projects analysis.

Keywords: tolerant parsing; robust parsing; lightweight parsing; partial parsing; island grammar; parser generation

DOI: 10.15514/ISPRAS-2018-30(4)-1

For citation: Goloveshkin A.V., Mikhalkovich S.S. Tolerant parsing with a special kind of "Any" symbol: the algorithm and practical application. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 4, 2018. pp. 7-28. DOI: 10.15514/ISPRAS-2018-30(4)-1

1. Introduction

Tolerant parsing is a parsing technique differing from the detailed whole-language (so-called baseline) parsing needed to build a full-featured compiler for a certain programming language. The main feature of the approach is the ability to capture points of interest inside the program, while all the code that does not contain such points can be skipped with no or minimal analysis performed. From developer's

perspective, this feature allows her to focus on the structure of the points of interest, providing a minimal description of the irrelevant area. Tolerant parsing is usually called *lightweight* because tolerant grammar tends to be much shorter than the baseline one.

There are several reasons for the tolerant parsing to be the most suitable option for the program analysis

- **Language embedding**: Some program artifacts assume the usage of multiple languages in one source file. In yacc-like grammars describing the syntax-directed translation, actions performed on a parsing step are expressed in terms of a certain general-purpose language. This means that the parser developed to capture the grammar structure must be tolerant to all the possible variations of these language snippets. A possible application of a tolerant grammar parser is described in [1]. A detailed description of the embedded language tolerant parsing is given in [2].
- **Full grammar inaccessibility**: Tolerant grammar imprints the developer's notion of what places inside the program are the most important in the context of the current task. Its structure and the mapping between the grammar entities and the language constructs are transparent to the programmer from the very beginning and can be further refined in accordance with the in-the-wild testing results. On the contrary, the baseline grammar usage requires a prior exploration and comprehension. This process is proved to be time-consuming [3] and can be impossible due to proprietary issues or manual baseline parser writing [4].
- **Domain-specific idioms**: In a certain project, some local domain-specific patterns can be applied [4]. 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 at these patterns, ignoring the underlying structure that allows one, in particular, to perform the impact analysis [5].
- **Incorrect program processing**: Syntax errors can be handled by the whole-language parser with some sophisticated error recovery mechanisms [6, 7]. These mechanisms are heuristic by the nature and do not guarantee the successful parsing resumption, as well as the preservation of the built parts of the parse tree. Tolerant parser is able to skip irrelevant error-containing areas. At the same time, tolerant parsing can be broken by the mismatch of the elements structuring the program (e.g. by the absence of a block closing bracket in C#). Specific error handling techniques allowing recovering from this category of errors are described for the *bridge grammars* [8, 9], a special kind of the island grammars.

The contributions of this paper are: 1) a modification of the standard LL(1) parsing algorithm aimed at *island grammars* tolerant parsing paradigm and designed to simplify irrelevant code skipping by means of a special `Any` symbol, this symbol is

used in a tolerant grammar to mark an irrelevant code without specifying its structure; 2) a compiler generator with a built-in tolerant grammar description language containing `Any` as a part of the standard syntax; 3) a lightweight grammar of the C# programming language for this generator; 4) an experimental evidence of the applicability of the generated tolerant C# parser for large-scale software projects analysis.

The remainder of the paper is organized as follows: a brief overview of the existing tolerant parsing techniques is provided in Section 2, in Section 3 the main goals of the current research are listed, in Section 4 we discuss related work and outline limitations of the closest analogues of our approach, in Section 5 the modification of the standard LL(1) parsing algorithm aimed at `Any` symbol processing is introduced. The tolerant grammar for the C# programming language is presented in Section 6, this section also includes a sufficient volume of experimental data obtained by applying the generated tolerant parser to a real-world software source code. In Section 7 a summary of the theoretical and practical contribution of the paper is provided.

2. Tolerant parsing techniques

Three basic tolerant parsing techniques considered in [2, 4, 5, 10–13] are fuzzy parsing, island grammars and skeleton grammars.

Fuzzy parsing is based on the notion of *anchors*, specific tokens that mark the beginning of the constructs of interest. The formal definition of a fuzzy parser is provided in [10, 11]. The grammar used by the fuzzy parser actually consists of a number of smaller grammars. Each of them has its own start symbol with a production rule starting with the anchor. The main concern of the fuzzy parsing technique is that parsing process is tightly coupled with anchor tokens and can be error-prone in case these tokens appear outside of the points of interest.

Skeleton grammar construction is described in [12]. The skeleton grammar partially shares its structure with the baseline grammar. Rules describing points of interest are complemented with baseline grammar rules needed to derive those points from the start symbol (this process is called *root completion*). After the root completion, special *default productions* are formulated for all the undefined nonterminal symbols appearing in the rules added. The key precondition making this process possible is the baseline grammar accessibility. As noticed in Section 1, most often this is not the case, besides, baseline grammar comprehension is quite time-consuming and requires some additional effort.

Island grammars technique is in the focus of our research. We believe that the concept of an island grammar is not well-known, so we provide its formal definition in accordance to [4, 5], despite the fact that this definition is not further referenced.

Definition 1. Given a context-free grammar $G = (N, T, P, S)$, where N is a set of nonterminal symbols, T is a set of terminal symbols, P is a set of production rules, $S \in N$ is a specified start symbol, and a set of constructs of interest $I \subset T^*$ such that

$\forall i \in I, \exists \omega_1, \omega_2 \in T^*: \omega_1 i \omega_2 \in L(G)$, where $L(G)$ denotes the language generated by G. An *island grammar* $G_I = (N', T', P', S')$ for $L(G)$ has the following properties:

1) $L(G) \subset L(G_I)$;
2) $\forall i \in I, \exists n \in N': n \overset{*}{\Rightarrow} i$ and $\exists \omega_1, \omega_2 \in T^*: \omega_1 i \omega_2 \notin L(G) \wedge \omega_1 i \omega_2 \in L(G_I)$;
3) $K(G) > K(G_I)$.

The first property means that G_I generates an extension of $L(G)$, the second means that the syntax analyzer for G_I recognizes constructs of interest from I in at least one sentence that is not recognized by the parser for G. The third property introduces the function $K(G)$ denoting the grammar complexity.

Informally speaking, island grammar consists of detailed productions describing certain constructs of interest (the *islands*) and liberal productions that catch the remainder (the *water*). Island productions form a set of *patterns* to be matched by the points of interest. However, patterns are not enough to overcome two important island grammars side effects called *false positives* and *false negatives* [12]. In case relevant code snippets look similar to the irrelevant ones, they can be confused by the parser, as a result, the irrelevant code will be recognized as the point of interest and some points of interest will be missed, there also can be a parse error. To minimize the mismatch, iterative refinement is needed for patterns as well as for *anti-patterns* matching irrelevant code.

