Bounded Seas
— Island Parsing Without Shipwrecks

Jan Kurš, Mircea Lungu, and Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch

Abstract. Imprecise manipulation of source code (semi-parsing) is useful for tasks such as robust parsing, error recovery, lexical analysis, and rapid development of parsers for data extraction. An island grammar precisely defines only a subset of a language syntax (islands), while the rest of the syntax (water) is defined imprecisely.

Usually, water is defined as the negation of islands. Albeit simple, such a definition of water is naive and impedes composition of islands. When developing an island grammar, sooner or later a programmer has to create water tailored to each individual island. Such an approach is fragile, however, because water can change with any change of a grammar. It is time-consuming, because water is defined manually by a programmer and not automatically. Finally, an island surrounded by water cannot be reused because water has to be defined for every grammar individually.

In this paper we propose a new technique of island parsing — bounded seas. Bounded seas are composable, robust, reusable and easy to use because island-specific water is created automatically. We integrated bounded seas into a parser combinator framework as a demonstration of their composability and reusability.

1 Introduction

Island grammars [1] offer a way to parse input without complete knowledge of the target grammar. They are especially useful for extracting selected information from source files, for reverse engineering and similar applications. The approach assumes that only a subset of the language syntax is known (the islands), while the rest of the syntax is undefined (the water). During parsing, any unrecognized input (water) is skipped until an island is found.

The common misconception is that water should consume everything until an island is detected. Such a water is easy to define, but it causes composability problems. To be specific, such a water does not allow islands to be embedded into the optional or repetitive rules without giving misleading results. To be correct, water should stop when any of a number of possible islands is encountered. Small changes in the grammar may radically change the nature of the water.

To define an island grammar that will return the unambiguous and correct result we have to define specific water manually for each particular island, contrary to one global water. Yet island-specific water is fragile, hard to define and

B. Combemale et al. (Eds.): SLE 2014, LNCS 8706, pp. 62–81, 2014.
© Springer International Publishing Switzerland 2014

it is not reusable. It is fragile, because it requires reevaluation by a programmer after any change in a grammar. It is hard to define, because it requires the programmer's time for detailed analysis of a grammar. It is not reusable, because island-specific water depends on rules following the island, thus it is tailored to the context in which the island is used — it is not general.

In this paper we suggest a new technique for island parsing: *bounded seas*. Bounded seas are composable, reusable, robust and easy to use. The key idea of bounded seas is that specialized water is defined for each particular island (depending on the context of the island) so that an island can be embedded into optional or repetitive rules. To achieve such composability, an island is never searched behind a boundary defined by the rule following the island. To prevent fragility and to improve reusability, we describe how to compute water automatically, without user interaction. To prove feasibility, we integrated bounded seas into Petit Parser [2], a PEG–based parser combinator framework [3].

The contributions of the paper are: a) a description of bounded seas — a composable, reusable, robust and easy method of island parsing; b) a formalization of the process leading to an island grammar for PEGs; c) and an implementation of bounded seas in a PEG-based parser combinator framework.

Structure. Section 2 motivates this work by presenting the limitations of island grammars with an example. Section 3 presents our solution to overcoming these limitations by introducing bounded seas. Section 4 presents a sea operator for PEGs, which creates bounded sea from an arbitrary PEG expression. Section 5 discusses implementation, applicability of bounded seas in GLL and presents a Java code analysis case study that compares bounded seas with island grammars. Section 6 presents other semi-parsing techniques and highlights similarities and differences between them and bounded seas. Finally, section 7 concludes this paper.

2 Motivating Example

Let us consider the domain specific source code from Listing 1.1. We don't have a grammar specification for the code, because the parser was written using *ad hoc* techniques and the parser code is proprietary. Let us suppose that our task is to extract class and method names.

```
class Shape
    Color color;

    method getColor {
        return color;
    }
    int uid = UIDGenerator.newUID;
endclass
```

Listing 1.1. Source code of the `Shape` class in a proprietary language

2.1 A Naive Island Grammar

To extract the method names, we need a parser. To write a parser, we need a grammar. Because the grammar can easily consist of a hundred rules (*e.g.*, ≈ 80 for Python, ≈ 180 for Java) and since we do not want to spend many hours defining them, we define an island grammar in PEG (see Appendix A) with fewer than ten rules as in Listing 1.2. We initially assume that each class body contains just one method.

The `method` rule is an island. The `methodSea` rule represents a '*method*' island surrounded by water. The `methodSea` rule is defined imprecisely: water skips everything until '*method*' is found. Similarly we define the `methodBody` rule, which consumes an open curly bracket and then skips everything until the closing curly bracket is found.

```
start       ← class
class       ← 'class' id classBody 'endclass'
classBody   ← methodSea

methodSea   ← (!'method' .)* method (!'endclass'.)*

method      ← 'method' id methodBody
methodBody  ← '{' (!'}' .)* '}'

id          ← letter (letter / number)*
letter      ← 'a' / 'b' / 'c' ...
number      ← '1' / '2' / '3' ...
```

Listing 1.2. Our first island grammar

Composability Problems. The `methodSea` rule in the grammar in Listing 1.2 uses the naive definition of water. It will work as long as we do not complicate the grammar.

Suppose we now allow multiple classes in a single file (`start ← class*`). Parsing the input in Listing 1.3 should fail because `Shape` does not contain a method. However the result, no matter whether we use PEG or CFG, is only one class — `Shape` (instead of `Shape` and `Circle`) — with a method `getDiameter`, which is wrong.

