MASTER IN COMPUTER SCIENCE

Master Thesis
Philosophisch-naturwissenschaftliche Fakultät der
Universität Bern

Error Handling in PEG Parsers

Local Error Recovery for PetitParser

submitted by
Michael Rüfenacht
13. July 2016

supervised by
Prof. Dr. Oscar Nierstrasz
Jan Kurš

Software Composition Group
Institute of Computer Science
University of Berne, Switzerland

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

Parsing Expression Grammars (PEGs) provide a convenient, highly expressive and concise formalism to specify top-down parsers. The characteristics of PEGs solve numerous problems arising in generative approaches, especially in terms of ambiguity and the composition of programming languages. Nevertheless, corresponding parsers often suffer from poor error handling which is aggravated by the fact that classical error handling techniques — designed for non-backtracking, top-down parsing algorithms — are not directly applicable to PEGs. However, the input to a parser in practice is likely to contain syntax errors, especially in the context of tools requiring human interaction such as Integrated Development Environments (*IDEs*). Meaningful feedback and the ability to recover from typing errors and misconceptions are crucial to a programmer's productivity or the reliability of tools including parsing processes in general. To enable improved error handling capabilities, consisting of error recovery as well as error correction, we present the foundations on how to overcome the limitations of a backtracking PEG parsing combinator. As a proof of concept we created an automated and language agnostic implementation of the error handling scheme for PetitParser. We adapt the scheme to both the lexical as well as the syntactical level of a grammar and subsequently combine the adaptations into a layered error handling scheme. Our approach incorporates a classical local error recovery approach also known as acceptable-set recovery. The engine gathers its recovery information directly from the syntactic structure of the parser reducing the involvement of a programmer and the implementation of *ad-hoc* recovery mechanisms (such as error productions) to a bare minimum. We describe the necessary steps to enable error handling in existing parsers and evaluate the recovery quality of our implementation in the form of a case study of the JSON grammar.

Contents

1
Introduction

Parsing, the recognition and analysis of input strings based on formal grammars is a fundamental concept of computer science and inherent to software engineering. As a consequence, researchers have developed numerous systems to formally define grammars and algorithms to verify if a sentence is a member of the corresponding language [1, 3, 24]. Nevertheless, in practice especially systems requiring human interaction are likely to encounter inputs which do not fit their specification. These strings are considered to be erroneous. Compilers and applications such as integrated development environments (IDEs) which provide generic toolsets for language and software development are expected to be able to deliver meaningful feedback to their users at all times; especially in the presence of errors. A programmer expects an application to be robust, to deliver meaningful syntactical and semantical information and maintain – at least partial – formatting of the input. To fulfill such requirements these systems need to provide sophisticated error handling capabilities consisting of error reporting, error recovery and error correction. Manually enhancing a parser to be able to perform such operations is a rather tedious and error prone task. Automated — or at least systematic — syntax error handling is a complex topic having its origins in the 70 [34, 35, 47]. There exists a variety of error handling approaches and formalizations for LL as well as LR languages and their respective top-down or bottom-up parsing algorithms [7, 26, 53, 61] usually formalized using *context-free grammars* (CFGs).

In contradiction to CFGs, *parsing expression grammars* (PEGs) [17] are a relatively new approach to specify formal languages. They provide a concise syntax to specify a grammar and the corresponding backtracking top-down parser consisting of the lexical as well as the syntactical specification. Having the lexical structure defined directly in the grammar removes the necessity of a separate lexical analyzer /

scanner. PEGs are closed under composition thus enabling the composition and embedding of languages. PEGs rely on an ordered choice combinator which simplifies the handling of ambiguous rules and makes the resulting parse inherently deterministic. Despite these desirable characteristics and their conceptual elegance, PEGs suffer from poor error handling capabilities. Backtracking obfuscates the state of the parser at the time an error is encountered by requiring the parser to check all remaining branches [10, 24]. A failing choice always returns the latest occurring failure emitted by the last remaining rule. Due to the conceptual differences to CFGs, classical ways of reasoning on grammars and known error handling techniques are not directly applicable to PEGs.

In this thesis we elaborate an approach on how to overcome the limitations of PEGs in error handling. We leave the original formalism mostly unchanged and implement our solutions on top of its mechanics. Our approach is automatic and language agnostic,[1] making its application to existing parsers generic and reducing the involvement of a language specialist to a bare minimum. We develop the foundations for improved error detection in PEG parsers giving the error handling parser the ability to detect syntax errors at the position they appeared. This thesis contains informal descriptions on how to recreate the state of a parser using a special kind of configuration, analyze the types of syntax errors and compute corresponding ways to resume the parsing using a new concept called remainder grammars. Based on these ideas we developed a local error handling scheme relying on acceptable-sets which is adapted for the lexical as well as the syntactical level of the underlying grammar. The resulting scheme is designed to work with parsers defined by PEGs and therefore consistently relies on its mechanics. This means we have created a backtracking error handling scheme which in cases of ambiguity behaves as the prioritized choice implies: it succeeds with the first succeeding alternative. To evaluate the quality of the scheme, we propose and briefly describe a randomized version of mutation testing for different types and different numbers of errors. As a consequence we are able to generate corresponding cases based on a random set of valid input strings. The contributions of our work to error handling in PEGs consist of:

- An assessment of the problems arising in error handling for PEGs with focus on the characteristics to enable error recovery.
- The description of an automated, language agnostic, local error handling scheme to cope with PEGs. The scheme is based on existing error recovery approaches and explicitly targets the characteristics of PEGs and their implementation as backtracking, scannerless parser combinators.
- The derivation of concrete error handling strategies targeting syntactical as well as lexical elements of the formalism relying the fundamental workings of the general scheme.
- A proof of concept implementation of the presented strategies for the PetitParser framework.
- A brief introduction to our methodology to automatically evaluate and measure the quality of the created implementations.
- The collection and discussion of empirical data gained by performing a case study considering the grammar of the *Javascript Object Notation* (JSON).

[1] Within the restrictions of a PEG

The remainder of this thesis is organized as follows: In Chapter 2 we give a more specific introduction to the topic. Chapter 3 elaborates the difficulties PEGs and the PetitParser framework cause in terms of error handling. Chapter 4 presents our attempts to solve the aforementioned caveats. Chapter 5 contains results of our evaluation as well as the underlying methodology. In Chapter 6 we briefly describe the state of research in terms of related work while Chapter 7 concludes and proposes future work.

2
Prerequisites

2.1 Parsing Expression Grammars

Parsing expression grammars (PEGs) [17] form a concise, recognition-based formal foundation to specify languages as recursive, backtracking top-down parsers. PEGs combine the lexical as well as the syntactical specification of a language into a single formalism, removing the necessity of a separate lexical analyzer (scanner) [55, 62]. Formally, a PEG is a 4-tuple:

$$G = (V_N, V_T, R, e_s)$$

V_N is a finite set of nonterminals whose members are denoted using uppercase letters. V_T is a finite set of terminal symbols whose members are denoted using lowercase letters. V_N and V_T are distinct sets, meaning $V_N \cap V_T = \emptyset$. R is a finite set of productions, consisting of rules r having the form $A \leftarrow e$, where e is called a parsing expression and $A \in V_N$. For every nonterminal A exists exactly one rule r. R is therefore a function $R(A)$, mapping a nonterminal A to a parsing expression e.[1] Finally, e_s defines the start expression of the grammar. All parsing expressions can be inductively created using the terminal expressions given in Table 2.1 and their composition listed in Table 2.2. Our tables depict the full set of syntactic elements and their desugared versions resulting in an abstract version of PEGs as given by Ford as well as *e.g.* Medeiros [39, 40].

Parsing expressions can either succeed by returning the consumed input prefix or fail by emitting a

[1] The righthand side of the rule.

Expression	Desugared	Description	Precedence
' '		terminal string	5
$[a - z]$	'a' / 'b' / ... / 'z'	character classes	5
.		any element (of V_T)	5
ϵ		the empty word	5

Table 2.1: terminal expressions

Expression	Desugared	Description	Precedence
e_1 / e_2		alternative composition (prioritized choice)	1
e_1 e_2		sequential composition	2
&e	!!e	and predicate (non consuming lookahead)	3
!e		not predicate (non consuming inverted lookahead)	3
$e*$		e, repeated zero or more times (kleene closure), possessive	4
$e+$	e $e*$	e, repeated one or more times, possessive	4
$e?$	e / ϵ	optional expression	4
(e)		grouping	5

Table 2.2: expression combinators

failure state f. The invocation of a parsing expression e is specified by Ford as $(e, w) \Rightarrow (n, o)$ where w is the input to consume, n a step counter, $o \in V_T^* \cup \{f\}$ the produced output and $\Rightarrow$ is a mapping representing the recognition process. Every parsing expression can succeed without having to consume the full input, qualifying PEG based parsing as a prefix matching algorithm. *Parsing expression languages* (PELs, *i.e.* the languages which can be expressed by PEGs) are closed under union, intersection and complement which enables the combination of PEG parsers and therefore language embedding. Languages implemented using other formalisms *i.e.* requesting a scanner can usually not be combined. Their lexical specifications are likely to be ambiguous or contradictory [62].

Despite the syntactic similarities to CFGs and regular-expressions (REs), there exist some fundamental differences. The main difference between PEGs and CFGs is the way disambiguation is achieved. CFGs allow ambiguous choices and therefore nondeterministic behavior which has to be resolved using – usually bounded – lookahead. PEGs rely on a prioritized choice combinator which succeeds with the first succeeding alternative or backtracks on failure. Prioritization means that rules are combined and matched in the order they are specified. A PEG, in contradiction to a CFG, hence describes how to parse a language [11]. As a consequence, the parse-tree of a PEG based parser is always unambiguous and deterministic. If the prioritized choice is not sufficient to achieve disambiguation, additional distinction is established by using non-consuming *and-* or *not-predicates* giving PEG based parsers infinite lookahead. In contrast to CFGs which follow a generative approach, the language L a PEG describes is defined through the input the corresponding parser is able to recognize (referred to as recognition-based). Furthermore, repetition quantifiers are defined to be possessive [18]. They consume as much input as possible up to the first occurrence of a failure whereas greedy matching usually involves backtracking.

Like recursive top-down parsing in general, PEGs do not support left recursion. There are proposals to overcome this limitation [40, 65] for *packrat parsers* [16] which basically are PEG parsers with memoization. Grammars without left recursion and without expressions of the form $e*$ where e can accept the empty word are considered to be *well-formed* which means they terminate. A parsing expression is considered to be *nullable* if it can accept the empty word ϵ and is therefore an element of a set denoted as NUL shown in Table 2.3. Note that we assume that it is not sufficient to qualify an expression as nullable if it is able to succeed without consuming input due to the non-consuming semantic predicates. Also it is worth mentioning that having expressions of the form $!e$ whereas $e \in NUL$ would always fail (or always succeed for $\&e = !!e$) independently from the input, causing them to be useless.

Expression	Condition
e_1 / e_2	$e_1 \in NUL \vee e_2 \in NUL$
e_1 e_2	$e_1 \in NUL \wedge e_2 \in NUL$
$e*$	
ϵ	

Table 2.3: Members of the set NUL describing nullable parsers. We left out the and- as well as the not predicate, since it does not make any sense to put a nullable parser within one of the predicates. The nullability of all other parsing expressions is computable using these rules.

In the remainder of this thesis we will always assume that the involved grammars are *well-formed* and are not nullable (*i.e.* $e_s \notin NUL$, the grammar cannot accept the empty word). It is not possible to distinguish between a PEG acceping the empty word or accepting every input which means a corresponding grammar always succeeds. Also, we assume that string literals consisting of multiple characters are syntactic sugar for character sequences. Exemplary grammars formalized using PEGs can be found in Appendix A.1 and Appendix A.2.

2.2 PetitParser

PetitParser [49, 50] is a parser combinator [13, 29, 30] framework implemented in Pharo/Smalltalk.[2] PetitParser provides common parsing extensions such as semantic actions or memoization [15] on top of the PEG formalism. PetitParser enables the programmatic composition of parsers making them reusable by design. Every parser is a concrete implementation (*i.e.* an instance) of a parsing expression e (the terms therefore might be used interchangeably in this thesis). As a consequence, the implementation of a grammar is a directed graph of parsers as depicted by Figure 2.1. Having the whole grammar available as an object graph is an advantage over the original implementations in functional languages. It enables analysis and pre-processing as well as modifications such as canonicalization of the grammar. All relations defined for PEGs also hold for these corresponding parsers. On top of the fundamentals of PEGs, PetitParser also provides a specific kind of parsers, namely token parsers which provide a grouping mechanism to model classical tokens which we will make use of.

[2] http://pharo.org

PetitParser is furthermore part of *Moose*[3], a platform for source code and data analysis. *Moose* and PetitParser provide tool support for the creation of parsers reaching from *domain specific languages* (DSLs) [28] to full-featured general-purpose programming languages.

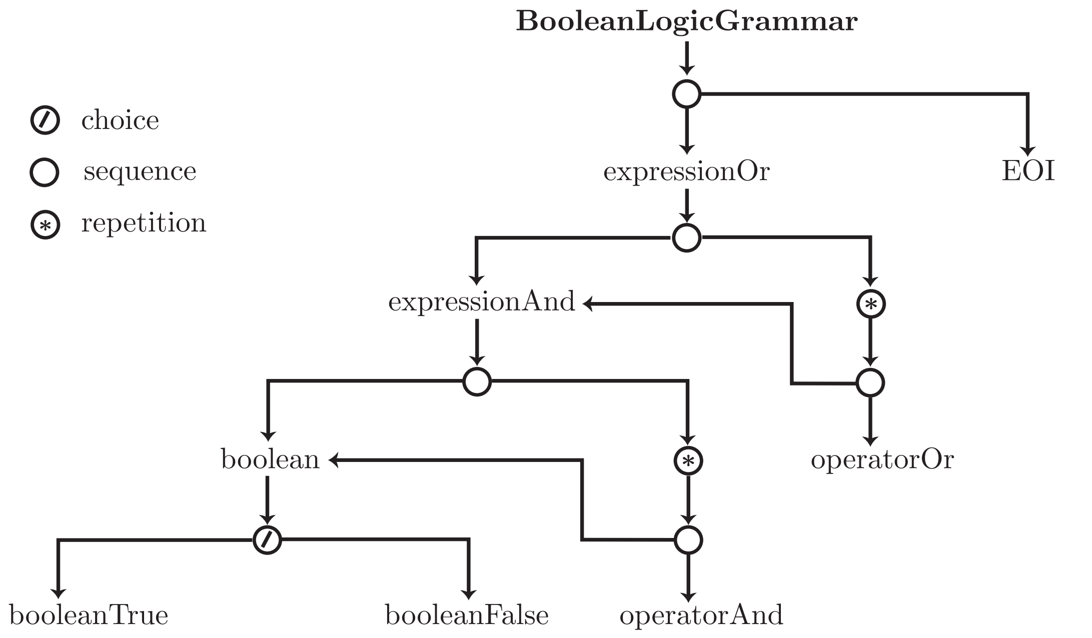

Figure 2.1: The simplified object graph of the boolean logic grammar. Every node represents a concrete parser instance and reusing them can cause cycles.

2.3 Error Handling

The handling of erroneous input by a parser denoted as error handling consists of three dependent stages: error detection, error recovery and error correction. It describes the behavior of a parser in presence of a syntax error. In general, error handling schemes in parsing exhibit different grades of impact on the resulting parse. Error handling can range between quitting the parsing process on error to a full repair of the erroneous input. An overview of error handling schemes and their characteristics can be found in standard parsing literature or surveys [7, 24, 26, 61]. Due to the subtle differences in the terminology we present a general overview of the terms and techniques used in this thesis.

2.3.1 Error Detection and Reporting

As mentioned, the input to a parser is likely to contain errors. The first requirement to a language translation system such as a parser is its ability to detect and report syntax errors. Reporting the sheer presence of a syntax error is not a sufficient error detection capability for our purposes. Especially – even though hard

[3] http://moosetechnology.org

to detect – the position an error occurred at is an essential information for further processing. Spenke *et al.* [57] consider error position to be a pragmatical term without any complete formalization. Section 3.1 elaborates this problem in more detail. Dain (in reference to Peterson [47]) describes errors as so called *parser defined errors* in a way suitable to our purposes.

> *The parser defined error in an incorrect string, with respect to a language, marks the point at which a prefix of the string ceases to be a prefix of the language. [7]*

Parsing algorithms which are able to locate an error at the symbol it occurred (such as table-driven LL and LR parsers) have the *correct-prefix property* [3, 35]. Besides the position an error occurred at, an error handling scheme should produce a meaningful message containing more information about the error. PetitParser incorporates a solid foundation concerning the automated creation of *meaningful* error messages according to Fords initial work on PEGs. Horning [27] proposes the usage of a *declarative* message in combination with an *operational* message. While the former describes the syntactical context (consisting of expectations in terms of the grammar), the latter describes the actions taken to overcome an error and therefore depends on the type of the encountered error. Even though (*declarative*) error messages are considered to be an important part of error reporting from a user's perspective [42], we will not focus on their creation in this thesis since they are of little use for automated processing. In summary we can state that a parsing error is defined through its position in the input string and its classification. All syntactical errors in programming languages can be expressed in terms of string differences using three types of errors and transformations to cause them [25]:

Addition: An error caused by the *insertion* of an unexpected symbol to the input stream.

Omission: An error caused by a missing symbol probably due to a *removal*.

Substitution: An error caused by a wrongly placed (or *substituted*) symbol in the input stream.

All these transformations can be reverted using one of these transformations. The exact meanings of the terms *transformation* and *symbol* are discussed in Chapter 4. For now a symbol is considered to be a *token*. Tokens are either *keywords*, having one single instance, or *token classes* which have multiple instances (*e.g.* strings or identifiers) [56].

