Parsing Ruby with an Island Parser

Bachelor Thesis

Rathesan Iyadurai
from
Ersigen BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

18. April 2016

Prof. Dr. Oscar Nierstrasz
Jan Kurš

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Ruby is a challenging language to parse because of a large and ambiguous grammar that is not only scarcely documented but subject to change on minor releases.

Therefore, third-party Ruby parsers for software analysis tools either translate the standard implementation's parser code rule for rule or deal with compatibility issues.

In this thesis we propose an alternative approach by using the island grammar methodology. Island grammars only extract the structures of interest (islands) with precise rules while skipping over the rest (water). This makes them fitting for quick and robust data extraction for software development tools.

We present a parsing expression grammar based island grammar for Ruby that is able to extract classes, modules and methods. As verification we measure precision, recall and error rate of our parser's output to one generated by the jRuby parser.

Contents

1
Introduction

Writing a parser for Ruby is particularly challenging, because of its ambiguous syntax and lack of formal specification. In this thesis, we demonstrate how to approach the challenges of implementing a parser that is able to extract a subset of the Ruby language while keeping a grammar as minimal as possible.

Having a method to partially extract structure can be beneficial in any situation where a robust parser for the language is not available or where it would take too much effort to implement one. We focus on extracting class, module and method definitions. They provide a source model that can be useful for software analysis platforms, such as Moose[1], or other reengineering or developer tooling purposes.

In order to generate a parser that is capable of extracting only class, module and method definitions, while ignoring any other unrecognised input, we use an island grammar [7]. An island grammar describes constructs of interest (the islands) with precise productions and uses more lenient, imprecise productions to describe the rest (the water). In our case classes, modules and methods represent the islands. We implement such an island grammar with the Pharo[2] implementation of PetitParser[3], a dynamic parsing framework.

We validate our approach by measuring the precision, recall and error rate of our parser against the jRuby parser.[4] After each improvement over a naive implementation, we report how the new rules affect the results. Listing 1 shows Ruby code with a

[1] `http://www.moosetechnology.org/`
[2] `http://pharo.org/`
[3] `http://scg.unibe.ch/research/helvetia/petitparser`
[4] `https://github.com/jruby/jruby-parser`

class nested inside of another class. The parser's output should preserve nesting and differentiate instance methods from class methods, as seen in Listing 2.

```
class Dog

  attr_accessor :age

  def initialize(age)
    raise "NOO" if age < 1

    @name = "foo class bar"
    @age  = age
  end

  class NestedDog

    def self.class_method
      puts "I'm in a class method"
    end

  end

end
```

Listing 1: A class `Dog` with a nested class `NestedDog`

```
Dog(
  .initialize
  NestedDog(
    .self.class_method
  )
)
```

Listing 2: Desired output from parsing the Ruby code from Listing 1

The output as seen in Listing 2 is then compared to the same output generated by the jRuby parser in order to measure precision, recall and error rate.

Chapter 2 gives an introduction to parsing expression grammars and PetitParser since they serve as building blocks for our island parser. Chapter 3 describes some of the problems of (semi) parsing Ruby, while also introducing island grammars as a tool that might ease the pain. It also explains how to find grammar specifications for Ruby. Chapter 4 presents an overview of our island parser implementation. It lists the major challenges, how we tackle them, and how each solution helps our parser perform against the jRuby parser. The full source code can be found on SmalltalkHub[5], but we also present a list of the required production rules in appendix A.

Chapter 5 concludes this thesis.

[5] `http://smalltalkhub.com/#!/~radi/RubyParser`

2
PetitParser

In order to better understand the Ruby grammar illustrations in the rest of the thesis, this chapter serves as a brief introduction to parsing expression grammars (PEGs) [3] and PetitParser [5, 8], a parsing framework for Pharo, which is strongly based on PEGs. PetitParser allows us to generate an island parser for Ruby by implementing Ruby's grammar rules in PEG-like Smalltalk syntax.

2.1 Parsing Expression Grammars

Parsing Expression Grammars (PEGs), as introduced by Ford [3], offer a rule system to *recognize* language strings, while previously known systems, such as context free grammars (CFGs) [1], offer a generative way to describe a language. This recognition-based system corresponds closely to top-down parsing, making PEGs more suitable for describing programming languages. Moreover, PEGs are unambiguous, meaning a successfully parsed string has one parse tree, whereas CFGs can lead to multiple trees.

```
Expr    ← Sum
Sum     ← Product (('+' / '-') Product)*
Product ← Value (('*' / '/') Value)*
Value   ← [0-9]+ / '(' Expr ')'
```

Listing 3: PEG describing basic arithmetic operations on natural numbers

Two important concepts in PEGs are *terminal* and *nonterminal* symbols. They are the lexical elements in a grammar. Terminals are the elements of a language and they are mostly represented in single quoted string literals (`' '`), although character classes (`[ ]`) offer a shortcut for allowing a whole range of terminals. Nonterminals are rules that expand to other rules. Listing 3 shows a PEG that recognizes basic arithmetic operations on natural numbers. `'+'` and `')'` are examples for terminals, `[0-9]` is a character class representing numbers from `0` to `9` as terminals and `Expr`, `Sum`, `Product` and `Value` are nonterminals. So the PEG itself is built up by a set of production rules of the form `A ← e`, whereas `A` is a nonterminal and `e` can be another nonterminal, terminal or an expression connected with some operators. Table 2.1 lists all valid PEG operations.