To reduce the need for anti-patterns description and refinement, indeterministic parsing techniques are usually used. GLR [14, pp. 381–391] and GLL [15] parsers are capable to apply multiple parse actions for the same token in case of an ambiguity and continue parsing the program in all ways. However, they have a number of disadvantages: indeterministic parsing is hard to trace and debug, may return multiple parse trees that need some extra processing, and in case the islands look similar to the water, a parsing result can be extremely unpredictable. From the latter it follows that one still has to describe and refine some anti-patterns.

3. Problem statement

The key assumption of the current research is that tolerant parsing can be performed with a deterministic algorithm, while patterns and anti-patterns forming the tolerant grammar can be simplified and partially eliminated by making the algorithm capable to match and skip some token sequences which have no explicit definition in the grammar.

The key goals of the current research are:

1) to design an LL(1) parsing algorithm with built-in notion of a special `Any` grammar symbol that provides skipping of the token sequences that are not explicitly described in the grammar;

2) to develop a compiler generator with an integrated language for LL(1) grammars writing, supporting `Any` symbol usage and automatic syntax tree construction;
3) to implement a tolerant island grammar for the C# programming language in the format supported by the generator below; the grammar is supposed to contain water anti-patterns simplified with `Any` symbol;
4) to test parser's applicability to the analysis of large-scale software projects.

The developed tool is planned to be used for lightweight parsing of software projects and their further sustainable concern-based markup.

4. Related work

4.1 Coco/R

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 [16, p. 14], a special symbol `ANY`, which denotes any token that is not an alternative to that `ANY` symbol in the current production, is predefined in generated parsers. For a given grammar, an individual set of admissible tokens is connected with each `ANY` entry. Initially all the sets consist of all the tokens defined in the grammar, then at the parser generation stage the alternatives of `ANY` symbols are removed from the corresponding sets to make the situation when a parser has to make a choice between `ANY` and some explicitly specified token unambiguously solvable in favor of the explicit option. Further we will call these alternatives *rivals*, in order to avoid terminological confusion with alternatives forming grammar rules.

The major shortcoming of `ANY` implementation in Coco/R is that the intuitive principle of the explicitly specified token priority is both incomplete and excessively restrictive. As a result, there are grammars for which parsers generated by Coco/R do not parse some programs valid from the developer's point of view. Some examples of such Coco/R grammars are shown in fig. 1. Lower case is used for terminal symbols, `{}` denotes zero or more repetitions of bracketed elements.

Excessive restrictiveness manifests itself for the iteration `{ANY}`, for which the same set defines admissible tokens both for the first position in the sequence corresponding to `{ANY}` and for the rest positions. For the grammar in Fig. 1a, the set `{b, c}` corresponds to `ANY`. The token `d` is excluded from the set to make parser capable to finish `{ANY}` matching and match `d` explicitly. The token `a` is also excluded, that makes all the strings starting from `a` being matched by the first alternative with explicitly written `a` token in the beginning. However, the latter exclusion leads to the fact that the string `bad$` is not recognized by the parser. Note that the first token of the input — token `b` — is enough to choose the right production for `A` nonterminal, and the next `a` token cannot be treated as the

beginning of the first alternative. If separate sets were used for ANYs from the iteration, a could be added to the set of admissible tokens for the second and subsequent positions in {ANY}, and this would not lead to an ambiguity.

The lack of outer context analysis for nonterminal symbols leads to incompleteness of the constraints that are imposed on the ANY admissible tokens set. In Fig. 1b, ANY has no rivals within the rule, so the set of admissible elements consists of all the tokens defined in the grammar. As a result, the generated parser is not capable to recognize the intuitively recognizable string abcd$. Once {ANY} processing starts, the parser reaches the end of the input stream treating each token as a part of the sequence corresponding to {ANY}. Outer context analysis for nonterminal B shows that token d is in FOLLOW(B), so it may appear after ANY iteration. Hence d must be deleted from admissible tokens set and be matched explicitly.

```
A = a b c | {ANY} d.      A = B d | b B.  B = a {ANY}.
```

(a) (b)

Fig. 1. The grammars illustrating ANY implementation shortcomings in Coco/R.

At the same time, static analysis of the outer context is too coarse to be a good solution. For the grammar in Fig. 1b, B appears in two different contexts and the restriction derived from the first context (B d alternative) is not relevant for the second one (b B alternative). In the second context, ANY has no inner or outer rivals. With statically defined admissible tokens set, two contexts are mixed and string bad$ is not recognizable again. On the other hand, after choosing an alternative for A, the more precise information about what can follow {ANY} is available. That is, with dynamic decision making at the parsing stage, the set of programs recognizable by the parser can be extended.

In the current paper, the symbol Any is described. Unlike the ANY symbol in Coco/R, it corresponds to the sequence of zero or more tokens, not a single token. In its implementation, all the shortcomings listed above are eliminated and the decision about the current token's admissibility at Any position is made at the parsing stage.

4.2 LightParse

The tool for lightweight LALR(1) parsers development called LightParse [17] also supports the use of Coco/R-like Any symbol. LightParse application is similar to what we plan to do: generated lightweight parsers are used for concern-oriented source code markup [18]. LightParse performs static construction of the sets of tokens allowed at Any position and inherits all the Coco/R ANY implementation limitations. Besides, LightParse grammar is not directly used to generate the parser. Instead, it is translated to the YACC-like format supported by the standard LALR(1) parser generator GPPG, then GPPG produces the parser. In the translated grammar, every entry of Any symbol is presented as a nonterminal symbol with single-element alternatives, by an alternative for each of the admissible terminal symbols.

To get the valid YACC-like grammar without the nonterminal outer context analysis, LightParse imposes additional restrictions on `Any` usage: this symbol is not permitted to be in the end of the alternative, except for the start symbol productions. The presence of the intermediate grammar processing stage leads to inconsistency between the source grammar vocabulary, which is used by the grammar developer too, and the terms used in messages issued by the GPPG generator when some parser generation errors appear. Our `Any` implementation does not assume additional grammar adaptations for making the grammar 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 dynamically at the parsing stage. This eliminates the limitations of LightParse `Any` symbol.