Things do not get better when we allow multiple repetitions of `methodSea` s in a `classBody` (`classBody ← methodSea*`). The parser will stay confused and depending on the technology (PEG, CFG), the result will be either incorrect (PEG) or ambiguous (CFG). In case of the ambiguous results, it is nice to know that one of the many results is correct, but how can we know which one?

```
class Shape
    int uid = UIDGenerator.newUID;
endclass

class Circle
    int diameter;

    method getDiameter {
        return diameter;
    }
endclass
```

Listing 1.3. Source code of `Shape` and `Circle` classes

2.2 An Advanced Island Grammar

To make the `methodSea` composable we must make it possible for it to be embedded into optional (`?`) or repetition (`+` , `*`) rules. Thus, we define the grammar as in Listing 1.4. This new definition can properly parse multiple classes in a file with an arbitrary number of methods in a class.

```
start      ← class*
class      ← 'class' id classBody 'endclass'
classBody  ← (methodSea)*

methodSea  ← (!'method' !'endclass'.)*
               method
             (!'method' !'endclass'.)*

method     ← 'method' id methodBody
methodBody ← '{'
             (
               (!'}' !'{' .)*
                 methodBody
               (!'}' !'{' .)*
             )*
             '}'

id         ← letter (letter / number)*
letter     ← 'a' / 'b' / 'c' ···
number     ← '1' / '2' / '3' ···
```

Listing 1.4. Complete and final island grammar

One can see that the syntactic predicates in the `methodSea` are more complicated. They have been inferred from the rest of the grammar by analyzing what tokens can appear behind the `method` island.

Ease of Use, Robustness, and Reusability Problems. The limitations of defining the `methodSea` by hand are illustrative of the general problems of semi-parsing:

1. Such a definition is time-consuming to produce because it requires programmer's time to analyze the grammar.
2. The definition is fragile, because the predicates need to be re-evaluated after any change in a grammar (*e.g.*, adding inner classes will result in adding `!'class'` into the predicates).
3. Last but not least, the `methodSea` is tailored just for the grammar in Listing 1.4 *e.g.*, it cannot be re-used in a grammar that does not use `'endclass'` as a keyword.

3 Bounded Seas

3.1 The Sea Operator in a Nutshell

We have shown that water must be tailored both to the island within the sea and to the surroundings of the sea (*e.g.*, `methodSea` in Listing 1.4). In this paper, we define a *bounded sea* to be an island surrounded by context-aware water.

To automate the definition of bounded seas we introduce a new operator for building tolerant grammars: *the sea operator*. We use the notation `~island~` to create sea from `island`, which can be a terminal or non-terminal. Instead of having to produce complex definitions of sea, a programmer can use the sea operator which will do the hard work. Listing 1.5 presents how the grammar in Listing 1.4 is defined using the sea operator:

```
class      ← 'class' id classBody 'endclass'
classBody  ← methodSea*

methodSea  ← ~method~
method     ← 'method' id methodBody

methodBody ← '{' ~(methodBody / ϵ)~* '}'

id         ← letter (letter / number)*
letter     ← 'a' / 'b' / 'c' ···
number     ← '1' / '2' / '3' ···
```

Listing 1.5. Island Grammar from Listing 1.4 rewritten with the sea operator

A rule defined with the sea operator (*e.g.*, `~method~`) maintains the composability property of the advanced grammar since by applying the sea operator we search for the island in a restricted scope. Moreover, such a rule is reusable, robust, and uncomplicated to define.

Conceptually two ideas are fundamental for bounded seas.

1. Water is defined for each island so that the search for an island will never cross the boundary defined by the rule that follows the island. For example, `method` islands will be searched only within a class and not in a whole file. The search boundary ensures composability.
2. Water computation in bounded seas is fully automated. The sea is created using the sea operator `~island~`. Once the sea is placed in the grammar, the grammar is analyzed and appropriate water is created without user interaction. This way the sea can be placed in any grammar. In case the grammar is changed, the seas are recomputed automatically. Automatic water computation ensures ease of definition, robustness, and reusability.

Bounded seas can be integrated into a parser combinator framework, a highly modular framework for building a parser from other composable parsers [4]. The fact that a bounded sea can be implemented as a parser combinator demonstrates its composability and flexibility. In fact, the original motivation for this work was the desire to have a reusable approach to semi-parsing that can be integrated with parser combinators.

3.2 The Sea Boundary

A sea boundary limits the scope within which the island can be searched. Water cannot consume anything beyond the boundary. The boundary of the sea consists of the input accepted by any parsing expression that can appear immediately after the island. For example in case of `A ← ~'a'~ (B / C)` the boundary of `~'a'~` is any input accepted either by `B` or by `C`.

The sea boundary ensures composability. With help of the boundary we can search for methods in a class without the risk that other classes will interfere. This was the issue for the input in Listing 1.3 and the non-bounded grammar from Listing 1.2, which found the `getDiameter` method in the `Shape` class.

The predicates of island-specific water have to be set up so that they stop water in two cases: first, when an island is reached; second, when the boundary is reached. If the boundary is reached before the island is found, water stops and the sea fails. The fact that sea can fail implies that sea can be embedded into optional or repetition expressions. For example, we can define the superclass specification as an optional island.

```
~classDef~ ~superclassSpec~? classBody 'endclass'
```

If `superclassSpec` is not present for the particular class, it will simply fail when reaching `classBody` instead of searching for `superclassSpec` further and further. The same holds for repetitions.

```
classBody ← ~method~*
```

This rule will consume only methods until it reaches '*endclass*' in the input string, since `endclass` forms the boundary of `~method~`, so methods in another class cannot be inadvertently consumed.

We first define bounded seas generally, and later provide a PEG-specific definition.

Definition 1 (Bounded Sea). A *bounded sea* consists of a sequence of three parsing phases:

1. **Before-Water:** Consume the input until an island or the boundary appears. Fail the whole sea if we hit the boundary. Continue if we hit an island.
2. **Island:** Consume an island.
3. **After-Water:** Consume the input until the boundary is reached.

