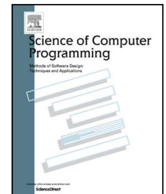
Science of Computer Programming 150 (2017) 31–55

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico



Mining structured data in natural language artifacts with island parsing

Alberto Bacchelli [a,*], Andrea Mocci [b], Anthony Cleve [c], Michele Lanza [b]

[a] Department of Informatics, University of Zurich, Switzerland
[b] REVEAL @ Software Institute, Università della Svizzera italiana (USI), Switzerland
[c] Faculty of Informatics, University of Namur, Belgium

A R T I C L E I N F O

Article history:
Received 11 December 2015
Received in revised form 18 November 2016
Accepted 13 June 2017
Available online 1 August 2017

Keywords:
Mining software repositories
Unstructured data
Island parsing

A B S T R A C T

Software repositories typically store data composed of structured and unstructured parts. Researchers mine this data to empirically validate research ideas and to support practitioners' activities. Structured data (e.g., source code) has a formal syntax and is straightforward to analyze; unstructured data (e.g., documentation) is a mix of natural language, noise, and snippets of structured data, and it is harder to analyze. Especially the structured content (e.g., code snippets) in unstructured data contains valuable information.

Researchers have proposed several approaches to recognize, extract, and analyze structured data embedded in natural language. We analyze these approaches and investigate their drawbacks. Subsequently, we present two novel methods, based on scannerless generalized LR (SGLR) and Parsing Expression Grammars (PEGs), to address these drawbacks and to mine structured fragments within unstructured data. We validate and compare these approaches on development emails and Stack Overflow posts with Java code fragments. Both approaches achieve high precision and recall values, but the PEG-based one achieves better computational performances and simplicity in engineering.

© 2017 Published by Elsevier B.V.

1. Introduction

Programmers are supported by a variety of development tools, such as version control systems (e.g., GIT), issue tracking systems (e.g., BugZilla), and electronic communication services (e.g., web forums or mailing lists), that accumulate and record a wide range of data about the development, evolution, and usage of software projects. By mining this data, researchers extract facts to validate research hypotheses and to support practitioners' day-to-day activities.

Many software repositories archive data that comprises structured information, artifacts either written by humans for a machine (e.g., source code) or generated by a machine for humans (e.g., execution traces). We call such information *structured software data*, because of its clearly structured syntax, defined through a formal grammar. Knowledge within structured data artifacts can be extracted and modeled with well-established parsing techniques.

Other software repositories archive information produced by humans for humans (e.g., emails and forum posts) and used to exchange information among project stakeholders. We call it *unstructured software data*, because it is written according

* Corresponding author.
E-mail addresses: bacchelli@ifi.uzh.ch (A. Bacchelli), andrea.mocci@usi.ch (A. Mocci), anthony.cleve@unamur.be (A. Cleve), michele.lanza@usi.ch (M. Lanza).

http://dx.doi.org/10.1016/j.scico.2017.06.009
0167-6423/© 2017 Published by Elsevier B.V.

to non-formal grammars, including natural language (NL). The knowledge encoded in these repositories is not available in other artifacts: T. Gleixner, principal maintainer of the Real-Time Linux Kernel, explained that "*the Linux kernel mailing list archives provide a huge choice of technical discussions*" [1]. Moreover, Question and Answers (Q&A) online services, such as Stack Overflow,[1] are filling "*archives with millions of entries that contribute to the body of knowledge in software development*" [2].

To obtain objective and accurate results from software repositories, data quality is of utmost importance: The extracted facts must be *relevant*, *unbiased*, and their contribution *comprehensible*.

Unstructured software data artifacts require the most care in terms of data quality, because they leave complete freedom to the authors. The first step in ensuring data quality for unstructured software data is to conduct an accurate pre-processing phase [3] to reduce irrelevant or invalid data. Assuming this phase is correctly conducted, the main problem with current approaches is that textual artifacts are treated as mere *bags of words*: a count of which terms appear and how frequently. The bag of words approach has been proven useful by Information Retrieval (IR) researchers on well formed NL documents, generated by information professionals (e.g., journalists, lawyers, and doctors) [4]. However, most unstructured software engineering artifacts are not written with the same care and *comprise NL text interleaved with other languages*. These languages are made of the structured fragments often occurring in technical discussions, such as code fragments, stack traces, and patches. When IR methods (e.g., VSM [4]) are used on multi language documents, they are less effective [5].

An analysis of artifacts containing both structured and unstructured fragments should exploit IR on relevant and well-formed NL sentences and use parsers on structured content. Although structured elements are defined through a formal grammar, separating them from NL is nontrivial because structured elements are rarely complete and well-formed in NL documents (e.g., method definitions may appear without body), thus not respecting their own grammar, and NL text snippets might respect the formal grammar accidentally (e.g., NL words are valid code identifiers), thus creating ambiguities.

This article investigates how to mine structured fragments embedded in NL artifacts. Our investigation stands on the shoulders of *island parsing* [6]: A technique to *extract* and *parse* structured data found within artifacts containing arbitrary text to extract and parse interesting structures, the "islands," from a "sea" of uninteresting strings. Our article makes the following contributions:

- An analysis of the state of the art of using island parsing to mine structured fragments in NL software artifacts (Section 3).
- iLander, an approach based on SGLR to extract structured fragments in NL-based artifacts (Section 4).
- A benchmark comprising 185 emails pertaining to three open source software system, in which we manually annotated embedded Java fragments.
- An assessment of the practical performance and effectiveness of iLander (SGLR based) in extracting Java source code fragments, using the aforementioned benchmark.
- An analysis of the limitations of SGLR, leading to the investigation of an alternative technique, based on *Parsing Expression Grammars* (PEGs) [7] (Section 5).
- PetitIsland, an island parsing framework based on PEGs, implemented using the parser framework PetitParser [8].
- An assessment of the practical performance and effectiveness of PetitIsland on the email benchmark.
- An improved benchmark, based on Rigby and Robillard dataset [9], comprising 188 Stack Overflow posts pertaining to three project tags.
- A description of the usage and effectiveness of our PetitIsland framework on the Stack Overflow benchmark and in support of external studies, for which we successfully extracted other kinds of structured fragments, such as stack traces and patches (Section 6).

Our results show that our approach based on the state of the art in island parsing (i.e., SGLR) achieves excellent results in terms of precision (99%) and recall (95%). However, the SGLR approach is not practical when used to parse fragments embedded in NL documents, because it suffers from serious performance issues, due to the precedence and filtering mechanism available in SGLR. The approach based on PEGs that we propose, instead not only shows that it achieves similar performances in terms of accuracy, but also empirically demonstrates that the performances allows its practical usage in real world scenarios.

2. Motivating example

Unstructured software data is noisy: It is not formal, it includes irrelevant data, and its content might be wrong, incomplete, or include jargon. Unstructured software data contains fragments written in languages other than NL, such as source code, execution traces, or patches. To extract *relevant* and *correct* information from unstructured data, we need to remove the unwanted data, recognize the different languages, and structure the content to enable techniques able to exploit the peculiarities of each language.

[1] http://stackoverflow.com/.

```
(1)  Alice wrote:
(2)  > On Mon 23, Bob wrote:
(3)  >> Dear list,
(4)  >> When starting up ArgoUML on my MacOS X system (Java 2)
(5)  >> it throws a NullPointerException very soon. You'll find the
(6)  >> trace below. I hope someone knows a solution. Thanks a lot!

(7)  >> Exception in thread "main" java.lang.NullPointerException
(8)  >> at
(9)  >> javax.swing.event.SwingSupport.fireChange(SwingChange.java)
(10) >> at javax.swing.AbstractAction.setEnabled(AbstractAction.java)
[...]
(11) >> at uci.uml.Main.main(Main.java:148)

(12) > I'm sorry I can't help you Bob but thanks for sharing the stack...
(13) > Alice.
(14) > --
(15) > "Beware of programmers who carry screwdrivers." --L. Brandwein

(16) Alice, I believe we must change Explorer.java to fix Bob's problem:
(17)   public void setEnclosingFig(Fig each) {
(18)     super.setEnclosingFig(each);
(19)     if (each != null || (each.getOwner() instanceof MPackage)) {
(20)       m = (MPackage) each.getOwner(); }

(21) The problem is in the condition, I attach the diff with this version:
(22) --- src/org/argouml/ui/explorer/Explorer.java (revision 14338)
(23) +++ src/org/argouml/ui/explorer/Explorer.java (working copy)
(24) @@ -147,1 +147,1 @@
[...]
(25)        super.setEnclosingFig(each);
(26)    -   if (each != null || (each.getOwner() instanceof MPackage)) {
(27)    +   if (each != null && (each.getOwner() instanceof MPackage)) {
(28)        m = (MPackage) each.getOwner(); }

(29) I hope this change is fine by you, if so, please apply it =)
(30) Cheers, Carl.
(31) -- I used to have a sig, but it took up much space so I got rid of it!
(32) -----------------------------------------------------------------
(33)  To unsubscribe, e-mail: dev-...@argouml.tigris.org
(34)  For additional commands, e-mail: dev-...@argouml.tigris.org
```

NL text   source code   patch   stack trace   noise

Fig. 1. Example development email with mixed content.

For example, let us consider the example development email in Fig. 1. Due to the variety of structured fragments included, not written in NL, if we consider the content of such a document as a single bag of words, we will obtain a motley set of flattened terms without a clear context, thus severely reducing data quality and the amount of available information.

By automatically extracting, parsing, and modeling the embedded structured data, we can support tasks such as:

- **Traceability recovery** In Fig. 1, the email is referring to several classes (e.g., `'Main'`, `'Fig'`, and `'MPackage'`), but only the class `'Explorer'` is critical to the discussion: It causes a failure and the email's author changed it to provide a solution. We realize the importance of `'Explorer'` by reading line `'16'` and by reading which file was modified in the patch (lines `'21-22'`). As part of our investigation on email archives [10], we often found this pattern: Artifacts mentioned in NL parts or headers of patches are more relevant to the discussion than artifacts mentioned in other contexts (e.g., stack traces). A traceability method based on bags of words (e.g., [5]) cannot recognize in which context references to artifacts appear. Such a method can only use the number of occurrences to assign a higher weight to certain terms [4], leading to imprecise results. A weighting based on occurrences would give the most relevance to class `'MPackage'` (mentioned five times), which is only marginal to the discussion.
- **Fact extraction.** To know the facts expressed in structured fragments (e.g., code snippets, patches, stack traces, or system logs), one can use ad-hoc parsers [11]. In Fig. 1, using a parser for patches, one recognizes that the file being modified is `'Explorer'` (lines `'22-23'`). Ad-hoc parsers can only be applied to the expected data structure, and not to mixed content, because they are not robust enough to manage unexpected data: The structured parts must be parsed specifically in the document.
- **Stop words removal.** To better characterize documents, IR research suggests to remove *stop words*, i.e., common words [4], thus assigning a higher weight to the peculiar terms of each document. This is suboptimal when applied to documents with a mixture of structured and unstructured content: One reduces the redundant information in NL parts, but also deletes information in parts with a different vocabulary (e.g., source code). For example, deleting the

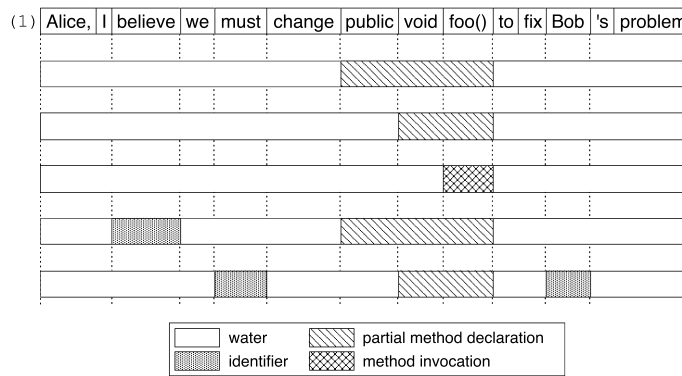


Fig. 2. A simple NL sentence with ambiguous structured fragments.

stop word `'each'` from the content of the document in Fig. 1 means also deleting a variable name in a code fragment (lines `'17-20'`) and a patch (`'25-28'`). This is suboptimal, since variable names provide relevant information [12].
- **Document summarization.** Due to the amount of data produced during software system evolution, researchers investigated how to expose only the significant parts to reduce information overload (e.g., [13]). The proposed techniques are tailored to specific types of artifact (e.g., code [14], NL documents [15]) and cannot be applied to documents that includes a mixture of languages and structures. Each technique must be applied only on specific parts of the text.