2.3.2 Error Recovery and Error Repair

As elaborated, error detection can be seen as the ability to report and describe a single syntax error. The ability of detecting an error and leaving the remainder of the input unprocessed as well as not delivering any kind of parsing result might not be sufficient for an error handling scheme. If described in terms of a *finite state automaton*, a parser goes into a dedicated error state as soon as it encounters an error. It will not be able to do any more transitions which are necessary to keep on processing the remaining input. The process of transposing the parsers internal state from an erroneous state into a valid one is called error recovery.

> *Somehow, the internal state of the parser must be adapted so that the parser can process the rest of the input. This adaptation of the internal state is called error recovery. [24]*

In the following we will call this process resynchronization. Having the possibility to resynchronize the parser with the input by skipping symbols enables the error handling procedure to resume and detect errors beyond the first error. Nevertheless, the ability to detect all – or at least multiple – syntax errors in the current input string does not enable the parser to produce a parsing result such as the *concrete* or the *abstract syntax tree* (AST). To generate a valid AST, required for further analysis, the error handling must be able to transform the erroneous string in such a way that it becomes syntactically valid.

> *The process of transforming the erroneous input string into a syntactically valid form is called error correction.*

The term correction implies the result to be *correct* which is not necessarily the case. That's why we use the term error repair in the following, since it has less strict implications (on correctness). Syntax error recovery can therefore considered to be either correcting or non-correcting. The majority of error handling approaches can further be differentiated by two more properties: The amount of context they take into account to overcome or repair the erroneous part of the input and the metrics they use to decide which form of repair should be done. The metrics a recovery scheme uses to decide how to fix or skip the erroneous input directly affect the way the parsing is resumed, which is part of the error recovery.

2.3.2.1 Error Repair

Repairing errors essentially consists of modifying strings using the aforementioned transformations (insertion, removal or substitution). A repair is the process of applying transformations to the input in a way that the resulting sentence is a member of the language. There are two levels of errors. Errors at the lexical level — sometimes referred to as spelling errors — are corrected by modifying single characters. The transformation of one string into another one is applied relying on criteria describing the *similarity* of strings [57] defining strings to be *similar* if

1. they have the same length and differ in one character only.
2. one string can be transformed into the other by adding one single character.
3. one string can be transformed into the other by interchanging two neighboring characters.[4]
4. one string is a prefix of the other one.

Analogous, syntactical errors are repaired by the addition, the removal or the substitution of *tokens*. While this concept appears to be rather trivial there usually exist multiple ways of repairing an erroneous

[4] Note that the interchange of two characters cannot be described by a single transformation.

string. To decide which transformation should be favored over others, mainly two metrics are relevant: the minimum-distance metric[5] [2] and least-cost metrics [63]. The minimum-distance metric always prioritizes a repair which consists of the minimal number of transformations. Least-cost metrics rely on a cost based decision and prioritize the repair which has the lowest costs. Graham *et al.* [20, 21] propose to assign a rating to all symbols of the grammar in the context of addition, removal and substitution. This rating is called *modification costs*. They are combined with *safety costs*, measuring the *reliability* of a correction. The error handling scheme will always – independently from the amount of involved transformations – pick the repair which has the lowest accumulated costs.

2.3.2.2 Local Error Handling

Local error handling operates at the error position by modifying the remaining, erroneous suffix of the input. All local approaches rely on *acceptable-sets* [24] (we will call them recovery sets), a set of symbols which are acceptable at or after the error location. The members of these sets are used as anchors to resynchronize the parser. If an error is detected, the parser skips input symbols until a member of the recovery set is matched. The skipped part at the beginning of the suffix is replaced by a corrected version. Afterwards the state of the parser is adapted to the corresponding state indicated by the matched anchor and resumes the parsing process. Local error handling approaches do not modify the consumed prefix and mostly differ in the way they compute their recovery set (forming a family of error handling schemes). A good example is Stirling's Follow Set Recovery [58] where multiple definitions of the follow set are presented.

Often local schemes are combined with regional error recovery [5, 19]. Sippu [56] states that a possibility is to only perform a local correction if there is only one single token transformation available. This often is too restrictive to create satisfying results. More sophisticated approaches involve local least-cost recovery approaches [4] relying on aforementioned metrics. The most interesting concept with regard to PEGs is the concept of *continuations* introduced by Röhrich [52], assuming that the erroneous suffix of the input string can be expressed as a continuation of the grammar. A *continuation grammar* allows to compute appropriate recovery sets without evaluating all possible parser states and removes the necessity to restart the parsing at the beginning which is especially important in the context of a backtracking parser.

2.3.2.3 Regional Error Handling

Regional error handling has an extended context taken into account to recover from and repair errors. In contrast to local error handling, regional error handling operates on a bounded region around the error position, taking a fixed amount of symbols before and after the error into account. The exploration of these symbols is done using forward and backward moves. In regional error handling the input as well as the parsing stack is modified [7]. These schemes are restricted to *bottom-up* parsers having a parsing stack and are only listed for completeness. Examples for regional error recovery, also called phrase-level recovery,

[5] According to the Hamming distance for strings.

were originally presented by Wirth [66] and Leinius [34].

2.3.2.4 Global Error Handling

Global error handling approaches try to repair the full input using a global context. They try to transform the erroneous input into a string which is a member of the language taking all possible corrections into account. Therefore they are often considered to be impractical in terms of performance [7]. Global error handling can be seen as regional error handling having the bounded region expanded to the full input using the same metrics to detect the most plausible repair. Examples are presented by Peterson [47], Pai and Kieburtz [44] or Burke–Fisher [6].

3 PEGs and Error Handling

We propose an error handling scheme suitable for PEG parsers, usually meaning it has to handle errors in a backtracking, recursive, top-down parser combinator. Our approach should be automated and language-agnostic reducing the involvement of a programmer *i.e.* a language specialist to a bare minimum. So far, the only way to add extended error handling capabilities to PEGs we know of is the usage of manually implemented *ad-hoc* solutions such as error productions. Speaking of error productions, we mean the addition of rules able to skip erroneous fragments of the input as shown in Listing 1.

```
start           ← _ expressionOr (EOI / errorProduction)
errorProduction ← (!EOI .)* EOI
```

Listing 1: An extension to the boolean logic grammar (see Appendix A.1) which skips symbols up to the end of input if it is not able to recognize the full input. The implementation of such rules requires knowledge of the language and manual modification of the parser.

Automated error recovery is able to extract the necessary data directly from the underlying grammar or in our case the parser itself. The fact that PEG parsers rely on the prioritized choice mechanics and can be implemented without using a separate lexical analyzer elegantly solves numerous problems arising in terms of ambiguity and language composition. While these are desirable properties, they cause corresponding parsers to perform badly in the presence of errors. PEG parsers by default exhibit almost none of the properties required to adapt known error handling schemes.

3.1 Backtracking and Ambiguity

PEGs are a recognition based formalism describing recursive top-down parsing with backtracking. In contradiction to table-driven parsers which are considered to have rather precise error locating capabilities, backtracking parsing algorithms perform badly in the presence of errors [10]. They inherently obfuscate the position of the syntax error in the input stream by reporting the latest occurring error. Ford [16] states that conventional LL and LR parsers perform a deterministic left-to-right scan and stop and report an error whenever there is no way to proceed. Backtracking parsers do not have this property often referred to as the aforementioned *correct-prefix property* (whereas the ability to report errors on their first encounter is called the *immediate-error-detection property* [12]). Even though there seems to be no formal description of this property, it can intuitively be described as the process of failing at the position the input ceases to be part of a correct prefix of the language. The prioritized choice requires a parser to check all available choices (or branches) until one of them succeeds or it returns the latest encountered failure. As soon as a parsing expression e_1 is unable to recognize the current portion of the input string (possibly due to a syntax error) it returns a failure f. If f occurs in a prioritized choice $e_c \leftarrow e_1 \ / \ e_2$, the parser backtracks to the position it tried to recognize expression e_1 and proceeds with e_2. This behavior inherently obfuscates the source of a syntax error, not only in terms of the error location but also in terms of the rule context (and thus the generated error message). This results in the reporting of errors located far from the position of the actual syntax error. Consider an example given in Listing 2 where the parser would locate the source of the error at the beginning of the input instead of the end.

```
[ "an" , "array" ␣
```

Listing 2: An incomplete array in JSON missing the closing delimiter at the end (denoted by ␣). Due to the backtracking a PEG parser would report the latest failure of the rule $value \leftarrow ... \ / \ array \ / \ number$, namely ‘Expected digit at position 0’ occurring in the `number` production.

Also, the semantics of a single failure state f mostly indicating the necessity to backtrack leads to another problem: PEGs do not have an explicit error state. As a matter of fact f might even cause expressions to succeed[1] and therefore its meaning fundamentally depends on the actual context. The state f only implicitly indicates the presence of a syntax error if it is emitted by the starting rule e_s signaling that the parser is not able to recognize the input. Therefore the *non-emptyness* of grammars is an important criterion since we would never encounter a failure otherwise. As a consequence, the detection of a syntax error happens post-mortem not preserving any information on the state of the parser at the point of failure. Even worse, backtracking parsers might have multiple different states at the point of failure which have to be taken into account. The prioritized choice requires the programmer to order rules by decreasing amount of specificity, probably being ambiguous *e.g.* having the same prefix.[2] Failures within ambiguous prefixes or at the beginning of a choice appear at the same position within different rules.

[1] not-predicate
[2] Ignoring the possiblity of left-factoring ambiguous rules.

3.2 Scannerless Parser Combinators

The main problem an error handling scheme faces in the context of a parser implemented as a scannerless parser combinator is the absence of structural information. The inclusion of the lexical specification into the formalism causes PEGs to degenerate to character-level grammars in general. Detecting and reporting errors in terms of expected characters results in rather cryptic error messages. Also, there is no notion of hierarchy. While a two-pass approach relying on a scanner separates lexical from syntactical elements of the source, scannerless parsers lack any form of literal distinction.[3] As a consequence, repairs in the form of insertion, deletion or substitution of characters might get complicated and ineffective [8] and there is no way of validating a speculatively applied repair. Tokens as generated by a scanner do not only carry information about the type of the input, they also mark word boundaries if recovered correctly, which means consecutive tokens are accessible upfront. Having the word boundaries at hand furthermore enables the application of spelling correction to a restricted area. In a scannerless parser, the recognition of tokens (if present) depends on their syntactical order which causes the recovery process and lexical analysis [64] — *e.g.* important for syntax highlighting — to be more complex. Also, relations like first set and follow set are not applicable anymore or at least of little use to known schemes. This lack of hierarchy is also aggravated by the fact that parser combinators such as PetitParser encode all parsing expressions as an instance of a parser (or a function). Without any concrete type information, the object graph does not provide sufficient structural information to an error handling scheme. In general, it is undecidable which expressions deliver the necessary informations to overcome an error.

3.3 Prefix Matching

The ability to detect errors and having the correct-prefix property does not necessarily ensure a correct error location (but still can lead to a correct error recovery) [24, 57]. The error-location identified by a programmer, taking semantic information or a general context such as the layout into account might differ from the error location the parser detects. This is usually the case if erroneous code forms a valid prefix or even a full sentence of the language as depicted in Listing 3. While this seems to be a generally unsolvable problem in parse error handling, the ability of a parsing expression to succeed without consuming the full input makes it even worse. PEGs inherently tend to leave remaining input unprocessed. One might overcome this weakness by adding an end-of-input marker (in PEGs specified as not-any `EOI ← !.`) at the cost of losing the composability of languages.

```
"array" ]
```

Listing 3: An array with a missing opening delimiter '['. The string will correctly be recognized as specified in $value \leftarrow .../\ string\ /\ array\ /\ number$, leaving the remainder unprocessed without failing if the grammar does not include the end-of-input marker.

[3] This is why we did not yet give any definition of the term *symbol*.

4 The Error Handling Scheme

The idea of our error handling scheme originates from continuations as introduced by Röhrich [52]. Continuations are based on the partition of an input string w into a correct prefix u and an incorrect suffix v and the subsequent repair of v. We divide our approach into four stages as shown in Figure 4.1.

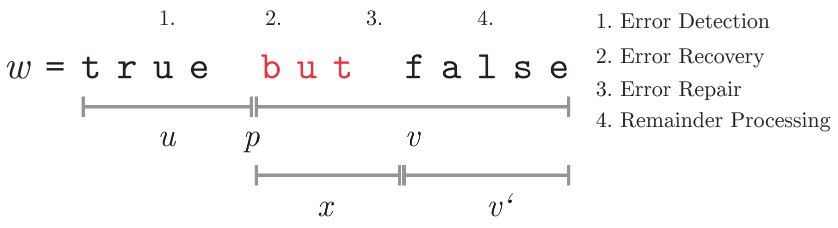

Figure 4.1: The erroneous input w failing at position p (due to the erroneous `but` operator), separating the input into a correct prefix u and an incorrect suffix v. x denotes the part of the suffix v we are going to repair before we resume the parsing on v'.

1. Error Detection is the process of identifying failures as syntax errors as described in Section 4.1. If the error handling parser detects an error it restarts in a recording mode. The recording run builds up a tree-like data structure consisting of parsing records representing the state of every parser invocation during the recognition. Using the resulting tree structure we determine so called error candidates, a collection of leaf nodes heuristically identified as possible syntax errors. Error candidates provide us access to the involved parsing expressions and allow us to determine the valid prefix u, in our example $u = true␣$.

2. Error Recovery given in Section 4.2, describes how to overcome the erroneous part x of the input and find a synchronization point marked by the beginning of v'. We compute a recovery set used to resynchronize the parser by skipping input until one of its members (*i.e.* the boolean false) is detected. To skip the erroneous portion of the input we use a parser which can be formalized using PEGs. Based on the matched members of the recovery set we determine the state the parser might be able to resume from and the type of the detected syntax error.

3. Error Repair elaborated in Section 4.3 detects and classifies possible types of the encountered error and the corresponding ways to repair them using transformations. In the example we could either substitute $x = but$ with the operator and or the operator or.

4. Remainder Processing describes how to proceed the parsing after repairing an error. For all chosen ways to repair the erroneous part x found during the error repair, we compute a *remainder grammar*. The remainder grammar formalizes the language of the remaining suffix v' given the state of the parser at the point of failure as shown in Section 4.4.

Subsequently, the recovery scheme is recursively invoked on the remaining suffix v' proceeding until no valid way to resume was found or the input is fully consumed. A successful error recovery attempt results in the collection of transformations used to repair the errors. Our error handling scheme is backtracking as well as the parser and evaluates all locally available ways of recovering or repairing. Instead of relying on cost based decisions to identify the most plausible way of handling an error, we stick to the semantics of the prioritized choice and succeed with the first working attempt to overcome an error. While the criteria an error handling scheme should fulfill are rather uniformly defined by the literature (*e.g.* Spenke *et al.* [57]) we focus on feasibility and quality ignoring considerations of performance (except the parsing of valid input which should not be influenced in terms of performance by the error handling).

All positions p including the error position used in our approach can be seen as the position directly before the character they describe. This decision was taken having the implementation as a stream in mind. To be consistent we preserve this definition for all our elaborations. The separation of an input string w into a (correct) prefix u and an (erroneous) suffix v is called a partition of w at a position p which in general is the error position. Since we describe all of the concepts in a rather informal way we depict all the statements using a contrived boolean-logic-grammar which can be found in Appendix A.1 or *Javascript Object Notation* (JSON) specified in Appendix A.2.

4.1 Error Detection

As aforementioned, PEGs are neither able to detect syntax errors explicitly nor do they specify an explicit error state. To overcome this limitation, we use a *farthest failure heuristic* to classify a failure as an error. Informally, the heuristic prioritizes the parses creating the longest correct prefix u of an input $w = uv$ instead of returning the latest occurring failure (analogous to preserving the longest parse in compilers [54]). All parsing expressions failing at the position p, farthest to the right, are encoded into error candidates. This approach intuitively suits the mechanics of PEGs, which — if correctly specified — consume as much input as possible up to a failure before they backtrack. A possible formalization of the *farthest failure position* for PEGs can be found in the work of Maidl *et al.* [38] encoding errors as pairs consisting of the error position and a list of parsing expressions. Due to the limitations of PEGs in terms of error detection, elaborated in Chapter 3, the usage of a farthest failure heuristic imposes difficulties on the proposed solution:

Ambiguity: The farthest failure heuristic does not uniquely identify a single failing expression as the source of a syntax error. On the contrary, default PEG semantics are likely to encounter multiple failures at the same position especially if the recognition fails at the beginning of a choice or in case of ambiguity. Therefore we propose that error candidates are collected preserving the order of their occurrence given by the depth-first invocation order of a recursive top-down parser.

Time of detection: As mentioned, the presence of a syntax error cannot be detected until the occurrence of a failure at the starting rule e_s. The farthest failure heuristic establishes a correct prefix property, but no immediate error detection property. To identify the position of a failure as farthest failure position is not feasible unless all possible branches are evaluated. Unless we don't use any kind of backtracking restrictions (*e.g.* cuts [41] as used in Prolog), the detection of syntax errors is done *post-mortem* and requires us to restart the parsing process at least the first time the error handling routines are triggered.

Heuristic: Even though the farthest failure heuristic fits the semantics of PEGs, the recognized, correct prefix does not necessarily correspond to the intentions of the programmer (see Section 3.1). Even worse, the programmer might accidentally create a correct prefix due to a misconception or a spelling error. This problem is ignored for now since it would require us involve far more complex heuristics during the error handling process.

Literals: According to their definition, PEGs do not always degenerate to character level grammars. The string literals act as a grouping mechanism causing the farthest failure heuristic to be too coarse grained to detect character errors within a word which is necessary to implement lexical repairs. To overcome this problem and to be consistent throughout the thesis we assume that all string literals are representing single characters.

Modifying the implementation of a PEG parser towards preservation of the farthest failure position is not a complex task. Nevertheless, simply having the knowledge of a more accurate position and generating more descriptive error messages is of little use for our purposes. Therefore we gather the necessary data — such as the involved parsing expressions — to recover from and repair syntax errors in an additional step elaborated in Subsection 4.1.1. Further elaborations on *longest match heuristics* can *e.g.* be found in the work of Yang *et al.* [67, 68].