3.3 The Context Sensitivity of Bounded Seas

To make bounded seas useful we decided for a context-sensitive behaviour. A bounded sea recognizes different substrings of an input depending on what surrounds the sea. There are two cases where the context-sensitivity emerges:

1. A bounded sea recognizes different input depending on what immediately follows the sea.
2. A bounded sea recognizes different input depending on what immediately precedes the sea.

Let us demonstrate on rules from Listing 1.6 and two inputs '*..a..b..*' and '*..a..c..*'. On its own, `A` recognizes any input with `'a'` and `B` recognizes any input with `'b'` (see rows 1-4 in Table 1).

```
A  ← ~'a'~
B  ← ~'b'~

R1 ← A          R2 ← B
R3 ← A 'b'      R4 ← A 'c'
R5 ← A  B
```

Listing 1.6. Rules for the context-sensitive behaviour demonstration

However, when the two islands are not alone, their boundary can differ, depending on the context. The boundary of `A` is `'b'` in `R3` and the boundary of `A` is `'c'` in `R4`. Therefore `A` consumes different substrings of input depending whether called from `R3` or `R4` (see rows 5-8 in Table 1).

A more complex case of context-sensitivity, which we call the *overlapping sea problem*, arises when one sea is immediately followed by another. Consider, for example, rule `R5`, where the sea `A` has as its boundary `B`, which is also a sea. Note that the before-water of `B` should consume anything up to its island `'b'` or its own boundary, *including the island of its preceding sea* `A`. Now,

Table 1. The seas `A` and `B` recognize different inputs depending on a context

	Rule	Input	Result
1	`R1 ← A`	'*..a..b..*'	`A` recognizes '..a..b..'
2	`R1 ← A`	'*..a..c..*'	`A` recognizes '..a..c..'
3	`R2 ← B`	'*..a..b..*'	`B` recognizes '..a..b..'
4	`R2 ← B`	'*..a..c..*'	`B` fails
5	`R3 ← A 'b'`	'*..a..b..*'	`A` recognizes '..a..' `'b'` recognizes 'b'
6	`R3 ← A 'b'`	'*..a..c..*'	`A` recognizes '..a..b..' `'b'` fails
7	`R4 ← A 'c'`	'*..a..b..*'	`A` recognizes '..a..b..' `'c'` fails
8	`R4 ← A 'c'`	'*..a..c..*'	`A` recognizes '..a..' `'c'` recognizes 'c'
9	`R5 ← A B`	'*..a..b..*'	`A` recognizes '..a..' `B` recognizes 'b..'
10	`R5 ← A B`	'*..a..c..*'	`A` recognizes '..a..c..' `B` fails

the before-water of `A` should consume anything up to either its island `'a'` or its boundary `B`. But the very search for the boundary will now consume the island we are looking for, since `B`'s before-water will consume `'a'`! We must therefore take special care to avoid a "shipwreck" in the case of overlapping seas by disabling the before-water of the second sea.

4 Bounded Seas in Parsing Expression Grammars

Starting from the standard definition of PEGs (see Appendix A), we now show how to add the sea operator while avoiding the overlapping sea problem. To define the sea operator, we need the following two abstractions:

1. **The water operator** *consumes uninteresting input* Water ($\approx$) is a new PEG prefix operator that takes as its argument an expression that specifies when the water ends. We discuss this in detail in subsection 4.1.
2. **The NEXT function** *determines the boundary of a sea.* Intuitively, $NEXT(e)$ returns the set of expressions[1] that can appear directly after a particular expression e. The details of the NEXT function are given in subsection 4.2.

Definition 2 (Sea Operator). Given the definitions of $\approx$ and NEXT, we define the sea operator as follows: $\sim$`e`$\sim$ is a sequence expression

$$\approx(\texttt{e} \texttt{ / } next_1 \texttt{ / } next_2 \texttt{ / } \cdots\ next_n)$$
$$\texttt{e}$$
$$\approx(next_1 \texttt{ / } next_2 \texttt{ / } \cdots\ next_n)$$

where $next_i \in NEXT(e)$.

That is, the before-water consumes everything up to the island or the boundary, and the after-water consumes everything up to the boundary.

[1] The NEXT function is modelled after FOLLOW sets from parsing theory, except that instead of returning a set of tokens, it returns a set of parsers.

4.1 The *Water* Operator

The purpose of a water expression is to consume uninteresting input. Water consumes input until it encounters the expression specified in its argument (*i.e.*, the *boundary*). We must, however, take care to avoid the overlapping sea problem. If two seas overlap (one sea is followed by another), the second sea bounds the first one. The second sea has to disable its before-water as illustrated in subsection 3.3. We detect overlapping seas as follows: if sea s_1 is invoked from the water of another sea s_2, it means that the water of s_1 is testing for its boundary s_2 and thus s_2 has to disable its before-water. To distinguish between nested seas (*e.g.*, `~'x' ~island~ 'x'~`) and overlapping seas, we test the position where this sea was invoked. In case of nested seas the positions differ, and in case of overlapping seas they are the same.

Definition 3 (Extended Semantics). In order to detect overlapping seas, we extend the semantics of a PEG $G = \{N, T, R, e_s\}$ with a stack of invoked expressions and their positions. For standard PEG operators there is no change except that an explicit stack S is maintained. We define a relation $\Rightarrow_G$ from tuples of the form (x, S) to the output o, where $x \in T^*$ is an input string to be recognized, $S \notin N$ is a stack consisting of tuples (e, p), where $p \geq 0$ is a position and e is a parsing expression, and $o \in T^* \cup \{f\}$ indicates the result of a recognition attempt. The distinguished symbol $f \notin T$ indicates failure. Function $len(x)$ returns a length of an input x. Function $(e, p) : S$ denotes a stack with tuple (e, p) on the top and stack S below. For $((x, S), o) \in \Rightarrow_G$ we write $(x, S) \Rightarrow o$.