The aforementioned tasks only constitute a subset of significant problems that would benefit from a parsing approach able to analyze documents where structured and unstructured data are intertwined.

Since we strive to create an approach that is usable in real world scenarios in software engineering tasks, we define the following requirements:

- Enabling the automated and precise recognition of the structured fragments contained in the parsed document, while preserving their meaningful contextual properties;
- Be sufficiently generic and extensible to be easily reused in various application contexts, considering structured fragments of different nature, expressed in different languages and formats;
- Face the noisy nature of heterogeneous NL documents, potentially leading to ambiguous parse trees;
- Be scalable, to enable the processing of very large sets of textual artifacts in reasonable time.

In the following, we investigate the alternatives of implementing parsing of structured elements in NL artifacts by considering the above requirements.

3. State of the art

To parse structured fragments in NL language, an island grammar may reach the full complexity of context-free languages in terms of expressiveness. In literature, island parsing has been tackled with different techniques: regular expressions[2] (REs), SLGR parsing, and generalized top-down (GTD) parsing.

3.1. Island parsing concepts

Before discussing the state of the art about the use of island parsing to extract structured fragments in NL artifacts, we briefly introduce the concepts related to island parsing. Island parsing refers to the task of finding constructs of interest (the islands) among noisy information (the sea of water). It is based on *island grammars*, which define "detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water)" [6]. In our case, structured fragments (e.g., source code fragments) are the islands to be extracted, and the rest (e.g., NL sentences) is the water. Island grammars are inherently ambiguous, and the degree of ambiguity depends on the fragments that need to be extracted. Consider the sentence in Fig. 2, and consider an island grammar which extract and parses Java identifiers, partial method declarations (i.e., without body) and possible method invocations. Because of ambiguity, the number of possible parse trees is enormous. Fig. 2 shows some possible (flattened) parse trees.

First, since the method *foo* has no parameters, it can be considered as both a method declaration or an invocation with the preceding keywords considered as water. Second, a lot of words in NL are valid Java identifiers and thus they can be both

[2] Regular expressions cannot perform real *parsing* but only pattern recognition, thus we refer to approaches using REs as *island recognition* approaches.

parsed as such or as water. To filter out undesired parse trees, island parsing require parsing that support disambiguation mechanisms. Typically, such mechanisms are used to prefer complex productions over simple ones, or to identify a subset of interesting identifiers with a particular lexical structure.

3.2. Island recognition with regular expressions

In a pioneering approach [16], Murphy and Notkin proposed a lexical approach based on REs to extract models of a software system from diverse artifacts. Software engineers could obtain consistent models from any kind of textual artifacts concerning software, by: (1) defining patterns (using REs) that describe source code constructs of interest (e.g., function calls and definitions); (2) establishing the operations to be executed whenever a pattern is matched in an artifact being scanned; and (3) implementing post-processing operations for combining information from individual files into a global model.

Existing approaches for the recognition of code and other structured fragments from NL artifacts (i.e., [17,18,9]) use regular expressions to deal with the extraction, mainly for engineering simplicity and performance reasons.

Rigby and Robillard implemented an automated code element resolution tool, named ACE, to get the qualified name of code elements (in freeform text and code snippets) [9]. ACE performs three stages in the code element identification process; in the first stage they use an island recognition approach to identify code-like terms from each document. The approach "*is composed of a set of regular expressions that are approximations of* [some] *constructs in the Java Language Specification.*"

Bettenburg et al. devised INFOZILLA, a tool to recognize patches, stack traces, source code snippets, and enumerations in the textual descriptions of issue reports [17]. INFOZILLA is composed of four independent filters, one per category, which are used in sequence to process the text. Each filter is based on text matching implemented through regular expressions. The authors reported results on the effectiveness of INFOZILLA in *differentiating* documents (i.e., deciding whether they contain or not each category): It achieved excellent results in this task. The authors did not conduct further code extraction and provided little details about the island parser implementation. Subsequently INFOZILLA was used in a research investigating the features that are important when submitting bug reports [19].

We created BESC [18], an automatic method to *identify* lines of JAVA code in emails. We found that the last character is a good indicator of the nature of a line and implemented simple rules, mainly based on regular expressions. For example, we detected most JAVA lines by selecting lines ending with curly brackets or semicolons. BESC is not a full-fledged parsing approach: It loses the context between lines and cannot be used for fact extraction and modeling.

Regular expressions are not enough. REs support language structures that are less expressive than those necessary to parse programming languages. In fact, programming languages are not regular languages, and contain recursive structures (e.g., nested blocks) of context-free languages, which cannot be matched by REs. For example, with REs alone, it is impossible to correctly parse parenthesized expressions or nested blocks, which are present in almost every programming language. For this reason, REs are limited in both parsing and extraction of code as structured fragments embedded in NL artifacts.

A solid island recognition approach for code fragments immersed in a sea of NL requires—at least—the capability of recognizing context-free languages. Although the existing approaches can perform simple extraction and might appear simpler to implement with REs, the context-free structure of programming languages hardens even the simple extraction.

Let us consider the extraction of code fragments in a programming language with nesting of blocks like JAVA or C and in particular consider the nesting of conditional structures, such as if statements and loops. For example, INFOZILLA extracts only conditionals and loops followed by a block of code:

```
if (i > 0) {
  /* my code here */
}
```

However, both JAVA and C allow single statements as the body of loops and conditionals. For example, this is a valid snippet of code in JAVA:

```
for (int i = 0; i < list.len; i++)
  if (list.get(i).equals(x))
    return true;
```

In INFOZILLA such a fragment would not be extracted. An approach that includes single statements as bodies of loops and conditionals would need recursive structures, and thus a formalism that is more expressive than REs. An alternative could be to write specialized REs that reach a certain depth of nesting, but this implies a loss of generality and the use of complicated RE definitions.

Evidence of the presence of loop and conditional statements without blocks can be found in the StackOverflow posts. Considering the latest StackOverflow data dump available,[3] we counted the number of posts containing at least a conditional or loop statement, and then we calculated the amount of posts containing at least an occurrence of such statements without subsequent block. For posts about JAVA (579,270), we counted 11% (62,722) of such posts, while for C posts (186,182) we counted approximately 17% (31,753) of the posts. We conclude that, both from a theoretical and a practical point of view, approaches based on REs are prone to low precision and recall for the extraction task.

[3] http://www.clearbits.net/creators/146-stack-exchange-data-dump.

The use of REs may become problematic also from a language engineering point of view. Intuitively, it is harder to construct specific REs that capture specific subset of context-free structures, such as nested loops without blocks, because they must be manually derived from the grammar of a programming language. Instead, it would be preferable to reuse, if possible, the rules of a grammar by using an extraction mechanism that supports them.

3.3. Island parsing with SGLR and GTD

Island parsing can be approached using a full-fledged parsing technique for context-free languages; however, some parsing techniques are more appropriate for island parsing. In particular, island parsing benefits from *scannerless parsing* and the support for *ambiguous grammars*.

Scannerless parsing. Parser generators for programming languages separate the *tokenization* phase from the parsing phase. In the former phase the input code is *scanned* with REs to classify elements as keywords, identifiers, operators, or other tokens (e.g., brackets); in the latter phase the tokens are given as input to the parsing algorithm. Island grammars are ambiguous also at the token level. For example, it is impossible to determine, without its context, if the word `'public'` belongs to a NL sentence (e.g., "*This is a* public *API.*") or to a structured element like a Java class definition (e.g., `'public class Tree();'`). For this reason, the separation between tokenization and parsing in island grammars is inopportune, and may complicate the grammar structure for parsing. For this reason, island parsing approaches favor *scannerless parsing* [6].

Ambiguous grammars. Island parsing requires handling of ambiguous grammars. Few parsing algorithms support the full flexibility of context-free grammars with ambiguities, such as *Generalized LR* [20], a bottom-up parsing technique, with its scannerless version called *SGLR*, or the *Earley* [21] algorithm. Both require $O(n^3)$ time complexity in the worst case. The only distinction in performance between GLR and the Earley algorithm regards non-ambiguous grammars; GLR has the advantage, with respect to Earley, that the cubic complexity is reached only when parsing ambiguous structures, and so reaching the worst case depends on the amount of ambiguity in the grammar. Thus, concerning island parsing, the two approaches perform similarly.

A possible alternative is to use *generalized Top-Down (GTD) parsing* [22] that supports ambiguous grammars through backtracking. The disadvantage of this approach is that it results in very long parse times in the context of island parsing [23]. The most efficient implementation of such technique has been recently proposed by Frost, Hafiz and Callaghan [22], and it requires polynomial ($\Theta(n^3)$) time complexity to parse ambiguous context-free grammars. This approach is bound to perform in a similar way to *SGLR* in the context of island parsing of structured fragments in NL artifacts. Other top-down alternatives (e.g., GLL [24] that runs in linear time on LL grammars and facilitates the building and reading of grammars) are available and present similar very long parse times in the context of island parsing.

The pioneering work that uses SGLR for island parsing is Mangrove by Moonen et al. [6]. They showed how island grammars may allow the derivation of robust parsers for programs written in a particular *programming* language; it has been implemented by using the *Syntax Definition Formalism* (SDF) [25]. The approach has been used to extract and parse specific code fragments of COBOL, like conditionals, embedded in different COBOL dialects. To the best of our knowledge, no existing approach uses SGLR to perform extraction of structured fragments in NL artifacts.

An important related work that uses island parsing for a software engineering task is the work by Synytskyy et al. [26]. They describe a technique to extract fragments in different programming languages from dynamic web pages written in ASP; the approach uses GTD parsing as implemented in TXL programming language [27]. The work has no experimental evaluation; thus, it would be interesting to consider the use of top-down parsing approaches in the context of island parsing of structured fragments in NL artifacts.

Kurš et al. investigate the definition of the water production in island grammars and propose the concept of *bounded seas* to improve the robustness of island parsers and to make grammars easier to reuse and define [28].

3.4. Other related work

Bird et al. proposed an approach to measure the acceptance rate of patches submitted via email in open-source software projects [29]. They classified emails with source code patches, but provided little information about their extraction techniques and no details on the evaluation benchmark. Tang et al. addressed the issue of cleaning email data for subsequent text mining [30]. They proposed a sequence approach to clean emails in four passes: *1*) non-text filtering, *2*) paragraph, *3*) sentence, and *4*) word normalization. Dekhtyar et al. discussed challenges and opportunities related to using text mining techniques to software artifacts written in NL [31]. Basten and Klint describe DeFacto [32], a generic fact extraction technique based on the ASF[4]+SDF technology, which includes SGLR parsing. The technique consists in annotating the grammar of a language of interest with fact annotations. Based on those annotations, local facts are automatically extracted from *actual* source code by a generic fact extractor. Specific software analysis tasks may then start by further enriching the extracted elementary facts. In Basten and Klint's approach, each extracted *source code fact* corresponds to *one* particular syntax production.

[4] Algebraic Specification Formalism.

Table 1
Basic grammar notation.

Metasyntax	Description
`a → b`	production: The element `'a'` can be replaced by `'b'`
`a b`	the element `'a'` is followed by the element `'b'`.
`a?`	the element `'a'` is optional.
`a*`	the element `'a'` appears zero or more times
`a+`	the element `'a'` appears one or more times.

Dean et al. proposed *agile parsing*, to support static analysis techniques by integrating the original programming language grammar with a set of rules that support the analysis task at hand [33]. The approach uses TXL and in particular ambiguity resolution techniques similar to the ones used in island parsing. Wu et al. propose a component-based LR parsing approach (CLR) that allows to compose separate parsers for grammar components, with loose coupling, thus potentially enabling the definition of ambiguous grammars such as island ones [34].