4.1.1 Parsing Records

Backtracking parser combinators do not preserve any notion of the parser's state. We could modify the semantics of PEGs which would affect the performance of parsing valid input (*e.g.* by preserving a parse or prediction stack), the only basic performance-related criterion we care about. Instead, to recreate the state the parser was in at the point of failure, the engine restarts the parsing in a recording mode after the original recognition has failed. During the recording phase, we collect data containing the context of every expression e (*i.e.* the context of every invocation of a parser) of the grammar G and compose them in a tree structure consisting of *parsing records* denoted as N_e. A parsing record represents a tree node having relations to its parent as well as its children as depicted in Figure 4.2 and could be formalized as a 7-tuple

$$N_e = (e, p_{start}, p_{end}, result, N_{parent}, index, children)$$

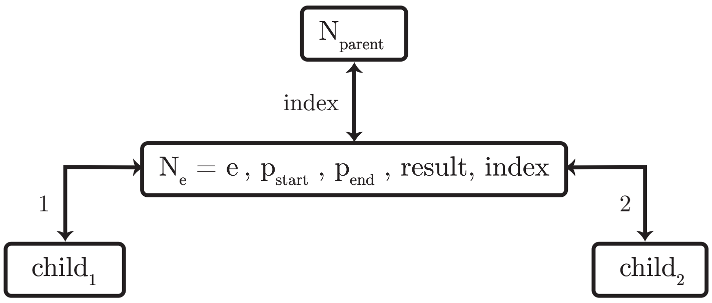

Figure 4.2: Parsing records are tree nodes collected during the recording run. All nodes store references to their parent node, their child nodes, the input partitions during the invocation as well as the involved parser. The value of index denotes the position of a node within the children of its parental node as indicated by the labels at the connecting edges.

Besides the relations to N_{parent} and $children$, maintaining the tree structure, the record contains the $result$ created by the parsing with default PEG semantics. The partition of the input at the time of invocation — and hence the progress the parser made — is encoded using positions namely p_{start} and p_{end} whereas p_{end} is the farthest position occurring in its child nodes. Finally, $index$ encodes the position of the parser in the context of its original parent (*e.g.* e_1 / e_2). The $index$ is important because the relation to a parent of a parsing expression is not unique since rules can generally be reused, which makes bottom-up computations impossible in general. Basically we materialize the full parse tree up to the error which allows us to define additional relations, perform bottom-up computations and determine the correct input partitions to resume the parsing. The root node of the tree structure is denoted as N_{e_s} recorded at the invocation of the starting rule e_s. An according example is presented in Subsection 4.1.2.

4.1.2 Error Candidates

Given the tree structure generated by the recording run and the farthest failure position $p_{farthest}$ we can identify error candidates. Error candidates encode all terminal expressions which were supposed to succeed at the error position. They are stored as a collection of parsing records fulfilling the condition

$$p_{end} = p_{farthest} \land (e \in V_T \lor e = !e_1) \land result = f$$

ordered by their invocation-order *i.e.* their occurrence in a depth-first tree traversal of the corresponding nodes. Informally, we collect all failing terminal parsers at the point the parser made the most progress. In a concrete implementation $p_{farthest}$ does not have to be known upfront and can be re-evaluated during the traversal of the parsing records. The inclusion of the not-predicate $!e$ might be surprising but has a simple reason: We neither want to fix code which is intended to fail nor are we able to distinguish syntax errors from an intended disambiguation.

Let's take a step back and have a look at an example given in Figure 4.3. The figure illustrates how the tree structure consisting of parsing records — given an erroneous input — looks and which nodes are identified as error candidates. Relying on the resulting data structure, the stage of error detection is able to find the parsing expressions which failed at the error position and can access the corresponding context.

4.2 Error Recovery

After having the ability to identify possible errors in the form of error candidates and persist the context at the point of failure we need to be able to identify the erroneous prefix x of the remainder $v = xv'$ as previously shown in Figure 4.1. This prefix defines the region where the error recovery as well as the error repair take place. To detect this area, local error handling schemes make use of a set of expressions which are acceptable at or to the right of the point-of-failure called acceptable-sets. Acceptable-sets are used by procedures to skip the erroneous region up to a member of the set. The members of an acceptable-set are also known as markers or fiducial tokens [44]. Our approach relies on a basic version of follow set recovery [58], which means that the acceptable-set principally consists of the members of the follow set. Since our definitions have got some specific properties we refer to our version of acceptable-sets as recovery sets.

All sets in our definitions need to be ordered in a way we can preserve the semantics of taking the first successful error recovery attempt analogous to the prioritized choice. For the sake of simplicity we assume that all set-based definitions in the following preserve the order given by the first addition of an element to the set (which is also preserved under union). To skip the erroneous region we use a *skipping parser* which is defined in terms of PEGs.

4.2.1 First Set, Dynamic Follow Set and Recovery Set

To compute the recovery set for any member of the error candidates we need some basic notion of the parsing expressions which are allowed to appear to the right of the failing parser. The concepts of first set and follow set are well known in the context of CFGs. An according definition for a function $first(e)$ generating the first set in terms of PEGs is given in Table 4.1. Alternative definitions for PEGs, similar to ours, are given by Redziejowski [48] or Mascarenhas *et al.* [11].

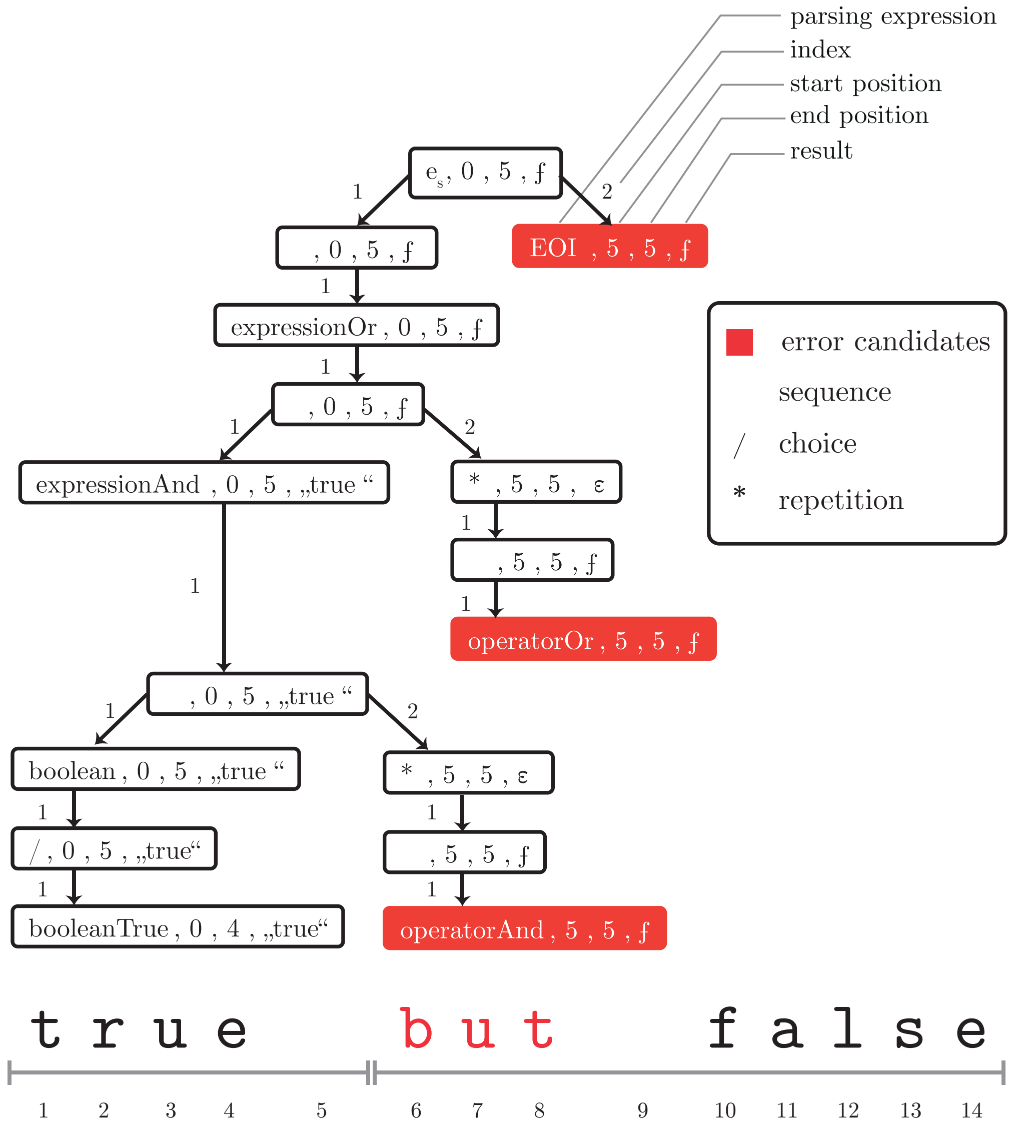

Figure 4.3: Visualization of the tree structure collected using the parsing records during the recording run. The leaf records failing at the furthest position $p_{farthest} = 5$ are marked as error candidates. The error candidates are collected as $(N_{operatorAnd}\ N_{operatorOr}\ N_{EOI})$ indicating that the parser expected ‘and’, ‘or’, or the end-of-input. The input is partitioned at position 5.

Expression	first set	Condition
e	$\{e\}$	$e \in V_T$
ϵ	$\{\$\}$	
e_1/e_2	$first(e_1) \cup first(e_2)$	
$e_1\ e_2$	$first(e_1)$	$\$ \notin first(e_1)$
	$first(e_1) \cup first(e_2)$	$\$ \in first(e_1) \wedge \$ \in first(e_2)$
	$(first(e_1) - \{\$\}) \cup first(e_2)$	$\$ \in first(e_2) \wedge \$ \notin first(e_2)$
$!e$	$\{!x \mid x \in first(e)\}$	-
$e*$	$first(e) \cup \$$	-

Table 4.1: The first set for PEGs. We use the $ as a sentinel, denoting the empty word which is preserved under union.

Based on the first set we develop a specific version of follow set. Informally the classical follow set consists of all terminal symbols which can appear to the right of a parsing expression independently from its current invocation context. To implement an accurate local error handling scheme we need to rely on a definition of follow set which depends on the actual parsing history or in other words, which corresponds to the current sentential form of the grammar. Using a standard definition of follow set is not sufficient since parsing expressions which are reused within the grammar might be part of different productions making the corresponding follow set too general. A definition of a follow set depending on the parsing history is sometimes referred to as *dynamic follow set* [59]. Computing these upfront would require us to evaluate all possible sentential forms of the grammar which in general are not finite. Therefore we use the parsing records to compose these sets bottom-up which is made possible by the tree structure given by the parsing records. We use a function $dynfollow$ taking a parsing record as its argument and a helper function $follow\_at$ which can take a parsing record as well as a parsing expression as its first argument and the index i provided by the parsing-record as its second argument shown in Table 4.2. Be aware that the parsing records themselves do not contain the members of the follow set since we did not yet encounter them during the parsing.

Expression	Dynamic-Follow-Set	Condition
$dynfollow(N_e)$	$follow\_at(N_{parent}, index)$	
$follow\_at(N_{e_s}, i)$	$\{\$\}$	
$follow\_at(N_e, i)$	$follow\_at(e, i)$	$\$ \notin follow\_at(e, i)$
	$follow\_at(e, i) \cup follow\_at(N_{parent}, index)$	$\$ \in follow\_at(e, i)$
$follow\_at(e_1\ e_2, 1)$	$first(e_2)$	
$follow\_at(e*, i)$	$first(e) \cup \{\$\}$	
$follow\_at(e, i)$	$\{\$\}$	otherwise

Table 4.2: The definition of dynamic follow set for parsing records N_e and parsing expressions e. The algorithm recursively traverses up the tree denoted as $follow\_at(N_{parent}, index)$ as long as the sentinel $ is a member of the set or the root node N_{e_s} is reached.

With regard to the parsing records given in Figure 4.3 we can now compute dynamic follow sets for the collected error candidates *e.g.* $N_{operatorAnd}$:

$$\begin{aligned}
dynfollow(N_{operatorAnd}) &= follow\_at(N_{parent}, 1) \\
&= follow\_at(N_{operatorAnd\ boolean}, 1) \\
&= follow\_at(operatorAnd\ boolean, 1) \\
&= first(boolean) \\
&= first(booleanTrue\ /\ booleanFalse) \\
&= first(booleanTrue) \cup first(booleanFalse) \\
&= \{booleanTrue\} \cup \{booleanFalse\} \\
&= \{booleanTrue, booleanFalse\}
\end{aligned}$$

To illustrate the benefits of a dynamic follow set the basic example is not sufficient. Due to the simplicity of the boolean-logic-grammar, there exists no difference between a follow set and the dynamic follow set. Booleans are always followed by an operator or the end-of-input and the operators themselves are always followed by a boolean. Hence, consider boolean values within the JSON grammar (given in Appendix A.2). The default follow set of a boolean without regard to the context is { `!. , ',' , '}' , ']'` } but a boolean can appear as a

1. standalone value **`true`** where dynamic follow set = { `!.` }
2. member of an array `[ `**`true`**` ]` where dynamic follow set = { `',' , ']'` }
3. value within an object `{ "boolean" : `**`true`**` }` where dynamic follow set = { `',' , '}'` }

As one can see the dynamic follow set is more precise, corresponds to the current sentential form of the grammar and will therefore reduce the possibility of accepting wrong symbols in case of errors.

To finally define the recovery set itself we need to take two special situations into account. First the presence of additional symbols preceding the expected expression. In this case the expected parsing expression e identified as error candidate might appear to the right of the error position after an arbitrary number of erroneous symbols. We therefore add an aliased version of e

$$E' \leftarrow e$$

to the recovery set. The alias is necessary because the parsing expression can be part of its own dynamic follow set. We need to be able to distinguish between the original parsing expression which is a member of the error candidates and its occurrence in the dynamic follow set further discussed in Section 4.3.

Second, the set might contain the sentinel $\$$ meaning that there was no non-nullable parsing expression to the right of the error candidate. Hence, $\$$ marks the end of the grammar and we therefore remove it from the dynamic follow set and add the end-of-input expression at the end. The resulting definition of the recovery set is given in rule 4.1 and 4.2.

$$recovery\text{-}set(N_e) = dynfollow(N_e) \cup \{E'\} \qquad | \ \$ \notin dynfollow(N_e) \tag{4.1}$$

$$recovery\text{-}set(N_e) = (dynfollow(N_e) - \{\$\}) \cup \{E'\} \cup \{!.\} \qquad | \ \$ \in dynfollow(N_e) \tag{4.2}$$

Every recovery set of an error candidate can be seen as a *local* recovery set. We could elaborate the nature of an error using every single recovery set and backtrack to the error position. Instead we create a (*global*) recovery set [23] consisting of the union of all local recovery sets to avoid scanning the whole input for every error candidate. Given the error candidates from the previous example, the computation of the global recovery set is straightforward:

$$error\text{-}candidates = (N_{operatorAnd}, N_{operatorOr}, N_{EOI})$$

$$\begin{aligned}
recovery\text{-}set(N_{operatorAnd}) &= dynfollow(N_{operatorAnd}) \cup \{operatorAnd'\} \\
&= \{booleanTrue, booleanFalse\} \cup \{operatorAnd'\} \\
recovery\text{-}set(N_{operatorOr}) &= dynfollow(N_{operatorOr}) \cup \{operatorOr'\} \\
&= \{booleanTrue, booleanFalse\} \cup \{operatorOr'\} \\
recovery\text{-}set(N_{EOI}) &= dynfollow(N_{EOI}) \cup \{EOI'\} \\
&= (\{\$\} - \{\$\}) \cup \{EOI'\} \cup \{!.\} \\
&= \{EOI', !.\}
\end{aligned}$$

$$\begin{aligned}
recovery\text{-}set &= recovery\text{-}set(N_{operatorAnd}) \\
&\cup recovery\text{-}set(N_{operatorOr}) \\
&\cup recovery\text{-}set(N_{EOI}) \\
&= \{booleanTrue, booleanFalse, operatorAnd', operatorOr', EOI', !.\}
\end{aligned}$$

One might note that the end-of-input is now redundant within the set due to the aliasing. The end-of-input expression is a special case which has to be handled by the concrete implementation and depends on the mapping from a parsing expression onto a string. In practice, the end-of-input is always mapped to the empty string ϵ.

4.2.2 Skipping Parser

Having the possibility to collect error candidates and compute an appropriate recovery set enables us to skip the erroneous region of the the input. We therefore create a parser formalized by a parsing expression in the form of a prioritized choice whose options are the members of the recovery set. The resulting skipping parser can be formalized as a parsing expression e_{skip} shown in Listing 4 using negation. By invoking this parser on the erroneous suffix v we get the partition of the suffix $v = xv'$, indicated by not emitting any failure *i.e.* $(e_{skip}, v) \Rightarrow (*, x)$. The erroneous portion x denotes the area we will try to repair and therefore implicitly defines the remaining input v'. If $x = \epsilon$ the skipping parser has not done any progress and directly encountered a member of the recovery set which is important for the error repair.

```
recoverySetExpression ← recovery-set₁ / ... / recovery-setₙ
e_skip                ← (!recoverySetExpression .)* !!recoverySetExpression
```

Listing 4: The definition of the skipping parser used to overcome the erroneous region until a member of the recovery set was encountered. We ensure that none of the members of the recovery set succeeds and consume the encountered symbol using the any operator. The final expression `!!recoverySetExpression` ensures that we did not skip the full input up to the end without encountering one of the expected symbols. The usage of the not-predicate prevents the skipping parser from doing any more progress and therefore preserves the position within the input.