Operator	**Description**
`' '`	Literal string
" "	Literal string
`[ ]`	Character class
.	Any character
(e)	Grouping
$e?$	Optional
$e*$	Zero or more
$e+$	One or more
$\&e$	And-predicate
$!e$	Not-predicate
$e_1\ e_2$	Sequence
e_1/e_2	Prioritized Choice

Table 2.1: PEG operators

One of the crucial differences to CFGs and also the key reason to why PEGs are unambiguous, is the prioritised choice operator `/`. It selects the first match in PEG, meaning that an expression e_1/e_2 successfully returns its result if e_1 succeeds. Should e_1 fail, it attempts e_2 from the start. This behaviour gives programmers better control over which parse tree will be generated from a grammar.

Two very practical operators are $\&$ and !, which act as syntactic predicates. The expression $\&e$ will try to consume `e`, remember whether it was successful or not, and backtrack to the starting point. On the contrary, `!e` fails when $\&e$ succeeds, but succeeds when $\&e$ fails. For instance, the expression (!"′".)$*$ can be used to consume anything until a ′ is encountered, which could prove useful for consuming string literals.

2.2 PetitParser DSL

PetitParser lets us write PEG rules down in Smalltalk code and automatically generates a parser. It is quite different to other popular parser generators, because it combines four alternative parser methodologies, namely scannerless parsers [9], parser combinators [4], parsing expression grammars and packrat parsers [2].

```
value   := #digit asParser plus / expr.
product := value , (($* asParser / $/ asParser) trim, value) star.
sum     := product , (($+ asParser / $- asParser) trim, product) star.
expr    := sum end.
```

Listing 4: A PetitParser implementation of the calculator PEG in Listing 3

Listing 4 shows the a PetitParser implementation of the PEG in Listing 3. The syntax almost feels like a domain specific language (DSL) for writing PEGs thanks to the message send syntax of Smalltalk. Most of the PEG operators are implemented as unary or binary message sends in Smalltalk. PEGs + operator, for instance, is implemented as the unary message `plus`, which can be send to any PetitParser object to create a parser that accepts one or more of whatever the receiver of `plus` accepted. Thus `#digit asParser plus` creates a parser that accepts one or more digits. Ultimately, to find out whether our parser recognizes a string, we send the message `parse: aString` to a parser, *e.g.,* `expr parse: '1 + 1'`.

Terminal Parsers

The `asParser` message is defined on a number of core classes to conveniently convert Smalltalk objects to terminal parsers. Calling `asParser` on a `ByteString` or `Character` instance returns a parser that accepts that string or character, *e.g.,* `'foo' asParser` accepts the string `foo`, `$+ asParser` accepts the character `+`.

`asParser` on `ByteSymbol` returns prebuild parsers depending on the name of the `ByteSymbol`. We have already introduced `#digit asParser` in Listing 4. The resulting parser accepts digits from 0 to 9. Another example is `#any asParser`, which parses any character. Table 2.2 shows commonly used terminal parsers. An exhaustive list can be found by browsing the class side methods of the `PPPredicateObjectParser` class.

Parser	**Description**
`$a asParser`	Accepts the character `a`
`'foo' asParser`	Accepts the string `foo`
`#any asParser`	Accepts any character
`#digit asParser`	Accepts the digits 0-9
`#letter asParser`	Accepts the letter a-z and A-Z
`#word asParser`	Accepts any alphanumeric character
`#newline asParser`	Accepts a newline

Table 2.2: Common PetitParser Terminal Parsers

Parser Combinators

What we refer to as parsers are plain Smalltalk objects in PetitParser and these objects can be combined with message sends. As an example, `$+ asParser` returns a `PPLiteralObjectParser` that accepts the character `+`. To mimic PEGs' `/`, we can simply send the `/` message with another parser object as an argument, hence `$+ asParser / $- asParser` returns a `PPChoiceParser`, which behaves exactly like the PEG `'+' / '-'`. Table 2.3 shows commonly used parser combinators with their PEG equivalents if one exists.

Parser	Description	PEG equivalent
`p1 , p2`	Parses `p1` followed by `p2` (sequence)	$p_1 \, p_2$
`p1 / p2`	Parses `p1`, if `p1` fails move on to `p2` (choice)	p_1 / p_2
`p star`	Parses zero or more of `p`	$p*$
`p plus`	Parses one or more of `p`	$p+$
`p optional`	Parses `p` if possible	$p?$
`p and`	Parses `p` without consuming	$\&p$
`p not`	Parses `p` without consuming and succeeds when `p` fails	$!p$
`p negate`	Consumes one character if parsing of `p` fails	$(!p.)$
`p end`	Parses `p` and succeeds at the end of the input	

Table 2.3: Common PetitParser Combinations and PEG equivalents

Action Parsers

Finding out whether a given string conforms to a grammar might already be satisfactory in some cases, but more than often one wishes to perform actions on recognising a string. PetitParser meets these demands with action parsers. Table 2.4 lists common action messages that can be sent to a parser object.

Actions	Description
`p flatten`	Flattens the result to a `ByteString`
`p trim`	Trims whitespace before and after `p`
`p trim: trimParser`	Trims whatever `trimParser` can consume
`p ===> aBlock`	Calls `aBlock` with a list of nodes passed as a block argument

Table 2.4: Common PetitParser Actions

3
Parsing Ruby in the Wild

Ruby is a dynamic, open source programming language[1] with an easy to read syntax and not so easy to read grammar specifications. In this chapter, we explain why one might want to write a custom parser for a language like Ruby. We also discuss what makes Ruby especially challenging to parse, why we chose the island grammar methodology for our implementation and how to find grammar specifications to support the implementation process.