3.5. Summary

Parsing structured fragments from NL software engineering artifacts requires techniques that support context-free structures, ambiguous grammars, and scannerless parsing. The existing approaches using REs are limited for such task. By the analysis of the state of the art, we found that SGLR is a good candidate for our application of island parsing. Hereafter, in Section 4, we analyze the applicability of SGLR, discussing the benefits and limitations from a theoretical point of view, and by implementing it we also discover the practical applicability. An alternative is the use of top-down based approaches. While GTD could be a candidate, a different top-down parsing technique, called *parsing expression grammars (PEGs)*, seems promising to explore. PEGs are a powerful formalism that is essentially recognition-oriented, and have a non-comparable expressiveness with CFGs [7]; in fact, PEGs are able to recognize typical classes of languages associated with top-down parsers (like LR(k) grammars), but they can also recognize a limited set of non-context-free languages. Because of their recognition-oriented nature, and the use of ordered choice, PEGs parse languages *in linear time* [7]. For this reason, in Section 5, we describe an alternative approach to extract and parse structured fragments which uses PEGs.

4. iLander: island parsing with SGLR and ASF+SDF

In this section, we investigate the use of SGLR to perform extraction of structured fragments from NL artifacts. Inspired by previous similar applications of island recognition for mining unstructured data, our island parsing approach targets Java code fragments.

A number of choices in the implementation of an island grammar depend on the mining task at hand and the consequent abstractions that one wants to derive from the fragmented information in a NL document. As an experimental scenario, we consider the same situation proposed by the pioneering approach by Murphy and Notkin [16]: We suppose we want an island parser able to extract models of a software system from diverse software artifacts. In other words, we consider a scenario in which we are interested in recovering structural information (e.g., classes and methods, and their definitions and relations) about the software system discussed in NL artifacts.

We devise iLander, our SGLR approach, by using the ASF+SDF Meta-Environment [25].

4.1. Notation

We adopt a EBNF-like notation, as presented in Table 1. With this notation we define, for example, the syntax of a Java method header as in the following:

```
MethodHeader → Modifier* MethodRes MethodDeclarator Throws?
```

4.2. SGLR algorithm and ASF+SDF

By using SDF, we can define context-free grammars in a modular way, thus facilitating the derivation of an island grammar from any existing programming language grammar. Within SDF, we can use SGLR, which does not impose any restrictions on the grammar. This property is essential when parsing artifacts based on island grammars, which are ambiguous by nature. For *ambiguity management*, we make use of the disambiguation constructs, such as priorities, restrictions, or preference attributes to favor one particular production when several alternatives exist. The main benefits of using SGLR algorithm and ASF+SDF for island parsing are:

- Support of context-free grammars and scannerless parsing;
- Support for ambiguous grammars, with their resolution mechanisms.

```
1  Because of problems with "argoHome" location, I imported
2  java.net.URLDecoder and added the line
3  URLDecoder.decode(argoHome); just below this one:
4  public void loadModulesFromDir(String dir) in ModuleLoader.java.
5
6  Another problem in ModuleLoader: since the cookbook explains
7  how to make a PluggableDiagram (that's exactly what I am doing,
8  so I extend this class), I have not found where the JMenuItem
9  returned by method getDiagramMenuItem() in PluggableDiagram
10 is attached in Argo menus. It seems this is not yet
11 implemented, even though PluggableDiagrams implements Diagram.
12
13 So I have added those lines:
14 void append(PluggableDiagram aModule) {
15   ProjectBrowser.TheInstance
16      .appendPluggableDiagram((PluggableDiagram)aModule); }
17
18 Such modifications must be reflected in ProjectBrowser.
```

Listing 1: Example document enclosing structured fragments.

Table 2
Considered Java island productions in iLander.

Nonterminal	Description
`CompilationUnit`	class declaration with imports
`ClassDecl`	complete class declaration
`MethodDecl`	complete method declaration
`ConstructorDecl`	complete constructor declaration
`IncompleteClassDecl`	incomplete class declaration
`IncompleteMethodDecl`	incomplete method declaration
`IncompleteConstructorDecl`	incomplete constructor declaration
`FieldDeclaration`	class field declaration
`MethodInvocation`	method invocation
`ConstructorInvocation`	constructor invocation
`IfThenStatement` `IfThenElseStatement`	conditional blocks
`TryStatement`	try/catch blocks
`WhileStatement` `ForStatement, DoStatement`	loops
`ClassRelationship`	implements/extends relations
`Block`	alone blocks

The potential disadvantages of SGLR for island parsing are:

- The worst-case time complexity of SGLR is $O(n^3)$ reached with highly ambiguous grammars: island grammars;
- The ambiguity resolution mechanism of SGLR in ASF+SDF is performed after parsing. For this reason, the parser constructs every alternative parse tree, and then applies a filter to derive the preferred ones [35].

By implementing iLander, we determine whether the SGLR algorithm can be effectively used to support our specific application of island parsing in practice. In particular, we can verify whether and how the combination of cubic time complexity and post-parsing filtering impacts the performances.

4.3. *Specifying a Java island grammar with* ASF+SDF

Common programming language grammars, such as the Java one, describe different constructs at different levels of complexity, which range from identifiers and keywords (e.g., `'private'` and `'while'`) to whole compilation units (i.e., the full definition of a Java class in its file).

In principle, we could define an island as a piece of code that can be reduced to any possible nonterminal in the grammar.

Consider the NL document in Listing 1. An interesting contained fragment could be a piece of code that can be parsed and reduced to a nonterminal `'MethodDecl'` (e.g., lines 14 to 16, Listing 1). Table 2 summarizes the nonterminals of the Java programming languages that we extract as source fragments.

Irrelevant productions. The choice of productions is related to the mining task at hand. In our scenario (i.e., recovering structural information about a system from discussions in NL artifacts), certain fragments are irrelevant. For example, isolated expressions or generic statements seldom carry relevant information in terms of methods or classes of the system. Instead, they carry structural information if, and only if, they contain specific sub-operands (as in the case of expressions) or are specific statements. Consider the sentence *"I think you should refactor the return statement, so that it returns* `getInte-`

```
(1)  On 4/12/07 Bob wrote:
(2)  [...]
(3)  > public class Bicycle {
(4)  >
(5)  >   void changeCadence(int newValue) {
(6)  >     cadence = newValue;
(7)  >   }
(8)  >
(9)  >   void changeGear(int newValue) {
(10) >     gear = newValue * 2;
(11) >   }
(12) Bob I believe you should change this method to fix the bug.
(13) > }
```

Fig. 3. Examples of incomplete class declaration with partial body.

ger()/n.". In the valid JAVA expression 'getInteger()/n', the interesting information is the method invocation. Thus, instead of choosing the nonterminal 'Expression' as a valid fragment, we choose only the subexpressions that carry information about the system structure, such as 'MethodInvocation' and 'ConstructorInvocation'.

Incomplete productions. Incomplete productions are a source of differentiation from a traditional programming language grammar. Consider line 4 in Listing 1: It explicitly references a method signature but the body is missing. The standard JAVA grammar includes only method definitions that are followed by either a semicolon (when they appear in interfaces) or the whole method body. By considering only the fragments that can be reduced to a nonterminal in the standard grammar, we lose relevant information, such as the incomplete method declaration in Listing 1. For this, we also extract incomplete information (e.g., 'MethodDecl') corresponding to a subset of a production that does not reduce to any nonterminal in the standard programming language grammar.

Considering every possible incomplete production is a source of ambiguity, which in turn would affect the performance of the fragment extractor. Incomplete productions must be selected according to the kind of code fragments that need to be extracted from the NL artifact, and according to the potential further analyses that need to be performed on the extracted fragments. Our island grammar includes incomplete productions of nonterminals representing declarations of methods, constructors, and classes. Such incomplete productions do not require a final semicolon or a block with the body of the construct. For example, the following production has been introduced to extract incomplete constructor declarations:

```
IncompleteConstructorDecl → Modifier* ConstructorDeclarator Throws?
```

By supporting incomplete productions, we can also extract entity declaration even when a body is incomplete or contains water. For example, for a class declaration with a partial body, our method extracts a single fact for the declaration, as an incomplete class declaration, and parses the partial body as if it was a sentence of the island grammar.

Let us consider the document in Fig. 3: The declaration of the class 'Bicycle' is not correct, because line 12 is not valid JAVA. In this case, we still extract a fact for the incomplete declaration of class *Bicycle* and parse the partial body as it was a sentence of the island grammar, thus extracting the two complete method declarations (i.e., 'changeCadence(int)' and 'changeGear(int)'). An alternative is to support *islands with lake* [6]. Due to engineering difficulties we have not been able to support this construct in ILANDER, using SGLR, while we successfully implemented it with PEGs (see Section 5.2).

'ClassRelationship' is another kind of incomplete fragments that we consider, for it contains structural information: It expresses relations of inheritance or implementation between classes. For example, in line 11 of Listing 1, we find two potential class names separated by the keyword 'implements'; from this, we derive that there is an implementation relation between the two entities and that 'Diagram' is an interface.

Ambiguity resolution: preferring islands. Island grammars are inherently ambiguous: Water is defined as *anything* that is not an interesting fragment (i.e., an island), thus it should be *avoided* and islands *preferred*. Moreover, certain islands are a subpart of more comprehensive ones: For example, consider the complete method declaration in Listing 1 (lines 14–16). As shown in Table 2, at the same level of 'MethodDecl', we have 'Block' and 'IncompleteMethodDecl'. Thus, the construct in the example is ambiguous: It can be parsed either as a 'MethodDecl' or as a sequence of 'IncompleteDecl' plus *Block*. The solution that reduces to a single nonterminal, which keeps the binding among the parts, should be privileged.

Language definition systems, such as SDF, that support ambiguous grammars provide specialized constructs to resolve ambiguities. To select between alternative derivations, we use the disambiguation mechanism provided by SDF, which is based on the following two keywords:

'avoid': The parser removes alternative derivations that have 'avoid' at the top node, *iff* no other derivations have 'avoid' at the top node.
'prefer': The parser removes all other derivations that do not have 'prefer' at the top node.

By using the 'avoid' keyword for any reduction to water we prefer islands. We exploit the same mechanism to give preference to some island structures over others. By using SGRL in SDF, avoiding water is concise, from an engineering perspective, but declaring precedences among islands is wordy. Let us consider the following three productions:

```
isValidSource(#IncompleteMethodDecl) = true when
   void #MethodDecl := #IncompleteMethodDecl

isValidSource(#IncompleteMethodDecl) = true when
   #PrimType #MethodDecl := #IncompleteMethodDecl

isValidSource(#IncompleteMethodDecl) = true when
   #Identifier #MethodDecl := #IncompleteMethodDecl,
   isAClassName(#Identifier) == false
```

Listing 2: “isValidSource” ASF function.

```
extractCodeFragments(#Source,#ExtractedSourceFragments)
                        = #ExtractedSourceFragments
(MethodInvocation : getLocation(#MethodDecl) :
   #Identifier()) when
  #IncompleteMethodDecl := #Source,
  #MethodRes #MethodDecl := #IncompleteMethodDecl,
  #Identifier() := #MethodDecl,
  isValidSource(#IncompleteMethodDecl) == false
```

Listing 3: “extractCodeFragment” ASF function.

```
CompilationUnit
ClassDecl
IncompleteClassDecl
```

These three productions are in decreasing order of contained information: The second should be considered only if the first is not available, and the third should only be considered if the first two are unsuccessful. To make this explicit with the `'avoid/prefer'` resolution mechanism, we have to write the following productions, which add further indirection:

```
Island → CompilationUnit prefer
Island → ClassDecl'
ClassDecl' → ClassDecl prefer
ClassDecl' → IncompleteClassDecl
```

Besides this syntactical hurdle, a critical aspect of `'avoid/prefer'` resolution mechanism is that it is performed *after parsing*: The parsing mechanism in ASF+SDF first constructs all the possible parse trees, then performs filtering according to the resolution rules. Given the high degree of ambiguity of the island grammar, always constructing all the possible parse trees could negatively impact the performances of iLander.