4.3 Bounded seas

In [19], an extension of the regular parsing algorithm is described for parsing expression grammars (PEG). It is intended to automatically deduce anti-patterns for water which is supposed to be context-aware, i.e., specific for each particular island in the input. This approach named *bounded seas* is integrated in PetitParser tool which allows one to implement PEG-based parsers. Bounded seas are intended to completely eliminate the need for water rules explicit description in island grammars. A rule element of the form `~island~` is treated as a triple `before-water island after-water`. The key property of the water is that it never consumes any input from the right context of the bounded sea. The right context can be derived statically from the grammar or dynamically from the parser state. For the `after-water` entity, right context is set with an expression consisting of all the possible expressions that can directly follow `after-water`, separated with the ordered choice operator. Right context for the `before-water` consists of the `island` expression itself and the corresponding `after-water` boundary expression which both are ordered choice operands. Water expression succeeds when the corresponding right boundary expression succeeds.

Checking all the possible boundaries assumes backtracking, which leads to a sufficient time overhead. Since backtracking is a basic technique for PEG due to ordered choice operator presence, it is usually optimized with *packrat* parsing [20], which makes parsing time linearly dependent on the length of the program. Similar technique is used to eliminate potentially exponential complexity of bounded seas analysis. However, execution time decrease is achieved at the cost of a significant increase in the amount of memory used. Despite the right context exploration complexity, bounded seas are not able to make a globally correct decision on when water skipping should be ended. It is outlined in [19] that expressions forming the sea boundary actually recognize only prefixes of the possible boundaries, and boundaries form an LL(k) language where k depends on the particular situation. So, being designed to eliminate the need for anti-patterns presence in a grammar,

bounded seas, however, do not guarantee successful distinguishing islands from water without any explicit hints about the water content.

Besides, PetitParser itself is Smalltalk-based[1] and is intended for use in a closed ecosystem of Pharo virtual machine and Moose framework, so generated parsers have extremely low integrability into an arbitrary project.

The approach presented in the current paper has less overhead because it does not use backtracking at all. It performs a linear input processing and use the modified `FIRST` set building algorithm to find a boundary for `Any`. Though in [19] standard `FIRST` and `FOLLOW` sets from LL(1) parsing theory are named insufficient to recognize the boundary, it is demonstrated in Section 6 that with proper formulation of anti-patterns, the use of a modified `FIRST` set is enough to successfully analyze large-scale software project sources. `Any` symbol is used instead of explicit description of some parts of patterns and anti-patterns, that makes the island grammar significantly shorter and simplifies the grammar development process. Our parser generator is implemented in C#, thus it can be used with projects written on any .NET Framework-supporting language.

5. «Any» symbol implementation

We are mainly focused not on the individual islands capturing but on the extraction of the program hierarchical structure up to a certain level and tend to name the relevant code not *islands* but *land*, so the developed parser generator was named LanD[2] (by coincidence, it is also an acronym of «language description»). Table-driven predictive LL(1) parsing algorithm [21, pp. 220–228] was selected as the simplest and most suitable for debugging option for water skipping integration.

5.1 Formal definition of a simplified grammar

We introduce into the grammar the special terminal symbol `Any` to mark places where zero or more tokens from the irrelevant area can be matched. We denote by $\mathrm{lhs}(p)$ and $\mathrm{rhs}(p)$, respectively, the left and the right part of the production p. Notation $x \in \mathrm{rhs}(p)$ for $x \in N \cup T$ means that $\mathrm{rhs}(p) = \alpha_1 x \alpha_2$, where $\alpha_1 \in (N \cup T)^*, \alpha_2 \in (N \cup T)^*$. $\mathrm{SYMBOLS}(\gamma)$ is used for the set of terminal symbols needed to compose all the $\omega : \gamma \overset{*}{\Rightarrow} \omega, \gamma \in (N \cup T)^*, \omega \in T^*$. Through the symbol `Any`, we formulate the concept of a simplified grammar.

Definition 2. Let $G = (N, T, P, S)$ be a context-free grammar, $Any \notin T$. *Simplified* with respect to G is the grammar $G_s = (N_s, T_s, P_s, S_s)$ defined as follows:

1) $S_s = S$;

[1] PetitParser was also ported to a number of other languages, but those ports are experimental and are not updated with state-of-the-art features such as bounded seas.

[2] https://github.com/alexeyvale/SYRCoSE-2018

2) $P_s = \{p \in f(P) \mid \text{lhs}(p) = S_s \vee \exists p' \in P_s : \text{lhs}(p) \in \text{rhs}(p')\}$, where $f: P \to \{p = A \to \alpha \mid A \in N, \alpha \in (N \cup T \cup \{Any\})^*\}$ is the mapping that satisfies the following criteria:
   a) $\exists P' \subseteq P : P' = \{p \in P \mid f(p) \neq p\}$, $P' \neq \emptyset$,
   b) $\forall p \in P \setminus P'$, $f(p) = p$,
   c) $\forall p \in P'$, $\exists n \in \mathbb{N}$: p is representable in the form $A \to \alpha_1\gamma_1\beta_1\alpha_2\gamma_2\beta_2 \dots \alpha_n\gamma_n\beta_n$ and $f(p)$ is representable in the form $A \to \alpha_1 Any \beta_1 \alpha_2 Any \beta_2 \dots \alpha_n Any \beta_n$, where $\forall i \in [1..n]$, $\alpha_i\gamma_i\beta_i \in (N \cup T)^*$, and $\forall i \in [1..n]$, $\forall a \in \text{FOLLOW}(A)$, $\text{SYMBOLS}(\gamma_i) \cap \text{FIRST}(\beta_i\alpha_{i+1}\gamma_{i+1}\beta_{i+1} \dots \alpha_n\gamma_n\beta_n a) = \emptyset$;
3) $N_s = \{A \in N \mid \exists p \in P_s : \text{lhs}(p) = A\}$;
4) $T_s = \{a \in T \mid \exists p \in P_s : a \in \text{rhs}(p)\} \cup \{Any\}$.

Intuitively, P_s contains productions for the start symbol of G_s and productions for all the nonterminals which are reachable from the start symbol. Note that, according to items 3 and 4, $\forall p \in P_s$, $\text{lhs}(p) \in N_s$, $\text{rhs}(p) \in (N_s \cup T_s)^*$, i.e. P_s really satisfies the production set definition for a context-free grammar.

The definition of the mapping f means that some of the strings generated by G contain substrings which can be replaced with `Any`, then we obtain strings generated by G_s. In the absence of grammar simplification options developer has to work with grammar G, which can correspond to the baseline language grammar, as well as be a specially written more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. If `Any` symbol is supported by the grammar and the corresponding generator, anti-patterns forming a set P' can be substantially simplified. Symbol `Any` can be written instead of the parts denoted by γ_i in production's right hand side in case these parts satisfy the criterion 2c of the definition 2. Verification of this criterion is possible only when solving a direct problem: when the grammar G_s is generated based on the existing G. In a real situation, there is no grammar G and the developer has to solve the inverse problem: she manually writes a simplified grammar G_s, assuming that her knowledge of the particular island patterns and the general structure of the program is close to the ground truth — the structure of the baseline grammar G — and also considering parts denoted by `Any` satisfy the criterion. When this is not the case, unparsed or incorrectly parsed programs appear at the testing phase, this means that the grammar should be refined. This process usually takes several iterations.