3.1 Parsing Ruby for Development Tools

Around 2005, a now very popular web framework called Ruby on Rails[2] was released and shed a lot of light on what Ruby was capable of. This made Ruby not only a strong candidate for the web, but a prominent choice as a general-purpose programming language. Since then, countless applications have been written in Ruby and naturally many development tools followed.

Tools that automatically refactor code or generate coverage reports require a full abstract syntax tree representation to comprehend the semantics of a language. To generate an such a tree these tools oftentimes use a full parser derived from the standard

[1] `https://www.ruby-lang.org/en/`
[2] `http://rubyonrails.org/`

implementation[3]. Popular examples for Ruby are parser[4], ruby_parser[5], jruby-parser[6] or the standard library Ripper[7].

One of the complex challenges of parsing Ruby is understanding its local ambiguities. When the parser encounters `x` inside of a method definition, it does not know if `x` is a local variable or a method call, since Ruby allows you to omit the parentheses when calling methods. To deal with ambiguities like these, the parser has to know about all the local variables in scope.

Ruby also provides a lot of different string literals that require context sensitive parsing. An example is the "here document"[8]. It allows you to write multiline blocks of text by delimiting them with an identifier of your choosing as seen in Listing 5.

```
str = <<FOOBAR

I am a very
long, multi-line
text.

FOOBAR
```

Listing 5: Ruby heredoc

Ruby's philosophy is to allow developers to express themselves the way they like. It thereby provides many different ways to express the same thing in the syntax alone. The literals `["a", "b", "c"]` and `%w(a b c)` create the same array of strings. As does `%w|a b c|` or `%w?a b c?`. Keywords that usually enclose lexical closures can be omitted in a short-hand construct as presented in Listing 6.

```
# This conditional
if foo?
  return bar
end

# can be written as
return bar if foo?
```

Listing 6: Two `if` constructs doing the same thing require multiple productions to differentiate them when parsing

[3] https://github.com/ruby/ruby/blob/dfec9d978804fbd53df9e6b93e5801985b2b3130/parse.y
[4] https://github.com/whitequark/parser
[5] https://github.com/seattlerb/ruby_parser
[6] https://github.com/ruby/ruby/blob/dfec9d978804fbd53df9e6b93e5801985b2b3130/parse.y
[7] http://ruby-doc.org/stdlib-2.0.0/libdoc/ripper/rdoc/Ripper.html
[8] http://ruby-doc.org/core-2.2.0/doc/syntax/literals_rdoc.html#label-Here+Documents

To address these challenges most third-party parsers seem to be simply translating the standard implementation parser code to some other parser generator.[9] The main issue with this approach is that they have a hard time keeping up with the standard implementation, since the core developers of Ruby might change syntax in patchlevel releases.[10]

A more robust approach would be to implement custom parsers that are tailored to the tools they serve. These parsers should be able to ignore different dialects and ambiguities in structures their tools are not interested in. A dependency analysis tool for a classical object oriented systems cares about classes and the messages their objects send to each other. Therefore, its parser should only have to recognise and validate classes, methods and method calls. Such a parser can be implemented with a special kind of grammar — an *island grammar*.

3.2 Island grammars

Island grammars, as described by Moonen [6], allow us to extract information from a language without having to know about the full grammar. The main idea is to write precise grammar rules only for the structures of the language we are interested in — the *islands*. For the rest of the language we skip any input until an island occurs. This rest of the language is referred to as *water*.

For Ruby, or any contemporarily used programming language, one can find a full grammar implementation, translate it to the parsing framework of choice and be done with it. This approach would result in a full parser that could easily provide us with the information we need for any kind of software analysis tool. The obvious disadvantage is the enormous sacrifice of time, effort and sanity, but island grammars treasure other unapparent advantages over full grammars. These not only make them an easy choice for our task, but a fast and robust way of partially extracting information from source code of any kind.

Firstly, island parsers are more light-weight than the full implementation as a result of the smaller number of production rules. A smaller grammar also means fewer ambiguities and complexity. Fewer developers are needed to maintain such a parser.

Another advantage is robustness. An island grammar can automatically deal with different dialects and syntax errors in the water as a side effect of how water rules are defined. Say we are only interested in Ruby method definitions. A simplified method definition production starts with recognising the keyword `def` followed by an identifier, some precise argument rules, the method body and eventually an `end` keyword. The method body is water to us, thus we choose a very naive and imprecise rule that should

[9] `https://github.com/seattlerb/ruby_parser/blob/5edec536a1647a64c6c499a9f4dc40a05bf2e9d0/lib/ruby19_parser.y`

[10] `https://github.com/whitequark/parser#compatibility-with-ruby-mri`

consume everything until it encounters the enclosing `end`, *e.g.*, `'end' asParser negate star`. One consequence is that the parser may encounter the old `Hash` literal {`:key => value`} or the newer one introduced in Ruby 1.9 {`key: value`} inside of a method definition and succeed anyhow, even though we never defined the rules for the different notations. As a matter of fact, the method body rule will blindly accept anything until it encounters an `end`, so even future dialects are possibly taken care of already.

Some languages never had their grammar publicised or have a proprietary parser that was hand written with a grammar that never existed beyond the single mind of an already deceased author. Writing a full grammar for such a language might not be possible at all or will definitely not pay off when one is only interested in a small part of it.

Additionally, there are languages that contain other languages, for instance by embedding them in the likes of web templating engines[11]. These languages, when one tries to fully comprehend them, require a parser that can handle multiple languages, whereas an island grammar can be used to define rules for only the language one is interested in. Thanks to the robustness of the water, the other languages will be successfully skipped.

These properties make island grammars a very fitting tool for our undertaking. We write precise production rules for our islands, namely class, module and method definitions, while keeping the rules for other constructs, such as conditional blocks, imprecise. In addition to that, a small set of general rules for the overall structure will suffice to successfully recognise a Ruby program.