Ambiguity resolution: choosing water. Let us consider the fragment: ‘by method getDiagramMenuItem()’ (Listing 1, line 9). The island grammar described in the previous section would select ‘method getDiagramMenuItem()’ as a valid `'IncompleteMethodDecl'` fragment: ‘method’ is a valid identifier, and thus a lexically valid return type for the Java grammar. In this case, an element is wrongly recognized as a larger island, instead of water plus a smaller island.

To solve this problem, we exploit the naming conventions of the programming language. Java, as well as other programming languages, prescribes how to capitalize artifact names. Java naming conventions [36] prescribe that class names must start with a capital letter. The word ‘method’ violates the naming convention. We exploit this to exclude reductions to incomplete method declarations where the supposed return type is likely to be invalid. For every extracted fragment, we define an asf function (i.e., *isValidSource*) which takes a source fragment as input and returns true *iff* the fragment is valid. The base case for that function is true for any fragment. For incomplete method declarations, we define *isValidSource* to return true iff the return type is *void*, a primitive type, or an identifier that respects to naming conventions. With the asf syntax, the rule is declared with three different cases, as in Listing 2.

Similar definitions are done for potential constructor declarations, which must respect valid naming conventions.

In the previous example, the ‘method getDiagramMenuItem()’ is not a valid method declaration, but it can be restructured as a method invocation fragment ‘getDiagramMenuItem()’. We define an ASF transformation rule to translate what was parsed as a method declaration to a method invocation (Listing 3).

The rule takes a fragment parsed as an incomplete method declaration violating the naming conventions, extracts the identifier corresponding to the method name, and produces a source fragment of type `'MethodInvocation'`. Like in the case of avoid/prefer disambiguation mechanism, the ASF transformation rules are applied post-parsing.

4.4. Empirical evaluation

We evaluate iLander by performing a case study with NL artifacts pertaining to real world software systems.

Subject systems. To allow the replicability of our experiment, we focus on open source software systems, whose data is fully available in public repositories. To improve generalizability, we picked systems from different domains that are

Table 3
Systems considered in the evaluation.

System	Website	Emails		
		Mailing list	With code	Sample
ArgoUML	argouml.tigris.org	24,876	12%	50
Freenet	freenetproject.org	22,095	9%	39
Mina	mina.apache.org	12,869	29%	99

developed by distinct free software communities. Among these projects, the developers are likely to use different development styles, as well as to make different use of the related NL artifacts. We consider the three open source systems (OSS) developed in Java depicted in Table 3.

Subject NL artifacts. A rich archive of NL artifacts in open source systems are development mailing lists. We focus on this kind of artifact not only for the availability of a large number of data points, but also because email data is noisy: It can contain extra line breaks, extra spaces, and special character tokens; it can contain spaces and periods mistakenly removed, words misspelled, badly cased or non-cased [37]. This makes mailing list data a veritable acid test to evaluate our approach.

In the 3rd column of Table 3, we report the size of the development mailing lists of the considered systems, after removing automatically generated emails. We randomly chose a statistically significant sample of emails *with code* to inspect. Based on our previous work on classifying emails and lines containing source code (i.e., Besc [18]), we know the proportion of messages with code in the chosen mailing lists (reported in the 4th column of Table 3). We use this information to calculate the size of significant samples [38]. The 5th column reports the number of emails from which we randomly selected samples.

Evaluation metrics. To analyze the effectiveness of our extraction technique from two perspectives, we measure the capacity of our method in locating the chosen structured elements and assess whether the grammar production assigned to the fragments is correct. We *precision* and *recall* [4]:

$$Precision = \frac{|TP|}{|TP + FP|} \quad Recall = \frac{|TP|}{|TP + FN|}$$

In the formulas, TP (true positives) are code fragments correctly extracted, FN (false negatives) are correct code fragments not extracted, and FP (false positives) are code fragments incorrectly extracted.

We can describe *precision* as the fraction of the extracted fragments that are correct, and *recall* as the fraction on the total number of correct fragments.

Text normalization. Before manually creating a benchmark with all the emails in our samples, and computing results with iLander, we processed emails to normalize the text: (1) We remove the email metadata and the occurrences of the characters (i.e., $>$) used to mark different quotation levels; (2) we normalize patches by removing the lines marked as deleted (to avoid recovering what is explicitly no longer valid) and the + signs at the beginning of the lines; and (3) we normalize stack traces removing incorrect line breaks by using a regular expression.

Tackling performance issues with content splitting. The text of emails contains a high number of ambiguities, generated by NL and incomplete or scattered fragments. As a result, we found that iLander requires up to hours (using a eight core server with 40 GB of RAM) to parse emails longer than 150 lines. Since we would like to use iLander in real world scenarios (i.e., handle tens of thousands of documents in a reasonable time), we try to reduce ambiguities—and thus the parsing time—by creating an heuristic to split the email in self-contained blocks.

First, the splitting process divides the body in multiple parts according to the quotation levels: Each time we encounter a change in the quotation level we create a new split, preserving the correct order. Different quotation levels are already separate blocks, which respect the intention of the email's author, thus they are not disruptive with respect to the meaning of the parts. We analyze each split, and apply further splitting techniques when it is longer than 50 lines:

- *Code patches* Starting from the beginning of the split, when we encounter the header of a code patch, we create a new split. Even when they are referring to the same file, blocks of patches are independent, and can be treated separately by the parser, without losing information.
- *Stack trace lines* Starting from line 50 (to avoid the generation of too many splits), we split when we encounter a stack trace line. These lines can be analyzed separately without losing any contextual information.
- *Natural language lines:* If we reach line 80 without having split before, we try to find lines with only NL, because they can be separated without breaking the structure of any code fragment. To recognize NL lines, we use a technique we previously devised [18], by which we can determine, with a considerable precision, whether a line is code. Since the method does not assure perfection, we split when it finds four consecutive non-code lines.
- *Forced splitting:* If we reach line 150 without any split, we divide the content as soon as we encounter a line not containing an open parenthesis, or curly brackets. This forced splitting can alter the context, but, in practice, is applied to a very small fraction of emails.

Table 4
Systems' mailing lists and results.

System	Results	
	Precision	Recall
ArgoUML	99%	95%
Freenet	99%	95%
Mina	99%	94%

```
1  ActionOpenProject.getInst().openProject(ur
2  l);
```

Listing 4: Truncated code fragment.

With the splitting, we managed to reduce the parsing time of emails from hours to several minutes. Nevertheless, in practice iLander took eight hours to process our benchmark of 183 emails.

Results. After the text normalization phase, we manually inspected each email and labeled all structured fragments with the correct grammar production. Conducting the normalization phase before the annotation allowed us to verify the normalization process. Afterward, we manually inspected and compared the output of iLander to the benchmark. Table 4 shows the results obtained by the approach applied to the 183 emails we manually labeled.

iLander achieved very high precision on the complete dataset for all three systems, with almost no NL words classified as parts of structured fragment. The recall values are also considerably high, thus the method recognized almost all the required fragments. We found no error in the grammar productions reported in the recognized fragments.

By manually inspecting the entire output of our technique, even though severely time-consuming, we gained a qualitative knowledge of the cause of the few errors generated. The majority of the false negatives (which affect the recall) were caused by the irregular text of emails. In particular, a few code fragments were truncated in the middle of an identifier, thus hindering a correct complete identification (although the surroundings were correctly recognized).

In Listing 4, the `'url'` of the invocation `'openProject'` is truncated.

iLander outputs a false negative only recognizing the first method invocation `'ActionOpenProject.getInst()'`. Examples of false positives are the strings *Java(TM)* and *developer(s)* wrongly reported as method invocations. These rare cases can be removed through post-processing, e.g., with statistical parsing.

Summary. The SGLR-based implementation achieved high precision and recall when applied to emails coming from real-world projects. However, the approach showed profound practical limitations. It is hard to implement for non-language engineers: Despite previous knowledge of one of the authors with ASF+SDF, the implementation required more than 400 researcher-hour. Moreover, the approach does not scale to long documents, probably due to the characteristics of SGLR and post-parsing ambiguity resolution. As a result, we had to split longer emails into chunks.

5. PetitIsland: island parsing with PEGs

PetitIsland is our approach to island parsing of structured fragments in NL based on PEGs. We implemented it with PetitParser, a parser framework that supports scannerless parsing, parser combinators, and PEGs [8]. We adopted the Smalltalk version due to its convenient syntax.[5]

5.1. *PEGs and* PetitParser *notation*

Instead of expressing a language in a generative manner, like context-free grammars, PEGs avoid expressing equivalent nondeterministic choices between alternative productions. They use an *ordered* choice operator (usually denoted with `'/'`), which lists each alternative in a prioritized order. As a result, in a PEG, the first of the alternatives that successfully matches a given text is chosen. For example, the following PEG parses arithmetic expressions with sums and products, giving products priority without the need for further disambiguation.

```
Term →  Prod '+' Term / Prod
Prod →   Number '*' Prod / Number
```

In PetitParser, grammars are specified by means of parser objects (Table 5) that are composed into other parsers using parser combinators (Table 6[6]).

For example, a parser for an identifier—in the form of a letter followed by zero or more letters or digits—can be implemented as follows (we also provide a EBNF-like translation of each production):

[5] http://scg.unibe.ch/research/helvetia/petitparser.
[6] The entire list can be found in www.lukas-renggli.ch/blog/petitparser-1.

Table 5
Terminal Parsers in PetitParser, plus EBNF-like metasyntax.

Terminal	Metasyntax	Parser created
`$a asParser`	`'a'`	Parser for the character `'a'`.
`'abc' asParser`	`"abc"`	Parser for the string `'abc'`.
`#any asParser`	`.`	Parser for any character, new line and spaces included.
`#digit asParser`	`#d`	Parser for the digits `'0..9'`.
`#letter asParser`	`#l`	Parser for the letters `'a..z'` and `'A..Z'`.
`#uppercase asParser`	`#L`	Parser for the letters `'A..Z'`.

Table 6
Parser Combinators in PetitParser, plus EBNF-like metasyntax.

Combinator	Metasyntax	Parser created
`p1 , p2`	`p1 p2`	Parser that combines parser `'p1'` followed by `'p2'`.
`p1 / p2`	`p1 / p2`	Parser that tries parser `'p1'`, *iff* that fails uses `'p2'`.
`p star`	`p*`	Parser that uses parser `'p'` zero or more times.
`p plus`	`p+`	Parser that uses parser `'p'` one or more times.
`p not`	`^p`	Parser that uses parser `'p'` and succeed when it fails, but does not consume its input.
`p end`	`p$`	Parser that uses parser `'p'` and succeeds at the input end.

```
identifier := #letter asParser ,
         (#letter asParser / #digit asParser) star.
```

EBNF-like: `identifier → #l (#l / #d)*`

The expressions `'#letter asParser'` and `'#digit asParser'` return parsers that accept a single character of the respective character class; the `','` operator combines two parsers into a sequence parser; the `'/'` operator combines two parsers into an *ordered choice* parser, and the `'star'` operator accepts zero or more instances of this *ordered choice* parser.

By subclassing the PetitParser class that implements composite parsers (i.e., `'PPCompositeParser'`), grammars can be defined as parts of a class. Each production is implemented with an instance variable and a method returning the grammar of the rule. For example, we can re-define our parser for identifiers through the following class `'PPIdentifier'`[7]:

```
PPCompositeParser subclass: #PPIdentifier
  instanceVariables: 'validCharacters'
```

Beginning with the mandatory method `'start'`, which specifies the starting production, the class `'PPIdentifier'` declares the following methods:

```
PPIdentifier>>start
  ^#letter asParser , validCharacters
```

`start → #l validCharacters`

```
PPIdentifier>>validCharacters
  ^(#letter asParser / #digit asParser) star
```

`validCharacters → (#l / #d)*`

We expanded the identifier production into two productions to highlight how instance variables and methods are used. The result is a parser made of a graph of connected parser objects, which can be used to parse input text:

```
parser := PPIdentifier new.
parser parse: 'ex1'.     This returns an abstract syntax tree.
parser parse: '2ex'.     This returns a parse failure.
```

PEGs parsing is performed using *packrat parsing* [39]. Packrat parsing provides the same power and flexibility of top-down parsing with backtracking, but instead of requiring super-linear time complexity, it exploits memoization to guarantee linear parse time complexity. While memoization is inherently space-intensive, this is generally not an issue on modern machines [39,7].