Notice that despite the parser is built according to grammar G_s, a program from $L(G)$ is needed to be parsed. The modified LL(1) algorithm uses the criterion 2c to translate the program to the language $L(G_s)$.

5.2 Parsing algorithm modification

In fig. 2a the modified LL(1) parsing algorithm is presented. The highlighted lines distinguish it from the standard algorithm. In the given pseudo-code parsing stack is

accessed through the `Stack` variable, input buffer is accessed through the lexical analyzer object `Lexer` with methods `NextToken` returning the next token from the input stream and `CurrentToken` returning the last token that was read. The variable `t` corresponds to an additional buffer for the current token, `M` denotes the parsing table. The grammar G_s is a regular LL(1) grammar where `Any` is a regular token, therefore parsing table construction algorithm remains unmodified and the construction itself is carried out in the standard way.

```
Stack.Push($);
Stack.Push(S);

X := Stack.Peek();
t := Lexer.NextToken();
while (X ≠ $) do
  if (X = t) then
    if (t = Any) then                          (1)
      Stack.Pop();
      t := Lexer.CurrentToken();
      while (t ∉ FIRST'(Stack) and t ≠ $) do
        t := Lexer.NextToken();
      end while;
      if (t = $ and $ ∉ FIRST'(Stack)) then
        error();
      end if;
    else                                       (2)
      Stack.Pop();
      t := Lexer.NextToken();
    end if;
  elif (M[X,t] = X → Y1Y2...Yk) then           (3)
    Stack.Pop();
    for (i from k to 1) do
      Stack.Push(Yi);
    end for;
  elif (t = Any) then                          (4)
    error();
  else                                         (5)
    t := Any;
  end if;
  X := Stack.Peek();
end while;

if (t = $) then                                (6)
  accept();
else
  error();
end if;
```

(a)

```
BuildFirst'():
  foreach (A ∈ N) do
    MemorizedFirst'[A] := ∅
  end foreach;
  changed := true;
  while (changed) do
    changed := false;
    foreach (A → α ∈ P) do
      MemorizedFirst'[A] ∪= FIRST'(α);
      if (MemorizedFirst'[A] is changed) then
        changed := true;
      end if;
    end foreach;
  end while;
```

(b)

```
FIRST'(α = Y1Y2...Yk):
  first := ∅;
  for (i from 1 to k) do
    if (Yi∈ T\{Any}) then
      first ∪= {Yi};
      break;
    elif (Yi∈ N) then
      first ∪= MemorizedFirst'[Yi]\{ε};
      if (ε ∉ MemorizedFirst'[Yi]) then
        break;
      end if;
    end if;
  end for;
  if (∀i ∈ [1..k]: ε ∈ MemorizedFirst'[Yi]
      or Yi = Any) then
    first ∪= {ε};
  end if;
  return first;
```

(c)

Fig. 2. Modified algorithms: (a) LL(1) parsing algorithm, (b) FIRST set memorization algorithm, (c) FIRST set building algorithm

Modification of the parsing algorithm is caused by the fact that parser do a more complicated job than checking if the program is valid with respect to G_s. While

parser is generated by the simplified with respect to some G grammar G_s, the program derived by G comes as the input. As tokens are received from the input stream, the modified parser should translate the program from $L(G)$ to $L(G_s)$, then it can check the syntactic correctness of the translated part.

When the terminal symbol on the top of the parsing stack does not match the current token in `t` or when a nonterminal symbol `X` is on top of the stack and there is no record in the cell `M[X, t]`, the standard LL(1) algorithm reports an error because there is no explicit option available to continue parsing, and possibly starts an error recovery routine. For the modified algorithm, this situation is normal because, as it was said, the program does not belong to the language the parser is generated for. In case `Any` is on the top of the stack or the `M[X, Any]` cell is not empty, the modified algorithm tries to replace with `Any` some sequence of tokens from the input stream, making the transition from the text from $L(G)$ to the text from $L(G_s)$. Replacement is based on the criterion 2c: the set of tokens forming the replaced sequence must not intersect with the set of tokens which are possible `Any` successors in accordance with the parsing stack state. The successors set is called `FIRST'`, it is built by the modified version of the standard `FIRST` algorithm. This modification is discussed in Section 5.3. Obviously, $L(G) \subseteq L(G_s)$, because at the `Any` position not only valid $L(G)$ program subsequence can be replaced, but also an arbitrary sequence of tokens from the complement of a successors set. This makes parser less sensitive to possible errors in water regions.

It is possible to draw some parallels between the modification given and well-known error recovery algorithms [21, pp. 228–231, pp. 295–297]: `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. There are grounds for such an analogy. The program parsed is erroneous in terms of G_s. Replacing tokens with `Any`, the parser looks for a place from which the program satisfies the grammar again. However, behind a skin-deep similarity, there is a fundamental difference in goals, implementation and results obtaining by the algorithms. Standard error recovery is performed when a program processed is clearly incorrect. The main goal of the recovery is to resume parsing at any cost. Some significant results of the previous analysis can be discarded, and a significant part of the input stream, possibly containing some points of interest, is discarded too. In addition, recovery is not guaranteed to be successful. According to Section 5.1, the goal of `Any` processing is the translation of a presumably valid $L(G)$ program to $L(G_s)$. 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 discarding totally predictable. One can be sure that the parts of the input stream replaced with `Any` belongs to the water and can be skipped without loss of the land. Furthermore, as it was previously noted, predictable and correct replacement with `Any` is possible in some cases even for programs that are incorrect with respect to G.

Further, speaking of the fact that `Any` successfully replaces a sequence of tokens of the input program, we will simply say in some cases that `Any` *matches* this sequence. Keep in mind, that, as shown below, this process is more complex than the standard token matching.

5.3 The problem of consecutive «Any»

To get tokens denoting the end of the sequence that corresponds to `Any`, the first intention is to build the standard `FIRST` set for a parsing stack, treating the symbols on the stack as a string starting from its top. Unfortunately, there is a case when the standard `FIRST` algorithm is not enough. Sometimes two or more `Any` tokens can follow each other at the beginning of the sentences which can be derived from the stack. For a grammar

```
A = Any B C; B = a | ; C = Any c;
```

the `FIRST(Stack)` set built when the first `Any` is processed equals to `{a, Any}`. The `Any` token is never returned by the lexical analyzer, so, there is no chance that parser will recognize a string with no `a` tokens. As a result, a part of $L(G)$ remains uncovered by the parser, and the valid with respect to G_s program `Any Any c` will never be recognized, because there is no input program that can be transformed to it. For the example input `bbbc$`, `Any` processing starts at the first `b` and fails at the endmarker symbol `$`.