3.3 Finding grammar specifications

We have the means necessary to implement a PEG based island parser with PetitParser, but without any knowledge of Ruby's grammar we would have to reinvent the wheel. A formal grammar specification, if one exists, will support writing a parser of our own.

Unfortunately, Ruby does not have a formal grammar specification that all implementations rely on. There are however official documents describing the syntax and semantics of Ruby such as the Ruby Draft Documentation[12]. Another document that can be found is a compact BNF description from the University of Buffalo[13]. Both of them are rather outdated, but the latter provides an incomplete but quick overview.

For the most official and recent version of the grammar we recommend the parser source code of the C implementation[14]. At the time of writing, the file has more than

[11] https://en.wikipedia.org/w/index.php?title=Comparison_of_web_template_engines&redirect=no

[12] https://www.ipa.go.jp/files/000011432.pdf

[13] http://www.cse.buffalo.edu/~regan/cse305/RubyBNF.pdf

[14] https://github.com/ruby/ruby/blob/dfec9d978804fbd53df9e6b93e5801985b2b3130/parse.y

10’000 lines of hard to read code. However, the main repository provides a script that is able to extract a more readable version of the grammar in YACC syntax. Listing 7 presents how to access the latest grammar from the Ruby source code. Credits go to the users in the “Ruby Grammar” stack overflow post.[15]

```
git clone https://github.com/ruby/ruby
cd ruby
ruby sample/exyacc.rb < parse.y
```

Listing 7: Fetching the Ruby source code and extracting the grammar

[15] https://stackoverflow.com/questions/663027/ruby-grammar

4
Implementation

In this chapter we present an overview of the problems we encountered while writing an island parser for Ruby and how we approached them. We would be selling snake oil if we described island grammars as the silver bullet for semi-parsing Ruby. Some of the problems presented below are problems that full parser implementations have to deal with as well, such as context sensitive parsing of string literals. Others are exclusive to island parsing because of the imprecise nature of the water productions.

We directly illustrate the relevant parser code in the solution sections of each problem. For a complete picture we recommend looking at the full source code hosted on SmalltalkHub[1] since the snippets provided here are simplified for illustration purposes.

In order to validate our implementation, we measure precision and recall against the jRuby parser.[2] As sample data we use 100 files from the following six, fairly popular, open source projects Rails[3], Discourse[4], Diaspora[5], Cucumber[6], Valgrant[7] and Typhoeus[8]. We picked those files randomly to provide our parser with a large variety of Ruby code while making sure that they contained structures that are difficult to parse.

We present the results iteratively after each improvement over a naive island grammar.

[1] `http://smalltalkhub.com/#!/~radi/RubyParser`
[2] `https://github.com/jruby/jruby-parser`
[3] `https://github.com/rails/rails`
[4] `https://github.com/discourse/discourse`
[5] `https://github.com/diaspora/diaspora`
[6] `https://github.com/cucumber/cucumber`
[7] `https://github.com/mitchellh/vagrant`
[8] `https://github.com/typhoeus/typhoeus`

4.1 A naive island grammar

We start out with a naive island grammar. In Listing 8 the bold text is what our island productions should consume. Everything else should be consumed by our water.

```
class Dog

  attr_accessor :age

  def initialize(age)
    raise "NOO" if age < 1

    @name = "foo class bar"
    @age  = age
  end

  def bark
    if age > 3
      puts "wuff"
    else
      puts "wiff"
    end
  end

end
```

Listing 8: What we want to extract from this class modelling canine behaviour

The class and method definitions look similar. Their scopes are both enclosed by a keyword `end`. For method definitions for instance, we would implement a rule such as `'def' asParser, identifier, body, 'end' asParser`, whereas `body` would allow nested class and method definitions.

The structures we are not interested in, *e.g.,* `attr_accessor :age`, `puts "wuff"`, should ideally be skipped by a water rule that we wrap our islands with. Intuitively, that rule should be a negation of what our islands are interested in, *e.g.,* `('class' asParser / 'def' asParser / 'end' asParser) negate star`.

Listing Listing 9 shows how we arrange our island and water rules to achieve that behaviour.

```
"Defined in a subclass of PPCompositeParser called RubyGrammar"

primary
  ↑ (water, ((classDef / methodDef , water) star))

water
  ↑ ((classDef / methodDef / 'end' asParser) negate) star
```

Listing 9: Entry point and basic structure of the parser

Validation

We can now generate some output that looks similar to the one presented in Listing 10 from both the jRuby parser and our implementation. This structure allows us to easily measure precision, recall and error rate. As a consequence of nesting the methods inside of classes or modules, we automatically consider scope for true positives.

```
Dog(
  .initialize
  .bark
  .self.some_class_method
)
```

Listing 10: Easily comparable structure

The error rate is the amount of files our parser failed to parse over the total amount of files. For instance, when our parser fails to recognise an enclosing `end` keyword of a method definition, it will throw a parser error.

Table 4.1 presents the results of our first iteration.

Precision	**Recall**	**Error Rate**
1.00	1.00	0.94

Our naive implementation performs terribly against the jRuby parser with an error rate of 0.94, meaning that most files resulted in a PetitParser error. The most obvious and major issue being that it keeps matching the wrong `end`.