As one can see, skipping symbols in a scannerless parser is different from popping symbols off a parsing stack like it is done in classical parsers using a scanner. As an example we reconsider the previous situation in Figure 4.3. If we invoke the corresponding skipping parser on the remaining suffix $v =$`but false` we will end up in the situation depicted in Figure 4.4 delivering the input partition $v = xv'$.

```
recoverySetExpression ←   booleanTrue
                        / booleanFalse
                        / operatorAnd'
                        / operatorOr'
                        / EOI'
                        / !.
```

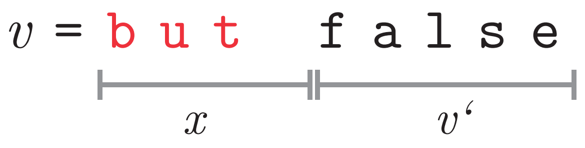

Figure 4.4: If the skipping parser succeeds recognizing a prefix x of v we get a partition of $v = xv'$, which results from the invocation $(e_{skip}, v) \Rightarrow (*, x)$. If $x = f$ we were not able to find a point to resynchronize the parser and our error handling scheme fails.

4.3 Error Repair

The main work of the error repair lies in the classification of the type of the occurred error and therefore identification of an according transformation to repair it. All of the required information, necessary to identify the type an error has, is delivered by the error recovery procedure, respectively given by the members of the recovery set succeeding at the end of the skipping phase.

4.3.1 Transformations

Syntactical as well as lexical errors and the corresponding way of repairing them can be expressed using three types of transformations namely substitution, insertion and removal. An overview is given in Table 4.3. There are multiple ways of expressing these transformations in terms of each other. We prefer to define all of them using a basic substitution applied to the erroneous remainder $v = xv'$. A substitution can be defined as

$$substitution_{x,s} = sv'$$

where x is the prefix of the partition of $v = xv'$ and s is the string that replaces x. As a matter of fact s is a string which is acceptable by the parsing expression encoded in the error candidate. A removal can be described as a substitution with an empty string ϵ whereas an insertion is defined as a substitution using the concatenation of the insertion value with the prefix x:

$$removal_{x,s} = substitution_{x,\epsilon} = \epsilon v'$$

$$insertion_{x,s} = substitution_{x,sx} = sxv'$$

Error	Description	Caused by	Repaired with
addition	caused by an additional erroneous symbol in the input	insertion	removal
omission	caused by a missing element in the input	removal	insertion
substitution	caused by a mistakenly placed element in the input	substitution	substitution

Table 4.3: Classifications of errors and their corresponding transformations.

4.3.2 Error Classification

To gather information on how to fix the erroneous part of the input we need to be able to draw conclusions on the type of error that occurred. To do so, we take the information about which members of the recovery sets succeeded at the end of the skipping phase. We define a rule set to determine plausible transformations corresponding to the type of the encountered syntax error. We consider these transformations to be plausible because the current context given by the state of the parser and the partition of the input do not provide enough information to unambiguously decide how to repair an error. In terms of a PEG, every transformation may represent a different branch to proceed the parsing with, especially if the involved rules are ambiguous. Therefore, the error-repairing procedure needs to take all plausible transformations for all error candidates into account.

To classify the syntax error we need to detect which parsing expressions of the recovery set were able to recognize a prefix of the remaining input v'. Let's define another ordered set consisting of expressions which are members of the recovery set of an error candidate N_e and do not fail if applied to v' (all these expressions indicate the beginning of a valid suffix).

$$match_{N_e} = \{m \mid m \in recovery\text{-}set(N_e) \wedge (m, v') \Rightarrow (n, V_T^*)\}$$

Using this set of parsing expressions whose members are denoted as m (to avoid naming collision with the originally failed expression e encoded in the parsing record N_e), we can describe the classification of syntax errors for all $m \in match_{N_e}$. To be able to express the repairing attempts in terms of a transformation we further need a function

$$s(e) = s \text{ such that } (e, s) \Rightarrow (*, s)$$

which allows us to map a parsing expression onto a string. Since parsing expressions in general cannot be uniquely mapped onto a string, we leave the decision about what to insert to the concrete implementation.

Addition: If $m = E' \Rightarrow transformation = removal_x$. E' is the aliased version of e according to Subsection 4.2.1, defined to distinguish the original parser from its occurrences in its own recovery set. The originally failing expression was found later in the input. We assume that the portion x we skipped was inserted by mistake and hence needs to be removed as shown in Figure 4.5.

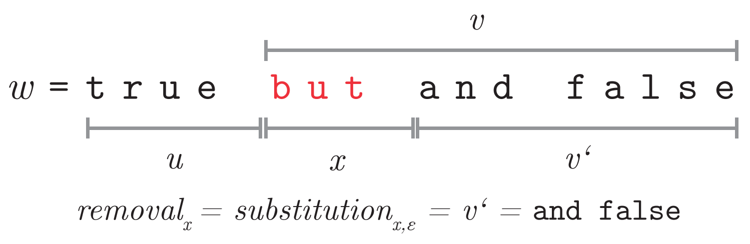

Figure 4.5: Transformation in presence of an addition. The error repair matches $E' \leftarrow operatorAnd$ representing the parsing expression originally expected at the error position (member of the error candidates) which therefore indicates the beginning of v'.

Substitution: If $m \in dynfollow(N_e) \wedge x \neq \epsilon \Rightarrow transformation = substitution_{x,s(e)}$. The expression m was a member of the dynamic follow set of e and appeared to the right of the error position as expected. Since $x \neq \epsilon$, the skipping parser encountered mistakenly placed, erroneous input before the expected expression m. To repair the error we substitute the erroneous input x with an input portion which adheres to the expression e expected at the point of failure shown in Figure 4.6.

Omission: If $m \in dynfollow(N_e) \wedge x = \epsilon \Rightarrow transformation = insertion_{x,s(e)}$. The expression which was matched was a member of the dynamic follow set and directly appeared at the error position ($x = \epsilon$) instead of the originally expected expression e. Therefore we insert $s(e)$

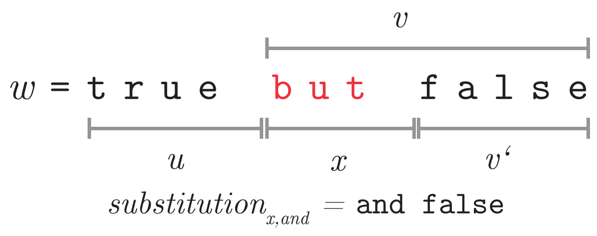

Figure 4.6: Transformation in presence of a substitution. The error repair matches $m \leftarrow booleanFalse$ at the beginning of v'. The expression m is a member of the dynamic follow set of e and we can therefore replace x with $s(e) =$'and'.

between the error position and the string representation of the matched member of its dynamic follow set as depicted by Figure 4.7.

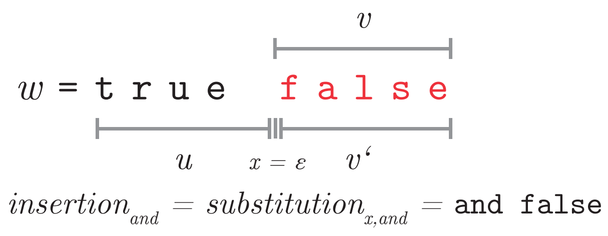

Figure 4.7: Transformation in presence of an omission. The error repair matches $m \leftarrow booleanFalse$ at the beginning of v'. Since m is a member of the dynamic follow set of e and $x = \epsilon$ we can insert 'and' before v'.

Let's reconsider the previous example and the partition $v = xv'$ recognized by the skipping parser as depicted by Figure 4.4. Using the recovery sets of all error candidates, the suffix v consisting of $x = but\textvisiblespace$ and $v' = true$, we compute the matches by collecting the succeeding expressions giving us the necessary information which transformations are plausible:

First, take the given recovery sets of the error candidates

$$\begin{aligned}
recovery\text{-}set(N_{operatorAnd}) &= \{booleanTrue, booleanFalse, operatorAnd'\} \\
recovery\text{-}set(N_{operatorOr}) &= \{booleanTrue, booleanFalse, operatorOr'\} \\
recovery\text{-}set(N_{EOI}) &= \{EOI', !.\}
\end{aligned}$$

and match all of its members against the remaining input $v' = false$ (they usually only recognize a prefix)

$$\begin{aligned}
(booleanTrue, v') &\Rightarrow (1, f)\\
(booleanFalse, v') &\Rightarrow (1, false)\\
(operatorAnd', v') &\Rightarrow (1, f)\\
(operatorOr', v') &\Rightarrow (1, f)\\
(EOI', v') &\Rightarrow (1, f)\\
(!., v') &\Rightarrow (1, f)
\end{aligned}$$

and the resulting sets $match_{N_e}$ containing all succeeding members of the recovery set

$$\begin{aligned}
match_{N_{operatorAnd}} &= \{booleanFalse\}\\
match_{N_{operatorOr}} &= \{booleanFalse\}\\
match_{N_{EOI}} &= \emptyset
\end{aligned}$$

Since $booleanFalse$ is a member of $dynfollow(N_{operatorAnd})$ as well as of $dynfollow(N_{operatorOr})$ and $x \neq \epsilon$ we end up having two plausible *transformations*

$$(substitution_{x,s(operatorAnd)}\ ,\ substitution_{x,s(operatorOr)})$$

intuitively indicating that we could replace $x = but$ with operator ‘and’ or operator ‘or’.

4.4 Remainder Processing

In the previous sections we elaborated how to identify a syntax error, detect the erroneous region x, classify the encountered error and encode it accordingly as a transformation. Identifying a single error and delivering plausible ways to repair it might be a basic proposal on how to improve the error handling of a PEG parser but provides an incomplete behavior in terms of error recovery. The described mechanics are not able to detect multiple errors within the input and leave the remaining portion v' unprocessed. Furthermore, we are missing a way to verify a particular repair.

A naive approach to confirm the validity of a repair could be to apply plausible transformations to the input and restart the parsing at the beginning of every resulting, modified input. Besides the fact that we parse the valid prefix u for every recovery attempt, it is hard to determine if the previously applied transformation has introduced new errors and causes the algorithm to examine a variety of unnecessary repairs. Instead, we created our own way to formalize and process the remaining input — comparable to continuations presented by Röhrich [52] — which is suitable for PEGs. Due to the fact that we detect errors post-mortem, two main problems arise at this stage of the error handling scheme: First, we need a way to encode the parsing history to recreate the parser's state to proceed from. Second, we need to express the grammar of the remaining input v' in terms of PEGs. While the previous stages work analogously to known approaches, the following concepts are specific to our particular solution.

4.4.1 Parser Configurations

To express a parser's call stack we use *configurations*. Configurations basically are paths in the object graph, leading from the starting rule to an arbitrary parser. This concept is not restricted to the concrete parser and can be applied to PEGs as well. To denote such paths we use ordered collections $(c_n)_{n\in\mathbb{N}}$ of length n, consisting of indices every parsing expression has in the context of its parent as depicted by Figure 4.8.

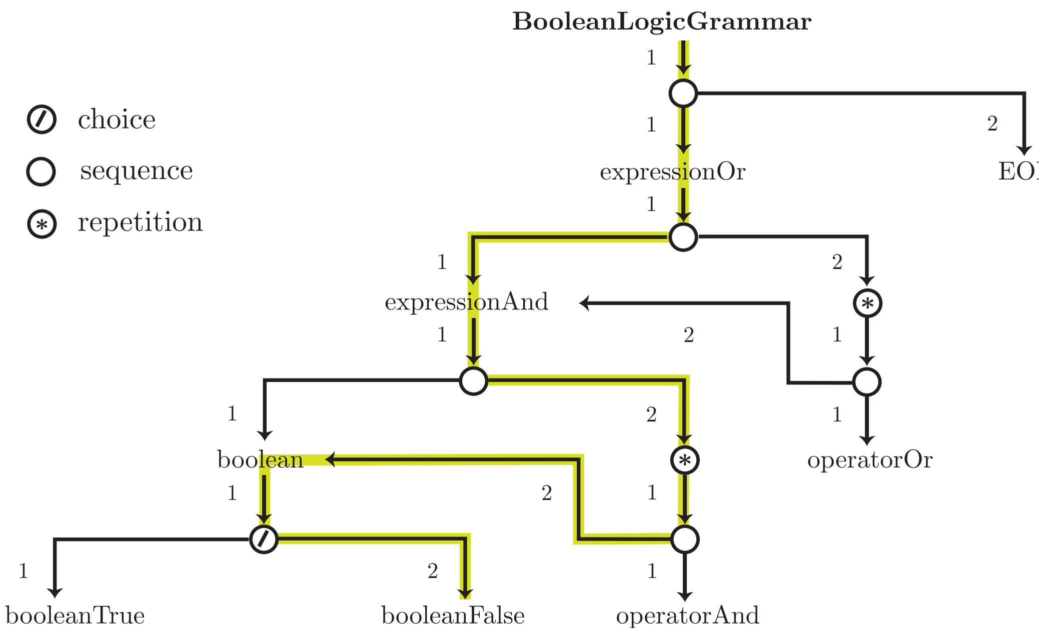

Figure 4.8: A configuration, denoting the occurrence of `booleanFalse` within the `operatorAnd` expression. All possible call stacks and therefore all states the parser can reach can be encoded using a path in the graph and the input partition it occurred at. The corresponding configuration c in this case is the sequence (`1 1 1 1 1 2 1 2 1 2`) pointing to the expression $booleanFalse$.

The $index$ value, encoded within the parsing records enables us to compute the configuration of a parser at a parsing expression *i.e.* the error candidates by traversing the tree from the bottom, up to the root node. During the computation of the recovery sets we also keep track of their members' configurations which represent the state of the parser at the beginning of v'. Note that configurations in the literature [58] are sufficient to express a parser's state. Our definition in addition requires the input partition (*i.e.* u and v) it occurred at to be able to do so and proceed the parsing from the corresponding state. This is due to the fact that a certain configuration might be reached having consumed a different input. The state of a parser can be described as having a configuration c at a certain input partition uv. In general a backtracking parser can have multiple configurations at a certain partition, each representing a different branch within the grammar.

4.4.2 Remainder Grammars

Using a configuration enables us to express a *remainder grammar*, an equivalent or at least similar concept to continuations. The resulting parsing expressing formalizes the language of the remaining input with respect to the current sentential form of the original grammar. Remainder grammars are partial grammars of the original grammar e_s, enabling us to avoid the necessity to parse the valid prefix u for every possible repair. A figurative way to explain the meaning of a remainder grammar as a subgrammar of e_s is depicted by Figure 4.9.

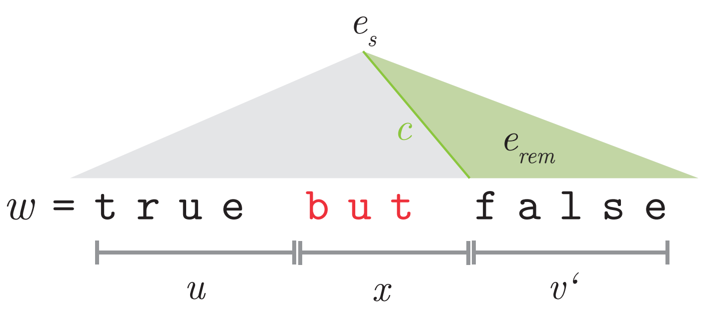

Figure 4.9: A graphical way to depict the meaning of a remainder grammar. The whole triangle represents the full grammar e_s. The configuration c represents a seam, denoting the frontier between the already processed parts of the grammar and the remaining parsing expression e_{rem}.

We define a function $rem(e, c)$, whereas e is a parsing expression, computing the PEG formalizing the remaining input at a certain configuration c as given in Table 4.4. Using the inductive definition of PEGs causes the configurations to contain only 1 and 2 with the exception of the repetition quantifiers which will increase the value with every invocation. The remainder grammar of a grammar G at a configuration c is defined as $rem(e_s, c)$.

Expression	Remainder	Condition
rem(e , c)	e	$c = () \lor e \in V_T$
rem(e, c)	rem($R(e)$, c')	$c = 1\ c' \land e \in V_N$
rem(ϵ , c)	ϵ	
rem($e_1\ e_2$, c)	rem(e_1, c') e_2	$c = 1\ c'$
	rem(e_2, c')	$c = 2\ c'$
rem(e_1 / e_2, c)	rem(e_1, c')	$c = 1\ c'$
	rem(e_2, c')	$c = 2\ c'$
rem(!e, c)	!rem(e, c')	$c = 1\ c'$
rem($e*$, c)	rem(e, c') $e*$	$c = \bullet\ c'$

Table 4.4: Remainder grammars for PEGs. The first rule in the table enables us to use configurations which do not reach down to a terminal parsing expression. Note that non-terminal parsing expressions $e \in V_N$ access their right hand side $R(e)$ using $index = 1$.

To illustrate the concrete computation of a remainder grammar, reconsider Figure 4.8 showing a configuration $c = (\ 1\ 1\ 1\ 1\ 1\ 2\ 1\ 2\ 1\ 2\ )$ pointing to $booleanFalse$. The computation of the corresponding remainder grammar $rem(e_s, c)$ and the resulting expression e_{rem} is shown in Table 4.5. In this case e_{rem} could *e.g.* recognize the current remainder $v' = false$.

remainder grammar	c
rem(e$_s$, c')	1 c'
rem(expressionOr EOI, c')	1 c'
rem(expressionOr, c') EOI	1 c'
rem(expressionAnd (operatorOr expressionAnd)*, c') EOI	1 c'
rem(expressionAnd, c') (operatorOr expressionAnd)* EOI	1 c'
rem(boolean (operatorAnd boolean)*, c') (operatorOr expressionAnd)* EOI	2 c'
rem((operatorAnd boolean)*, c') (operatorOr expressionAnd)* EOI	1 c'
rem(operatorAnd boolean, c') (operatorAnd boolean)* (operatorOr expressionAnd)* EOI	2 c'
rem(boolean, c') (operatorAnd boolean)* (operatorOr expressionAnd)* EOI	1 c'
rem(booleanTrue / booleanFalse, c') (operatorAnd boolean)* (operatorOr expressionAnd)* EOI	2 c'
rem(booleanFalse, c') (operatorAnd boolean)* (operatorOr expressionAnd)* EOI	()
booleanFalse (operatorAnd boolean)* (operatorOr expressionAnd)* EOI	

Table 4.5: The computation of the remainder grammar at configuration $c = (\ 1\ 1\ 1\ 1\ 1\ 2\ 1\ 2\ 1\ 2\ )$ pointing to `booleanFalse`.