To make the parser capable to cope with a simplified grammar that allows consequent `Any` symbols in some derivations, it is needed to modify the standard `FIRST` algorithm on the basis of the definition 2. According to it, `Any` denotes the place where the matched sequence from an input program may be empty. In the example above, the `c` terminal which is explicitly presented in the grammar can be treated as the end of the sequence to replace, if we assume that the sequence matched by the second `Any` is empty. Acting under this assumption, the modified algorithm should expand the `FIRST` set with the tokens that may follow the last of the subsequent `Any` symbols. This turns the standard `FIRST` set into the `FIRST'` used in Fig. 2a.

In fig. 2b and fig. 2c, modified algorithms for `FIRST'` construction are presented. In fig. 2b, there is an adopted version of the algorithm from [14, pp. 239–240]. It performs non-recursive construction of `FIRST'` sets for all the nonterminals in the grammar. The sets constructed are memorized in the `MemorizedFirst'` dictionary. The original algorithm is proven to be finite, the same proof is valid for the adopted version. `FIRST'` itself is presented in Fig. 2c. Note that `Any` is not placed in the `FIRST'` set.

As shown in Section 6.1, when to match a sequence of `Any` is the only available option for processing some part of the input, `FIRST'` helps to find the actual input subsequence corresponding to the whole sequence of `Any` symbols. Technically, in this case input subsequence is matched by the first `Any`, the following `Any` symbols

match empty sequences. This is the only possible solution for a simplified grammar, because to say for sure how to precisely establish a pairwise match between the parts of the input subsequence and consecutive `Any` symbols, we need more information about the original G grammar. A similar problem called *overlapping seas* is discussed in [19]: when one sea may follow another, it is impossible to distinguish between the `after-water` of the first sea and the `before-water` of the second, so the second water is believed to be empty.

The suggested `FIRST'` modification is proven to be enough to develop a working tolerant grammar for the real programming language.

6. Experiments

6.1 Model example

Consider the following grammar:

```
A = B Any C; B = a | Any c; C = Any b | c;
```

The corresponding parsing table is presented in Table 1. The rows correspond to the nonterminal symbols defined in the grammar, the columns correspond to the tokens that may appear in the buffer `t`. Each cell contains the alternative that should be applied when the row nonterminal is on the top of the parsing stack and the column terminal is the lookahead token. The work of the modified parsing algorithm for a given input string is described in Table 2. Each row corresponds to the iteration of the outer `while` cycle in Fig. 2a, the last row corresponds to the action that takes place right after exiting the cycle. The numbers in the **Action** column correspond to the conditions numbered in Fig. 2a, the number of the true condition for the current iteration is placed in the table cell.

Table 1. Parsing table for the model example

	a	**b**	**c**	**Any**	**$**
A	$A \to B\ Any\ C$			$A \to B\ Any\ C$	
B	$B \to a$			$B \to Any\ c$	
C			$C \to c$	$C \to Any\ b$	

This example illustrates some of the advantages of our `Any` implementation, that were declared earlier. In contrast to the situation discussed for the Coco/R parser generator and the grammar in Fig. 1a, at the 4th iteration, the first `a` token in the input is included in the sequence being matched by `Any`, because the `Any` symbol is really rivalled by `a` only at the 1st iteration where the choice between $B \to a$ and $B \to Any\ c$ productions has to be made. The 7th iteration reveals the situation specified in Section 5.3: there is a derivation where two `Any` follow each other. Searching for all the tokens that may appear after `Any` in $A \to B\ Any\ C$ in accordance to the parsing stack, the `FIRST'` algorithm looks beyond the `Any`, which is in the beginning of $C \to Any\ b$, and considers `b` as the possible successor

of the sequence that should be matched by the current `Any`. As mentioned earlier, in case `Any` is immediately followed by other `Any` symbols, a sequence of input tokens of the maximum possible length is replaced with the first `Any`, and subsequent `Any` symbols correspond to zero-length subsequences of the input.

Table 2. Tracing table

	Stack	**Input**	**X**	**T**	**Action**	**Remark**
1	$ A	bacaab$	A	b	(5)	
2	$ A	bacaab$	A	Any	(3)	
3	$ C Any B	bacaab$	B	Any	(3)	
4	$ C Any c Any	bacaab$	Any	Any	(1)	$\mathrm{FIRST}'(c\ Any\ C) = \{c\}$
5	$ C Any c	caab$	c	c	(2)	
6	$ C Any	aab$	Any	a	(5)	
7	$ C Any	aab$	Any	Any	(1)	$\mathrm{FIRST}'(C) = \{b, c\}$
8	$ C	b$	C	b	(5)	
9	$ C	b$	C	Any	(3)	
10	$ b Any	b$	Any	Any	(1)	$\mathrm{FIRST}'(b) = \{b\}$
11	$ b	b$	b	b	(2)	
12	$	$	$	$	(6)	

Dynamically performed computation of the set of symbols that may follow `Any` takes into account the actual outer context for the alternatives that are matched (this context is formed by the elements that are lower on the stack, than the current alternative), rather than all the possible outer contexts which can arise according to the grammar.

6.2 Real-world repositories analysis

To test the algorithm on real source code repositories, the island grammar for the C# programming language was developed. The generated parser was applied to the repositories of three industrial projects ranked from the smallest to the largest by the number of files with a source code: the LanD project itself (93 files), PascalABC.NET[3] (2725 files), and Roslyn[4] (8027 files). PascalABC.NET is a programming language which combines Pascal syntax with .NET framework functionality. The corresponding project consists of compiler and IDE sources. Roslyn is a pair of open-source compilers for C# and Visual Basic. Roslyn project includes compiler sources and lots of test files capturing different complex and uncommon variants of a C# program. The number of files in the corresponding repositories relevant at the time of experiment conducting is given in brackets.