[7] See www.lukas-renggli.ch/blog/petitparser-2 for more detailed examples.

Advantages and drawbacks. The main benefits of using PEGs, packrat parsing, and PetitParser for island parsing are:

- The support for parsing with linear time complexity;
- The support for scannerless parsing;
- The support for parser combinators, which helps better reuse of parsers expressed with the PetitParser notation.

We previously discussed how PEGs have an expressive power that is not comparable with context-free grammars; the potential disadvantages are:

- Reuse of typical grammar structures can be harder because some productions need to be rewritten, e.g., left-recursive rules cannot be directly implemented into PEGs. However, EBNF syntax is supported by PetitParser;
- PEG does not produce all ambiguous parse trees as it only generates the first one that is correct, based on the specified ordering. This implies that the decision on which production should be consider cannot take into account the result of other high-level productions found in the document.
- Memoization required by Packrat parsing could negatively affect the performances of PetitIsland.

Similarly, to iLander, by actually implementing PetitIsland, we determine how these theoretical advantages and drawbacks affect the approach to verify whether it can be used effectively to support our specific application of island parsing in practice.

5.2. Specifying island grammars with PetitIsland

Parsing combinators and PEGs allow us to concisely write and reuse parsers. From an engineering perspective, especially for non-experts of language engineering, this is a considerable advantage over SGLR and ASF+SDF. With parsing combinators and PEGs, we could define the basis of our entire island parsing approach in four productions. By using PetitParser, we implemented this in the class `'PPIsland'`:

```
PPCompositeParser subclass: #PPIsland
  instanceVariables: 'island water waterBlob'
```

We define the first production in the `'start'` method:

```
PPIsland>>start
  ^(island / water) plus end
```

```
start → (island / water)+$
```

The `'start'` production builds on the ordered choice provided by PEG (Section 5.1): We specify that the `'island'` production has precedence over the `'water'` one (i.e., `'island / water'`). Moreover, we set (using `'plus'`) that the text might contain one or more occurrences of `'island'` and/or `'water'` and must be parsed to the end (using `'end'`).

The second production we define is the `'island:'` method:

```
PPIsland>>island: aParser
  island := aParser
```

In this method, the `'island'` production must be declared externally and passed to the `'PPIsland'` parser as an argument (usually done when the `'PPIsland'` class is instantiated). Thanks to parsing combinators, we can leave the definition of island(s) external and have an approach to island parsing that is reusable out-of-the-box, under any definition of `'island'`.

The third and fourth productions regard the water:

```
PPIsland>>water
  ^waterBlob / #any asParser
```

```
water → (waterBlob / .)
```

```
PPIsland>>waterBlob
  ^(#letter asParser / #digit asParser) plus
```

```
waterBlob → (#l / #d)+
```

The most conservative solution to define `'water'` would use the expression `'#any asParser'`, which consumes one single character of any kind (see Table 5). In practice, with this approach, the `'start'` production first tries to match an `'island'`, then, in case of failure, it matches and consumes any character (i.e., `'water'`, defined as `'#any asParser'`), and it starts again from the subsequent character. Structured fragments do not start in the middle of a word, so we also

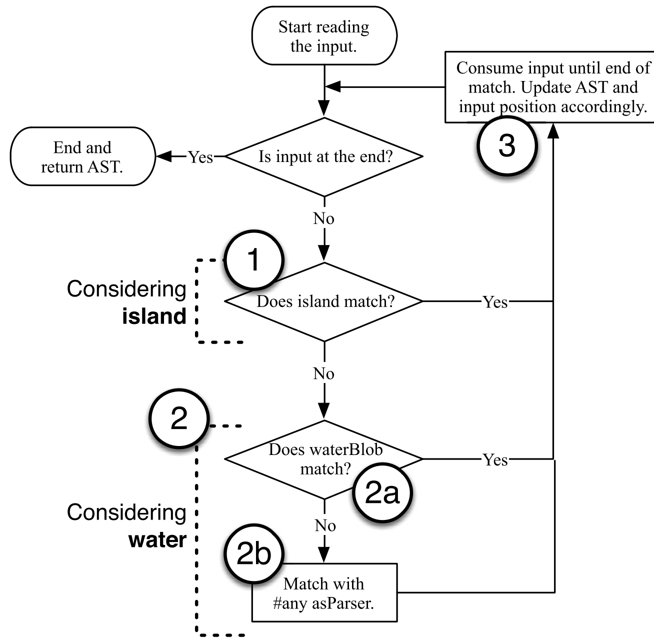


Fig. 4. The parsing process, showing the role of precedences.

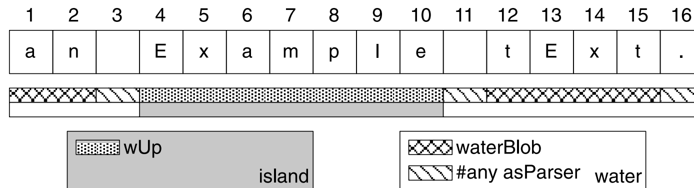


Fig. 5. Parts matched with a small island parser.

defined `waterBlob`, i.e., a production that consumes an entire word instead of a single character (also speeding up the parsing).

To illustrate our island parsing approach, we present an example in which we plug a small parser into it. Let us assume we want to create a parser to recognize, within an arbitrary text, all the occurrences of words starting with an uppercase letter. First, we write a parser for the content we want to extract (i.e., words starting with uppercase letters):

```
wUp := #uppercase asParser, #letter asParser star
```

$$\texttt{wUp} \rightarrow \texttt{(\#L / \#l)*}$$

We plug this parser into a new instance of `PPIsland`, creating a customized island parser, to parse an example text:

```
islandParser := PPIsland newWithIsland: wUp.
islandParser parse: 'an Example tExt.'.
```

Fig. 4 depicts the parsing process, underlining the flow and precedences of the parsers. Fig. 5 depicts the final result.

The `islandParser` starts from position 1 (Fig. 5) containing the first character `a`. According to the `start` production, which specifies the main precedence, `islandParser` tries to find a match with the `island` parser (Point 1, Fig. 4). The `island` parser corresponds to the plugged `wUp` parser. Since `wUp` fails, the `islandParser` does not consume the input and rolls back to the `water` parser (Point 2). The `water` parser tries to match with `waterBlob` (Point 2a). From position 1, `waterBlob` successfully matches (and consumes the input, Point 3) up to position 2 included (Fig. 5); it also updates the resulting AST accordingly. The `islandParser` continues from position 3 trying to match other islands and water (due to the `plus` in the `start` rule), always giving precedence to the former. From position 3 the only parser that matches is the `#any asParser` (thus reaching Point 2b in Fig. 4), so a single character is consumed. From position 4, the `wUp` parser matches the input and consumes it up to position 11. The process goes on similarly until it reaches the end of input (due to the `end` in the `start` rule).

By using a transformation or subclassing `PPIsland`, it is possible to ignore water and return an AST with only islands.

```
(1) Here you find a code example to answer your question:
(2)  public void setEnclosingFig(Fig each) {
(3)   ...
(4)   if (each != null || (each.getOwner() instanceof MPackage)) {
(5)     m = (MNamespace) each.getOwner(); }
(6)   ...
(7)  }
(8) The key point is the condition. [...]
```

Fig. 6. Structured fragment embedding water and islands: Island with lakes.

Islands with lakes: In certain documents (e.g., emails) structured fragments often embed parts extraneous to their grammar. In Fig. 6 the lines 2–7 contain a method declaration in which parts of the content are omitted and replaced with ellipses (lines 3 and 6). These fragments are named *islands with lakes* [6], because the islands embed water.

We subclassed `'PPIsland'` with `'PPIslandWithLakes'` and overrode the `'start'` method, to support such constructs:

```
PPIsland subclass: #PPIslandWithLakes
  instanceVariables: 'startParser stopParser'

PPIsland>>start
  ^startParser ,
     (island / (stopParser not , water) ) star , stopParser
```

`start → startParser (island/(^stopParser water))* stopParser`

This parser also requires `'startParser'` and `'stopParser'` to define the boundaries. Again, any definition of island can be plugged. After the starting boundary is found (i.e., `'startParsers'` matches), the parser tries to match zero or more islands and lakes (i.e., `'(island / (stopParser not , water) ) star'`). When no island is matched, `'PPIslandWithLakes'` tries to match with `'water'`, but only if it does not find a `'stopParser'`. Whenever it finds something matched by the `'stopParser'`, `'PPIslandWithLakes'` stops.

To clarify the functioning of our island with lake parsing approach, we show how we can define a parser for recognizing the structured fragment in Fig. 6. If we already defined a parser able to recognize the header of a method declaration (e.g., the one in line 2 in Fig. 6), we would define the parser for the method declaration body, and combine them:

```
methodDeclHeader := [ definition of parser ]
methodDeclBody := PPIslandWithLake
    newWithIsland: blockStatement
    start: ${ asParser
    stop: $} asParser.
methodDecl := methodDeclHeader , methodDeclBody.
```

We embed the `'methodDecl'` parser into a new instance of island parser (as previously done with the `'wUp'` parser):

```
islandParser := PPIsland newWithIsland: methodDecl.
```

This `'islandParser'` is able to correctly parse the example in Fig. 6, thus extracting the method declaration and its non-water content.

5.3. *Specifying a Java island grammar with* PetitIsland

We present an approach based on PetitIsland to parse target Java fragments. We consider the same experimental scenario used for iLander, in order to compare their results: We suppose we want an island parser able to extract models of a Java software system from development emails.

Starting from the Java Language Specification [40], we created a class `'PPJavaSyntax'` by extending the PetitParser class `'PPCompositeParser'` and implemented each production in the language specification with an instance variable and a method returning the grammar of the production (see Section 5.1 for details). The authors of the language specification described many productions with left-recursive rules, as in the following:

```
TypeDeclarations →
  TypeDeclarations TypeDeclaration |
  TypeDeclaration
```

Since left-recursive rules cannot be directly implemented into PEGs [7], we re-wrote the rules to avoid left-recursion. The Java grammar has a starting production, which we implemented in `'PPJavaSyntax'` through the following method:

Table 7
Considered JAVA productions, in priority order.

Nonterminal	Description
`compilationUnit`	class declaration with imports
`packageDecl`	package declaration
`importDecl`	import declaration
`typeDecl`	class or interface declaration
`methodDecl`	method declaration
`incompleteTypeDecl`	incomplete class or interface declaration
`incompleteMethodDecl`	incomplete method declaration
`strictFieldDecl`	"strict" field declaration
`creatorWithOptSemicolon`	constructor invocation with optional semicolon
`assertStatement`	assertion predicate
`ifStatement` `switchStatement`	conditional blocks

`forStatement`
`whileStatement`
`doStatement`
	loops

`breakStatement` `continueStatement`	execution statements
`tryStatement` `throwStatement`	exception statements
`synchronizedStatement`	mutual exclusion statements
`returnStatement`	method return statement
`classRelationship`	implements or extends relations
`strictVariableDecl`	"strict" variable declaration
`strictExpressionStatement`	"strict" expression statement
`strictMethodInvocation`	"strict" method invocation
`strictAnnotation`	"strict" annotation

```
PPJavaSyntax>>compilationUnit
  ^(annotations optional , packageDecl) optional , importDecl star , typeDecl plus
```

```
compilationUnit → (annotations? packageDecl)?
                      importDecl* typeDecl+
```

This can be plugged into an instance of `'PPIsland'` by defining it as the `'island'` (see Section 5.2). In this way, however, only this production would be recognized within NL documents, while many others that often appear (e.g., method invocations or declarations, if or do statements) would be lost. For this reason, we defined a catalogue of productions, listed in Table 7, that we want to recognize regardless of their surrounding context of NL sentences.

Some of the grammar productions (Table 7) are directly translated to our approach from the JAVA language specification. For example, this is the case for conditional blocks (e.g., `'ifStatement'`), loops, and execution or exception statements. Other productions, described in the following, are derived or inspired from the original ones specifying correct JAVA syntax and from other programming customs, e.g., naming conventions. Such novel productions are needed to support island parsing and are a source of differentiation from traditional programming language grammars; they are needed, e.g., to parse incomplete fragments or island with lakes. We implemented a JAVA PEG grammar for island parsing in a novel class: `'PPJavaIsland'`, which subclasses `'PPJavaSyntax'`, thus, we only had to implement the changed and new productions.