We define $\Rightarrow_G$ inductively as follows (without any semantic changes for standard PEG operators):

$$\textbf{Empty:}\ \frac{x \in T^*}{(x, (\epsilon, p) : S) \Rightarrow \epsilon}$$

$$\textbf{Terminal (success case):}\ \frac{a \in T, x \in T^*}{(ax, (a, p) : S) \Rightarrow a}$$

$$\textbf{Terminal (failure case):}\ \frac{a \neq b \qquad (a, \epsilon, S) \Rightarrow f}{(bx, (a, p) : S) \Rightarrow f}$$

$$\textbf{Nonterminal:}\ \frac{A \leftarrow e \in R \qquad (x, (e, p) : S) \Rightarrow o}{(x, (A, p) : S) \Rightarrow o}$$

$$\textbf{Sequence (success case):}\ \frac{\begin{array}{c}(x_1x_2y, (e_1, p) : S) \Rightarrow x_1 \\ (x_2y, (e_2, p + len(x_1)) : S) \Rightarrow x_2\end{array}}{(x_1x_2y, (e_1e_2, p) : S) \Rightarrow x_1x_2}$$

$$\textbf{Sequence (failure case)}:\ \frac{(x, (e_1, p) : S) \Rightarrow f}{(x, (e_1e_2, p) : S) \Rightarrow f}$$

$$\textbf{Sequence (failure case 2):}\ \frac{\begin{array}{c}(xy, (e_1, p) : S) \Rightarrow x \\ (y, (e_2, p + len(x)) : S) \Rightarrow f\end{array}}{(xy, (e_1e_2, p) : S) \Rightarrow f}$$

$$\textbf{Alternation (case 1):}\ \frac{(xy, (e_1, p) : S) \Rightarrow x}{(x, (e_1/e_2, p) : S) \Rightarrow x}$$

$$\textbf{Alternation (case 2):}\ \frac{\begin{array}{c}(x, (e_1, p) : S) \Rightarrow f \\ (x, (e_2, p) : S) \Rightarrow o\end{array}}{(x, (e_1/e_2, p) : S) \Rightarrow o}$$

$$\textbf{Repetitions (repetition case):}\ \frac{\begin{array}{c}(x_1x_2y, (e, p) : S) \Rightarrow x_1 \\ (x_2, (e*, p + len(x_1)) : S) \Rightarrow x_2\end{array}}{(x_1x_2y, (e*, p) : S) \Rightarrow x_1x_2}$$

$$\textbf{Repetitions (termination case):}\ \frac{(x, (e, p) : S) \Rightarrow f}{(x, (e*, p) : S) \Rightarrow \epsilon}$$

$$\textbf{Not predicate (case 1):}\ \frac{(xy, (e, p) : S) \Rightarrow x}{(xy, (!e, p) : S) \Rightarrow f}$$

$$\textbf{Not predicate (case 2):}\ \frac{(xy, (e, p) : S) \Rightarrow f}{(xy, (!e, p) : S) \Rightarrow \epsilon}$$

Definition 4 (Water Operator). With the extended semantics of PEGs we can define a prefix **water operator** $\approx$. It searches for a boundary and consumes input until it reaches a boundary. If the water starts a boundary of another sea, it stops immediately. Function $seasOverlap(S, p_1)$ returns true if there is a pair $(\approx e, p_2)$ on a stack S where $p_1 = p_2$ and e is any parsing expression and returns false otherwise. $x \in T^*$, $y \in T^*$, $z \in T^*$ and function $substring(x)$ returns set of all substrings of x.

$$\textbf{Overlapping seas case:}\ \frac{seasOverlap(S, p)}{(x, (\approx e, p) : S) = \epsilon}$$

$$\textbf{Boundary found case:}\ \frac{\begin{array}{c}(yz, (e, p) : S) \Rightarrow y \\ (x'', (e, p + len(x')) : S) \Rightarrow f \ \ \forall x = x'x''x'''\end{array}}{(xyz, (\approx e, p) : S) = x}$$

$$\textbf{End of input case:}\ \frac{(yz, (e, pos(x)) : S) \Rightarrow f}{(xyz, (\approx e, p) : S) = xyz}$$

In case of *directly nested seas* (*e.g.*, $\sim\sim$`island`$\sim\sim$) we obtain the same behaviour as with $\sim$`island`$\sim$. The function *seasOverlap* returns true in case a sea is directly invoked from another sea without consuming any input. Applying the rule *Overlapping seas* from Definition 4, water of the inner sea is eliminated and the boundary is the same for the both seas. Therefore $\sim\sim$`island`$\sim\sim$ is equivalent to $\sim$`island`$\sim$.

4.2 The NEXT Function

The purpose of the NEXT function is to determine the boundary of a sea. The boundary is an expression that consumes whatever follows the sea. Consider the

```
code         ← (~class~/~struct~)* mainMethod
class        ← 'class'  ID classBody
stuct        ← 'struct' ID sbody
mainMethod   ← 'public' 'method' 'main' methodBody

classBody    ← ...
sbody        ← ...
methodBody   ← ...
ID           ← ...
```

Listing 1.7. Definition of code that consists of classes and structures followed by main method

grammar in the example from Listing 1.7. The `code` is defined in a way that it accepts an arbitrary number of class and structure islands in the beginning (classes and structures can be in any order) and there is a main method at the end. Intuitively, another class island, a structure island or a main method can appear after a class island.