4.4.3 Resume the Parsing

Having the possibility to encode the state of a parser and compute remainder grammars, we are now able to resume the parsing after we detected the collection of plausible transformations. We therefore keep track of the configurations of all members m of the $match$ sets described in Subsection 4.3.2. Remember, all expressions m were able to recognize the beginning of v' and hence indicate that the remainder grammar is given by their corresponding configurations. While we are able to identify if remainder grammars of a grammar G are identical (they *e.g.* have the same configuration), their equality in terms of the expressed language in general cannot be determined as it is impossible to determine if PEGs are ambiguous [14]. As a consequence we need to examine all possible remainder grammars by recursively invoking the error handling scheme on the remainder v' and backtrack until it is able to succeed or eventually fails. Note, that allowing our approach to fail is a fundamental difference to Röhrich's proposal which subsequently modifies the input until a valid parsing run is achieved. The behavior Röhrich describes requires the parser to have a unique state at the error position which is not the case in a backtracking parser. The error handling scheme therefore works similar to the prioritized choice and backtracks until the first transformation is verified by being able to process the remaining input (either by not finding any more errors or by being able to fix them). To succeed with the first working attempt without any further prioritization of a particular repair might advance the error handling scheme to generate spurious errors (possibly ignoring a more accurate error repair induced by another transformation). A more sophisticated way to prioritize a repair would be to rate and prioritize the corresponding transformations by using metrics [4, 60] to reduce the probability of creating spurious errors. To proceed, the error handling scheme is recursively invoked on the remaining input v' using the remainder grammar as new base grammar e_s.

Let's again have a look at our initial example and recap the usage of configurations and remainder grammars. We detected the error position, computed the error candidates, skipped the erroneous region x and identified plausible transformations based on the matching members m of the recovery set. We described the matching sets as

$$\begin{aligned} match_{N_{operatorAnd}} &= \{booleanFalse\} \\ match_{N_{operatorOr}} &= \{booleanFalse\} \\ match_{N_{EOI}} &= \emptyset \end{aligned}$$

As already mentioned, the error handling scheme in general has multiple configurations to proceed from, depicted in Figure 4.10. Even though both configurations point to the same parsing expression they do not necessarily express the same parsing history and therefore indicate a different remainder grammar. Since we proceed all options in-order, we start by computing the remainder grammar (see Table 4.5) at configuration $c_1 = (\ 1\ 1\ 1\ 1\ 1\ 2\ 1\ 2\ 1\ 2\ )$ given by $m = booleanFalse$ where $m \in match_{N_{operatorAnd}}$.

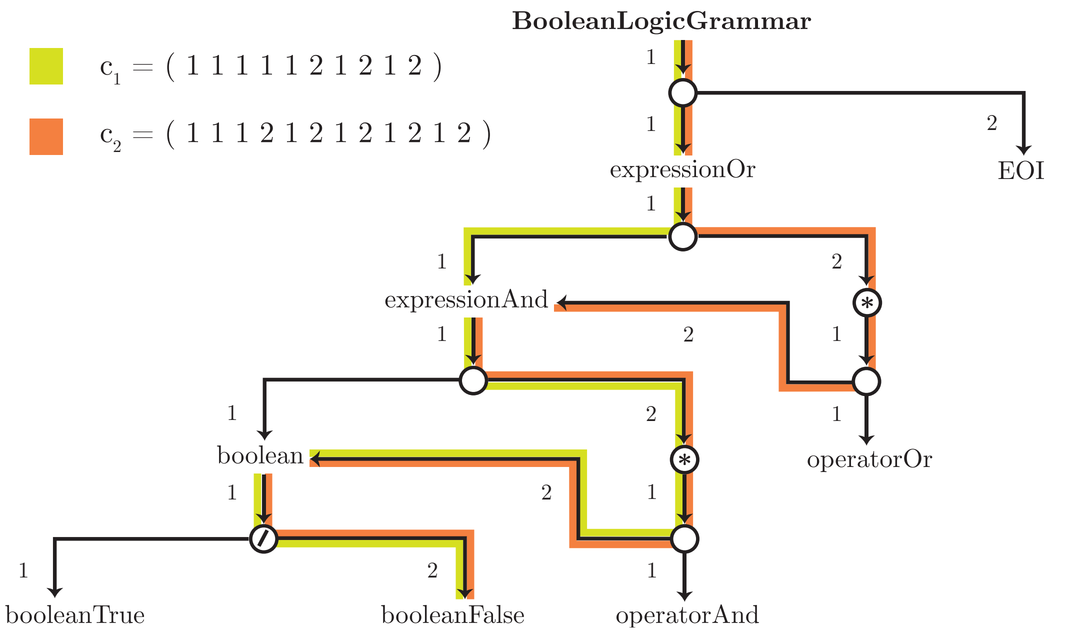

Figure 4.10: The boolean-logic-grammar and configurations pointing to $m = booleanFalse$, a member of the respective recovery sets of $N_{operatorOr}$ and $N_{operatorAnd}$. The depicted configurations indicate two remainder grammars to proceed the parsing with.

Invoking the error handling scheme using $e_s = e_{rem}$ on the remaining input $w = v' = false$ results in a successful parse and we accept the transformation ($substitution_{\text{but},s(operatorAnd)}$) as the correct way to repair the input. If the recursive invocation of the scheme would fail we would examine c_2 to verify $substitution_{\text{but},s(operatorOr)}$ which is also correct but is not validated due to the prioritization of the first working attempt.

4.5 Error Handling Strategies

In the previous sections we described how basic error handling mechanisms can be adapted to PEGs. In practice the concrete error handling strategies have some special characteristics causing subtle differences to the fundamental workings we elaborated. A concrete strategy handles errors on a specific level of granularity in terms of the modified symbols and the resulting input partitions. Concrete strategies therefore extract specific error candidates and recovery sets. We propose two basic strategies, namely a token error handling strategy (token strategy) and a lexical error handling strategy (lexical strategy) which can also be combined. While in the previous examples the erroneous area x was uniform for all repairing attempts, in practice the strategies can choose to define x (and therefore the remaining input v') for every transformation they detected. In the following we describe these strategies in terms of their distinctions to the general approach.

4.5.1 Symbols

Up to now we did not provide any clear notion what symbols exactly are. We considered them to be tokens to bridge the gap to classical error handling approaches, designed for parsers relying on a scanner. In contrast, scannerless PEG parsers in general recognize the input one character after the other, each representing a symbol as described in Section 2.1. We therefore consider characters to be lexical symbols (recognized by all $e \in V_T$). PEGs do not provide any more coarse grained structural information such as expressions representing full tokens. Without going into detail of a concrete implementation, we identify tokens as a specific set of expressions whose members represent a dedicated type of terminal symbol. In the following definitions, tokens are specified by expressions which are members of a set denoted as

$$V_{Token}$$

which allows us to deliver an alternative notion of terminal expressions. They allow us to define different versions of first set, dynamic follow set and therefore the recovery set, suitable for a concrete strategy. The usage of dedicated token parsers (`PPTokenParser`) is already common practice for parsers implemented using the PetitParser framework. V_{Token} in the context of PetitParser consists of all parser instances within the grammar qualifying themselves as a token parser.

4.5.2 Token Strategy

As the name indicates, the token strategy operates on terminal symbols recognized by parsing expressions $e \in V_{Token}$. The token strategy closely conforms to the error recovery and the error repair algorithms we described in a general fashion, except its notion of terminal parsing expressions. Therefore, to describe the strategy it is sufficient to refine a few criteria:

Error Candidates: Error candidates have to fulfill the slightly modified condition

$$p_{end} = p_{farthest} \wedge (e \in V_{Token} \vee e = !e_1) \wedge result = f$$

meaning that an error candidate is either a token or a not-predicate. This allows us to group composite expressions into tokens and exclude expressions such as whitespace from the algorithm.

Input Partition: While in general the initial input partition is done at the error position, the token strategy partitions the input at the beginning of the token encoded within the error candidates. Due to the fact that we use tokens the error position is not the same as the position the parsing expression was originally invoked. Preserving the farthest failure position to be the beginning of the erroneous area x would result in transformations which are applied within a token as shown in Figure 4.11.

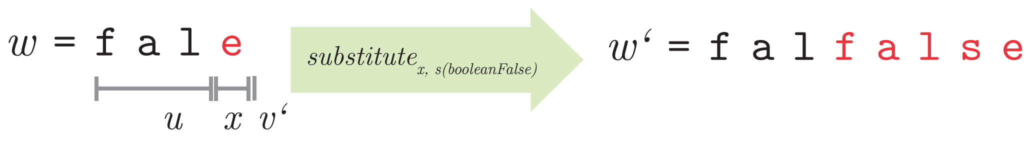

Figure 4.11: An erroneous token substitution caused by the input partition uxv'. Therefore we partition the input at $start$ of the corresponding error candidate, might causing x to be different for every transformation

First Set: As mentioned, the concrete token strategy has a different notion of terminal symbols. We therefore redefine two rules to compute the first set as shown in Table 4.6. Redefining these two definitions causes the first set as well as the dynamic follow set to consist of parsing expressions $e \in V_{Token}$. It also indicates why we did not use nullability as a criterion to determine if we need to extend the first set as well as the follow set: using the sentinel as a marker for a possibly unclosed set allows us to reuse the majority of the definition without the necessity to introduce different notions of nullability, especially with regard to future extensions.

4.5.3 Lexical Strategy

Apart from the described differences the token strategy tightly adheres to the general approach we described in the previous sections. Since the token strategy is sometimes too coarse grained and might skip relatively long parts of the input due to subtle spelling errors, it is preferable to correct a token instead of fully

Expression	first set	Condition
e	$\{\$\}$	$e \in V_T$
e	$\{e\}$	$e \in V_{Token}$

Table 4.6: Changes to the general first set definition given in Table 4.1 for error handling on a token level in PEGs. We use the $ as a sentinel, denoting the empty word which is preserved under union.

removing or replacing it. A lexical strategy can be seen as an attempt to achieve a basic spelling correction following the characteristics of string similarity as elaborated in Subsection 2.3.2.1. We restrict our approach to all rules of similarity which can be expressed using the three basic transformations removal, insertion and substitution of a single character. In other words, we try to transform one area of the input into another by applying a single character transformation which can be achieved by adapting the general strategy. It is worth mentioning that a spelling correction in a scannerless parser has to be done on-the-fly in order to process full tokens. Since the recognition of tokens — in contrast to a separate scanner — is bound to their syntactical order, lexical repair has to be applied during the parsing as aforementioned in Section 3.2.

Error Candidates and Recovery Sets: The definition of error candidates is exactly as elaborated in Subsection 4.1.2. The first set as well as the follow set — and therefore also the recovery set— consist of terminal parsing expressions which are able to recognize a single character.

Input Partition and Skipping Behavior: If adapted to our general approach, we could specify the skipping parser as listed in Listing 5.

```
e_skip      ← !!recoverySetExpression / (. !!recoverySetExpression)
```

Listing 5: Parsing expression describing the lexical skipping parser. It skips at the maximum one arbitrary character given by the dot in the second part of the choice and fails if none of the expected expressions was able to succeed at the error position (or error position $+1$).

Using a skipping expression e_{skip} as specified has a major weakness: If the erroneous character at the error position is a member of the recovery set (and therefore $x = \epsilon$) the strategy fails in rather simple cases due to its partial inability to detect substitutions as well as additions. To overcome this limitation, the lexical strategy by default assumes that the error appeared directly at the error position and reapplies the detection of possible transformations at the following position. We therefore encode the erroneous part x of the input as part of the transformation without the requirement to be the same for all of them. The achieved improvement is shown in Figure 4.12.

Character Exchange: The last remaining criterion for string similarity which is not covered by a single character transformation is the exchange of consecutive characters. Our lexical strategy is able to detect

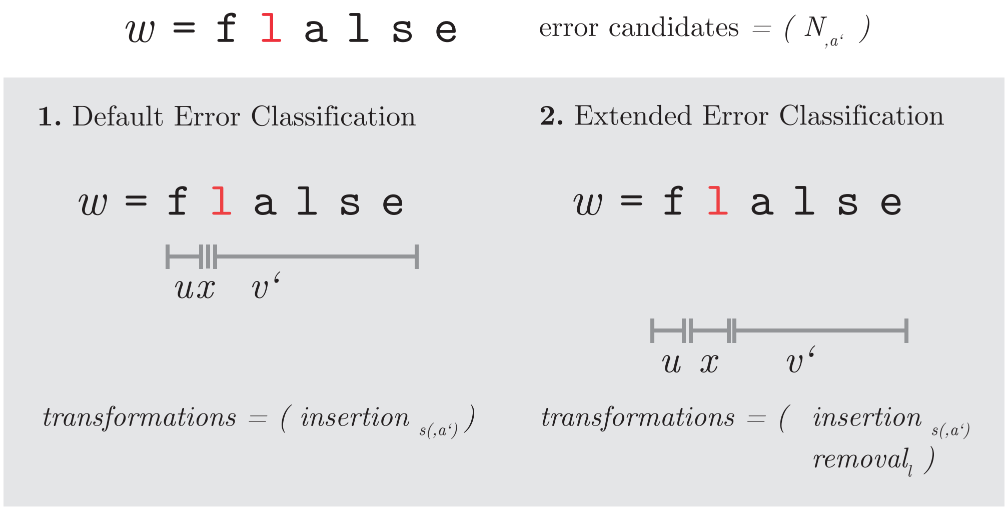
w = f l a l s e
error candidates = (N,a‘)
1. Default Error Classification
w = f l a l s e
u x v‘
transformations = (insertion s(,a‘))
2. Extended Error Classification
w = f l a l s e
u x v‘
transformations = (insertion s(,a‘) removal l)

Figure 4.12: An example depicting the weakness of the default skipping behavior within a general lexical strategy. **1.** Since the members of the recovery set of the error candidates are able to recognize a as well as l, the skipping stops immediately at the error position (albeit the ways to fix the input are obvious and expressible in terms of transformations). Therefore we would only be able to detect a plausible (but erroneous) insertion. **2.** As a consequence, we apply the the detection again assuming that $v' = alse$ and add the additionally detected removal of l to the collection of plausible transformations.

the exchange of two characters. Nevertheless, we are not able to proceed the parsing afterwards since we only know the configurations of the literal parsing expressions directly following the error candidate. To resume the parsing, it would require us to know the configurations of the parsing expressions to the right of the members of the recovery set which can be considered to be a follow follow set. These configurations would enable us to compute the the correct remainder grammar in presence of a character exchange. At the time of writing we did not yet implement a corresponding behavior.

4.5.4 Combined Strategy

Even though we consider the combination of strategies to be a matter of implementation it seems worth clarifying the motivation to do so. The basic token strategy is able to overcome a variety of errors, but sometimes removes larger areas – possibly containing multiple tokens – or replaces the area with a single token. On the other hand, the lexical strategy is rather precise and fine-grained but not as robust as the token strategy in terms of recovery. Therefore we propose the combination of both strategies as often done in classical error handling approaches. Nevertheless, the fact that PEG parsers do not need a scanner requires the lexical strategy to be applied upfront. In general we build a pipeline of strategies ordered by their granularity and apply them in the specified order. As soon as a strategy fails we use the next strategy as a fallback. Let’s again have a look at a slightly modified version of the basic example, having an additional error as shown in Figure 4.13.

$w =$ `t r u l e   b u t   f a l s e`

Figure 4.13: An extended version of the basic example having an additional single-character error within the first token.

On one hand, using the lexical strategy would cause the error handling to fail at the erroneous operator *but* (because there are multiple consecutive, erroneous characters) after being able to fix the erroneous token $trule$. On the other hand, using the token strategy would be able to succeed by detecting $substitution_{trule\ but\ false,true}$ which replaces the whole input with $true$ (the first member of the recovery set we are able to find is the end-of-input (EOI)). We therefore consider the token strategy to be more robust in general. The combination of both approaches is able to detect and repair both errors as shown in Figure 4.14.

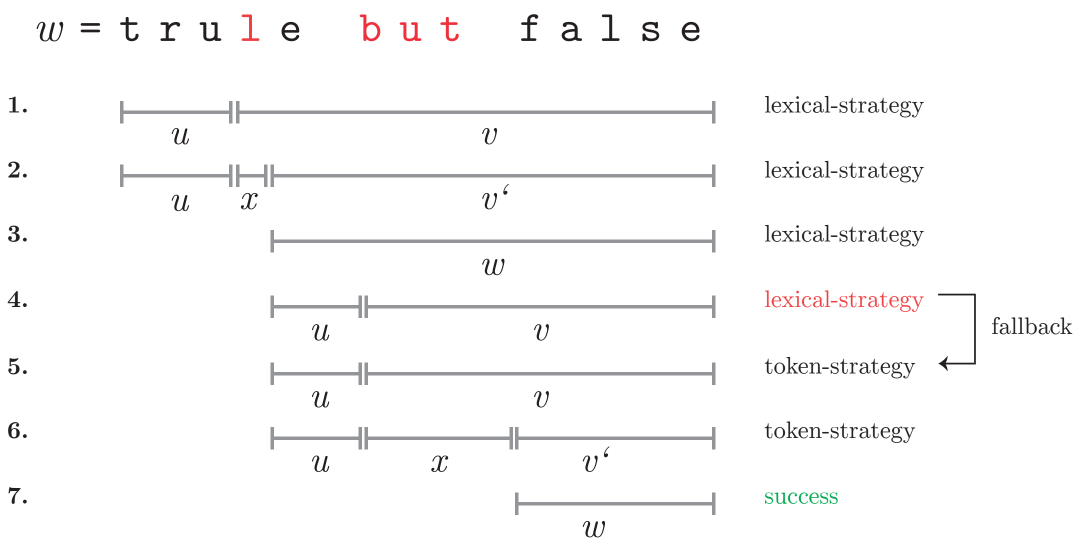

Figure 4.14: Application of the combined strategy. After the recursive invocation of the error handling in step 3, the lexical strategy is not able to detect any plausible transformations and fails in step 4. Therefore the error handling routine uses the next strategy in the pipeline which is able overcome the erroneous operator. Invoking the error handling procedure again in step 7 using the corresponding remainder grammar and the resulting remainder $w = v'$ succeeds. As a consequence, the scheme succeeds and was able to overcome all errors using $transformations = (removal_l\ ,\ substitution_{but,and})$.