[3] https://github.com/pascalabcnet/pascalabcnet

[4] https://github.com/dotnet/roslyn

```
namespace_content  =  opening_directive*! (attribute|namespace|namespace_member)*
opening_directive  =  ('using'|'extern') Any ';'
namespace =  'namespace' name '{' namespace_content '}'
namespace_member   =  name? (enum|delegate|class_struct_interface)
enum      = 'enum' name Any '{' Any '}' ';'?
delegate = 'delegate' name before_body? ';'
class_struct_interface = ('class'|'interface'|'struct') name Any '{' class_content_element* '}' ';'?
class_content_element  = attribute | keyword_marked_entities
                       | name (keyword_marked_entities | class_member_tail)
keyword_marked_entities =  enum | delegate | class_struct_interface | operator | event
operator           =  'operator' Any arguments class_member_tail
event              =  'event' name class_member_tail
class_member_tail =  before_body? (block init_value? | initializer | ';')
before_body        =  Any ':' (arguments|Any)*
initializer        =  init_expression | init_value
init_expression    =  '=>' (Any|block)* ';'
init_value         =  '=' (Any|block)* ';'
name               =  (ID|arguments|'extern') name_tail_element*
name_tail_element =  ID|arguments|'extern'|'.'|'?'|'<' name_tail_element* '>'|'[' Any ']'|','|'::'

attribute  =  '[' (Any|attribute)* ']'
block      =  '{'(Any|block)* '}'
arguments  =  '(' (Any|arguments)* ')'
```

Fig. 3. Rules of the tolerant C# grammar for LanD parser generator

Rules from C# tolerant grammar are presented in Fig. 3, the complete grammar can be found in LanD project repository[5]. Water rules are highlighted. Symbol `*` denotes zero or more element repetitions, `+` denotes one or more repetitions, `?` denotes an optional element, brackets `()` are used for grouping. Quantifiers of a special kind, `*!` and `?!`, are used to set the non-empty alternative priority in case the ambiguity is detected at the parsing table construction stage. With their help, in particular, the dangling else problem is solved in the Pascal language grammar:

```
if = 'if' Any 'then' operator ('else' operator)?!
```

In the C# grammar, the `*!` construct is used to distinguish between `extern alias` declaration and the header of a method written in an unmanaged code. Though these constructs do not appear at the same nesting level in real programs, they are allowed to do so according to the lightweight grammar. This results in ambiguity that needs an additional priority indication.

As it can be seen, `Any` is widely used for denoting places which are insufficient for points of interest capturing. Such irrelevant areas are inheritance specification and type restrictions in class definitions (`before_block` nonterminal), field and property initializers (`initializer` nonterminal and nonterminals which are directly derivable from it). The largest parts that are matched by `Any` are blocks of code in method bodies (`block` nonterminal). A detailed description of these areas would make the grammar several times longer. In the corresponding anti-pattern formulated with `Any`, only a minimal structuring information should be placed: boundary tokens `{` and `}` are specified and self-nesting is explicitly allowed to ensure that boundaries will be matched pairwise. This technique is also used for

[5] https://github.com/alexeyvale/SYRCoSE-2018/blob/master/LanD Specifications/sharp.land

`attribute` and `arguments` entities, so it can be said that it forms a sustainable grammar writing pattern.

`Any` also appears in some patterns, such as `enum`, `class_struct_interface`, `operator`, denoting lakes among the land. Lakes can mark irrelevant places as well as places for which we are interested only in the list of matched tokens, not in the correct subtree specifying the deeper structure.

Table 3. Numbers of unparsed files per C# grammar refinement iteration.

	LanD	**PascalABC.NET**	**Roslyn**
0	8	-	-
1	0	39	-
2	0	0	209
3	0	0	31
4	0	0	3

In Table 3, the quantitative data describing the grammar refinement process is provided. The first column contains the number of refinement iterations passed. In the table cells, there are numbers of files from each project which still cause parsing failure. Having started with the smallest project, the LanD itself, we included the bigger ones to the testing process as the grammar became refined enough to produce parser capable to parse all the files under consideration. For two refinement iterations, the number of errors for LanD and PascalABC.NET was reduced to zero. Surprisingly, even so we got a significant number of erroneously parsed and unparsed files for Roslyn (209 files out of 8027). Analyzing them we found out that it was caused by tuple types and tuple literals. It is one of the new features added to C# 7.0. These constructs may look exactly like method arguments, causing confusion during parsing. The problem was solved by the less restrictive class member patterns description: the entire header is matched by the `name` pattern which includes the `arguments` pattern. The `arguments` pattern matches method arguments as well as tuple types. A more accurate division of `name` into modifiers, a type, an entity name and arguments was moved at the automatically built syntax tree post-processing stage. Expression bodied properties became another cause of errors. They are widely used in Roslyn but are not presented in LanD and PascalABC.NET. To process them as the water, the `init_expression` anti-pattern was added and the `init_value` anti-pattern was refined.

At the last iteration of grammar testing and refinement, the number of errors is still non-zero. However, on closer inspection it was proved to be not a consequence of inaccuracies in the grammar structure. The first file[6] is a test file for the Roslyn compiler, it contains the text of the program in Shift-JIS encoding, which is used for Japanese, moreover, the class name is written in Japanese. The latter causes a lexical

[6] https://github.com/dotnet/roslyn/blob/master/src/Compilers/Test/Resources/Core/Encoding/sjis.cs

analysis error. We consider the usage of national alphabets for entity naming to be a rare case, but, if necessary, the `ID` token can be adopted as needed. The second file[7] also belongs to the testing infrastructure, it contains a meta-information in a form of invalid global code: there is a string field, declared directly inside the namespace but outside of the class. In the third file[8], the code containing `using` directives and a class definition is placed after the namespace definition. This code is enclosed in `#if false` preprocessor directive, so it is not compiled after the preprocessing stage. Our tolerant parser works with the pure sources and ignores the directives, so it justifiably treats this program as incorrect.

The resulting C# grammar is aimed at all-encompassing parsing of all the possible valid C# code variations from three real-world software projects, at the same time it is both tolerant with respect to code in places indicated with `Any`, and lightweight. For instance, the baseline C# parser description[9] for the industrial compiler generator ANTLR, which uses an extended LL(*) algorithm [22], contains 1159 lines, and lexical analyzer specification contains 1101 lines. The text of our tolerant LL(1) C# grammar has (including token definitions and different generator options) just 51 lines. Developing a parser for a certain project, one can make the grammar even more lightweight if some project-specific restrictions are known. In case some coding conventions are applied, land and water content become less variable. If a legacy code is parsed, one can be sure that the latest language features are not in use there, so the grammar is allowed not to contain patterns and anti-patterns for them.

At the next stage of the experiment, the syntax trees of the parsed files were used to calculate the numbers of successfully discovered LanD entities that we are interested in, solving the code markup task. As control numbers, the results of counting the same entities using syntax trees built by Roslyn were used. The entities were grouped into five categories: enums, classes, fields, properties, methods. The grouping is carried out in accordance with the hierarchy of classes representing the nodes of a syntactic tree in Roslyn. Entities which corresponds to Roslyn tree nodes of type `BaseFieldDeclarationSyntax` are marked as fields. These are fields themselves, as well as events described without access methods. Elements corresponding to nodes of types inherited from `BasePropertyDeclarationSyntax` are treated as properties. In addition to properties themselves, these are indexers and events with explicitly specified `add` and `remove` accessors. Methods correspond to `BaseMethodDeclarationSyntax` type: it is the parent type for method, constructor, destructor, and operator nodes.