The boundary has to be something "solid". An optional expression itself is not a good boundary, because it succeeds for any input. Consider a simple expression `~e~ 'a'? 'b' 'c'`. The `'a'?` can appear behind the *'island'* but `'b'` as well, if `'a'` fails. It is certainly not `'c'` because it always succeeds `'b'`. In this case we have to define a boundary as `'a?' 'b'` (not only `'a'?`).

Definition 5 (Abstract Simulation). In order to recognize a solid expression we define a relation representing an abstract simulation [5]. We define a relation $\rightharpoonup_G$ consisting of pairs (e, o), where e is an expression and $o \in \{0, 1, f\}$. We will write $e \rightharpoonup o$ for $(e, o) \in \rightharpoonup_G$. If $e \rightharpoonup 0$, then e can succeed on some input while consuming no input. If $e \rightharpoonup 1$, then e can succeed on some input while consuming at least one terminal. If $e \rightharpoonup f$, then e may fail on some input. We will use variable s to represent a $\rightharpoonup_G$ outcome of either 0 or 1. We will define the simulation relation $\rightharpoonup_G$ as follows:

1. $\epsilon \rightharpoonup 0$.
2. $t \rightharpoonup 1$, $t \in T$.
3. $t \rightharpoonup f$, $t \in T$.
4. $A \rightharpoonup o$ if $R_G(A) \rightharpoonup o$.
5. $e_1 e_2 \rightharpoonup 0$ if $e_1 \rightharpoonup 0$ and $e_2 \rightharpoonup 0$.
   $e_1 e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup 1$ and $e_2 \rightharpoonup s$.
   $e_1 e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup s$ and $e_2 \rightharpoonup 1$.
6. $e_1 e_2 \rightharpoonup f$ if $e_1 \rightharpoonup f$
7. $e_1 e_2 \rightharpoonup f$ if $e_1 \rightharpoonup s$ and $e_2 \rightharpoonup f$.
8. $e_1 / e_2 \rightharpoonup s$ if $e_1 \rightharpoonup s$
9. $e_1 / e_2 \rightharpoonup o$ if $e_1 \rightharpoonup f$ and $e_2 \rightharpoonup o$.
10. $e* \rightharpoonup 1$ if $e \rightharpoonup 1$

11. $e* \rightharpoonup 0$ if $e \rightharpoonup f$
12. $!e \rightharpoonup f$ if $e \rightharpoonup s$
13. $!e \rightharpoonup 0$ if $e \rightharpoonup f$

Because this relation does not depend on the input string, and there are a finite number of expressions in a grammar, we can compute this relation over any grammar [5].

Definition 6 (NEXT). If S is a stack of *(expression, position)* pairs representing positions and invoked parsing expressions and if $\overline{\triangle}(S)$ pops an element from the stack S returning a stack S' without the top element and if $s_n, s_{n-1}, ..s_2, s_1$ are expressions on the stack S (top of the stack is to the left, bottom to the right) and if $\square$ is a sequence formed from two sets of parsing expressions such that $S_1 \square S_2 = \{e_i e_j | e_i \in S_1, e_j \in S_2\}$, we define $NEXT(S)$ as a set of expressions such that:

- if $s_{n-1} = e_1 e_2$ and $s_n = e_1$ and $e_2 \not\rightharpoonup 0$ then $NEXT(S) = \{e_2\}$
- if $s_{n-1} = e_1 e_2$ and $s_n = e_1$ and $e_2 \rightharpoonup 0$ then $NEXT(S) = \{e_2\} \square NEXT(\overline{\triangle}(S))$
- if $s_{n-1} = e_1 e_2$ and $s_n = e_2$ then $NEXT(S) = NEXT(\overline{\triangle}(S))$
- if $s_{n-1} = e_1 / e_2$ and $s_n = e_1$ or $s_n = e_2$ then $NEXT(S) = NEXT(\overline{\triangle}(S))$
- if $s_{n-1} = e*$ and $e = s_n$ then $NEXT(S) = e \cup NEXT(\overline{\triangle}(S))$
- if $s_{n-1} = !e$ and $e = s_n$ then $NEXT(S) = \{\}$

5 Discussion

5.1 Implementation

As a validation of bounded sea composability and reusability we provide an implementation of bounded sea in a PetitParser framework.[2]

5.2 Java Parser Case Study

In the following section we compare four kinds of Java parsers.

1. **PetitJava** is an open source Java parser written using PetitParser [2] by the community around the Moose analysis platform[6]. We used the latest version available online[3].
2. **Naive Island Parser** is an island parser with water defined as a negation of an island. The sea rules in this parser can be reused, because they do not consider their surroundings and they are grammar-independent. The sea rules are defined in a simple form: consume input until an island is found, then consume an island.

[2] `http://scg.unibe.ch/research/islandparsing/sle2014`
[3] `http://smalltalkhub.com/#!/~Moose/PetitJava/`

3. **Advanced Island Parser** is a more complex version of the naive island parser. The water is more complicated to prevent the most frequent failures of island parser. The sea rules in this parser are hard-wired to the grammar and cannot be reused. The sea rules are customized for a particular islands.
4. **Island Parser with Bounded Seas** is an island parser written using bounded seas. The sea rules were defined using the sea operator.

The PetitJava parser parses Java code[4]. All the island parsers are very similar between themselves, with a similar number of rules. The island parsers were designed to extract only method names in a class. None of the parsers was optimized to provide a better performance.

In this section we compare the three island parsers (almost identical in a structure) written by the first author. Very probably, the advanced island parser can be updated so that it achieves better precision and better performance, but at the cost of considerable engineering work. We want to demonstrate that naive water rules do not work and that the advanced version of water is needed. Moreover, we want to confirm that with bounded seas we can get high precision and performance without the effort required to define an advanced island parser.

Table 2 displays the precision (P) and recall (R) with which the different parsers extract methods from six Java classes. The PetitJava parser had a perfect precision and recall for the cases where it did not fail. The PetitJava failures are due to incomplete and incorrectly specified rules.