4.6 Implementation

Based on the fundamental workings of error handling in PEG parsers we created a concrete implementation for the PetitParser framework. In general we propose the usage of an error handler (*i.e.* an error handling parser) which controls the parsing by delegating to the concrete parser representing the grammar. The error handler initiates the error handling procedure if the original parser fails. To perform the recovery it makes use of concrete strategies which detect the ways to overcome the current error *i.e.* the transformations and corresponding configurations to proceed the parsing from.

4.6.1 Error Handler

To clarify the inner workings of of our implementation and bridge the gap to the theoretical elaborations, it seems worth explaining the basic workflow of the error handler. The error handler initiates the concrete error handling if the original parser fails. Delegating to the original parser allows us to preserve all the characteristics and features of the original parser and prevents side effects in valid parsing runs (such as performance drawbacks in terms of speed or memory consumption). The error handler also performs the recording run and collects the necessary data in the form of parsing records. Furthermore, it controls the backtracking involved in the corresponding remainder processing. In other words, the error handler performs all steps of the algorithm which are uniform to the strategies. To analyze the errors, the error handler relies on a pipeline of strategies which operate on the previously recorded data to *e.g.* extract their own error candidates and detect transformations. In contrast to the theoretical elaborations, the implementation of a strategy does not perform the full error handling process. The implementation of a strategy only performs the error recovery and the error repair, returns the detected, plausible transformations and their according configurations to compute the resulting remainder grammars. Strategies only perform the steps which are specific to their particular way of analyzing the errors. The pipeline can be extended by an arbitrary number of strategies which are invoked in the given order using the current parsing record until the error-handler is able to succeed. Each strategy can be seen as a fallback to its predecessor in the pipeline which would allow us to add extended strategies such as a panic mode. The concept of the combined strategy is therefore implemented as an error handler using a pipeline where the token strategy is used as a fallback for the lexical strategy.

4.6.2 Custom Parsers and Parsing-Actions

Even though we claim our approach to be as generic as possible, in practice, programmers tend to make use of the programmatic freedom the PetitParser framework provides them. Custom parsers do not expose their syntactical meaning to the error handling scheme per default. As a consequence, to enable error handling in custom parsers, the programmer needs to specify how his implementation can be expressed as a PEG or at least using the default parsers of the PetitParser framework. The transformation of the original parser is done in an additional canonicalization step.

Furthermore, like most frameworks and parser generators PetitParser also allows the definition of *parsing-actions* [1]. Parsing-actions are callbacks invoked if a certain production of a grammar succeeds *e.g.* for the creation of ASTs. During the error handling process, these callbacks are removed from the parsers to avoid errors due to structurally unexpected results. Since we detect errors off-line we are not yet able to preserve the parsing result as a product of the error handling algorithm. Hence, to generate a valid

AST the invoking application still has to trigger an additional parsing run on the repaired input, where previously disabled actions are invoked as expected.

5
The Validation

To our best knowledge, there is no common standard for the automated evaluation of error handling approaches. As far as we know, there do not exist any test sets containing erroneous programs including real-wold syntax errors besides the cases collected by Ripley and Druseikis [51] for the Haskell programming language (which does not seem to be publicly accessible). Therefore we used a randomized and generic approach to seed syntax-errors into any type of source code using a reference parser which specifies the grammar; in our case the JSON format. While De Jonge *et al.* [9] propose an overview of metrics and elaborate the possibilities of automated parsing error recovery evaluation, we rely on a rather rudimentary rating to get an idea of the quality of our error handling scheme. Since the criteria as well as the nature of the involved algorithms and test sets — especially if generated automatically — seem to vary, the results of different approaches are hardly comparable.

5.1 Generation of Test Cases

To generate our test cases, we used a randomized approach relying on bug seeding, a form of mutation testing [31]. As a basic corpus we used different sets of JSON files extracted from of the most popular *NPM* (Node Package Manager)[1] packages. The syntactical correctness of source code within the environment of PEGs is validated through the parser's ability to recognize the input. As a consequence we need a reference parser to ensure that the original input was error-free and the modifications we made to the input

[1] `https://www.npmjs.com`

introduced actual errors. In general, the existence of a reference parser is not always given which makes the generation of positive as well as negative test cases dependent on a specification [70] (common to generative approaches).

To seed the errors, we tokenize the input using the corresponding reference parser and generate errors expressed in terms of the three basic transformations — as specified in Subsection 4.3.1 — on the lexical as well as on the token level. this results in three types of errors namely

token errors: Token errors affect a full token which means we remove a token, substitute one token with another, or insert an additional token. The tokens we insert as part of a substitution or an insertion are randomly sampled from the original parsing result.

lexical errors: Lexical errors are created by modifying single characters of a token. The corresponding transformations are randomly applied at the beginning, the end or within a token. Analogous to the token-errors, the characters required for substitutions and insertions are sampled from the original input by selecting a random, non-whitespace character.

mixed errors: Mixed errors are either lexical or token errors which are randomly seeded into the input.

Using a basic corpus of valid inputs, we are able to generate an arbitrary number of test cases containing an arbitrary number of errors.

5.2 Quality Measurement

To evaluate our approach, we solely focus on the quality in terms of the detected and repaired errors, ignoring criteria such as time or space efficiency and error messaging. When evaluating the quality of an error handling scheme, two main problems exist:

1. There is no formal notion describing the correctness of a repair and automated error recovery is comparable to human judgement [46]. In general, a repairing attempt can hardly be considered to be fully correct within an automated evaluation, informally meaning that there is not only a single valid way of handling a syntax error (as seen in the basic example in Figure 4.1 where it is possible to replace but with and as well as with or). Furthermore, the positions of the seeded errors and the detected repair are usually not the same due to surrounding whitespace which is differently treated based on the type of error.

2. There is no formalization of the presence of spurious errors introduced by the error handling scheme [57]. There are cases where spurious errors are forced by the seeded error and do not directly correspond to the behavior of the algorithm $e.g.$ by the insertion of opening delimiters of nested structures such as arrays.

These two factors indicate that we need a more permissive metric to rate our approach. We therefore introduce three categories, rating the behavior of a specific error handling strategy within a corresponding test case. We compare the number of errors seeded into the input to the number of repairs, both expressed in terms of transformations required to generate or fix the errors.

We consider the result of a test case to be

a) successful if the number of repairs is equal to the number of seeded errors.

b) suboptimal if the number of repairs is

- bigger than the amount of seeded errors, meaning that the scheme introduced spurious errors.
- smaller than the amount of seeded errors, meaning that the scheme was not able to detect all errors or discarded them during the repair.

c) failing if the amount of repairs is zero (the error handling scheme was not able to fix the code).

Based on the corresponding data we inspect two additional metrics which rate the behavior of the strategy: The failure rate, describing the ratio of failed test cases to the number of evaluated test cases and the suboptimal rate, denoting the ratio of suboptimal results to the evaluated test cases.

5.3 Results

We evaluated all strategies on a corpus of 300 files. For every file in the corpus we created test cases containing a fixed number of errors ranging from 1 to 5. These test cases were grouped into three suites containing either lexical, token or mixed errors, whereas mixed errors consist of a random combination of both error types. We ran all suites using all types of strategies resulting in a total of 4500 test cases per strategy. We analyzed the behavior of each strategy separately and compares their results to each other. Based on the gathered results we introduced an additional strategy called the inverted strategy, incorporating the advantages of the other strategies as elaborated in Subsection 5.3.4.

5.3.1 Lexical Strategy

The results of the lexical strategy are depicted in Figure 5.1. In general, the failure rate of the strategy seems to correlate directly with the amount of errors at a constant suboptimal rate. In contrast to the expectations, the lexical strategy generates fewer failures on token errors as well as mixed errors than on lexical errors (what it was designed for). Said behavior is a consequence of the error seeding algorithm as well as the underlying JSON grammar discussed in Section 5.4. Also — despite the restricted scope of a single character — the lexical strategy applies reparations in a more speculative way if the error is followed by whitespace (which is considered to be a valid member of the follow set). The insertion of single character tokens is a difference to classical spelling correction which only repairs existing tokens.

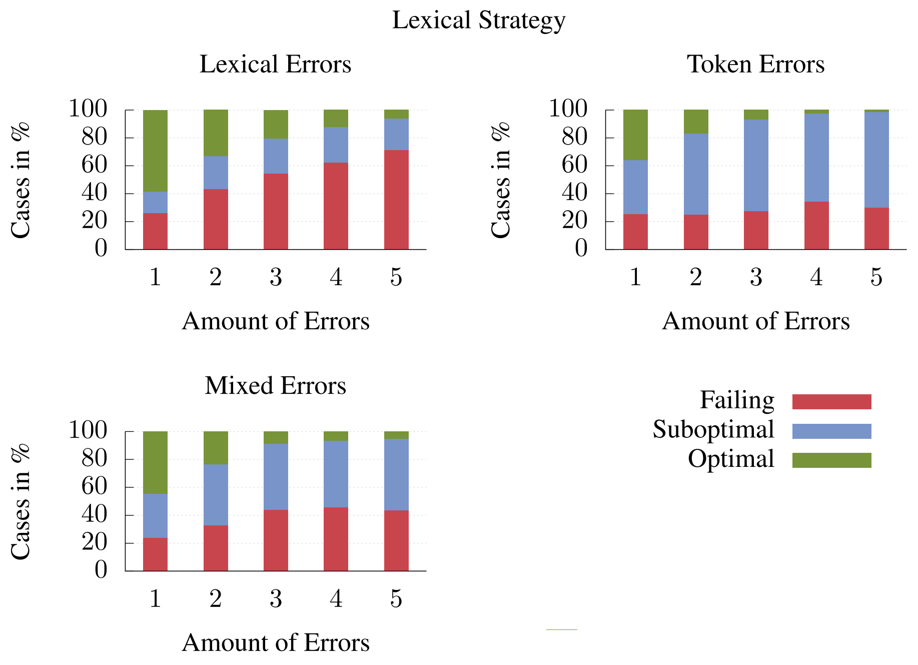

Figure 5.1: Results of the lexical strategy in presence of lexical, token or mixed errors. Surprisingly, the lexical strategy is able to overcome numerous token errors in our setup.

5.3.2 Token Strategy

Figure 5.2 depicts the results of of the token strategy. The token strategy has a low failure rate of ~ 5% but tends to create more suboptimal results with an increasing amount of errors. The token strategy can therefore be considered — as aforementioned — to be more robust than the lexical strategy. The token strategy was able to overcome all non-consecutive errors but failed in cases where the seeded modifications caused the input to have multiple consecutive errors.

5.3.3 Combined Strategy

Figure 5.3 depicts the results of of the combined strategy chaining the lexical and the token strategy. The strategy has an improved failure rate in comparison to the token strategy which resides below 1%. On the down side, there was a high suboptimal rate at all numbers of errors. The high amount of suboptimal results is caused by the increasing number of plausible repairs due to the chaining of strategies. Since the lexical strategy is more speculative in some cases, it is able to create a new context for the chained strategies.

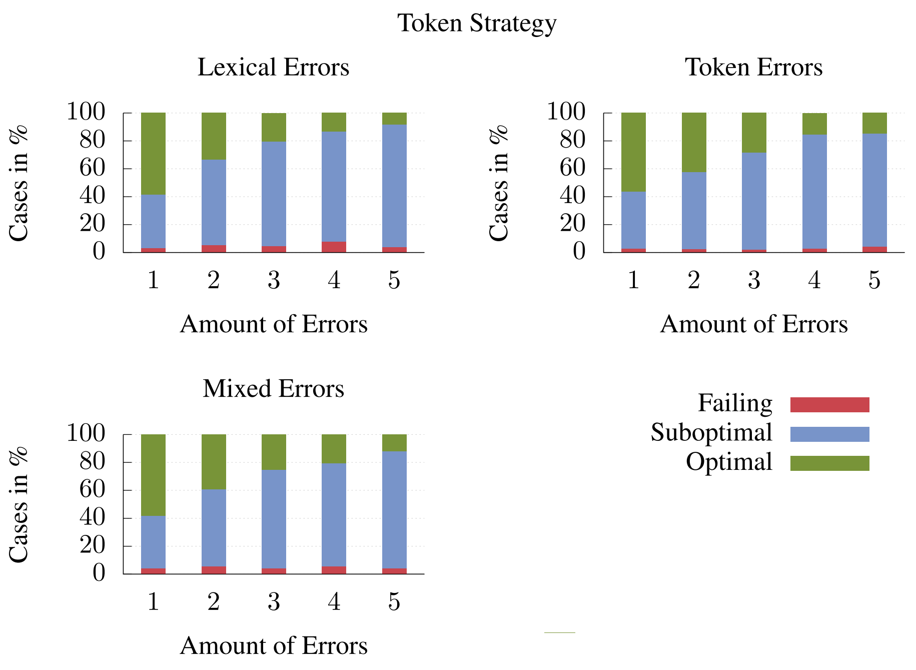

Figure 5.2: Results of the token strategy in presence of lexical, token or mixed errors. The token strategy has a low failure rate in the given setup but also tends to introduce suboptimal results.

5.3.4 Inverted Strategy

Based on the results of the previously defined strategies, we tried to create a strategy which exploits the advantages of the other strategies. We wanted to achieve the low failure rate of the combined strategy in combination with the slightly lower suboptimal rate of the token strategy. Since we use a pipeline of strategies on each error it seems obvious to combine the token and the combined strategy. Having a look at the pipeline reveals that combining these approaches basically leads to an application order of

$$\textit{token-strategy} \rightarrow \textit{lexical-strategy} \rightarrow \textit{token-strategy}$$

where $\rightarrow$ denotes the fallback. This means we would invoke the token-based recovery twice on the same error, or in other words, use the token strategy as its own fallback. It is therefore sufficient to reduce the pipeline to the first two steps. The resulting pipeline is using the lexical as a fallback for the token strategy which is the inversion of the combined strategy. We therefore call it the inverted strategy whose results are depicted by Figure 5.4. The inverted strategy is able to deliver a higher amount of optimal results at the same low failure rate of the combined strategy.

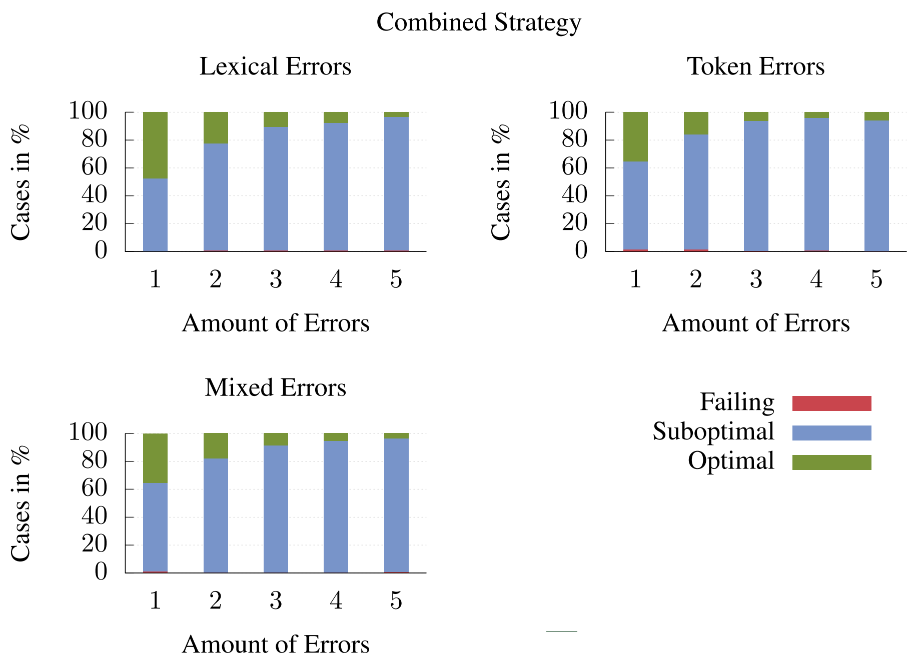

Figure 5.3: Results of the combined strategy in presence of lexical, token or mixed errors. The combined strategy is able to overcome nearly all syntax errors at the cost of inserting spurious errors or ignoring errors by skipping them.

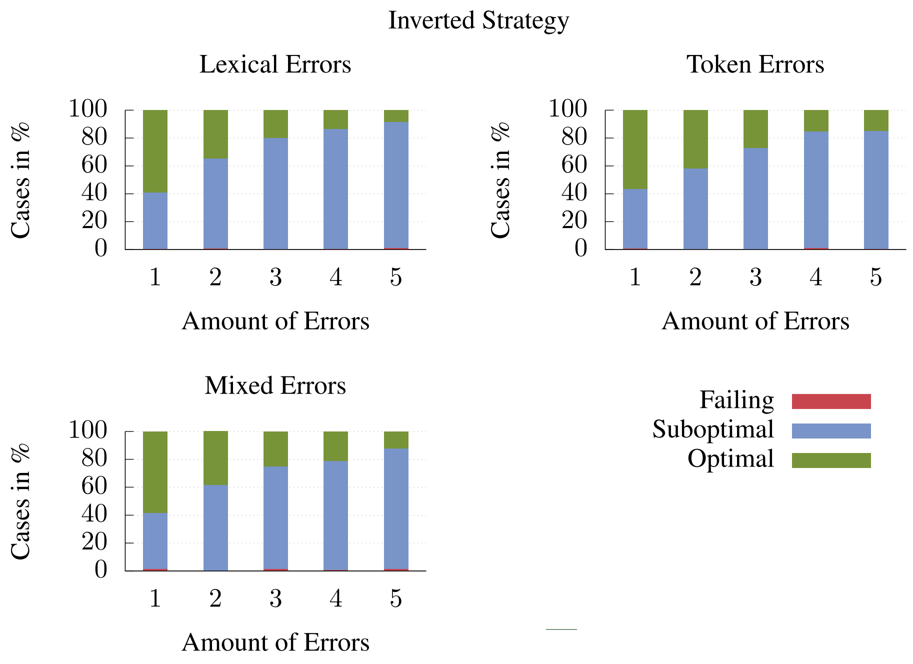

Figure 5.4: The results of the inverted strategy on all three types of errors. The inversion of the pipeline resulted in a failure rate which is equal to the combined strategy in conjunction to the slightly lower suboptimal rate of the token strategy.