We defined the productions, shown in Table 7, in the new method `'islands'`:

```
PPJavaIsland>>islands
  ^(compilationUnit / packageDecl / importDecl / [... continues with all the productions in Table 7, in order.])
```

```
islands → compilationUnit / packageDecl / importDecl / [...]
```

Then, we defined the island parser by plugging the parser for the consider productions into a new `'PPIsland'` instance:

```
javaProductions := PPJavaIsland new.
islandParser :=
  PPIsland newWithIsland: (javaProductions islands)
```

Irrelevant productions. The choice of irrelevant products depends on the task. Since we are addressing the same task of iLANDER, we discard the same productions (e.g., isolated expressions or generic statements) at the top level.

Incomplete productions. Incomplete productions are also analogous to those presented for iLANDER. As an additional example, we consider lines 2 and 3 in Fig. 7. By considering only the fragments that can be reduced to a nonterminal in the standard grammar, we might lose several structured fragments, such as method signatures not followed by a body or a semicolon. For this reason, we also extract fragments corresponding to a subset of a production that does not reduce to a nonterminal in the standard language specification.

```
(1)  […] how I solved it. WallSetter implements AsyncTask, you
(2)  should not forget to implement the method protected void
(3)  onPostExecute(String result).
(4)  Your other class, instead, should look like this:
(5)  public class Square extends Shape {
(6)    private length = 5;
(7)    public Square(){...}
(8)    ...
(9)    public double area(){ return length * length; }
(10) }
(11) So that your last class can call the method area() to get it.
```

Fig. 7. Example text with various source code fragments.

```
(1) This is the class implementing the Fibonacci algorithm:
(2)  package com.stackoverflow;
(3)  import java.io.*;
(4)  class Fibonacci {
[...]
(5)  }
(6) Please note that this is a recursive solution.
```

Fig. 8. Example text with a compilation unit.

Although every possible incomplete production can be supported, for this validation we limit ourselves to the most popular incomplete productions found in NL documents: incomplete method and type declarations. Such incomplete productions do not require a final semicolon or a block with the body of the construct. For example, we implemented the following production in `PPJavaIsland` to support incomplete method declarations:

```
PPJavaIsland>>incompleteMethodDecl
  ^methodModifiers optional, typeParameters optional , (voidType / ncrType) optional, identifier ,
     ncrStrictFormalParameters , emptySquaredParenthesis star , throws optional
```

```
incompleteMethodDecl → methodModifiers? typeParameters? (voidType / ncrType)? identifier
                ncrStrictFormalParameters emptySquaredParenthesis* throws?
```

We also consider class relationships as they contain interesting information to be extracted. They express inheritance and implementation relations between classes and interfaces. For example, in line `1` of Fig. 7, we find two potential class names separated by the keyword `implements`. From this fragment, we could derive that `AsyncTask` is an interface, and there is an implementation relation between the two entities.

We recognize these fragments as follows:

```
PPJavaIsland>>classRelationship
  ^ncrStrictIdentifier, (extends / implements), ncrStrictIdentifier
```

```
classRelationship → ncrStrictIdentifier (extends / implements)
                          ncrStrictIdentifier
```

Islands with lakes. Some of the considered productions, when appearing in arbitrary documents, might contain embedded water. This mainly affects productions with a body, such as the one in declarations and loops, that includes multiple statements. Fig. 7 shows an example of a class declaration (lines 5 to 10) and a method declaration (line 7) that contain water. To correctly support these cases, in `PPJavaIsland`, we overrode the methods defining body productions; for example, this is what the production for a `block` (used, for example, by loops) looks like to support embedded water:

```
PPJavaIsland>>block
  ^PPIslandWithLake newWithIsland: blockStatement
    start: ${ asParser stop: $} asParser.
```

Ambiguity resolution: ordering islands. Exploiting the PEG ordered choice, we can order the considered productions from the most comprehensive down, before plugging them into a new instance of `PPIsland`. In this way, we do not lose the binding among the parts in case of larger productions. For example, consider the fragment in Fig. 8. In this case, by first trying to match a `compilationUnit` (which also includes the optional `packageDecl`), we realize that the `Fibonacci` class is defined in the `com.stackoverflow` package. If we tried to match first the single `packageDecl`, we would have lost the connection between the package and the class declaration.

Table 8
Results with PetitIsland implementation, by system.

System	Results	
	Precision	Recall
ArgoUML	96%	96%
Freenet	97%	95%
Mina	97%	94%

Ambiguity resolution: naming conventions. The method named `'incompleteMethodDecl'` includes nonterminals that start with the prefix `'ncr'` and are extraneous to the official grammar (e.g., `'ncrType'`). The acronym `'ncr'` stands for *naming convention respectful*; the corresponding productions are included to reduce some of the ambiguities that might arise from parsing Java fragments in NL documents, by exploiting Java naming conventions [36]. This is the analog of the approach used in iLander, but has the advantage (thanks to PEGs) that solutions that are not valid at parsing time are immediately discarded instead of being post-processed at filtering time.

We also defined some productions whose name starts with, or includes, the prefix *strict* (e.g., `'strictAnnotation'` or `'ncrStrictIdentifier'`). In these cases, we exploit the scannerless parsing approach for disambiguation.

Consider, for example, the fragment 'I solved it. WallSetter implements AsyncTask' (line 1 in Fig. 7). By using a standard `'identifier'` (or even a `'ncrIdentifier'`) the `'classRelationship'` parser would match: 'it. WallSetter implements AsyncTask', thus recognizing `'it. WallSetter'` as an identifier. In fact, the Java grammar allows identifier with qualifiers separated by spaces. However, this is not recommended by the naming conventions. Since we do not have a separate tokenization phase, as we rely on scannerless parsing, we can specify that certain productions require a more "strict" tokenization, by not allowing whitespace between certain parts. For example, `'ncrStrictIdentifier'` does not allow whitespace in a type name, and `'strictAnnotation'` does not allow whitespace between the `'@'` and the subsequent identifier.

5.4. Empirical evaluation

We evaluated our Java island grammar implementation with PetitIsland using the same benchmark that we used to evaluate iLander. We used the same pre-processing normalization (Section 4.4), but it has been not necessary to split the content of the emails to reduce the ambiguities. In fact, even without reducing the length of emails, our PEGs based approach could parse the entire benchmark in less than two minutes (using a 4 core laptop with 8 GB of RAM).

Automated comparison. Since the text normalization phase and the quality of the benchmark were already assessed in the manual evaluation of SGLR, to improve the replicability of the current evaluation, we adopted an automated approach. We wrote a script to automatically compare what we extracted to what was labeled as code in the benchmark. The automatic comparison we set up is very strict. Consider the code fragment in Fig. 8. The expected outcome is a single code fragment, which corresponds to a compilation unit. If our approach did not recognize this as a single piece, but as two or more pieces (e.g., a `'packageDecl'`, followed by an `'importDecl'`, plus a `'typeDecl'`), we would have counted one FN and as many FPs as the separated fragments proposed (e.g., three). Also in the case of partial extractions, we count an incomplete fragment as both a FP and a FN. For example, if we extracted `'onPostExecute(String result)'` from Fig. 7 instead of `'protected void onPostExecute(String result)'`, we would have counted a FN (for the missed fragment) plus a FP (for the partially wrong extraction). Table 8 reports the results achieved in terms of precision and recall.

Summary. The implementation of PEG-based island parsing of Java fragments from NL artifacts achieved similar results to the SGLR-based approach, in terms of precision and recall, when applied to the same dataset of emails pertaining to real-world projects. As opposed to the SGLR-based approach, PetitIsland required a shorter implementation time, corresponding to 120 researcher-hour, despite all the authors being newcomers to the used PEG framework. Moreover, PetitIsland did not show practical time limitations. In fact, due to the characteristic of PEG and parsing time ambiguity resolution, the approach scales to long documents, without incurring in memory problems that could have been generated by the memorization technique. In particular, even though we did not split emails into chunks (as we had to do with SGLR), the PEG based approach took minutes to parse the input, instead of the hours necessary to the SGLR one.

6. PetitIsland: further evaluation

We tested our approach on three scenarios: (1) technical discussions; (2) classification of the lines of development emails, (3) higher level code parsing.

Table 9
Dataset description and results, by project tag.

Project tag	Answers	Fragments	Results		
			FP	FN	Precision-recall
Android	63	120	2	5	98%–96%
Hibernate	51	68	1	2	98%–97%
HttpClient	74	163	3	6	98%–96%

6.1. Stack overflow

To validate our approach to recognize Java code fragments in NL artifacts, we test it on Stack Overflow,[8] a web service where developers exchange knowledge in the form of questions and answers. Post authors can include fragments of source code and tag them, so that they appear formatted appropriately.

The advantage of using Stack Overflow data is that it is already code tagged by external people not involved with the evaluation, and that it corresponds to another real-world scenario for applying our approach. Moreover, the sample of questions considered in the benchmark is the same used by Rigby and Robillard [9], thus having the advantage that they previously pre-processed and verified it. The dataset is composed of 188 answers, taken from posts with project tags related to three Java applications: HttpClient, Hibernate, and Android.[9] The three project tags crosscut a diverse set of topics. We downloaded the dataset kindly provided by Rigby and Robillard[10] and adapted it to our task: We re-inspected all the 188 answers and fixed any incorrect tag. Incorrect tags regarded not tagged named code entities (e.g., 'ConstraintViolationException') in most cases (43 occurrences throughout all the documents) and untagged inline code (e.g., 'runOnUiThread()') in a few others (5 occurrences); in these cases we added the missing code tag. In a few other cases (4 occurrences), we removed the code tag to parts of sentences that were tagged by the sentence's author as a way to highlight a term.

With the benchmark in place, we applied our approach to island parsing to the raw text and used the same comparison approach used to evaluate PetitIsland on the email benchmark. Table 9 reports the results.

Error inspection. We manually inspected the errors generated by our approach to understand their causes and whether they could be addressed. Most errors were due to ambiguities that cannot be resolved without a deeper understanding of the meaning of the text. For example, in the sentence "*A new openConnection() method has been added*", our parser recognized a constructor invocation: `'new openConnection()'`. This error could only be avoided knowing that `'new'` was part of the discourse, rather than a valid fragment of code. Fixing this error with a lexical parsing approach like ours is possible by either excluding similar cases from the available productions or devising stricter rules for recognizing them. Both these solutions would fix such a false positive, but they could introduce new false negatives, with the final result of only rebalancing the trade-off between precision and recall.

Summary. We tested our approach to island parsing by extracting source code fragments from Stack Overflow posts. This task is useful for a number of applications, such as mining API usages [41], improving traceability methods [42,9], or extracting diverse models of a software systems [16]. Concerning accuracy performance, results show that our approach accomplishes the required task with a very low number of errors, in terms of both precision and recall. Moreover, concerning time performance, our approach confirmed the positive results of the previous evaluation: It took a computation time of less than one minute to parse 188 documents on a 4-core CPU with 8 GB of RAM.

6.2. Email content classification

Fig. 1 shows an example development email that embeds three different types of structured content: source code, patches, and stack traces. Since these "languages," together with noise, are very common to development emails, we decided to create a classification technique to distinguish these different parts [11]. In particular, we classify the lines of development emails in five categories: NL, noise,[11] source code, patches, and stack traces. Given our technique, researchers, before exploiting development email data, can apply a pre-processing phase to recognize the different parts, so that they can conduct the subsequent data analysis with the most appropriate method for each part.

Since the categories to recognize are not only made of structured content (i.e., we also classify NL and noise), we decided to use an hybrid approach: Exploiting island parsing to detect structured parts, and using machine learning to recognize NL and noise, and to merge the results. In this article, we focus on the implementation details concerning the island parsers that we created to recognize the structured parts. We have three parsers: source code, patches, and stack traces. We evaluated them on 1,493 emails taken from the development mailing list of four open-source software systems (namely ArgoUML,[12]

[8] http://stackoverflow.com/.
[9] http://hc.apache.org, http://www.hibernate.org, http://developer.android.com/about/index.html.
[10] http://swevo.cs.mcgill.ca/icse2013rr.
[11] We consider as noise the user signatures and mail headers, for example.
[12] http://argouml.tigris.org/.