Table 2. Precision (P) and recall (R) of the four tested parsers. We indicate with "-" the cases where the PetitJava parser fails with an error

Class	**PetitJava**		**Island**		**Advanced**		**Bounded**		**Method**
Name	P	R	P	R	P	R	P	R	**Count**
java.lang.Class	-	-	0.05	0.05	0.78	0.78	0.91	0.91	108
java.lang.Object	-	-	0.00	0.00	0.91	0.91	0.91	0.91	12
java.lang.Math	1	1	0.02	0.02	0.91	0.93	0.95	1.00	46
java.io.InputStream	1	1	0.00	0.00	0.88	0.88	1.00	1.00	9
java.io.FileInputStream	1	1	0.13	0.13	0.80	0.75	1.00	1.00	16
java.util.ArrayList	-	-	0.07	0.07	0.88	0.82	0.96	0.92	28

Table 3 synthesizes the information in Table 2 and adds information about the effort required to implement the given parser. The best precision and recall are achieved with the PetitJava parser, sacrificing simplicity and robustness. Island parsers provide very good robustness, but the naive island parser does not provide any useful output. The advanced island parser is comparable to the bounded island parser. This is not surprising, considering that bounded seas use the same techniques as the advanced island parser. Support for inner classes means that a parser can be extended to recognize inner classes and methods inside them. The non-bounded island parsers can only search for a flattened list of methods in contrast to bounded seas, which can support nested lists.

[4] In this paper, we exclusively consider Java 5 code.

Table 3. Comparison of parsing techniques. The 11 rules of the advanced parser are marked as a medium effort, because the extra rule is not trivial to infer and it is highly interconnected with the rest of the grammar.

	PetitJava	**Island**	**Advanced**	**Bounded**
P & R	very high	very low	high	high
Robustness	low	high	high	high
Supports inner classes	yes	no	no	yes
Effort	≈ 200 rules very high	10 rules low	11 rules medium	10 rules low

Table 4 presents a performance comparison of the parsers. The performance of the bounded seas parser is in all the test cases one order of magnitude better than that of the advanced island parser. The performance difference is due to the bounded sea parsers skipping water at the start of boundary of another bounded sea. The Naive and Advanced island parsers are slower than the PetitJava parser.

Table 4. The performance comparison of the four parsers shows that the performance of the bounded seas parser is on par with the one of the PetitJava parser

Class Name	**PetitJava** [ms]	**Island** [ms]	**Advanced** [ms]	**Bounded** [ms]
java.lang.Class	-	6921	25229	4733
java.lang.Object	-	1058	6000	351
java.lang.Math	941	2077	11000	875
java.io.InputStream	331	638	3135	325
java.io.FileInputStream	338	301	561	301
java.util.ArrayList	-	1025	3831	826

5.3 Generalized LL Parsing

In this paper we have discussed bounded seas for PEGs. However, the essence of bounded seas is not in the grammar formalism used but in the fact that water is specific for each island and it is computed automatically from a stack of invoked expressions. We argue that bounded islands are useful for Context Free Grammars (CFGs) [7] as well.

The key difference between PEGs and CFGs is that CFGs may return ambiguous results whereas PEGs cannot. Implementing an island grammar as a CFG may lead to ambiguous results even though only one of the results is desired. The undesired, remaining results are present only because of vaguely-defined water. This is problematic since it is hard to decide which of the results is the correct one. Bounded seas eliminate ambiguities by adopting a more precise definition of water.

Generalized LL Parsing [8,9] can handle any CFG, allows all the choices of CFGs to be explored in parallel, and, in case of ambiguity returns all the possible results. Bounded seas can be implemented in a GLL parser because their top-down nature allows for a stack of parsing expressions and they support syntactic predicates used in a boundary.

5.4 Terminal Expressions in NEXT

Let us consider the case when NEXT returns a set of terminal expressions. In that case the NEXT function behaves similarly to the FOLLOW function from LL parsing theory [10,11,12]. If $e_i \in NEXT(e)$ is a terminal symbol we avoid the problems with another sea in a boundary of a sea. It simplifies the implementation of bounded seas. On the other hand, the first terminal is only an approximation of the following expression and it does not provide enough precision.

To illustrate, let us return to the grammar from Listing 1.7. Suppose that $NEXT($ `class` $)$, instead of returning a set of parsing expressions, returns a set of first terminals { `'class'` , `'struct'` , `'public'` }. If there are other elements in the input that start with `'public'` (*e.g.*, '*public int i = 0.*'), they are indistinguishable from the `mainMethod` from the point of view of the NEXT function.

5.5 Limitations

To compute $NEXT(e_1)$ in a sequence e_1e_2 we need to know the e_2. Yet in some cases, *e.g.*, in monadic parser combinator [3] libraries, the right-hand-side of a sequence is a closure such as:

```
p1 >>= \\result -> p2
```

This means that `e2` is unknown until the result of `e1` is known. In case `e1` is a bounded sea, the result of `e1` cannot be computed before we know `e2`.

Therefore our approach is only applicable if we can compute the e_2 in a sequence e_1e_2 before parsing the e_1. This prevents our solution from being used in some libraries such as the monadic libraries mentioned earlier.

This also limits (but does not forbid) use in context-sensitive grammars, where e_2 depends on a result of e_1. The context sensitive rules such as $e_1e_2e_3$ where e_3 depends on a result of e_1 and e_2 is a bounded sea are allowed. Because we use a stack and we compute $NEXT$ during parsing, e_3 can be computed when e_2 starts the parsing.

Bounded seas do not allow for context-sensitive dependencies between an island and its border with one exception. When a sea is bounded by another sea, we disable water if another water is already invoked at the same position.