5.3.5 Comparison

The purpose of implementing multiple strategies and combining them was to improve the quality and robustness of the error handling scheme. We briefly mention some of the main insights of their comparison in terms of failure rate and the creation of suboptimal results. To get an overview of their differences concerning the quality of the error handling we aggregated the results of all strategies in relation to lexical errors in Figure 5.5, token errors Figure 5.6 and mixed errors Figure 5.7. While the lexical strategy has a higher tendency to fail if there are more errors, the other strategies show an increasing suboptimal rate depending on the amount of errors. Figure 5.8 shows the accumulated failure rate, Figure 5.9 the accumulated suboptimal rate for all strategies in relation to the amount of errors.

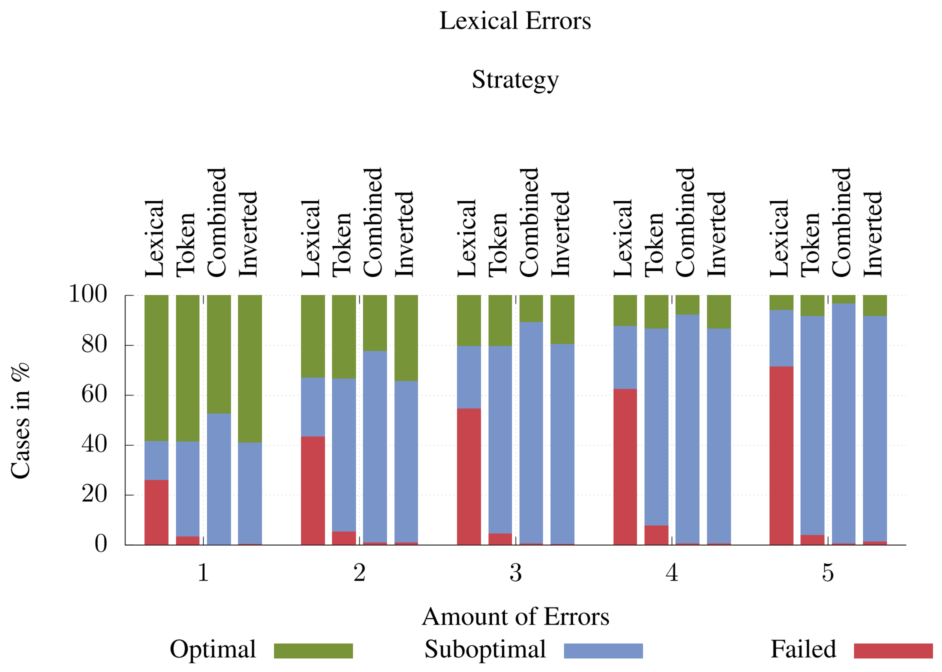

Figure 5.5: Comparison of strategies on lexical errors. While the token as well as the combined strategy introduce an increased amount of suboptimal results, the lexical strategy tended to fail with an increasing number of errors.

5.4 Discussion

Overall, the implementation of the presented strategies is able to fulfill the expectations. The token strategy, the combined strategy as well as the inverted strategy are able to overcome an arbitrary number of non-consecutive errors within the boundaries of our case study, but produce a relatively high number of suboptimal results. We consider this to be a consequence of adhering to the semantics of the ordered choice *i.e.* succeeding with the first working attempt to repair the input. While it seems that the inverted

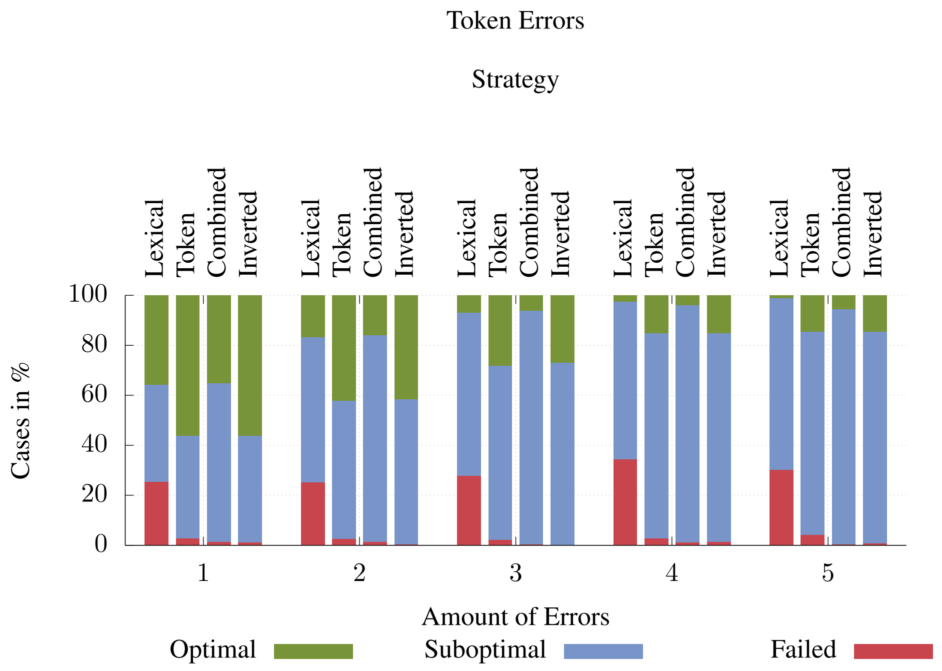

Figure 5.6: Comparison of strategies on a test suite containing full token errors. Analogous to the comparison on lexical errors in Figure 5.5, the coarse grained token strategy and the combined strategy generated more suboptimal results in presence of a higher number of errors.

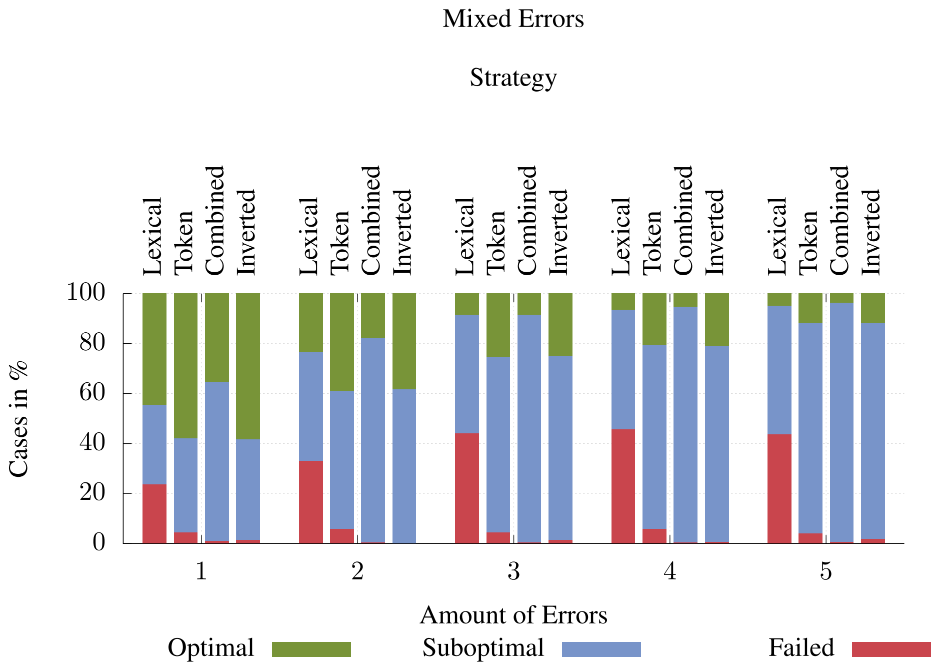

Figure 5.7: Comparison of strategies on mixed errors. The results are similar to the other error types showing an increasing suboptimal rate depending on the number of failures.

strategy delivered the best results it might be worth mentioning that this is a consequence of our rating in correspondence to the fact that we take the first working repairing attempt (negatively influencing the combined strategy). Before discussing some specific characteristics of our evaluation, two additional observations have to be mentioned upfront:

1. The lexical strategy performed better in the presence of token errors than it did in the presence of lexical errors having a significantly lower failure rate. This is a consequence of the way we seed errors (discussed in Subsection 5.4.2) and the underlying JSON grammar which often consists of single character tokens (briefly explained in Subsection 5.4.1).

2. The failure rate of the inverted strategy slightly differs from the failure rate of the combined strategy, even though it should be exactly the same. This difference is caused by test cases which timed out during execution and were therefore rated as failed (see Subsection 5.4.4).

5.4.1 JSON

As aforementioned, the lexical strategy was unexpectedly able to overcome numerous token errors. The reason lies in the nature of the JSON grammar. The grammar consists of several single-character tokens such as `','`, `':'` or the opening and closing delimiters of the nested structures *i.e.* arrays and objects. The lexical strategy is therefore able to fix full token errors as long as their length does not exceed a single

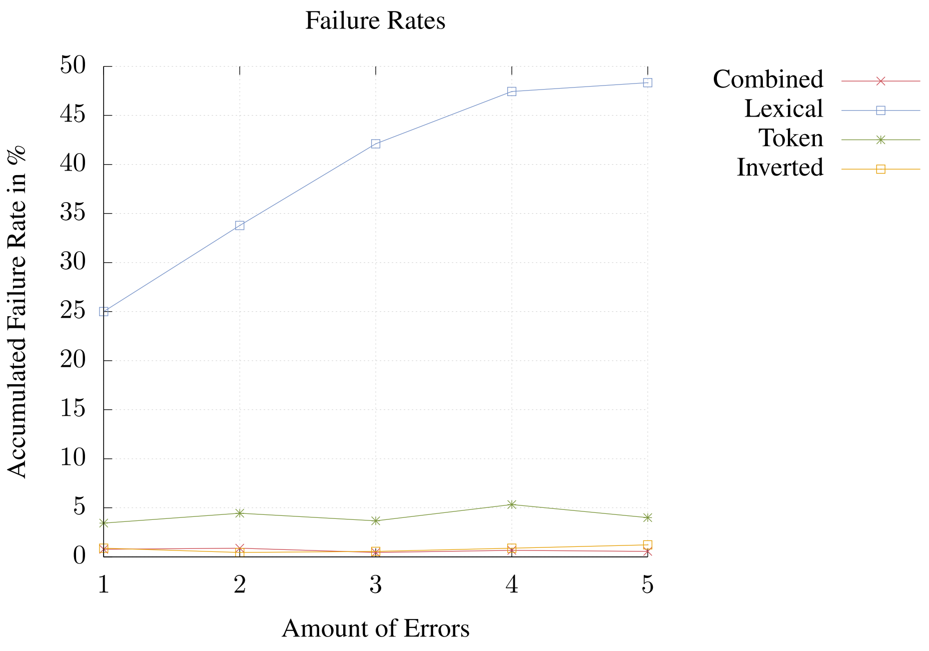

Figure 5.8: The accumulated failure rate for each strategy in direct comparison. The strategies using an extended pipeline, namely the combined strategy and the inverted strategy have a comparable low failure rate.

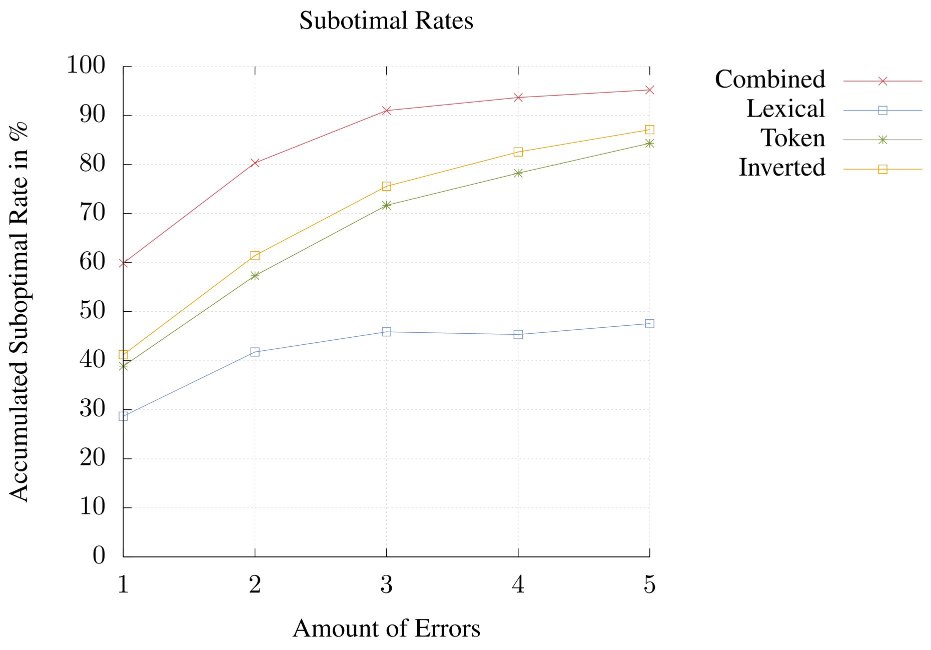

Figure 5.9: The accumulated rate of suboptimal results for each strategy in direct comparison. All strategies, especially the combined strategy tend to generate suboptimal results with an increasing amount of errors. By inverting the pipeline of the combined strategy, we were able reduce the suboptimal rate.

character. In addition, the grammar is relatively simple and has the LL(1) property, featuring a minimal degree of ambiguity. The implementation does not rely on the lookahead capabilities of PEGs, namely the and- as well as the not-predicate. We can only speculate on the impact of said structures on the error handling scheme.

Moreover, we did not elaborate the error proneness [36] of the JSON language itself. Nevertheless, it is worth mentioning that JSON in practice mainly consists of strings and nested data structures which are both claimed to be critical to error handling algorithms [8, 57]. Also, in contradiction to general-purpose programming languages, the JSON format does not contain identifiers which would complicate the error handling as well as the error-seeding procedure.

5.4.2 Error Seeding

Our approach to seed syntax errors has a high degree of randomization. Therefore, it is hard to draw conclusions on the probability of the errors to appear in real-world programs or their initial positioning within the source code. An error seeded at the beginning of a choice, preventing the parser from recognizing a prefix of a certain branch, increases the amount of error candidates and hence the amount of plausible transformations. As a consequence, our error handling approach has a higher probability of generating suboptimal results in these cases. While the way we seed token errors uses valid syntactical elements of the language for insertions, the process of seeding lexical errors makes use of randomly sampled characters. The higher degree of randomization seems to influence the robustness of the algorithm, indicated by the fact that all strategies have a higher failure rate on lexical errors.

Also we did not evaluate factors such as the characteristics of the seeded errors with regard to the density of their occurrence, the syntactical elements they affect [51] or the probability of their occurrence in real programs [69].

5.4.3 Quality Measurement

It has to be said that our rating system is rather permissive and coarse grained, especially in the cases creating suboptimal results. We did neither elaborate the impact of the chosen repair in terms of the affected (*e.g.* removed) tokens nor did we take the amount of spurious errors into account. Future versions of the automated evaluation might incorporate a more sophisticated rating scheme such as comparison of the original to the recovered AST [9].

5.4.4 Performance

Since we focused on feasibility and a generic description of the error handling scheme, we did not elaborate any performance related criteria. Nevertheless, during the evaluation of the algorithm we noticed long running test cases. Some of the test cases were even rated as failed because they ran into a timeout. In some cases the algorithm has a high complexity due to the backtracking and the evaluation of multiple options of repairing the input. Errors appearing deep in the call history *e.g.* caused by deeply nested structures generate complex remainder grammars and therefore excessive recording runs causing the performance to degrade in terms of speed and memory consumption.

6 Related Work

There is a plethora of work concerning error handling in classical parsing algorithms created for CFGs [7, 24, 26, 61]. Our own approach is based on the basic version of follow set recovery as presented by Stirling [58] in combination with remainder grammars which are comparable to the concept of continuations presented by Röhrich [52].

Since most of the problems arise due to the backtracking behavior of PEG parsers, an alternative approach to a recording run is to restrict the backtracking itself in the presence of failures creating an explicit error state. Hutton [29] presented a concept called the *nofail* combinator. The nofail combinator marks rules in the grammar which are not allowed to fail (which means that they are unambiguous). He stated the fact that two states (*i.e.* success and fail) are not sufficient to express the presence of errors in backtracking parsing algorithms effectively. Therefore, he introduced the *three values technique* which allowed tagging the parsing results with one of three states, namely `OK`, `Fail` and `Error`. Partridge and Wright [45] later on extended this concept to the *four values technique* with an additional state `EPSN`, indicating the occurrence of a failure in parsers without making any kind of progress *e.g.* optional parsers. The usage of mechanisms that restrict the backtracking of a parser allows errors to be detected at run-time.

Swierstra *et al.* [59] elaborate the possibilities of creating error-correcting combinator parsers for languages having the LL(1) property. Their approach also relies on local recovery using an acceptable set called the *noskip set*. Unlike our solution using PetitParser, their solution targets functional languages and corresponding sets need to be computed during the parsing itself (and are passed as a parameter to subroutines). One advantage of their approach is the fact that it works *on-line i.e.* during the parsing process and therefore allows the parsing of the input to be resumed without *off-line* processing.