[7] https://github.com/dotnet/roslyn/blob/master/src/Compilers/Test/Resources/Core/SymbolsTests/Metadata/public-and-private.cs

[8] https://github.com/dotnet/roslyn/blob/master/src/Workspaces/Core/Portable/Shared/Extensions/ObjectExtensions.cs

[9] https://github.com/antlr/grammars-v4/tree/master/csharp

In Table 4, the quantitative results are presented. For all projects in all categories, LanD detects more entities than Roslyn. The difference is caused by the conditional compilation directive `#if`, which is actively used in the projects under consideration. For example, in PascalABC.NET the `#if DEBUG` construct is widely used to enable debug output and additional information collecting, conditional compilation is also presented in the sources of the syntax analyzers, which are generated with GPPG.

Table 4. Number of entities found by Roslyn/LanD.

	Enums	**Classes**	**Fields**	**Properties**	**Methods**
LanD	13/14	94/95	390/390	248/253	431/436
PABC	356/363	4611/4622	16720/16753	12326/12350	42248/42386
Roslyn	437/441	21583/21622	19606/19737	21886/21919	108040/108400

Roslyn parser has an integrated preprocessor which resolves `#if` conditions and pass to the parsing stage only the appropriate parts of the code. LanD is a language-independent tool, so it does not have a built-in notion of directives. For a lightweight parser, directives are defined as single-line lexemes which are usually skipped. As a result, LanD statistics take into account all the entities regardless of whether or not they are enclosed in the `#if` directive with an undefined symbol. It should be noted that C# preprocessing is a fairly simple task. If necessary, the correct preprocessor can be easily written and applied to the text passed to the LanD-generated C# parser. However, this will lead to a loss of information about the areas excluded by the preprocessor.

7. Conclusion

In the present paper, the LL(1) parsing algorithm modification is proposed. This modification is intended for performing tolerant parsing based on the island grammars technique. The special `Any` symbol is integrated into the algorithm to add a capability to match token sequences which are not explicitly described in the grammar. With regard to island grammar development, the presence of `Any` simplifies the description of water and partially eliminates the need to describe the structure and variations of irrelevant areas. Besides, `Any` can be used for relevant code description in case this code contains lakes — areas for which we are interested only in pure token sequence, not in the structural information. Our `Any` implementation fixes the shortcomings of the closest analogues. It is more accurate and less restrictive in comparison with Coco/R and LightParse parser generators, it is also more simple than bounded seas approach, and still powerful enough to parse sources of large-scale software projects. It is experimentally proved that the lightweight parser of the C# language with built-in automatic construction of the syntax tree, which was developed by the authors of the current paper, makes it possible to successfully analyze the source codes of industrial software products and provides one hundred percent finding of points of interest. The developed generator

of lightweight parsers is planned to be used in solving the sustainable code markup problems.

Tolerant grammar description and syntax tree post-processing are supposed to be simplified by integrating the *Schrödinger's token* concept [13] into lexical and syntax analyzers. In particular, it can be useful for analyzing C# language where, along with reserved keywords, there are contextual keywords. Some of them (for example, words `where` and `partial`) directly affect the separation of land and water and the land structure analysis. Possible directions for further research are also a more intelligent resolution of the consecutive `Any` problem and integration of the `Any` symbol into LR(1) parsing algorithm.

References

[1]. Goloveshkin A.V. Searching and analysing crosscutting concerns in marked up programming language grammar. Izvestija vuzov. Severo-Kavkazskij region. Tehnicheskie nauki [University News. North-Caucasian Region. Technical Sciences Series], 2017, issue 3, pp. 29–34 (in Russian). DOI: 10.17213/0321-2653-2017-3-29-34.

[2]. Afroozeh 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, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers. Springer Berlin Heidelberg, 2013, pp. 224–243.

[3]. Van den Brand M., Sellink M.P.A., Verhoef C. Obtaining a COBOL grammar from legacy code for reengineering purposes. In Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications. BCS Learning & Development Ltd., 1997, pp. 6–16.

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

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

[6]. Graham S.L., Haley C.B., Joy W.N. Practical LR error recovery. SIGPLAN Notes, vol. 14, issue 8, 1979, pp. 168–175.

[7]. Burke M.G., Fisher G.A. A practical method for LR and LL syntactic error diagnosis and recovery. ACM Trans. Program. Lang. Syst., vol. 9, issue 2, 1987, pp. 164–197.

[8]. De Jonge M., Nilsson-Nyman E., Kats L.C.L., Visser E. Natural and flexible error recovery for generated parsers. Software Language Engineering: Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers. Springer Berlin Heidelberg, 2010, pp. 204–223.

[9]. Nilsson-Nyman E., Ekman T., Hedin G. Practical scope recovery using bridge parsing. Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers. Springer Berlin Heidelberg, 2009, pp. 95–113.

[10]. Koppler R. A systematic approach to fuzzy parsing. Software: Practice and Experience, vol. 27, issue 6, 1997, pp. 637–649.

[11]. Carvalho P., Oliveira N., Henriques P.R. Unfuzzying fuzzy parsing. 3rd Symposium on Languages, Applications and Technologies, ser. OpenAccess Series in Informatics (OASIcs), vol. 38, 2014, pp. 101–108
[12]. S. Klusener and R. Lämmel, Deriving tolerant grammars from a base-line grammar. In Proceedings of the International Conference on Software Maintenance. IEEE Computer Society, 2003, pp. 179–188.
[13]. Aycock J., Horspool R.N., Schrödinger's token. Software: Practice and Experience, vol. 31, issue 8, 2001, pp. 803–814.
[14]. Grune D., Jacobs C.J. Parsing Techniques: A Practical Guide (2nd Edition). Springer-Verlag, New York, 2008, 662 p.
[15]. Scott E., Johnstone A. GLL parsing. Electron. Notes Theor. Comput. Sci., vol. 253, issue 7, 2010, pp. 177–189.
[16]. Mössenböck H. (2014) The compiler generator Coco/R. Available at: http://ssw.jku.at/Coco/Doc/UserManual.pdf, accessed 02.03.2018.
[17]. Malevannyy M. Lightweight parsing and its application in development environment. Informatizatsiya i svyaz [Informatization and communication], 2015, vol. 3, pp. 89–94 (in Russian).
[18]. Malevannyy M.S., Mikhalkovich S.S. Context-based model for concern markup of a source code. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 2, pp. 63–78. DOI: 10.15514/ISPRAS-2016-28(2)-4.
[19]. Kurš J., Lungu M., Iyadurai R., Nierstrasz O. Bounded seas. Comput. Lang. Syst. Struct., 2015, vol. 44, pp. 114–140
[20]. Ford B. Packrat parsing: Simple, powerful, lazy, linear time. Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ser. ICFP '02. ACM, 2002, pp. 36–47.
[21]. Aho A.V., Lam M.S., Sethi R., Ullman J.D. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., 2006, 1000 p.
[22]. Parr T., Harwell S., Fisher K. Adaptive LL(*) parsing: The power of dynamic analysis. SIGPLAN Notes, vol. 49, issue 10, 2014, pp. 579–598.