Table 10
Email line classification results, by parser and line type.

Parser	Lines	TP	FP	Precision-recall
PPStackTrace	1,069	1,054	4	99%–99%
PPPatch	2,082	1,996	0	100%–96%
PPJavaIsland_v0.8	1,914	1,715	74	96%–90%

Freenet,[13] Apache JMeter,[14] and Apache Mina[15]), totaling 67,792 email lines. We decided to focus on *line* classification, instead of token classification, because *hybrid* lines (lines belonging to more than one category) account for less than 5% of the population in our sample set, and line classification let us use simpler, yet effective, heuristics to complement the island parsing approach. To mitigate the bias in the experiment, we include hybrid lines as separated instances.

The source code island parser is a preliminary and less refined version of the one presented in Section 6.1, while the remaining two, i.e., patch and stack trace, were created ad-hoc. Each of these parsers is a pluggable extension of PetitIsland, thanks to the usage of PEGs and parser combinators.

Stack trace island parsing. We first define some terminology to refer to the various parts of the structure of stack traces. Let us consider Fig. 1:

- The `'exceptionMessage'` refers to the NL message included at the beginning of stack traces (e.g., line 7);
- The atLine refers to a line that reports a method invocation occurred in a specific file (e.g., lines 8-11);
- The `'ellipsisLine'` is a line used to reduce lengthy stack traces and has the form: "*... <number> more*";
- The `'causedByLine'` is a line that might appear at any point in a stack trace to introduce a new nested trace and has the form: "*Caused by: <stacktrace>*".

We defined a parser class for parsing stack traces:

```
PPCompositeParser subclass: #PPStackTrace
  instanceVariables: 'stackTrace stackTraceLine atLine ellipsisLine […]'
```

Among the productions, we defined `'atLine'` and `'ellipsisLine'` because they have the most recognizable form. By plugging `'PPStackTrace'` into a new instance of `'PPIsland'` and testing our approach on the whole corpus we found no errors in extracting these parts of the stack trace:

```
PPStackTrace>>atLine
  ^at , qualifiedMethod ,
    leftParenthesis , classFile ,
     ((colon , number)
     / (comma, compiledCode)
     / (leftParenthesis , compiledCode , rightParenthesis)) optional , rightParenthesis

PPStackTrace>>ellipsisLine
  ^ellipsis, number, more
```

The `'exceptionMessage'` and the `'causedByLine'` elements have a mostly unpredictable structure (e.g., different Java virtual machine versions may output the same error message differently), thus they cannot be parsed with a specific grammar. To overcome this issue we use a double-pass approach: In the first pass, we recognize and mark all the occurrences of `'atLine'` and `'ellipsisLine'`; in the second pass, we look for each line that contains strings such as "exception", "error", "failure", etc. When such a line exists, if the next n lines belong to those lines marked in the first step, we classify it and all the lines up to the first `'atLine'` as `'stack trace'`. We empirically found the n value equals to 3, to be a good tradeoff between precision and recall. If we apply our stack trace parser to the email in Fig. 1, in the first pass, it will classify lines 8–11 as *stack trace*; in the second pass, it will consider lines 5 and 7 as `'exceptionMessage'` candidates, since they both contain the string "exception". Finally, it will only pick line 7, because in the next 3 lines there is an *atLine* element (in this heuristic, we also count the empty lines, such as the line between 6–7). The results achieved by this parser are reported in the first row of Table 10.

Patch island parsing. For the patch parser, we also define some terminology for their structure. Considering Fig. 1:

- The `'patchHeader'` refers to the first two lines of a patch, which contain the reference to the modified file and, optionally, the revision versions (e.g., lines 22–23);
- The `'patchBlockHeader'` refers to the lines detailing the modification done by the patch on a chunk (e.g., line 22);
- The `'patchBlock'` refers to all the lines in the chunk (e.g., lines 25–28).

[13] https://freenetproject.org/.
[14] http://jmeter.apache.org/.
[15] http://mina.apache.org/.

A single patch has only one 'patchHeader', while it might have multiple occurrences of 'patchBlockHeader' followed by the respective 'patchBlock'.

We devised a parser, 'PPPatch', adopting an approach similar to the one of the stack trace parser: We started from the most recognizable lines and expanded to include the more ambiguous ones. The parsing is done in a single pass: We wrote a production for the 'patchHeader', even if split on multiple lines, by using the tokens '--', '+++', and '@@' as hooks; then we generated a parser that first recognizes the 'patchBlockHeader' (thanks to its clear structure), then matches the following 'patchBlock'. The patch blocks are problematic, since they have variable length and their ending is not clearly defined. In fact, after the deleted and added lines (which are marked with initial '+' or '-' signs, as in lines 26–27), patches include *some* contextual lines: Their number may vary between zero and three, or more if not well formatted. Bird et al. tackled the patch block ending issue both by using the information about the range to be found in the 'patchBlockHeader' and by analyzing how a line starts (usually the context lines should be preceded by a space) [43]. However, in our dataset we found this information to be not reliable, because of unexpected line breaks and wrong formatting. For this reason, we implemented a lookahead heuristic that checks whether the lines after the '+' or '-' signs might be good candidates as patch. The heuristics checks whether the lines are source code, by using a simplified version of the 'PPJavaIsland' parser, and it classifies them as *patch*.

The complete results are reported in the second row of Table 10. As expected, since we used a conservative lookahead threshold (maximum four lines), we have a higher precision and lower recall. A manually inspection of the false negatives showed that the low recall is also due to some patch lines that have neither 'patchHeader' nor 'patchBlockHeader', thus being ignored by our parser.

Source code island parsing. Among the three classes with structured language (i.e., stack trace, patch, and source code), code is the most ambiguous. We used a preliminary version of the 'PPJavaIsland' presented in Section 5.3, which we call 'PPJavaIsland_v0.8'. We note that our island parser for source code would match most of the content of a 'patchBlock', because they do contain valid source code. This increases the number of false positives. For this reason, we chain the source code parsing to the patch parsing: We first detect the patches, then, on the lines *not* classified as patch, we run the code parser.

The complete results are reported in the third row of Table 10.

Summary. We tested our approach to island parsing by recognizing lines of different languages in development emails. This confirms the effectiveness of the island parsing approach with other kinds of structured data and shows how it is possible to build additional parsers for structured data and plug them into our framework [11].

6.3. Extracting source models from code artifacts

One of the first application of island parsers was the extraction of source code models from source code artifacts (e.g., COBOL files) [6]. Island parsing has the advantage, over traditional parsers, to be more robust and support the extraction of models from problematic source code, e.g., it can deal with source code that does not compile, is incomplete, or contains syntax and semantic errors. Island parsers can also help with legacy source code where the grammar is not fully available; or they are useful to avoid implementing complete parsers and dealing with the intricacies of writing rules for every "low level" productions, when these are not necessary for the models that researchers and data scientists need to extract.

This is a real-world scenario in which we successfully applied our approach to island parsing. Since PETITISLAND works with arbitrary text, we can also use it for extracting customized models from source code artifacts. In our previous work, we dealt with the problem of recovering traceability links between emails and source code [5] to verify whether lightweight lexical approaches based on text matching we devised [44] could be as effective as full-fledged IR techniques (i.e., vector space model, with *tf-idf* and latent semantic analysis).

We decided to compare the effectiveness of the linking techniques when dealing with diverse syntaxes and naming conventions. We considered three mailing lists pertaining to JAVA systems, one to a PHP system, one to an ACTIONSCRIPT system, and one to a C system. To conduct our comparison, we had to extract information from the source code of these systems written in four different programming languages. In particular, to apply IR techniques we had to extract a model with the name of the classes and the terms included in their definitions (as depicted in Fig. 9), so that we could compare their vocabulary (including, or not, keywords) with that of each candidate email. The traditional approach to extract the model is to use specialized parsers for each language and model their output. However, the specialized parsers were available in different programming languages, and generated AST in different formats that should have been visited with different procedures. Although we adopted this approach in our previous work [5], when we had to reproduce the experiments and extend them, we faced the drawbacks of using such diverse parsing approaches for model extraction, especially in terms of maintainability and evolution.

For this reason, we devised an approach, based on PETITISLAND, to extract the models from the source code. This allowed us to use the same technology for the parsing of each language, thus having consistent implementation, output, and subsequent transformation procedure. This improved the maintainability and extensibility of our analysis.

In our analysis, we were only interested in extracting type declarations and their bodies, so that we could extract type names and the terms contained in their body declarations and compare to the terms found in emails. For this task, a parser that recognizes every detailed productions, such as statements or expressions, is not necessary: A parser that recognizes

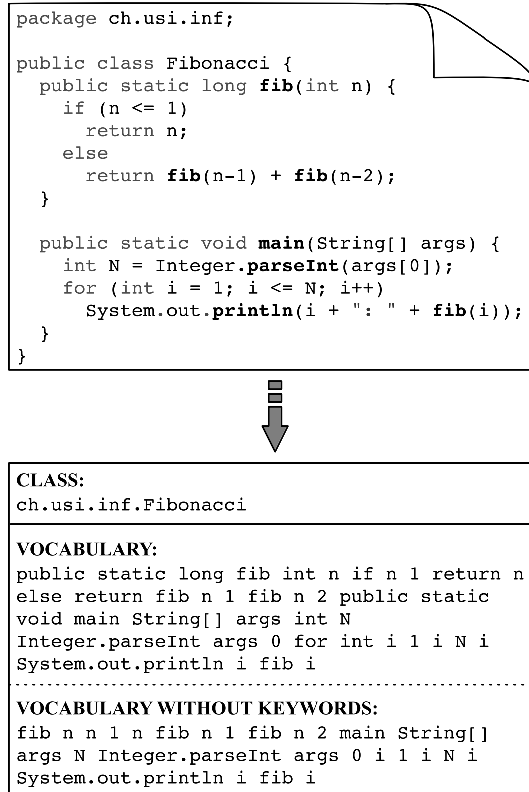


Fig. 9. Class modeling for text analysis.

type declarations and collects the text within their body (without parsing it) is sufficient. Implementing such a parser with our approach to island parsing is less time consuming than implementing a full-fledged parser. We implemented four specialized parsers, one for each considered language.

For Java we took advantage of `'PPJavaIsland'` and we reduced it to a minimal version. We removed all the productions more detailed than type declarations. For the other three programming languages, we wrote the parsers with a top-down approach, starting from the most comprehensive production (i.e., compilation units) down to type declarations. Being type declarations very high-level productions, our parsers required fewer than ten productions each.

To verify the quality of our parsing, we compared its output to the one generated by the specialized parsers we previously used [5]. For all the cases, except Java, the result of the different techniques were matching: We have been able to replicate the output of the other parsers, by applying our approach to island parsing and using a limited number of productions. In the case of Java, our parser was able to retrieve approximately 10% more classes and definitions than the specialized parser. We informed the authors of the specialized parser about this issue and they found a bug in their modeling procedure that got fixed in the subsequent release.

Summary. We showed that our approach can extract the fragments in which we are interested and conduct *fact extraction* by modeling the fragments.

7. Threats to validity

7.1. Construct validity

Construct validity threats regard the relation between theory and observation, i.e., measured variables may not measure conceptual variables.

Considering the development emails case study, to assess the island parsing phase of iLander, we relied on human judgment, both to label emails with the expected productions, and to evaluate the output. This process can be error-prone; to alleviate this, we did not directly evaluate the output on non-annotated emails, but we clearly separated the two phases. In the first one, we labeled emails without knowing the results of our approach. This allowed us to effectively verify all the expected source fragments. We decided to make use of human validation also for evaluating precision and recall. This choice is guided by the fact that the errors in an automated process would have been probably more significant than those of a human reviewer. Moreover, the human inspection allowed us to obtain a qualitative evaluation of our results.

In the case of the Stack Overflow case study, we relied on a benchmark provided by Rigby and Robillard [9]. An important aspect of Stack Overflow posts is that users are required to tag code fragments before post submission. We discovered errors in such tagging that we manually fixed.