Concerning error handling in scannerless parsers, De Jonge and Visser [8] propose a combination of several recovery techniques for generated *generalized LR parsers* (*GLR*) [32, 33]. Generalization allows the parallel evaluation of different parse trees based on different metrics instead of the sequential analysis we rely on. Their approach involves noise and water skipping techniques or island parsing [43] to skip erroneous input. Furthermore, they take the formatting of the code into account to detect erroneous regions, requiring the programmer to annotate a numerous of parts of their grammar.

Most of the work concerning error handling in the domain of PEGs focuses on error reporting, especially in terms of improved error messaging. As far as we know, there are no attempts to present a generic recovery approach which can be ported to other PEG based environments or backtracking parsers in general. Besides implementations improving the error messaging using labels *e.g.* Rats! [22] or PEGjs[1], there exist implementations of error recovery similar to our approach *e.g.* in Parboiled[2] or Papa Carlo[3]. Parboiled also uses a recording technique to determine the involved expressions, subsequently applies single character modifications and re-processes the whole input. Papa Carlo on the other hand, requires the programmer to explicitly mark rules as recoverable. As a consequence the corresponding rules will never fail and the framework tries to fix and re-process the input. To avoid the overhead caused by repeatedly parsing the valid prefix of the input, Papa Carlo uses memoization.

Analogous to the nofail combinator some approaches use *cuts* to modify the backtracking behavior. Cuts are a concept borrowed from logic programming languages *i.e.* Prolog, annotating points inside a grammar which are unambiguous. Basically, if a rule in the grammar fails after a cut, it is not necessary to try alternative parses. Mizushima *et al.* [41] propose the introduction of cuts to the PEG formalism to reduce the space consumption of packrat parsers. Besides the fact that cuts remove the necessity to store all intermediate results of a choice, they usually improve the error handling of backtracking parsers by implicitly introducing an additional (error) state. Based on the concept of suppressed backtracking *i.e.* cuts, Maidl *et al.* [37, 38] propose an approach using labelled failures. They extend the PEG formalism with the necessary semantics to throw, catch and handle errors using semantic actions. Their idea is inspired by the exception handling some higher level languages provide. Maidl *et al.* extend the PEG semantics at two relevant points: First, the behavior of the expressions is extended to preserve the farthest failure position, second, they introduce the notations and semantics for labelled errors. Annotating choices and parsers with labels allows bypassing the backtracking of the parser and enclose them in a scope (by catching the error at an arbitrary, enclosing choice). Furthermore, the error messages get enhanced by joining them at the corresponding label. A parser which is not allowed to fail – analogous to the nofail combinator – *e.g.* is annotated with $p / \Uparrow^{p\_error\_label}$.

Worth mentioning is Redziejowski's series of publications[4] on PEGs, especially on the application of classical parsing concepts to them [48]. Redziejowski presents algorithms and definitions of concepts such as follow set computation for PEGs. Although we do not focus on or investigate further into formalization, Mascarenhas *et al.* [11] provide a formalization of the relations between CFG and PEGs.

[1] `http://pegjs.org`
[2] `http://www.parboiled.org`
[3] `http://lakhin.com/projects/papa-carlo`
[4] `http://romanredz.se/pubs.htm`

7
Conclusion and Future Work

Error recovery is a complicated field of computer science, especially for backtracking parsing algorithms. As far as we know there are no automated error handling approaches targeting PEGs. We presented an automated, language agnostic, local error handling approach — incorporating error detection, error recovery as well as error repair — suitable for PEG based parsers. This thesis provides an assessment of the issues arising in PEG-based parsers and provides corresponding solutions to overcome them. We informally describe the foundations of a local, backtracking error handling scheme and provide four different variations, called strategies, of the basic algorithm. The presented theoretical foundations address the most relevant issues arising in error handling algorithms (besides the error messaging) and adhere to the semantics of the prioritized choice by succeeding with the first working recovery attempt. Unlike other approaches, we explicitly described how to express the state of a parser combinator in a simple way which furthermore enables dynamic analysis of the grammar itself *e.g.* subgrammar computations. Based on configurations we also presented a new way to express the remaining grammar given a certain configuration of the parser. We briefly described our approach to automated evaluation of error recovery schemes and applied it to our particular implementations for the PetitParser framework in a case study using the JSON grammar. Within the scope of the case study we collected and evaluated corresponding data and discussed the result with regard to the presented strategies.

In conclusion, our approach is able to overcome an arbitrary number of non-consecutive errors of different granularity in terms of the affected symbols. The presented strategies provided promising results, having a low failure rate but tend to create high numbers of suboptimal results meaning they succeed by either introducing spurious errors or discarding errors during the recovery phase. To confirm the achieved results and observe the impact of ambiguity and the unbounded lookahead of PEGs, additional evaluations involving more complex grammars are necessary. A first attempt to decrease the suboptimal rate of our approach could be the introduction of metrics comparing the local costs of a repair. Repairing attempts and their corresponding ways to proceed could be prioritized as it is done in local least-cost recovery approaches.

One issue of our scheme might be the fact that it is allowed to fail which is unavoidable under the given circumstances. To achieve an algorithm which does not fail, we would have to sacrifice the guarantee of producing a semantically valid version of the input by leaving skipped regions unrepaired. Currently we need an additional parsing run to get the recovered result. Up to now we are not able to preserve and recompose the results of the parse within the error handling procedure, a consequence of the implicit, off-line error detection. Improvements in this area could be achieved by introducing cuts which convert failures into an explicit error and trigger the error handling procedure during the parsing at run-time. Also, there is no reasonable error messaging included. The error reporting is considered to be the user-interface of error handling and would improve the usability in the case of failure by allowing observations on the performed repairing attempts. Future versions of the error handling scheme should therefore provide improved error reporting capabilities. To be useful in practice we should evaluate and optimize the implementation in terms of speed as well as memory efficiency.

While the results look promising the most interesting results of this thesis reside within the contributions to future work. The usage of configurations to express the state of a parser combinator might look trivial but opens up numerous possibilities in terms of optimizations and subgrammar computation. Keeping track of the configurations of the furthest-failing parsing expressions during the parsing could allow us to remove the necessity of a recording run and enable the computation of recovery sets without the parsing records. Configurations could also be used to resume a parser without the necessity to compute a remainder grammar and therefore allocate new parsers. To know if and how different configurations of a parser can be unified into a single state would allow us to uniquely represent a sentential form of a grammar and reduce the necessity of backtracking during the error handling to a bare minimum. In terms of subgrammar computation, configurations can be used to represent partitions of a grammar. Two configurations which diverge at a certain point representing bounds could allow us to form partitions of the original grammar. These partitions could be converted into recovering subsystems which we can apply to certain regions of the input and comparing the resulting costs. Reasoning on the correctness of such computations requires further investigations with a background in graph theory and goes beyond the scope of this thesis.

A
Appendices

A.1 Boolean Logic Grammar

A grammar specifying a boolean logic expression grammar formalized as a PEG. We left out the exact definition for _ *i.e.* spaces since giving all possible UTF-8 code points seems pointless.

```
start         ← _ expressionOr EOI
_             ← [ ]*
operatorOr    ← 'or' _
operatorAnd   ← 'and' _
booleanTrue   ← 'true' _
booleanFalse  ← 'false' _
boolean       ← booleanTrue / booleanFalse
expressionOr  ← expressionAnd ( operatorOr expressionAnd )*
expressionAnd ← boolean ( operatorAnd boolean )*
EOI           ← !.
```

Listing 6: Boolean Logic Grammar.

A.2 JSON Grammar

A grammar specifying *Javascript Object Notation* formalized as a PEG based on the formalisms given at `http://www.json.org`. For the sake of simplicity, escape codes in strings and an exact definition of whitespace (denoted as _) is omitted.

```
start              ← _ value
_                  ← [ ]*
comma              ← ',' _
dot                ← '.'
curlyBraceLeft     ← '[' _
curlyBraceRight    ← ']' _
squareBracketLeft  ← '{' _
squareBracketRight ← '}' _

booleanTrue        ← 'true' _
booleanFalse       ← 'false' _
boolean            ← booleanTrue / booleanFalse

null               ← 'null' _

number             ← '-'? integer numberFraction? numberExponential? _
numberInteger      ← ( [1-9] [0-9]* ) / '0'
numberFraction     ← dot [1-9]+
numberExponential  ← [eE] numberSign? [1-9]+
numberSign         ← '+' / '-'


string             ← stringQuote ( !stringQuote . )* stringQuote _
stringQuote        ← '"'

array              ← squareBracketLeft arrayItems? squareBracketRight
arrayItems         ← arrayItem ( comma arrayItem )*
arrayItem          ← value

object             ← curlyBracketLeft objectItems? curlyBracketRight
objectItems        ← objectItem ( comma objectItem)*
objectItem         ← string colon value

value              ← boolean / null / string / array / object / number
```

Listing 7: Json Grammar.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Alfred V. Aho and Thomas G. Peterson. A Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM Journal on Computing*, 1(4):305–312, 1972.

[3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[4] S. O. Anderson, R. C. Backhouse, E. H. Bugge, and C. P. Stirling. An Assessment of Locally Least-Cost Error Recovery. *The Computer Journal*, 26(1):15–24, 1983.

[5] Pierre Boullier and Martin Jourdan. A New Error Repair and Recovery Scheme for Lexical and Syntactic Analysis. *Sci. Comput. Program.*, 9(3):271–286, December 1987.

[6] Michael G. Burke and Gerald A. Fisher. A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery. *ACM Trans. Program. Lang. Syst.*, 9(2):164–197, March 1987.

[7] J. A. Dain. Syntax Error Handling in Languate Translation Systems. Technical report, Coventry, UK, UK, 1991. Research Report.

[8] Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg. Natural and Flexible Error Recovery for Generated Modular Language Environments. *ACM Trans. Program. Lang. Syst.*, 34(4):15:1–15:50, December 2012.

[9] Maartje de Jonge and Eelco Visser. Automated Evaluation of Syntax Error Recovery. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 322–325, New York, NY, USA, 2012. ACM.

[10] Pierpaolo Degano and Corrado Priami. Comparison of Syntactic Error Handling in LR Parsers. *Softw. Pract. Exper.*, 25(6):657–679, June 1995.

[11] Roberto Ierusalimschy Fabio Mascarenhas, Sérgio Medeiros. On the Relation between Context-Free Grammars and Parsing Expression Grammars. *CoRR*, abs/1304.3177, 2013.

[12] Charles N. Fischer, KC Tai, and DR Milton. Immediate Error Detection In Strong LL (1) Parsers. *Information Processing Letters*, 8(5):261–266, 1979.

[13] Jeroen Fokker. Functional Parsers. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 1–23, London, UK, UK, 1995. Springer-Verlag.

[14] Bryan Ford. Packrat Parsing : a Practical Linear-Time Algorithm with Backtracking. In *Proceedings of the International Conference on Functional Programming ICFP 2002*, 2002.

[15] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.

[16] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37/9, pages 36–47, New York, NY, USA, 2002. ACM.

[17] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[18] Jeffrey E.F. Friedl. *Mastering Regular Expressions*. " O'Reilly Media, Inc.", 2002.

[19] Susan L. Graham, Charles B. Haley, and William N. Joy. Practical LR Error Recovery. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '79, pages 168–175, New York, NY, USA, 1979. ACM.

[20] Susan L. Graham and Steven P. Rhodes. Practical Syntactic Error Recovery in Compilers. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 52–58, New York, NY, USA, 1973. ACM.

[21] Susan L. Graham and Steven P. Rhodes. Practical Syntactic Error Recovery. *Commun. ACM*, 18(11):639–650, November 1975.

[22] Robert Grimm. Better Extensibility Through Modular Syntax. *SIGPLAN Not.*, 41(6):38–51, June 2006.

[23] Josef Grosch. Efficient and Comfortable Error Recovery in Recursive Descent Parsers. *STRUCT. PROGRAM.*, 11(3):129–140, 1990.

[24] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques — A Practical Guide*. Springer, 2008.

[25] Paul E Hallowell Jr. Top-Down Parsing Syntax Error Recovery. Technical report, NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 1985.

[26] K Hammond and V J. Rayward-Smith. A Survey of Syntactic Error Recovery and Repair. *Comput. Lang.*, 9(1):51–67, August 1984.

[27] James J. Horning. What the compiler should tell the user. In *Compiler Construction: An Advanced Course*, pages 525–548. Springer Berlin Heidelberg, Berlin, Heidelberg, 1974.

[28] Paul Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3:39–60, 1997.

[29] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[30] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[31] Yue Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, Sept 2011.

[32] Bernard Lang. Parallel Non-Deterministic Bottom-Up Parsing. *SIGPLAN Not.*, 6(12):56–57, September 1971.

[33] Bernard Lang. Deterministic Techniques for Efficient Non-Deterministic Parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer Berlin Heidelberg, 1974.

[34] Ronald Paul Leinius. *Error Detection and Recovery for Syntax Directed Compiler Systems*. PhD thesis, 1970. AAI7024758.

[35] J.-P. Lévy. Automatic Correction of Syntax-Errors in Programming Languages. *Acta Informatica*, 4(3):271–292, 1975.

[36] Charles R. Litecky and Gordon B. Davis. A Study of Errors, Error-Proneness and Error Diagnosis in Cobol. *Commun. ACM*, 19(1):33–38, January 1976.

[37] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Exception Handling for Error Reporting in Parsing Expression Grammars. In André Rauber Du Bois and Phil Trinder, editors, *Programming Languages*, volume 8129 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2013.

[38] André Murbach Maidl, Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. Error Reporting in Parsing Expression Grammars. *CoRR*, abs/1405.6646, 2014.

[39] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. From regexes to parsing expression grammars. *Science of Computer Programming*, 2012.

[40] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. Left Recursion in Parsing Expression Grammars. In *Proceedings of the 16th Brazilian Conference on Programming Languages*, SBLP'12, pages 27–41, Berlin, Heidelberg, 2012. Springer-Verlag.

[41] Kota Mizushima, Atusi Maeda, and Yoshinori Yamaguchi. Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space. *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '10*, page 29, 2010.

[42] Rolf Molich and Jakob Nielsen. Improving a Human-Computer Dialogue. *Commun. ACM*, 33(3):338–348, March 1990.

[43] Leon Moonen. Generating robust parsers using island grammars. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings Eighth Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society, October 2001.

[44] Ajit B. Pai and Richard B. Kieburtz. Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table-Drive Parsers. *ACM Trans. Program. Lang. Syst.*, 2(1):18–41, January 1980.

[45] Andrew Partridge and David Wright. Predictive parser combinators need four values to report errors. *Journal of Functional Programming*, 6:355–364, 1996.

[46] Thomas J. Pennello and Frank DeRemer. A Forward Move Algorithm for LR Error Recovery. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 241–254, New York, NY, USA, 1978. ACM.

[47] Thomas G. Peterson. *Syntax Error Detection, Correction and Recovery in Parsers*. PhD thesis, Hoboken, NJ, USA, 1972. AAI7231175.

[48] Roman R. Redziejowski. Applying Classical Concepts to Parsing Expression Grammar. *Fundam. Inf.*, 93(1-3):325–336, January 2009.

[49] Lukas Renggli. *Dynamic Language Embedding With Homogeneous Tool Support*. PhD thesis, University of Bern, October 2010.

[50] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, pages 1–4, Malaga, Spain, June 2010.

[51] G.David Ripley and Frederick C. Druseikis. A Statistical Analysis of Syntax Errors. *Comput. Lang.*, 3(4):227–240, January 1978.

[52] Johannes Röhrich. Methods for the Automatic Construction of Error Correcting Parsers. *Acta Inf.*, 13(2):115–139, February 1980.

[53] Johannes Röhrich. Behandlung syntaktischer Fehler. *Informatik Spektrum*, 5(3):171–184, 1982.

[54] Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages*. Chalmers University of Technology, 1995.

[55] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 170–178, New York, NY, USA, 1989. ACM.

[56] Seppo Sippu and Eljas Soisalon-Soininen. A Syntax-Error-Handling Technique and Its Experimental Analysis. *ACM Trans. Program. Lang. Syst.*, 5(4):656–679, October 1983.

[57] Michael Spenke, Heinz Muhlenbein, Monika Mevenkamp, Friedemann Mattern, and Christian Beilken. A language independent error recovery method for LL(1) parsers. *Software: Practice and Experience*, 14(11):1095–1107, 1984.

[58] Colin Stirling. Follow Set Error Recovery. *Software: Practice and Experience*, 15(3):239–257, 1985.

[59] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*, pages 184–207, London, UK, UK, 1996. Springer-Verlag.

[60] Kuo-Chung Tai. Locally Minimum-distance Correction of Syntax Errors in Programming Languages. In *Proceedings of the ACM 1980 Annual Conference*, ACM '80, pages 204–210, New York, NY, USA, 1980. ACM.

[61] Peter N. van den Bosch. A Bibliography on Syntax Error Handling in Context Free Languages. *SIGPLAN Not.*, 27(4):77–86, April 1992.

[62] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[63] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *J. ACM*, 21(1):168–173, January 1974.

[64] TIMA WAGNER and Susan L Graham. General Incremental Lexical Analysis. *University of California, Berkeley*, 1997.

[65] Alessandro Warth, James R. Douglass, and Todd Millstein. Packrat Parsers Can Support Left Recursion. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '08, pages 103–110, New York, NY, USA, 2008. ACM.

[66] Niklaus Wirth. PL360, a Programming Language for the 360 Computers. *J. ACM*, 15(1):37–74, January 1968.

[67] Wuu Yang. On the look-ahead problem in lexical analysis. *Acta Informatica*, 32(5):459–476, 1995.

[68] Wuu Yang, Chey-Woei Tsay, and Jien-Tsai Chan. On the applicability of the longest-match rule in lexical analysis. *Computer Languages, Systems & Structures*, 28(3):273 – 288, 2002.

[69] Edward A. Youngs. Human Errors in Programming. *International Journal of Man-Machine Studies*, 6(3):361–376, 1974.

[70] S. V. Zelenov and S. A. Zelenova. Generation of Positive and Negative Tests for Parsers. *Programming and Computer Software*, 31(6):310–320.