4.2 Matching all `ends`

One of the very first and also very obvious problems we encountered was the `end` keyword. There are constructs other than class and method definitions that are enclosed with `end`. Listing 11 illustrates how our naive implementation would wrongly match `ends`.

```
1  class Dog
2
3    attr_accessor :age
4
5    def initialize(age)
6      raise "NOO" if age < 1
7
8      @name = "foo class bar"
9      @age  = age
10   end
11
12   def bark
13     if age > 3
14       puts "wuff"
15     else
16       puts "wiff"
17     end
18   end
19
20 end
```

Listing 11: Wrongly matched `ends`

The `if` conditional is also enclosed with an `end`. When parsing the method `bark`, the parser will preemptively assume it found an `end` on line 17. The actual `end` of `bark` on line 18 will be recognised as the `end` of the class definition, resulting in all following method definitions to be assigned to the outer scope.

Solution

We found the most straight-forward solution to be the identification all constructs that end with an `end` and write rules to capture them correctly, *i.e.,* match all starting keywords with an `end`. The extracted grammar specification from the steps described in section 3.3 will print an exhaustive list of these structures when grepped for `k_end`.

Concretely, this means that we have to extend our island and water production with those rules, as seen in Listing 4.2.

```
primary
  ↑ (water, ((classDef / methodDef / moduleDef
    / conditional / loop / ... , water) star))

water
  ↑ ((classDef / methodDef / moduleDef
    / conditional / loop / ... / 'end' asParser) negate) star
```

Validation

Precision	Recall	Error Rate
1.00	1.00	0.92

Still, our parser fails to parse most of the files. The second major issue we encountered is also an obvious one — string literals.

4.3 String Literals

Keywords might occur inside of strings, comments, regular expressions, *etc.*. Listing 12 shows how our parser will wrongly recognise the start of a class definition and again match the wrong ends as a result.

```
class Dog

  attr_accessor :age

  def initialize(age)
    raise "NOO" if age < 1

    @name = "foo class bar"
    @age  = age
  end

  def bark
    if age > 3
      puts "wuff"
    else
      puts "wiff"
    end
  end

end
```

Listing 12: The occurrence of the keyword `class` inside of a String literal confuses the parser

Solution

In order to ignore strings for instance we need to recognise them precisely. As a consequence we need to add an island for string literals. This means dealing with the following three quirks of Ruby:

1. Various string literals
2. String interpolation
3. Balanced brackets, parentheses and braces

The literals in Listing 13 all create the same string.

```
'foo class bar'
"foo class bar"
%?foo class bar?
%{foo class bar}
%<foo class bar>
%(foo class bar)
%[foo class bar]
%q{foo class bar}
%Q{foo class bar}
<<spongebob.strip
foo class bar
spongebob
```

Listing 13: Ruby offers a wide variety of strings literals

The `%`-literal allows any non-alphanumeric character as a delimiter. Instead of having to hardcode rules for every non-alphanumeric character, we chose to implement a context sensitive rule, that remembers the character after the `%`-sign and utilises it to compose a dynamic parser.

String interpolation in Ruby allows expression substitution inside of the double-quote, `%`-sign and heredoc literal. `"Foo #{expression} bar"` will result in a string with `#{expression}` replaced by whatever string `expression` evaluates to. Since string interpolation is delimited by curly braces and technically allows a Ruby program inside of it, we have to match all curly braces in our parser, too. Thankfully, only opening curly braces can result in closing curly braces.

When the enclosing character of the string literal occurs inside of the string, the programmer needs to escape it, *e.g.,* `"foo \" bar"` or `%(foo \) bar).` That is not necessary when the brackets, parentheses or braces occur in even pairs. The literal `%{foo } bar}` results in a syntax error, while `%{ { foo } bar}` does not. This behaviour can be guaranteed by checking for such a balanced pair when the opening character occurs inside of the string.

4.4 Keyword boundaries

We encountered another keyword related problem outside of string literals when consuming water. Since we need to check for the start or end of an island, which is in most cases a keyword, any keyword that occurs in water might wrongly trigger an island rule. Listing 14 highlights keywords that the water will not consume and either result in a wrong enclosure of an outer island, or the wrong start of a new island.

```
def foo
  endless
  bado
end.class
```

Listing 14: Problematic keyword occurrences in water

Solution

For our parser to recognise keywords correctly the keyword rules have to be surrounded by word boundaries. This means we need to look behind and ahead of a keyword and make sure that it is not surrounded by anything that could be inside of a word. The `do` in `bado` for instance is preceded by the character `a`, thus making this `do` an invalid keyword. The `end` in `end.class` on the other hand is followed by a period, and since a period is not a word character, this `end` is a valid keyword.

The `class` in `end.class` would be categorised as a valid keyword by our rules, because it is preceded by a period. It is however a method call and should thereby be an invalid keyword. To handle this case, we must disallow the period as a boundary as well, except for when it occurs ahead of an `end`. The reason being that Ruby allows you to directly call methods on literals such as the method definition in Listing 14, which might be enclosed by an `end`.

Validation

Precision	**Recall**	**Error Rate**
1.00	1.00	0.62

Recognising all keywords correctly results a major error rate reduction, but there is one last construct our parser has to know about to fully reduce the error rate.

