b
UNIVERSITÄT
BERN

# Recognising structural patterns in code

**A parser based approach**

**Bachelor Thesis**

Mathias Fuchs
from
Oberdorf SO, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

16. Mai 2017

Prof. Dr. Oscar Nierstrasz
Dr. Jan Kurš
Haidar Osman

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

# Abstract

Software complexity increases over time. This makes the analysis of systems increasingly difficult.

If we want to analyze a software system and focus on the structure, we require the creation of models. Agile Modeling tries to simplify this task. However, to create these models we need a parser for the source. This parser is sometimes missing or it requires a great deal of effort to create, especially when we are confronted with legacy code, unknown data formats, unknown domain specific languages and sometimes files with mixed languages and log files. To be able to model fast, in the spirit of Agile Modeling, we need to build parsers fast.

In this thesis we therefore investigate the possibility to automatically infer parsers of data serialization formats (*e.g.,* JSON, XML) and output the structure of the given source. To do this we created five grammar based building blocks (List, String, KeyValuePair, Command and Tag). These blocks can be combined in various ways to create the needed parser. This raises writing parsers one level higher, away from the "token" level and also enables us to automate the process.

We can successfully infer structure of formats such as JSON, XML and CSS. Our approach works particularly well on XML files, with an average f-measure of `1`. However it sometimes struggles with CSS, with an average f-measure of `0.88`. This is due a problem with the building blocks. They are based on island grammars and ignore the parts of the code that we do not care about. However it does not skip all the code that we want it to skip.

# Contents

# 1 Introduction

Analyzing complex systems requires parsing the source code to create models that abstract irrelevant information away and focus on the underlying structure of the system.

Creating these parsers is a time consuming activity and does not fit in the idea of Agile Modeling, which is the rapid construction of appropriate software models from a given source code[1]. To be able to build models fast we need a way to build parsers fast.

In this thesis we investigate the possibility to quickly and automatically infer parsers of data serialization formats (*e.g.,* JSON, XML *etc.*) and extract the structure of a given source.

```
{
   "AList":[
      "a",
      "b",
      "c"
   ],
   "key":"value",
   "nested":{
      "key2":"value2"
   }
}
```

Listing 1.1: Small JSON example code for which we want to extract the structure

[1]http://scg.unibe.ch/research/snf13

To understand what structure in these files actually means and how we can represent it, we analyzed various data serialisation formats (JSON, XML, CSS, TEX, HTML, BIBTEX, protocol buffer). The example in Listing 1.1 is a JSON source in its raw form.

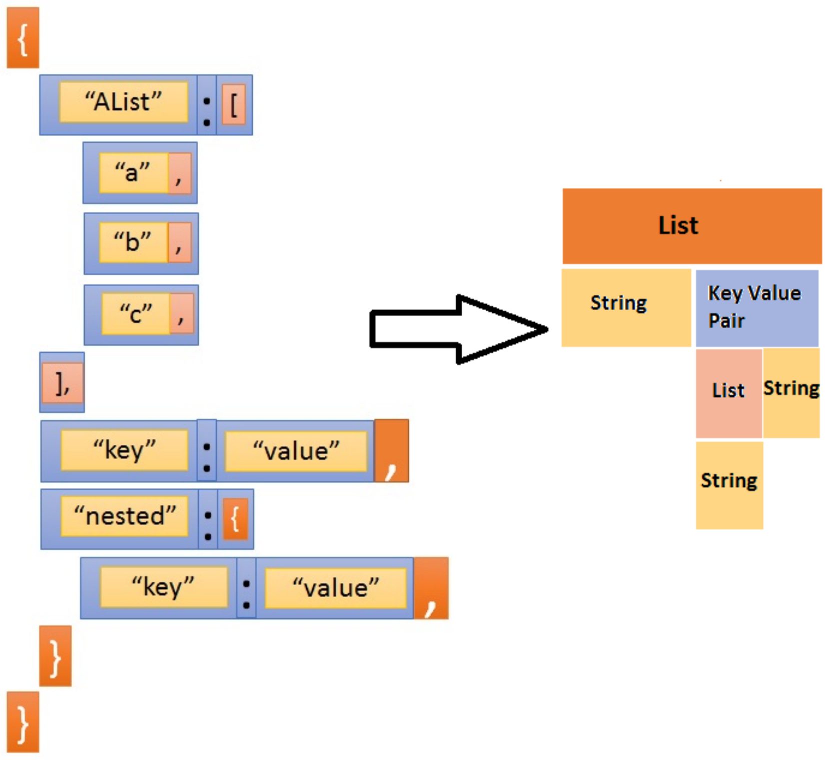


Figure 1.1: JSON example with different structural elements colored and labeled

When we look at Figure 1.1, which represents the code after analyzing it, we can easily see the structure of the file. We define structure as a set of containment rules *i.e.,* nodes that contain other nodes. An example of a possible tree-like structure representation can be found in Figure 1.3.

The structure of the given JSON example can be broken down to three different *building blocks*, which are commonly known programming elements: List, String, Key-Value Pair. The blocks also repeat themselves, because there can be nested structures.