6 Related Work

Noise Skipping Parsing. GLR* is a noise-skipping parsing algorithm for context-free grammars able to parse any input sentence by ignoring unrecognizable parts of the sentence [13]. The parser nondeterministically skips some words in a sentence and returns the parse with fewest skipped words. The parser is a modification of Generalized LR (Tomita) parsing algorithm [14].

The GLR* application domain is parsing of spontaneous speech. Contrary to the bounded seas presented in this work, GLR* itself decides what is a noise (water in our case) and where it is. In case of bounded seas the positions of a noise (water) are explicitly defined.

Fuzzy Parsing. The term fuzzy parser has been coined by Sniff [15], a commercial C++ IDE, that uses a hand-made top down parser. Sniff can process incomplete programs or programs with errors by focusing on symbol declarations (classes, members, functions, variables) and ignoring function bodies. In linguistics or natural language processing [16], the notion of fuzzy parsing corresponds to an algorithm that recognizes fuzzy languages.

The semi-formal definition of a fuzzy parser was introduced by Koppler [17]. Fuzzy parsers recognize only parts of the language by means of an unstructured set of rules. Compared with whole-language parsers, a fuzzy parser remains idle until its scanner encounters an anchor in the input or reaches the end of the input. Thereupon the parser behaves as a normal parser. Contrary to bounded seas, the fuzzy parsers represent a rather lexical approach, since they do not take a context-free structure into the account.

Island Grammars. Island grammars were suggested by Moonen in 2001 [1] as a method of semi-parsing to deal with irregularities in the artifacts that are typical for the reverse engineering domain. The idea of Moonen is based on a special syntactic rule called *water* that can accept any input. Water is annotated with a special keyword `avoid` that will ensure that water will be accepted only if there is no other rule that can be applied.

Contrary to Moonen, we propose boundaries (based on the NEXT function) that limit the scope in which water can be applied. Because each island has a different boundary, our solution does not use the single water rule; instead our water is tailored to each particular island.

Skeleton Grammars. Skeleton grammars [18] address the issue of false positives and false negatives when performing tolerant parsing by infering a tolerant (skeleton) grammar from a precise baseline grammar.

Our approach tackles the same problem as skeleton grammars: improving the precision of island grammars. They both maintain the composability property and both can be automated. The skeleton grammars use the standard first follow sets known from standard parsing theory [10,11,12] for synchronization (in similar way as we use a boundary). Contrary to skeleton grammars, bounded seas do not require a baseline grammar. Instead, bounded seas have to use a

NEXT set (returning set of expressions instead of set of tokens). Only this way can they achieve the required precision.

7 Conclusion

In this paper we presented bounded seas — composable, reusable, robust and easy to use islands. Contrary to traditional approach of island parsing, bounded seas do compute the scope where water can consume the input. We extended the semantics of PEGs to implement useful and practical bounded seas. The computation of a boundary is done by NEXT function, which inspired by the follow function from a standard parsing theory. The automation of the process that creates the bounded sea ensures that the bounded seas are easy to use and are not error-prone. The bounded seas presented in this work are context-sensitive.

As a validation of bounded seas composability and reusability we provide an implementation of bounded sea as a parser combinator in a PetitParser framework.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

References

1. Moonen, L.: Generating robust parsers using island grammars. In: Burd, E., Aiken, P., Koschke, R. (eds.) Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001), pp. 13–22. IEEE Computer Society (2001), doi:doi:10.1109/WCRE.2001.957806
2. Renggli, L., Ducasse, S., Gîrba, T., Nierstrasz, O.: Practical dynamic grammars for dynamic languages. In: 4th Workshop on Dynamic Languages and Applications (DYLA 2010), Malaga, Spain (2010)
3. Hutton, G., Meijer, E.: Monadic parser combinators, Tech. Rep. NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996)
4. Frost, R., Launchbury, J.: Constructing natural language interpreters in a lazy functional language. Comput. J. 32(2), 108–121 (1989), doi:doi:10.1093/comjnl/32.2.108
5. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: POPL 2004: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 111–122. ACM, New York (2004), doi:10.1145/964001.964011
6. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proceedings of the European Software Engineering Conference (ESEC/FSE 2005), pp. 1–10. ACM Press, New York (2005), doi:10.1145/1095430.1081707 (invited paper)

7. Chomsky, N.: Three models for the description of language. IRE Transactions on Information Theory 2, 113–124 (1956), http://www.chomsky.info/articles/195609--.pdf
8. Scott, E., Johnstone, A.: Gll parsing. Electron. Notes Theor. Comput. Sci. 253(7), 177–189 (2010), doi:10.1016/j.entcs.2010.08.041
9. Grune, D., Jacobs, C.J.: Generalized LL Parsing. In: Parsing Techniques — A Practical Guide, vol. 1, ch. 11.2, pp. 391–398. Springer (2008)
10. Grune, D., Jacobs, C.J.: Deterministic Top-Down Parsing. In: Parsing Techniques — A Practical Guide, vol. 1, ch. 8, pp. 235–361. Springer (2008)
11. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison Wesley, Reading (1986)
12. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation and Compiling Volume I: Parsing. Prentice-Hall (1972)
13. Lavie, A., Tomita, M.: Glr* - an efficient noise-skipping parsing algorithm for context free grammars. In: Proceedings of the Third International Workshop on Parsing Technologies, pp. 123–134 (1993)
14. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Norwell (1985)
15. Bischofberger, W.R.: Sniff: A pragmatic approach to a C++ programming environment. In: C++ Conference, pp. 67–82 (1992)
16. Asveld, P.: A fuzzy approach to erroneous inputs in context-free language recognition. In: Proceedings of the Fourth International Workshop on Parsing Technologies IWPT 1995, pp. 14–25. Institute of Formal and Applied Linguistics, Charles University (1995)
17. Koppler, R.: A systematic approach to fuzzy parsing. Software: Practice and Experience 27(6), 637–649 (1997), doi:10.1002/(SICI)1097-024X(199706)27:6<637:AID-SPE99>3.0.CO;2-3
18. Klusener, S., Lämmel, R.: Deriving tolerant grammars from a base-line grammar. In: Proceedings of the International Conference on Software Maintenance (ICSM 2003), pp. 179–188. IEEE Computer Society (2003), doi:10.1109/ICSM.2003.1235420