4.5 Modifiers

The most challenging constructs to recognise were statement modifiers. Modifiers are shorthands for conditionals or loops that usually are required to be enclosed with an `end`. Listing 15 demonstrates how an `if` modifier can be used as an alternative to an `if` block.

```
# block version of an if-conditional
if age > 1
  raise "NOO"
end

# modifier shorthand
raise "NOO" if age > 1
```

Listing 15: One-liner thanks to `if`-modifier

Listing 16 shows how our parser, without any knowledge about modifiers, would wrongly attempt to match an `end` keyword with an `if` of a modifier.

```
class Dog

  attr_accessor :age

  def initialize(age)
    raise "NOO" if age < 1

    @name = "foo class bar"
    @age  = age
  end

  def bark
    if age > 3
      puts "wuff"
    else
      puts "wiff"
    end
  end

end
```

Listing 16: Modifier `if` should not need to be enclosed by an `end`

What makes modifiers truly problematic is that without a full grammar, there is no obvious rule that differentiates the keyword of a normal block from the keyword of a modifier. For Listing 16, one could define a rule that recognises a modifier when a modifier keyword occurs inside of a line, *i.e.*, it is not at the beginning of a line. However, to our parser's misfortune, conditional blocks and loops are expressions that can be assigned to a variable for instance. Consequently, not being at the start of a line does not make a keyword a modifier keyword. As an example, in `some_variable = if condition?` the `if` is the start of a conditional block that continues on the next line.

Solution

Admittedly, we have not found a tried-and-true solution for recognising modifiers with a minimal island grammar. Instead, we analysed the usage of the problematic modifier keywords, namely `if`, `unless`, `while` and `until` in a corpus of 50+ popular Ruby libraries. The results showed that the only case where the starting keyword of a conditional block is in the middle of a line, is to assign its result with an equals sign. Therefore, a lookbehind for = sufficed to differentiate a modifier keyword from a regular one. Listing 17 shows the guards we used to achieve satisfactory precision.

```
modifier
  | modifierKeywords newline consumables guarded |

  modifierKeywords := kIf / kUnless / kWhile / kUntil.
  guarded := $= asParser / $+ asParser / $- asParser
            / $/ asParser / $% asParser / $* asParser / $( asParser.

  newline := #newline asParser.

  consumables := ($# asParser / $' asParser / $" asParser
            / $% asParser / reswords) not ,
            (newline / (guarded , modifierKeywords)) negate.

  ^ startOfLine , (consumables / string) plus,
    newline not , guarded not , modifierKeywords
```

Listing 17: Rule for successfully capturing modifier statements

Note that we also guard for characters such as + or − in order to recognise +=, −=, *etc.*

Validation

Precision	**Recall**	**Error Rate**
1.00	1.00	0.00

The modifier implementation reduces the error rate to 0.00 and we achieve full precision and recall.

In the end, we count about 40 production rules in order to precisely extract class, module and method definitions, whereas a lot of the rules are simple one-liners, *e.g.,* for correctly recognising keywords. This is a considerably small amount compared to the 160+ rules of the C parser, which the jRuby parser reimplements.

5
Conclusion

In this thesis we have presented an island grammar for Ruby that is able to extract class, module and method information. We have chosen island grammars, because we only need to know about a subset of Ruby's rather complex, scarcely documented grammar. To validate our implementation we have measured precision, recall and error rate against the jRuby parser.

We have successfully parsed a set of 100 files with full precision and recall without any errors by implementing about 40 production rules. Thus we can conclude that writing an island parser for Ruby is a feasible and robust approach for quick data extraction for software development tools.

Future Work

The island parser presented in this thesis provides an extendable skeleton for developing domain specific parsers for various tools. As an example, one can extend the method parser to handle external method calls and then generate an XML like file for Moose to analyse.[1]

[1] `http://www.themoosebook.org/book/externals/import-export/mse`

Acknowledgment

First and foremost, I would like to thank Jan Kurš for his invaluable support, expert advise and otherworldly patience. He kept motivating me in the darkest of times. I would also like to give special thanks to Professor Oscar Nierstrasz for giving me the opportunity to write my thesis at the Software Composition Group, which without exception consists of delightful people. Last but not least, I would like to thank my dear friend Michael for putting up with my grumblings and helping me pull this through.

A
List of production rules

This appendix lists all production rules of our island parser in alphabetical order. For familiarity, we tried to adopt the naming of the original production rules where possible.

A.1 argdecl

Experimental rule to extract argument declarations in method definitions. Only works when enclosed with parentheses.

```
argdecl
  ↑ (($( asParser , arglist optional , $) asParser)
      ==> #second) / (arglist optional , t)
```

A.2 arglist

Used by `argdecl` to extract a list of method arguments.

```
arglist
  | blockArg splatArg comma identifierSeq option1 |
  blockArg := ($& asParser , identifier) flatten.
  splatArg := ($* asParser , identifier optional) flatten.
  comma := $, asParser.

  identifierSeq := ((comma , identifier) ==> #second) star.

```

```
  option1 := (identifier ,
    identifierSeq ,
    ((comma , splatArg) ==> #second) optional ,
    ((comma , blockArg) ==> #second) optional).

  ↑ option1 /
    (splatArg , ((comma , blockArg) ==> #second) optional) /
    blockArg
```

A.3 begin

Rule to recognise `begin` blocks.

```
begin
  ↑ (kBegin , primary , kEnd)
```

A.4 blockComment

Rule to recognise block comments, which are started by a `=begin` statement.

```
blockComment
  | end |
  end := #startOfLine asParser , '=end' asParser.
  ↑ '=begin' asParser , (#any asParser starLazy: end) , end
```

A.5 bracePair

Rule to recognise and correctly match curly braces. This is necessary because we need a way to differentiate regular curly braces from the ones that enclose string interpolation.

```
bracePair
  ↑ (${ asParser , primary , $} asParser)
```

A.6 classDef

Rule to recognise class definitions.

```
classDef
  ↑ kClass , cpath , superclass optional , primary , kEnd
```

A.7 comment

Rule to recognise single-line comments in order to ignore them.

```
comment
  ↑ $# asParser trimBlanks,
    (#any asParser starLazy: #newline asParser),
    #newline asParser
```