After doing this analysis for multiple data serialisation formats, we had the idea to break down the whole process of parser building, away from the token representation of parsers, onto a higher level of these simple building blocks. This is an important step. It marks the core of our idea, because it enables us to automate the process of parser building and therefore recognize structure automatically.

We call these building blocks: *ParserFactories* (factories). We built five main factories: *ListParserFactory, StringParserFactory, KeyValueFactory, CommandParserFactory,* and *TagParserFactory*. Examples of the first three can be found in Figure 1.1.

The CommandParserFactory is used to recognize structure that can for example be found in CSS *e.g.,*

```
.selector {
    key:value;
    key-2:12px; }
```

Listing 1.2: Small CSS example

We used the word command in the name, because calling upon the key, executes its arguments. A command gives an order to its arguments.
The TagParserFactory is used to recognize structure that can for example be found in XML *e.g.,*

```
<a>b</a>
```

Listing 1.3: Small XML example

We used the word tag in the name, because we first found this element in XML files, and XML objects are always wrapped in tags[2].

We found the structure of the building blocks throughout all formats we analyzed and are able to build parsers using the building blocks for all these formats.

To implement these building blocks, which only parse the relevant structure that we need, we used island grammars [9] because a complete parser is typically not required [10]. Island grammars help us separate the parts of interest from the uninteresting parts. The parts that interest us are called *islands*. Everything else that is skipped is called *water*. The factories are implemented in Pharo[3] using PetitParser[4].

The process works by taking any given data serialisation file as input. Then we automatically build various parsers that could match the given input by using the factories. Each of these parsers tries to parse the given file and outputs the results. The results are ordered by validity and the best fit should appear on the top.

[2]`https://www.w3schools.com/xml/xml_syntax.asp`
[3]`http://pharo.org`
[4]`http://scg.unibe.ch/research/helvetia/petitparser`

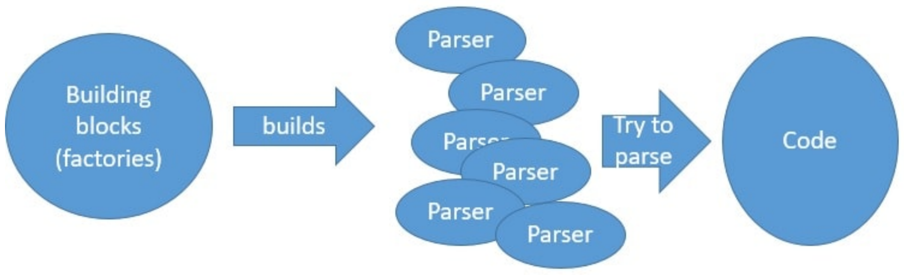


Figure 1.2: Factories produce parsers that try to parse the given source

In the end we are able to extract a human readable structure. In the following figure you see the structure representation of our small JSON example in a tree-like form:

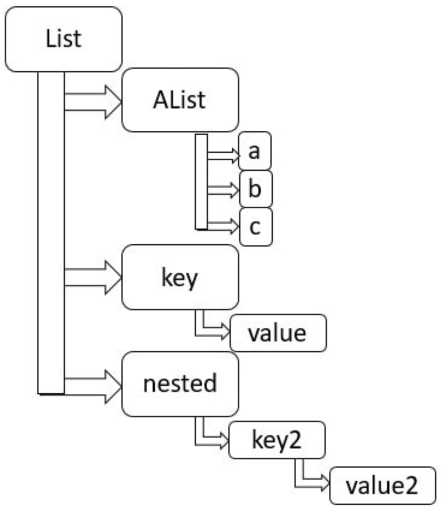


Figure 1.3: Structure of the JSON example in the form of a containment tree

This output represents the underlying structure and therefore facilitates model creation.

As a proof of concept we implemented Crassula which contains all the factories and combines them in a brute force way and outputs a list of different parsers that all try to parse the given code and output the structure.

For the validation we did an experiment where we measured the precision, recall and f-measure of Crassula against a correct parser. To do this we used ten files of XML, JSON and CSS and compared the results of the built parser against the correct parser for each file.

# 2 Related work

In this chapter we give an overview over other approaches towards structure recognition in code. Most of the work towards recognising structure in languages is done in the field of natural language processing. However for software languages there are not many approaches, especially not parser based ones.

**Example-Driven Reconstruction of Software Models:** This approach proposed by Nierstrasz *et al.* [10], proposes a way to rapidly and incrementally develop a parser by using the following steps:

- Specify mappings from source code examples to model elements
- Use the mappings to automatically generate a parser
- Parse as much code as possible
- reuse the exceptional cases to develop new example mappings
- iterate

The main benefits of this approach are the following:

- The engineer specifies mappings instead of grammar rules to build the parser. So he does not need to be an expert in parser technology.
- The model-building parser is developed quickly and iteratively.
- There is no need for a single consistent grammar. Multiple parsers based on different sets of examples can be used to parse the code base with different strategies.