Толерантный синтаксический анализ с использованием специального символа «Any»: алгоритм и практическое применение

А.В. Головешкин <alexeyvale@gmail.com>
С.С. Михалкович <miks@sfedu.ru>
Институт математики, механики и компьютерных наук им. И.И. Воровича,
Южный федеральный университет,
344090, Россия, г. Ростов-на-Дону, ул. Мильчакова, д. 8а

Аннотация. Толерантный синтаксический анализ позволяет найти области программы, представляющие интерес в контексте конкретной задачи, и извлечь информацию об их структуре. В то время как эти области должны быть подробно описаны в грамматике языка, другие части программы могут быть не описаны совсем или описаны менее детально, при этом генерируемый парсер должен признавать корректными все возможные вариации программы в нерелевантных областях, то есть, должен быть толерантным по отношению к ним. Островные грамматики — один из основных

способов реализации толерантного парсинга. Термином «остров» обозначаются релевантные области кода, термином «вода» — нерелевантный код. В настоящей работе описывается модифицированный LL(1) алгоритм со встроенной обработкой специального символа «Any», позволяющего сопоставлять последовательности токенов, не описанные разработчиком грамматики в явном виде. Применение данного алгоритма к островным грамматикам ведёт к сокращению описания воды и упрощению описания островов. Наша реализация «Any» является более безопасной для использования и менее ограничительной по сравнению с ближайшими аналогами в генераторах Coco/R и LightParse. Также она более предсказуема и требует меньших накладных расходов в сравнении с концепцией «ограниченных морей», внедрённой в PetitParser. На базе алгоритма реализован генератор компиляторов со встроенным языком описания островных грамматик. Как показано в разделе экспериментов, сгенерированный по островной грамматике языка C# толерантный парсер может быть успешно применён для анализа крупных промышленных проектов.

Ключевые слова: толерантный парсинг; устойчивый парсинг; легковесный парсинг; частичный парсинг; островная грамматика; генерация парсеров

DOI: 10.15514/ISPRAS-2018-30(4)-1

Для цитирования: Головешкин А.В., Михалкович С.С. Толерантный синтаксический анализ с использованием специального символа «Any»: алгоритм и практическое применение. Труды ИСП РАН, том 30, вып. 4, 2018 г., стр. 7-28 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(4)-1

Список литературы

[1]. Головешкин А.В. Поиск и анализ сквозных функциональностей в размеченной грамматике языка программирования. Известия вузов. Северо-Кавказский регион. Технические науки, 2017, вып. 3, стр. 29–34. DOI: 10.17213/0321-2653-2017-3-29-34.

[2]. Afroozeh 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, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers. Springer Berlin Heidelberg, 2013, pp. 224–243.

[3]. Van den Brand M., Sellink M.P.A., Verhoef C. Obtaining a COBOL grammar from legacy code for reengineering purposes. In Proceedings of the 2nd International Conference on Theory and Practice of Algebraic Specifications. BCS Learning & Development Ltd., 1997, pp. 6–16.

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

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

[6]. Graham S.L., Haley C.B., Joy W.N. Practical LR error recovery. SIGPLAN Notes, vol. 14, issue 8, 1979, pp. 168–175.

[7]. Burke M.G., Fisher G.A. A practical method for LR and LL syntactic error diagnosis and recovery. ACM Trans. Program. Lang. Syst., vol. 9, issue 2, 1987, pp. 164–197.

[8]. De Jonge M., Nilsson-Nyman E., Kats L.C.L., Visser E. Natural and flexible error recovery for generated parsers. Software Language Engineering: Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers. Springer Berlin Heidelberg, 2010, pp. 204–223.

[9]. Nilsson-Nyman E., Ekman T., Hedin G. Practical scope recovery using bridge parsing. Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers. Springer Berlin Heidelberg, 2009, pp. 95–113.

[10]. Koppler R. A systematic approach to fuzzy parsing. Software: Practice and Experience, vol. 27, issue 6, 1997, pp. 637–649.

[11]. Carvalho P., Oliveira N., Henriques P.R. Unfuzzying fuzzy parsing. 3rd Symposium on Languages, Applications and Technologies, ser. OpenAccess Series in Informatics (OASIcs), vol. 38, 2014, pp. 101–108

[12]. S. Klusener and R. Lämmel, Deriving tolerant grammars from a base-line grammar. In Proceedings of the International Conference on Software Maintenance. IEEE Computer Society, 2003, pp. 179–188.

[13]. Aycock J., Horspool R.N., Schrödinger’s token. Software: Practice and Experience, vol. 31, issue 8, 2001, pp. 803–814.

[14]. Grune D., Jacobs C.J. Parsing Techniques: A Practical Guide (2nd Edition). Springer-Verlag, New York, 2008, 662 p.

[15]. Scott E., Johnstone A. GLL parsing. Electron. Notes Theor. Comput. Sci., vol. 253, issue 7, 2010, pp. 177–189.

[16]. Mössenböck H. (2014) The compiler generator Coco/R. Available at: http://ssw.jku.at/Coco/Doc/UserManual.pdf, accessed 02.03.2018.

[17]. Малёванный М.С. Легковесный парсинг и его использование для функций среды разработки. Информатизация и связь, 2015, том 3, стр. 89–94.

[18]. Malevannyy M.S., Mikhalkovich S.S. Context-based model for concern markup of a source code. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 2, pp. 63–78. DOI: 10.15514/ISPRAS-2016-28(2)-4.

[19]. Kurš J., Lungu M., Iyadurai R., Nierstrasz O. Bounded seas. Comput. Lang. Syst. Struct., 2015, vol. 44, pp. 114–140

[20]. Ford B. Packrat parsing: Simple, powerful, lazy, linear time. Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ser. ICFP ’02. ACM, 2002, pp. 36–47.

[21]. Aho A.V., Lam M.S., Sethi R., Ullman J.D. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., 2006, 1000 p.

[22]. Parr T., Harwell S., Fisher K. Adaptive LL(*) parsing: The power of dynamic analysis. SIGPLAN Notes, vol. 49, issue 10, 2014, pp. 579–598.