A.8 conditional

Rule to recognise any kind of conditional, which are enclosed by an `end`.

```
conditional
  ↑ ((kIf / kUnless / kCase) ,
    primary ,
    kEnd)
```

A.9 cpath

Rule to recognise class or module names in `classDef` and `moduleDef`. They might be arbitrarily nested with `::`.

```
cpath
  ↑ ('::' asParser optional , identifier ,
    (('::' asParser , identifier) star)) flatten
```

A.10 doBlock

Rule to recognise any kind of `do`-`end` pair, *e.g.,* for anonymous blocks.

```
doBlock
  ↑ (kDo, primary , kEnd)
```

A.11 eigenDef

Rule to recognise a special kind of scope gate that changes `self`. As an example, Using `class << self` results in all methods defined inside that block to belong to the “eigenclass”, making them class methods, even though they look like regular instance methods.

```
eigenDef
  ↑ kClass , '<<' asParser trim , (identifier / kSelf) ,
    primary , kEnd
```

A.12 fname

Rule to recognise method names.

```
fname
  ↑ (operator /
    '..' asParser / '|' asParser / '' asParser / '&' asParser
    / '<=>' asParser / '==' asParser /
    '===' asParser / '=˜' asParser / '>'asParser / '>='asParser
    / '<' asParser / '<=' asParser / '+' asParser / '-' asParser /
    '*' asParser / '/' asParser / '%' asParser / '**' asParser
    / '<<' asParser / '>>' asParser / '˜' asParser / '+@' asParser /
    '-@' asParser / '[]' asParser / '[]=' asParser)
```

A.13 forLoop

Rule to recognise for-loops.

```
forLoop
  ↑ (kFor ,
      anything ,
      kDo optional ,
    primary ,
    kEnd)
```

A.14 heredoc

Rule to recognise here documents. This rule is context sensitive, because one might choose an arbitrary identifier to enclose the literal.

```
heredoc
  | startId end endId start heredocEnd |

  startId := identifier >=> [:context :cc |
    start := cc value.
    context globalAt: #endOfHeredoc put: start.
    start
  ].

  endId := identifier >=> [ :context :cc |
    end := cc value.
    end = (context globalAt: #endOfHeredoc) ifTrue: [
      end
    ] ifFalse: [
      PPFailure
        message: 'identifier does not match heredoc start'
        at: context position.
    ]
  ].

  heredocEnd := #newline asParser , endId trimBlanks.

  ↑ ('<<' asParser ,
    $- asParser optional ,
    startId ,
    (#any asParser starLazy: heredocEnd) ,
    heredocEnd) flatten
```

A.15 identifier

Rule to recognise identifiers. In Ruby, they might also start with an underscore.

```
identifier
  ↑ (#letter asParser / $_ asParser , word star) flatten
```

A.16 ignorable

Rule used by `primary` to ignore uninteresting bits, namely whitespace and comments.

```
ignorable
  ↑ comment / #space asParser
```

A.17 kBegin

The following rules are used to correctly recognise keywords by adding look-ahead and look-behind.

```
kBegin
  ↑ (word not previous, 'begin' asParser , word not) trim ==> #second
```

A.18 kCase

```
kCase
  ↑ (word not previous, 'case' asParser , word not) trim ==> #second
```

A.19 kClass

```
kClass
  ↑ (($. asParser / $: asParser / word) not previous,
    'class' asParser , ($. asParser / $: asParser / word) not)
    trim ==> #second
```

A.20 kDef

```
kDef
  ↑ (word not previous, 'def' asParser , word not) trim ==> #second
```

A.21 kDo

```
kDo
  ↑ (word not previous, 'do' asParser , word not) trim ==> #second
```

A.22 kEnd

```
kEnd
  ↑ (($. asParser / $: asParser / word) not previous, 'end' asParser ,
    ($: asParser / word) not) trim ==> #second
```

A.23 kFor

```
kFor
  ↑ (word not previous, 'for' asParser , word not) trim ==> #second
```

A.24 kIf

```
kIf
  ↑ (word not previous, 'if' asParser , word not) trim ==> #second
```

A.25 kModule

```
kModule
  ↑ (word not previous, 'module' asParser , word not) trim ==> #second
```

A.26 kSelf

```
kSelf
  ↑ (($. asParser / word) not previous, 'self' asParser ,
    ($. asParser / word) not) trim ==> #second
```

A.27 kUnless

```
kUnless
  ↑ (word not previous, 'unless' asParser , word not) trim ==> #second
```

A.28 kUntil

```
kUntil
  ↑ (word not previous, 'until' asParser , word not) trim ==> #second
```

A.29 kWhile

```
kWhile
  ↑ (word not previous, 'While' asParser , word not) trim ==> #second
```

A.30 loop

Rule to recognise while- and until-loops.

```
loop
  ↑ ((kWhile / kUntil) ,
    anything ,
    kDo optional ,
    primary ,
    kEnd)
```

A.31 methodDef

Rule to recognise method definitions. The `self` is not considered to be part of the method name and separately extracted to later determine whether it is a class method or not.

```
methodDef
  ↑ (kDef ,
      ('self.' asParser / (identifier , $. asParser) flatten) optional ,
    fname ,
    argdecl optional ,
    primary ,
    kEnd)
```

A.32 modifier

Rule to recognise modifiers. This is not a precise rule and will certainly fail for some edge cases.

```
modifier
  | modifierKeywords newline consumables guarded |

  modifierKeywords := kIf / kUnless / kWhile / kUntil.
  guarded := $= asParser / $+ asParser / $- asParser
            / $/ asParser / $% asParser / $* asParser
            / $( asParser.
  newline := #newline asParser.
  consumables := ($# asParser / $' asParser / $" asParser
            / $% asParser / reswords) not ,
            (newline / (guarded , modifierKeywords))
            negate.

  ^ (#lenientStartOfLogicalLine asParser ,
    (consumables / string) plus, newline not ,
    guarded not , modifierKeywords)
```