**Autogram:** In the paper Mining Input Grammars from Dynamic Taints [5], the authors proposed a tool called *Autogram*. Autogram automatically infers grammars by leveraging program execution. During execution time they track how the program is executed and from this they can derive the grammar. To track the execution steps Autogram uses dynamic tainting, which is a form of information flow analysis during the execution time of a program. In dynamic tainting we follow the execution flow of a program by tainting (marking) objects which then are tracked along the execution of the program.

**A Text Pattern-Matching Tool based on Parsing Expression Grammars:** This approach was introduced by Ierusalimschy[6]. Current pattern-matching tools often use regular expressions. But regular expressions have proven to be too weak for the task of pattern matching, since some patterns are hard or impossible to describe. Often tools that use regular expressions add a set of ad-hoc features to help the cause. The approach proposed by Ierusalimschy resulted in the library LPEG for the Lua scripting language. LPEG modifies slightly the original PEG syntax. In LPEG expressions are the main concept, not grammars. Grammars are only used when they are needed. An example of an LPEG grammar is:

```
pattern <- grammar/ simplepatt
grammar <- (nonterminal '<-' sp simplepatt)+
simplepatt <- alternative ('/' sp alternative)*
alternative <- ...
```

Listing 2.1: A simple example of a LPEG Grammar

So LPEG still accepts complete grammars. However it also accepts simple patterns like `[a-z]+` which has to be written as `S <-[a-z]+` to be accepted as a PEG.

**Recognising structural patterns in code. A k-means clustering approach:** In Walker's thesis [12] a k-means clustering algorithm is used to recognize structure of various (unknown) programming langauges. To be able to do a k-means clustering the data has to be in a vector representation. Three different representers are created and used to do the clustering [12]:

- **A type representer** creates a vector representation according to the tokens of the statement and creates the numbers for the vector accordingly. One example of a possible type representer: `0` is assigned for words, `1` for numerals and `2` for punctuation *e.g., if (x >1)* {*return 0;*} results in the vector: *0 2 0 2 1 2 2 0 1 2 2*.
- **A distance representer** calculates the distance information from a created type representer.
- **A weight representer** works as a filtering mechanism. It can assign weight to specific parts of a representation.

To cluster properly the parameter K has to be known. K is the number of clusters created. However in this case K is unknown. To calculate K algorithmically the Elbow method[1] is used. Four out of 24 possible representations of the approach worked fairly well for clustering on Java, having a v-measure[2] score equal or higher than 0.66. The representations that do well on Java also do a good job, clustering C#, XML and ABACUS log files.

**Automatic Token Classification. An attempt to mine useful information for parsing:** In this thesis [4] an approach was made to automatically infer keywords from a given source code to simplify the process of parser building for a given language.

Four different approaches are used:

- **The global method** assumes that keywords appear most frequently in the source code.
- **The coverage method** assumes that keywords appear in most files.
- **The newline method** assumes that keywords appear most frequently at the beginning of a newline.
- **The indent method** is an extension to the newline method. It expects keywords to appear at the beginning of a newline but before an indent.

The indent method is the best method overall, however it has the big limitation that it only recognizes the keywords that are at the beginning of a line. Therefore many other keywords are lost.

[1] `https://en.wikipedia.org/wiki/Elbow_method_(clustering)`

[2] `http://www1.cs.columbia.edu/~amaxwell/pubs/clustering_metric_paper.pdf`

# 3 Technical background

In this chapter we give the reader the necessary technical background to understand the thesis.

## 3.1 Parsing Expression Grammar

Parsing Expression Grammars (PEGs) were initially introduced by Bryan Ford [2]. PEGs are closely related to the family of top-down parsing languages and show great similarities to context free grammars (CFGs)[1]. A Context free grammar is, as the name says, context free. This means that a CFG only has one non-terminal on the left side and it has no context *e.g.,* $A \rightarrow bC$, where `A` and `C` are non-terminals and `b` is a terminal symbol. CFGs were originally designed as a formal tool for modeling and analyzing natural languages. They were quickly adopted for machine-oriented languages. However CFGs are not a perfect fit, because they are ambiguous whereas machine-oriented languages aim to be precise and unambiguous [2]. The PEG is basically an Extended Backus-Naur form (EBNF)[2], the meaning of which is determined by a top-down interpreter. The interpreter works from top-to-bottom and left-to-right [3]. PEGs can not be ambiguous. This means if a string parses, it has exactly one valid parse tree [3].

PEGs consist of:

- A finite set of non-terminal symbols.

[1]`https://en.wikipedia.org/wiki/Context-free_grammar`
[2]`https://en.wikipedia.org/wiki/Extended_backus-Naur_form`

- A finite set of terminal symbols that is disjoint from the set of non-terminals.
- A finite set of parsing rules
- An expression marked as the starting expression

The following table are the operators that are allowed in PEGs [2]:

| Operator | Type | Description |
|---|---|---|
| ’ ’ | primary | Literal String |
| ” ” | primary | Literal String |
| [ ] | primary | Character class |
| . | primary | Any character |
| (e) | primary | Grouping |
| e? | unary suffix | Optional |
| e* | unary suffix | Zero-or-more |
| e+ | unary suffix | One-or-more |
| &e | unary prefix | And-predicate |
| !e | unary prefix | Not-predicate |
| $e_1$ $e_2$ | binary | Sequence |
| $e_1$/ $e_2$ | binary | Prioritized Choice |

Table 3.1: Operators for Constructing Parsing Expression [2]

The main reason that PEGs are unambiguous is the *Prioritized Choice operator /* . If one of the alternative choices of an expression succeeds, it consumes what it recognized and continues to the next rule. If it recognizes nothing, it consumes nothing *e.g.,* if we have the expression $e_1$ / $e_2$, and $e_1$ succeeds, then it returns the result of $e_1$. If $e_1$ fails, it continues and tries to parse $e_2$. If $e_2$ succeeds it returns $e_2$. If $e_1$ and $e_2$ fail, it fails.