A Parsing Expression Grammars

PEGs were first introduced by Ford [5] and the formalism is closely related to top-down parsing. PEGs are syntactically similar to CFGs [7], but they have different semantics. The main semantic difference is that the choice operator in PEG is ordered — it selects the first successful match — while the choice operator in CFG is ambiguous. PEGs are composed using the operators in Table 5.

PEG Formalization

Definition 7 (PEG Definition). We use the standard definition as suggested by Ford [5]. A *parsing expression grammar* (PEG) is a 4-tuple G = {N, T, R, e_s}, where N is a set of nonterminals, T is a set of terminals, R is a set of rules, e_s is a start expression. $N \cap T = \emptyset$. Each r ∈ R is a pair (A, e), which we write A ← e, where A ∈ N, e is a parsing expression. The parsing expressions are defined inductively, if e, e_1 and e_2 are parsing expressions, then so is:

Table 5. Operators for constructing parsing expressions

Operator	Description
' '	Literal string
[]	Character class
·	Any character
(e)	Grouping
$e?$	Optional
$e*$	Zero-or-more repetitions of e
$e+$	One-or-more repetitions of e
$\&e$	And-predicate, does not consume input
$!e$	Not-predicate, does not consume input
$e_1\ e_2$	Sequence
e_1 / e_2	Prioritized choice

- ϵ, the empty string
- a, any terminal where $a \in T$
- A, any nonterminal where $A \in N$
- e_1e_2, a sequence
- e_1/e_2, a prioritized choice
- $e*$, zero or more repetitions
- $!e$ a not-predicate

The following operators are syntactic sugar:

- **Any Character:** · is character class containing all letters
- **Character class:** $[a_1, a_2, ...a_n]$ character class is $a_1/a_2/../a_n$
- **Optional expression:** $e?$ is e_d/ϵ, where e_d is desugaring of e
- **One-or-more repetitions:** $e+$ is e_de_d*, where e_d is desugaring of e
- **And-predicate:** $\&e$ is $!(!e_d)$, where e_d is desugaring of e

We will use text in quotation marks to refer to terminals *e.g.*, `'a'`, `'b'`, `'class'`. We will use identifiers `A`, `B`, `C`, `class` or `method` to refer to nonterminals. We will use `e` or indexed `e`: `e1`, `e2`, ... to refer to parsing expressions.

Definition 8 (PEG Semantics). To formalize the syntactic meaning of a grammar G = {N, T, R, e_s}, we define a relation $\Rightarrow_G$ from pairs of the form (e, x) to the output o, where e is a parsing expression, $x \in T^*$ is an input string to be recognized and $o \in T^* \cup \{f\}$ indicates the result of a recognition attempt. The distinguished symbol $f \notin T$ indicates failure. For $((e, x), o) \in \Rightarrow_G$ we will write $(e, x) \Rightarrow o$.

$$\textbf{Empty:}\frac{x \in T^*}{(\epsilon, x) \Rightarrow \epsilon}$$

$$\textbf{Terminal (success case):}\frac{a \in T, x \in T^*}{(a, ax) \Rightarrow a}$$

$$\textbf{Terminal (failure case):} \frac{a \neq b, \quad (a, \epsilon) \Rightarrow f}{(a, bx) \Rightarrow f}$$

$$\textbf{Nonterminal:} \frac{A \leftarrow e \in R \quad (e, x) \Rightarrow o}{(A, x) \Rightarrow o}$$

$$\textbf{Sequence (success case):} \frac{\begin{array}{c}(e_1, x_1x_2y) \Rightarrow x_1 \\ (e_2, x_2y) \Rightarrow x_2\end{array}}{(e_1e_2, x_1x_2y) \Rightarrow x_1x_2}$$

$$\textbf{Sequence (failure case 1):} \frac{(e_1, x) \Rightarrow f}{(e_1e_2, x) \Rightarrow f}$$

$$\textbf{Sequence (failure case 2):} \frac{(e_1, x_1y) \Rightarrow x_1 \quad (e_2, y) \Rightarrow f}{(e_1e_2, x_1y) \Rightarrow f}$$

$$\textbf{Alternation (case 1):} \frac{(e_1, xy) \Rightarrow x}{(e_1/e_2, x) \Rightarrow x}$$

$$\textbf{Alternation (case 2):} \frac{(e_1, x) \Rightarrow f \quad (e_2, x) \Rightarrow o}{(e_1/e_2, x) \Rightarrow o}$$

$$\textbf{Repetitions (repetition case):} \frac{\begin{array}{c}(e, x_1x_2y) \Rightarrow x_1 \\ (e*, x_2) \Rightarrow x_2\end{array}}{(e*, x_1x_2y) \Rightarrow x_1x_2}$$

$$\textbf{Repetitions (termination case):} \frac{(e, x) \Rightarrow f}{(e*, x) \Rightarrow \epsilon}$$

$$\textbf{Not predicate (case 1):} \frac{(e, xy) \Rightarrow x}{(!e, xy) \Rightarrow f}$$

$$\textbf{Not predicate (case 2):} \frac{(e, xy) \Rightarrow f}{(!e, xy) \Rightarrow \epsilon}$$