A.33 moduleDef

Rule to recognise module definitions, which basically look like class definitions without the optional super class declaration.

```
moduleDef
  ↑ kModule ,
    cpath ,
    primary ,
    kEnd
```

A.34 operator

Used by `fname` to determine whether we are dealing with a method name.

```
operator
  ↑ (identifier ,
    ($? asParser / $ asParser / $= asParser) optional) flatten
```

A.35 percentSignString

Rule to recognise various %-sign literals, which requires a lot of context-sensitive parsing.

```
percentSignString
  | openSeparator closeSeparator matchedBrackets |
  openSeparator :=
    #any asParser >=> [:context :cc | | ch closeCh openCh |

    ch := cc value.

    closeCh := ch.
    openCh := nil.
    (ch = ${) ifTrue: [ openCh := ${. closeCh := $} ].
    (ch = $<) ifTrue: [ openCh := $<. closeCh := $> ].
    (ch = $[) ifTrue: [ openCh := $[. closeCh := $] ].
    (ch = $() ifTrue: [ openCh := $(. closeCh := $) ].

    ch isAlphaNumeric ifFalse: [
      context globalAt: #openSeparator put: openCh.
      context globalAt: #closeSeparator put: closeCh.
      ch
    ] ifTrue: [
      PPFailure
        message: 'only non-alphanumerics are allowed as separators'
    ]
  ].

  closeSeparator := #any asParser >=> [ :context :cc | | ch |
    ch := cc value.
    ch = (context globalAt: #closeSeparator) ifTrue: [
      ch
    ] ifFalse: [
      PPFailure
        message: 'char does not match closeSeparator'
        at: context position.
    ]
  ].

  matchedBrackets := #any asParser and >=> [ :context :cc |
    | ch open close block |
    open := context globalAt: #openSeparator.
    close := context globalAt: #closeSeparator.
    ch := cc value.
    (ch = open) ifTrue: [  | openParser closeParser |
      openParser := open asParser.
      closeParser := close asParser.

      block := PPDelegateParser new.
```

```
      block setParser:
        openParser,
        (($\ asParser, openParser) / ($\ asParser, closeParser)
          / block / closeParser negate)
          star, closeParser.
      block name: 'matched brackets parser'.
      block parseOn: context.
    ] ifFalse: [
      PPFailure
        message: 'not the opening character'
        at: context position.
    ]
  ].

  ↑ ($% asParser,
    ($q asParser / $Q asParser / $w asParser
      / $W asParser / $r asParser) optional,
    openSeparator,
    ((($\ asParser, closeSeparator) / stringInterpolation
      / matchedBrackets / closeSeparator negate) star),
    closeSeparator) flatten
```

A.36 primary

Entry point of the parser. Encloses islands with water.

```
primary
  ↑ (anything,
      ((classDef / moduleDef / methodDef / eigenDef / loop
        / forLoop / begin / conditional / doBlock / string
        / bracePair / modifier, anything)
        ==> #first trim: ignorable) star)
```

A.37 reswords

Rule to recognise all keywords. Used by `water` to abort consumption.

```
reswords
  ↑ kBegin / kCase / kClass / kDef / kDo / kEnd / kIf / kModule
    / kUnless / kUntil / kWhile
```

A.38 string

Rule to recognise various string and regex literals.

```
string
  | doubleQuotes singleQuotes slash doubleString singleString regexp |
  doubleQuotes := $" asParser.
  singleQuotes := $' asParser.
  slash := $/ asParser.

  doubleString := (doubleQuotes ,
    (($\ asParser , doubleQuotes) / stringInterpolation
      / #any asParser starLazy: doubleQuotes) ,
    doubleQuotes) flatten.

  singleString := (singleQuotes ,
    (($\ asParser , singleQuotes)
      / #any asParser starLazy: singleQuotes) ,
    singleQuotes) flatten.

  regexp := (slash ,
    (('\\' asParser) / ($\ asParser , slash)
      / #any asParser starLazy: slash) ,
    slash) flatten.

  ˆ (doubleString / singleString / percentSignString
    / heredoc / regexp)
```

A.39 stringInterpolation

Used by `string` to recognise string interpolation. Technically, one could define a bunch of classes inside of string interpolation, but we have yet to find sane minds that do that.

```
stringInterpolation
  ↑ '#{' asParser , primary , $} asParser
```

A.40 superclass

Used by `class` to recognise super class declarations.

```
superclass
  ↑ (($< asParser trim , cpath) ==> #second)
```

A.41 t

Used to recognise an end-of-statement.

```
t
  ↑ (#newline asParser / $; asParser) trimBlanks plus
```

A.42 water

Rule to ignore everything we are not interested in.

```
anything
  ↑ ((comment / blockComment / (reswords / string
    / modifier / ${ asParser / $} asParser) negate) star)
```

A.43 word

Rule to recognise a Ruby word, which can contain underscores.

```
word
  ↑ #letter asParser / $_ asParser / #digit asParser
```

Bibliography

[1] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. `http://www.chomsky.info/articles/195609--.pdf`.

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

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

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

[5] Jan Kurs, Guillaume Larcheveque, Lukas Renggli, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. PetitParser: Building modular parsers. In *Deep Into Pharo*, page 36. Square Bracket Associates, September 2013.

[6] Daniel L. Moody. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35:756–779, 2009.

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

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

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