A small example to illustrate[3]:

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

Listing 3.1: Basic example of a parsing expression grammar (PEG)

In the above example non- terminal symbols are represented by a sequence of characters. They are the rules that expand to other rules. In this case: *Expr, Sum, Product, Value*.

Terminal symbols are mostly represented as strings in single quotes *e.g.,*’ ... ’ or single characters like digits or letters. PEGs can also contain character classes (see Table

[3]https://en.wikipedia.org/wiki/Parsing_expression_grammar

3.1) like `[0-9]` or `[a-z]`. These character classes are a known structure from regular expressions.

PEGs give us rules in the form of `R ← e`, where `R` is the name of the rule and represents a non-terminal, and `e` is the expression. The expression itself can reference non-terminals or terminals. This means the expression can refer to itself or other expressions.

## 3.2 Island grammars

*"An island grammar is a grammar that consists of detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water)"* [9]. Making use of the island grammar approach makes sense, if we only want to know about a part of the underlying syntax. While parsing, everything that is not a recognized input will be skipped.

Let us assume that our task is only to extract the descriptors which have a property[4] that starts with: key, from a given CSS file:

```
.selector {
    doNotCare:value;
    key:value1;
    doNotCare2:value2;
}
```

Listing 3.2: CSS example of which key:value1 should be extracted

In this case we do not care about the other descriptors (water) and only care about the descriptor whose property starts with key (island).

The island grammar for this example looks like this:

```
start ← content
content ← selector '{' water '}'
selector ← (!'{' ·)*
water ← (!'key' ·)* keyValuePair (!'}' ·)*
keyValuePair ← 'key' ':' value ';'
value ← (letter/number)*
letter ← [a-zA-Z]
number ← [0-9]
```

Listing 3.3: A simple example island grammar written as a PEG

[4]Words for CSS syntax found on:`https://www.w3schools.com/css/css_syntax.asp`

This grammar uses a basic definition of water. The `water` rule looks for something that is not `key`, then accepts it (with `.`) and repeats the process until `key` is found *i.e.,* it skips everything until `key` is found.

However in some cases such a naive implementation of an island grammar can result in problems. In particular when there is more than one island of interest *e.g.,* when we are interested in both selectors and key-value pairs. The approach of bounded seas [8] proposes a solution.

### 3.2.1 bounded seas

Usually water is defined as the negation of the island (see Listing 3.3), so when developing an island grammar, a language engineer has to create the water rule for each island individually.

To illustrate how fast the whole process can become complex, consider the following example:

```
.selectorA{
    doNotCare:value;
}
.selectorB{
    key:value1;
}
```

Listing 3.4: CSS example for which the grammar from Listing 3.3 fails

In this case the grammar from Listing 3.3 takes the rule-set with the `selectorA` and says that it found `key:value1` there. It does not realize that during its search, the first rule-set already closed and that the found declaration is in the rule-set with the declaration named `.selectorB`.

To solve this problem we have to adjust the grammar:

```
start ← content*
content ← selector '{' water '}'
selector ← (!'{' ·)*
water ← (!'key' !'}' ·)*
            keyValuePair
         (!'key' !'}' ·)*
keyValuePair ← 'key' ':' value ';'
keyName ← letter*
value ← (letter/number)*
letter ← [a-zA-Z]
number ← [0-9]
```

Listing 3.5: Adjusted CSS grammar that takes multiple rule-sets into account

As we can see in the adjusted grammar in Listing 3.5, the water rule is more complicated. It has been built by analyzing which tokens could appear after the rule `keyValuePair`. The water rule has to be adjusted every time changes in the grammar are made. However this grows very fast, because as the grammar becomes more complex, there will be more rules that have to be taken into account to tailor the water.

The bounded seas approach [8] proposes a solution for this. The grammar rule water from Listing 3.5 would simply be represented like this: $\sim$ `keyValuePair`$\sim$. It uses the newly introduced *sea operator:* $\sim$.

The informal definition of a bounded sea [8]:

**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.*

A bounded sea recognizes different input depending on what immediately precedes/ follows the sea. This means depending on which rule the bounded sea is called from the results may vary. In this thesis we therefore make use of the bounded seas approach. Bounded seas are implemented in the PetitParser framework in Pharo[5]. Our implementation of bounded seas will be described in the subsection 4.2.1.2.