7.2. Statistical conclusion

Statistical conclusion threats concerns the fact that the data is enough to support our claims. In our evaluation of JAVA island parsing in development emails, we considered sample sets that represent the population with an error of 9% at a confidence level of 95%. Instead, in the Stack Overflow case study, the sample set (again, the same benchmark of Rigby and Robillard [9]) represent the whole population of posts with an error of 8% at a confidence level of 95%.

7.3. External validity threats

These are concerned with the generalizability of results.

Considered programming language In our analysis and in our experiments, we considered only JAVA as a programming language whose structured fragments can be present in NL artifacts. Our results could potentially differ if we consider languages with sensibly different constructs. For example the SMALLTALK syntax is similar to that of NL (e.g., less punctuation) so it could be harder to derive an island grammar as we showed for JAVA. Thus, precision and recall of fragment recognition could be lower for different programming languages. However, covering a language like JAVA gives confidence about the applicability of our approach to a large class of languages with similar syntax, like C and C#. Moreover, we relied on JAVA specific naming conventions (e.g., camel casing) to disambiguate some constructs; similar—and thus equally exploitable—conventions are used in a number of other widely spread programming languages.

Parsing algorithm Other parsing approaches could exhibit better expressivity for ambiguity resolution, and possibly better performances for island parsing than the ones we considered. In our analysis, we covered the two typical classes of parsing techniques, that is, both bottom-up approaches (with SGLR) and top-down approaches (with PEGs). From a performance point of view, we showed how PEGs are particularly effective for island parsing of structured fragments in NL artifacts, mainly because they parse in linear time complexity. Their drawbacks i.e., memoization and not producing all ambiguous parse trees were not manifested in our experiments and did not impact performances. Concerning the latter, in fact, the possible connections among the high-level productions embedded in a document did not influence how precisely we could identify the different fragments. Studies can be designed and carried out to determine if and how this limitation impacts further analysis of the found structured productions.

Further, having implemented island parsing with a PEG-based approach opens the possibility to consider the use of Adaptable Parsing Expression Grammars (APEG) [45] as a ripe opportunity to automatically update non-standard productions and water rules as more edge cases appear in different contexts and more human input is provided.

Systems considered in island parsing evaluation We initially evaluated the fragment extraction and parsing on only three systems (ArgoUML, Freenet, Mina), and we considered development emails as NL documents. To further evaluate PETITISLAND, we applied it to parse code fragments in Stack Overflow posts and classify lines of development emails. While development emails and Q&A online services do not cover every possible NL artifact, they are significant examples, and relatively different each other in nature.

8. Conclusions

Software is, above all, a product by humans for humans. By having at our disposal all the structured information stored in unstructured NL artifacts, such as emails, IRC chats, documentations, bug comments, we can perform more accurate analyses on software systems and their evolution.

In this article we presented an approach to perform island parsing and mine structured information within NL artifacts. To implement island parsing, we considered two alternative parsing techniques, SGLR and PEGs, and implemented the island parser in both. We, then, evaluated the effectiveness of these approaches, finding that they reached high values of precision and recall for the extraction of structured fragments in development emails. However, the main difference we found was in the time performances: the SGLR approach required document splitting to be applicable to the email domain and required hours to parse the benchmark documents; the PEG-based approach, instead, did not require the splitting and parsed the benchmark in less than two minutes.

We conducted further assessment of the PEG-based approach on three additional case studies: (1) island parsing of JAVA code elements in 188 Stack Overflow posts, (2) recognizing and splitting of different structured languages (i.e., stack traces, patches, code fragments) within a JAVA development email, and (3) extracting facts from source code artifacts, written in four different programming languages. These applications showed how the approach can be used in practice and demonstrated its extensibility.

References

[1] T. Gleixner, The realtime preemption patch: pragmatic ignorance or a chance to collaborate?, in: Keynote of ECRTS 2010, 22nd Euromicro Conference on Real-Time Systems, 2010, http://lwn.net/Articles/397422/.
[2] C. Treude, O. Barzilay, M.-A. Storey, How do programmers ask and answer questions on the web? (NIER track), in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 804–807.
[3] N. Bettenburg, E. Shihab, A.E. Hassan, An empirical study on the risks of using off-the-shelf techniques for processing mailing list data, in: Proceedings of ICSM 2009, 25th IEEE International Conference on Software Maintenance, 2009, pp. 539–542.
[4] C. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008.
[5] A. Bacchelli, M. Lanza, R. Robbes, Linking e-mails and source code artifacts, in: Proc. of ICSE 2010, 32nd Int'l Conf. on Software Engineering, 2010, pp. 375–384.
[6] L. Moonen, Generating robust parsers using island grammars, in: Proceedings of WCRE 2001, 8th Working Conference on Reverse Engineering, 2001, pp. 13–22.
[7] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, SIGPLAN Not. 39 (1) (2004) 111–122.
[8] L. Renggli, S. Ducasse, T. Gîrba, O. Nierstrasz, Practical dynamic grammars for dynamic languages, in: Proceedings of DYLA 2010, 4th Workshop on Dynamic Languages and Applications, 2010.
[9] P.C. Rigby, M.P. Robillard, Discovering essential code elements in informal documentation, in: Proceedings of the 35th International Conference on Software Engineering, ICSE 2013, 2013, pp. 832–841.
[10] A. Bacchelli, M. Lanza, V. Humpa, RTFM (read the factual mails)–augmenting program comprehension with REmail, in: Proceedings of CSMR 2011, 15th IEEE European Conference on Software Maintenance and Reengineering, 2011, pp. 15–24.
[11] A. Bacchelli, T. dal Sasso, M. D'Ambros, M. Lanza, Content classification of development emails, in: Proceedings of ICSE 2012, 34th ACM/IEEE International Conference on Software Engineering, 2012, pp. 375–385.
[12] A. Kuhn, S. Ducasse, T. Gírba, Semantic clustering: identifying topics in source code, Inf. Softw. Technol. 49 (3) (2007) 230–243.
[13] S. Rastkar, G.C. Murphy, G. Murray, Summarizing software artifacts: a case study of bug reports, in: Proceedings of ICSE 2010, 2010, pp. 505–514.
[14] S. Haiduc, J. Aponte, A. Marcus, Supporting program comprehension with source code summarization, in: Proceedings of ICSE 2010, 2010, pp. 223–226.
[15] K.S. Jones, Automatic summarising: the state of the art, Inf. Process. Manag. 43 (2007) 1449–1481.
[16] G.C. Murphy, D. Notkin, Lightweight lexical source model extraction, ACM Trans. Softw. Eng. Methodol. 5 (3) (1996) 262–292.
[17] N. Bettenburg, R. Premraj, T. Zimmermann, S. Kim, Extracting structural information from bug reports, in: Proceedings of MSR 2008, 5th Working Conference on Mining Software Repositories, 2008, pp. 27–30.
[18] A. Bacchelli, M. D'Ambros, M. Lanza, Extracting source code from e-mails, in: Proceedings of ICPC 2010, 18th International Conference on Program Comprehension, 2010, pp. 24–33.
[19] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, C. Weiss, What makes a good bug report?, IEEE Trans. Softw. Eng. 36 (5) (2010) 618–643.
[20] M. Tomita, An efficient context-free parsing algorithm for natural languages, in: Proceedings of the 9th International Joint Conference on Artificial Intelligence, vol. 2, IJCAI'85, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1985, pp. 756–764, http://dl.acm.org/citation.cfm?id=1623611.1623625.
[21] J. Earley, An efficient context-free parsing algorithm, Commun. ACM 13 (2) (1970) 94–102.
[22] R.A. Frost, R. Hafiz, P. Callaghan, Parser combinators for ambiguous left-recursive grammars, in: P. Hudak, D.S. Warren (Eds.), Practical Aspects of Declarative Languages, in: Lect. Notes Comput. Sci., vol. 4902, Springer, 2008, pp. 167–181.
[23] A.D. Thurston, J.R. Cordy, A backtracking lr algorithm for parsing ambiguous context-dependent languages, in: Proceedings of CASCON 2006, Conference of the Centre for Advanced Studies on Collaborative Research, 2006, CASCON.
[24] E. Scott, A. Johnstone, Gll parsing, Electron. Notes Theor. Comput. Sci. 253 (7) (2010) 177–189.
[25] M.G.J. van den Brand, A.v. Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, J. Visser, The ASF+SDF meta-environment: a component-based language development environment, in: Proceedings of the 10th International Conference on Compiler Construction, 2001, pp. 365–370.
[26] N. Synytskyy, J.R. Cordy, T.R. Dean, Robust multilingual parsing using island grammars, in: Proceedings of CASCON 2003, Conference of the Centre for Advanced Studies on Collaborative Research, CASCON, 2003, pp. 266–278.
[27] J. Cordy, The TXL source transformation language, Sci. Comput. Program. 61 (3) (2006) 190–210.
[28] J. Kurš, M. Lungu, R. Iyadurai, O. Nierstrasz, Bounded seas, Comput. Lang. Syst. Struct. 44 (2015) 114–140.
[29] C. Bird, A. Gourley, P. Devanbu, Detecting patch submission and acceptance in oss projects, in: Proceedings of MSR 2007, 2007, p. 26.
[30] J. Tang, H. Li, Y. Cao, Z. Tang, Email data cleaning, in: Proceedings of KDD 2005, 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, 2005, pp. 489–498.
[31] A. Dekhtyar, J. Hayes, T. Menzies, Text is software too, in: Proceedings of MSR 2004, 1st International Workshop on Mining Software Repositories, 2004, pp. 22–26.
[32] H. Basten, P. Klint, Defacto: language-parametric fact extraction from source code, in: Proceedings of SLE 2008, International Conference of Software Language Engineering, 2008, pp. 265–284.
[33] T.R. Dean, J.R. Cordy, A.J. Malton, K.A. Schneider, Agile parsing in txl, Autom. Softw. Eng. 10 (4) (2003) 311–336.
[34] X. Wu, B.R. Bryant, J. Gray, M. Mernik, Component-based lr parsing, Comput. Lang. Syst. Struct. 36 (1) (2010) 16–33.
[35] E. Visser, Syntax Definition for Language Prototyping, Ph.D. thesis, University of Amsterdam, 1997.
[36] Sun Microsystems, Inc., Code conventions for the Java™ programming language, http://www.oracle.com/technetwork/java/codeconvtoc-136057.html, 1999.
[37] J. Tang, H. Li, Y. Cao, Z. Tang, Email data cleaning, in: Proceedings of SIGKDD 2005, 11th International Conference on Knowledge Discovery in Data mining, 2005, pp. 489–498.
[38] M. Triola, Elementary Statistics, 10th edition, Addison-Wesley, 2007.
[39] B. Ford, Packrat parsing: simple, powerful, lazy, linear time, functional pearl, SIGPLAN Not. 37 (9) (2002) 36–47, http://dx.doi.org/10.1145/583852.581483, http://doi.acm.org/10.1145/583852.581483.
[40] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, The Java Language Specification, 4th edition, Oracle, 2012.
[41] T. Xie, J. Pei, MAPO: mining api usages from open source repositories, in: Proceedings of MSR 2006, 3rd International Workshop on Mining Software Repositories, 2006, pp. 54–57.
[42] N. Bettenburg, S.W. Thomas, A.E. Hassan, Using code search to link code fragments in discussions and source code, in: Proceedings of CSMR 2012, 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 319–329.
[43] C. Bird, A. Gourley, P. Devanbu, Detecting patch submission and acceptance in OSS projects, in: Proceedings of MSR 2007, 4th International Workshop on Mining Software Repositories, 2007, pp. 26–29.
[44] A. Bacchelli, M. D'Ambros, M. Lanza, R. Robbes, Benchmarking lightweight techniques to link e-mails and source code, in: Proceedings of WCRE 2009, 16th IEEE Working Conference on Reverse Engineering, 2009, pp. 205–214.
[45] L.V. Reis, V.O. Di Iorio, R.S. Bigonha, An on-the-fly grammar modification mechanism for composing and defining extensible languages, Comput. Lang. Syst. Struct. 42 (2015) 46–59.