#### 3.2.1.1 Why to never reuse a bounded sea parser

A parser that uses the bounded sea implementation should never be reused. A bounded sea rule always evaluates the before-water, the island and the after-water. It is context sensitive. This means that for each island the before-water and the after-water can be different even if the island is the same. A bounded sea parser traverses the parse tree and takes all other rules into account. This means that the sea is tailored specifically in the context of this specific instance of the parser. This is why we always have to create a new parser based on the given specification and never reuse an already created one.

## 3.3 PetitParser

PetitParser (PP) gives the user the ability to implement PEGs in Smalltalk code. PP uses a combination of four alternative parser methodologies: *scannerless parsers [11],*

[5]`http://pharo.org`

*parser combinators*[6]*, PEGs [9] and packrat parsing [1].* A quick explanation of the three methodologies which were not already explained in this chapter:

- **Scannerless Parsers** combine scanner and parser into one.
- **Parser Combinators** are higher-order functions which accept multiple parsers as input and return a new parser as their output.
- **Packrat Parsers** always run in linear time (by using memoization), but require substantially more storage space.

To further simplify the process, PP provides a large set of ready-made parsers:

## 3.3.1 Terminal Parsers

The following is a table of the ready-made terminal parsers for PP [7]:

| **Terminal Parsers** | **Description** |
|---|---|
| $a asParser | Parses the character a. |
| ’abc’ asParser | Parses the string ’abc’. |
| #any asParser | Parses any character. |
| #digit asParser | Parses one digit [0..9] |
| #letter asParser | Parsers one letter [a..z] and [A..Z]. |
| #word asParser | Parses a digit or letter. |
| #blank asParser | Parses a space or tabulation. |
| #newline asParser | Parsers the carriage return or line feed characters. |
| #space asParser | Parses any white space character including new line. |
| #tab asParser | Parses a tab character. |
| #lowercase asParser | Parses a lowercase character. |
| #uppercase asParser | Parses an uppercase character. |
| nil asParser | Parses nothing. |

Table 3.2: Pre-defined list of PetitParser terminal parsers [7]

With the asParser message, you can create parsers that accept single characters ($a asParser) or parser which accepts a sequence of characters (’abc’ asParser). When you call asParser with the # symbol and a given name, it returns one of the pre-defined parsers from the PPPredicateObjectParser class.

[6]https://en.wikipedia.org/wiki/Parser_combinator

### 3.3.2 Parser Combinators

The parsers we create with PetitParser can be extended with different messages. For example if we create one of the terminal parsers: #letter asParser, which gives us an instance of PPPPredicateObjectParser, we can add messages to this Parser *e.g.,* star, which is the equivalent to $\star$ in PEGs. Below is a table of the different parser combinators that one can use with PetitParser [7]:

| **Parser Combinators** | **Description** |
|---|---|
| $p_1$, $p_2$ | Parses $p_1$ followed by $p_2$ (sequence). |
| $p_1$ / $p_2$ | Parses $p_1$, if that doesnt work parses $p_2$ |
| p star | Parses zero or more p |
| p plus | Parses one or more p. |
| p optional | Parses p if possible. |
| p and | Parses p but does not consume its input. |
| p negate | Parses p and succeeds when p fails. |
| p not | Parses p and succeeds when p fails, but does not consume its input. |
| p end | Parses p and succeeds only at the end of the input. |
| p times:n | Parses p exactly n times. |
| p min:n max:m | Parses p at least n times up to m times. |
| p starLazy:q | Like star but stop consuming when q succeeds |

Table 3.3: Pre-defined list of PetitParser parser combinators [7]

In the above table you can easily see the similarities to the PEG notation. Some are different *e.g.,* $p_1$`,` $p_2$, whereas compared to PEG notation: $p_1$ $p_2$. And *e.g.,* `p negate` equals the notation `(!p.)` in PEG.

The PP implements the example from Listing 3.1 like this:

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

Listing 3.6: Basic example of a PEG implemented with PetitParser

This implementation is very similar to the PEG implementation. However when we look at this implementation, we can see that there is a problem with the `expr:= sum`, because expr uses sum and sum is not defined yet. Sum in turn uses product, which is not defined and product uses value, which is not defined at this point and value uses expr again. So to allow this sort of indirect recursion, we have to implement it like this:

```
expr:= PPDelegateParser new.
sum:= PPDelegateParser new.
value:= PPDelegateParser new.
product:= PPDelegateParser new.


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

Listing 3.7: Basic example of a PEG implemented with PetitParser, taking into account variable definitions

One thing to mention is, that opposed to a bounded sea parser, a standalone and working parser (*e.g.,* string) can be reused as a sub-parser in another parser (*e.g.,* a list of strings). The reason for this is that these parsers do not require a specific context, as they are made of a fixed set of characters *e.g.,* string can start with a ” and anything that is not a ” or the escape character will be accepted.

A possibility where errors can happen is that PetitParser does not handle left-recursion *e.g.,* a parser that is defined like this: `test:= test, $a asParser` will never terminate. This is why when implementing a grammar one has to remove left recursive rules.

### 3.3.3 Action Parsers

A small example of the results by parsing with the created parser:

```
expr parse:'12+34'.
Output:
"#(#(#($1 $2) #()) #(#($+ #(#($3 $4) #()))))"
```

PP outputs different parts of the result in the form of Arrays. This form is hard to read for a human. To help with this PP also offers action parsers. Below is a table of the different pre-defined action parsers [7].

| **Action Parsers** | **Description** |
|---|---|
| p flatten | Creates a string from the result of p. |
| p token | Similar to flatten but returns a PPToken with details. |
| p trim | Trims white spaces before and after p. |
| p trim: trimParser | Trims whatever trimParser can parse (e.g., comments). |
| p ==>aBlock | Performs the transformation given in aBlock |

Table 3.4: Pre-defined list of PetitParser Action Parsers [7]

If we parse the given example again by using the *flatten* action, we get the following result:

```
expr flatten parse:'12+34'.
Output:
'12+34'
```

# 4 Crassula

We call our tool Crassula[1]. In this chapter we explain our idea, how we implemented it and what problems arose along the way.

## 4.1 Idea

After analyzing many different data serialisation formats (see chapter 1), we came to the conclusion that all languages analyzed by us can be broken down to a few common building blocks. With these building blocks we can represent all the data serialisation formats we analyzed and they enable us to extract the needed containment structure from the code.

[1]Crassula gets its name from the plant Crassula capitella, which is a plant that looks like lots of nested structure elements

### 4.1.1 Building blocks

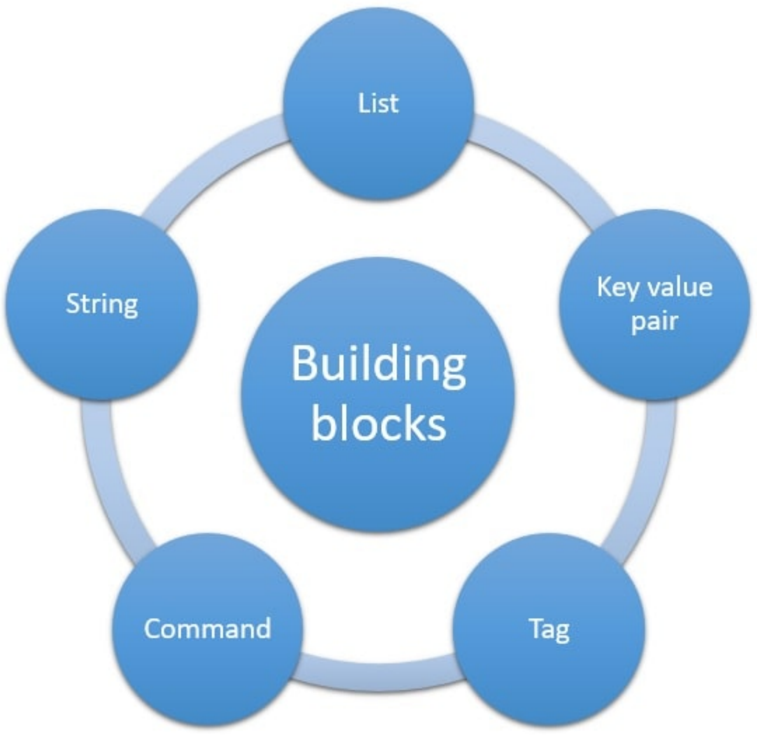


Figure 4.1: The five main building blocks that enable us to reconstruct the structure of a given data serialisation format

Figure 4.1 shows all the building blocks that are needed to build a parser for a given data serialisation format. All these building blocks work the same way. They always work by taking a set of arguments *e.g.,* a list always has an opening, a closing, the elements of the list and a delimiter between the different elements. These arguments are used by the building block to build an appropriate parser.

A brief explanation of each building-block:

**String**: A String is a sequence of characters, which is often represented between two signs to mark it as a String *e.g.,* `"string"` or `'string'`

The string building block consists of the following parts:

- **A character** that marks its beginning.
- **A character** that marks its end.
- **An escape character**[2] that is used to tell the compiler to interpret the escaped

[2]https://en.wikipedia.org/wiki/Escape_character

character differently than it normally would be treated. For example in Java \ is an escape character *e.g.,* `"Str\"ing"`. In this example you can use the character `"` inside the string, without the string being terminated at this point.

- **The content** of the string.

**List**: A list is commonly a container of different values *e.g.,*
`(elt1,elt2,elt3)`
Or a nested list:
`(elt1,((())))`

The list building block consists of the following parts:

- **An opening character** that marks its beginning. For example {,
- **A closing character** that marks its end. For example }.
- **A delimiter** that serves as a separator between the single values. For example: `,`.
- **An element** that can be anything that is inside the list. Since a list can be nested, the element itself can also be a list, so it is possible to have a list of lists of lists ...

**Key- value pair**: *e.g.,* `"key":"value"`. The key-value pair building block consists of the following parts:

- **A key** that is the name of the element.
- **A value** that can be any of the 5 elements, List, String, Command, Tag or another KeyValue pair.
- **A delimiter** that marks the end of the key and the start of the value.

**Command**: The command is for structures that look like this:

```
someKey {
    a:b;
    a:1;
    "a":"b";
}
```

Listing 4.1: Example of a command building block

We call it command, because calling upon the key, somewhat executes its arguments. Therefore it commands its arguments. A similar structure can for example be found in CSS or Protocol Buffer files. The command building block consists of the following parts:

- **A key** that is the name of the element.
- **Arguments** that can be any of the building blocks, but mostly is a list with elements.

**Tag**: A tag is a structure that looks like this:`<a>content</a>`.
The `<a>` structure is called a tag and the letter `a` is the name of the tag. This structure occurs in languages like XML/HTML. The tag building block consists of the following parts:

- **A character** that marks the opening of the tag *e.g.,* <.
- **A character** that marks the closing of the tag *e.g.,* >.
- **A character** that marks the character following the closing character that marks that it closes the tag, and doesn't open a new one *e.g.,* `/`.
- **The attributes** of the tag. A tag can have attributes *e.g.,* `<tag att=value/>`. These attributes are not relevant for the structure we want. Therefore we treat them as water.
- **The content** of the tag. It is not relevant for our containment structure. Therefore we treat it as water.

Using these building blocks we can automate the process of parser building and create a tool that outputs us the desired structure of a given source file. We implemented it with PetitParser for Pharo[3].

## 4.2 ParserFactory

We have five basic factories which represent the main building blocks explained above: StringParserFactory, CommandParserFactory, ListParserFactory, KeyValueParserFactory and TagParserFactory.

[3]`http://pharo.org`

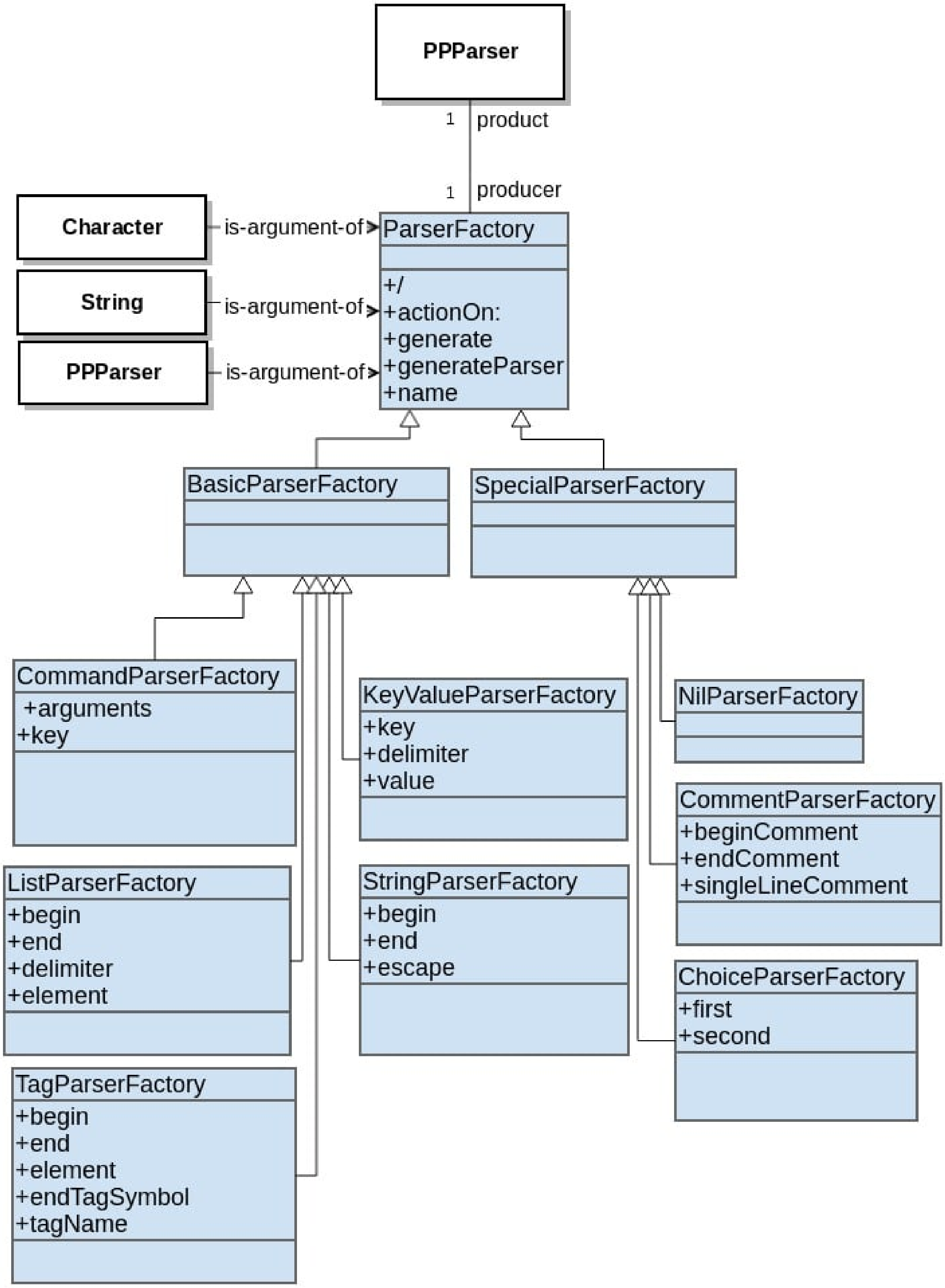


Figure 4.2: Overview of the ParserFactory implementation

Figure 4.2 presents an overview of the implementation.

We implement the ParserFactories making use of the PPParser class. We explain how the basic factories work by the example of the ListParserFactory. The other implementations all work in the same manner[4].

As mentioned in the subsection Building blocks, the list consists of an opening, a closing, a delimiter and an element. The creation of the list object in Pharo looks like this:

```
(ListParserFactory new)
    begin: someBeginning;
    end: someEnd ;
    delimiter: someDelimiter;
    element: someElement;
    yourself.
```

Listing 4.2: Creation of a ListParserFactory object in Pharo

A parser factory is created with a specific set of user-given arguments. Each method of the ParserFactory expects one argument *e.g.,* in Listing 4.2: `begin:  someBeginning` *etc.* The parser factory can be seen as a blueprint for a parser. The goal of a parser factory is to create a new parser according to this blueprint whenever the generateParser method is called. Everything that understands the message generateParser is considered a parser factory. A parser factory always accepts other parser factories as one of its arguments *e.g.,* in Listing 4.2, the arguments someBeginning, someEnd, someDelimiter and someElement all have to be implementors of generateParser *i.e.,* parser factories.

In our implementation Character, String, PPParser and ParserFactory are all implementors of generateParser, which means that they are a parser factories. If we call generateParser on a Character or a String it builds an instance of PPParser out of itself *e.g.,*'a' will become $a asParser. When we call generateParser on a PPParser we simply return itself, because it already is a parser. However because some cases have been identified too late in the design process, the basic parser factories (see Figure 4.2) currently only accept instances of PPParser as its arguments and no Characters or Strings.

An example for valid arguments of Listing 4.2:

```
someBeginning:= $( asParser.
someEnd:= $) asParser.
delimiter:= $, asParser.
element:= NilParserFactory new generate.
```

Listing 4.3: Example parameters for a ListParserFactory

[4]The full code can be found on this github repository: `https://github.com/Icewater1337/RecognisingStructuralPatterns`

The three most important methods are *generateParser*, *generate* and *actionOn:*.

- The **generate** method is the responsibility of the subclass to implement. This method generates the parser from the arguments that were used to build the factory. It creates the parser by using PetitParser. For an example see Listing 4.4.
- The **actionOn:** method changes the parser output to a tree form (see Figure 1.3) that represents the structure of the file.
- The **generateParser** method is the responsibility of the subclass to implement. It calls generate and then actionOn:

The most important method is the generate method, because this method actually executes the parser building process and creates the desired parser based on the given specification. The following example is the generate method of the ListParserFactory:

```
generate
    | parser |

    parser := begin trim,
    element optional,
    ((delimiter trim optional, element) nonEmpty star),
    end trim

    ˆ parser
```

Listing 4.4: ListParserFactory generate method

The generate method takes the arguments from the ListParserFactory and returns an instance of PPParser.

## 4.2.1 Special parser factories

There are special parser factories, which are not basic building blocks but assist the basic parser factories. One of them is the CommentParserFactory. We implemented it for convenience reasons to parse comments in the given source. However we can also parse for comments with the StringParserFactory.

#### 4.2.1.1 ChoiceParserFactory

It is the implementation that handles the equivalent to the PEG choice operator. It is the ParserFactory equivalent to the PPChoiceParser *e.g.,* `StringParserFactory new / KeyValueParserFactory new`. This returns an instance of ChoiceParserFactory, which in turn will return an Instance of PPChoiceParser when generateParser or generate is executed.

#### 4.2.1.2 NilParserFactory

Another special factory is the *NilParserFactory*. The problem is that water normally has to be tailored for each rule individually [8]. Therefore it was challenging to implement a water rule that automatically adjusts itself and therefore is suitable to work with the dynamic creation of parsers. However with the approach of bounded seas [8], explained in the bounded seas section, we came up with the following solution:

```
generate
    ^ (nil asParser sea) token
```

Listing 4.5: NilParserFactory implementation in Pharo. It is the implementation of the bounded seas for the factories

In this implementation we make use of the bounded sea implementation of PetitParser. As explained in the bounded seas section, the three parts of a bounded sea are:

- before- water
- island
- after-water.

The message sea of PPParser consumes everything (before water) until it finds the island, it then consumes the island and keeps on consuming everything until it finds text which is matched by another rule (after- water). A simple example:
`parser:= $a asParser sea`
`parser parse:'uhuasdaabc'.`
Output: `an OrderedCollection('uhu' $a 'sdaabc')`

This parser recognizes before (before-water) and after (after-water) the letter $a (island) as water, and recognizes the $a itself. The expression `nil asParser` parses nothing (see Table 3.3.1), therefore `nil asParser sea` parses everything until it parses nothing. And nothing will be parsed, as soon as another rule succeeds.

This approach works fairly well, but has some problems, which we explain in the validation chapter.

## 4.3 ParserGenerator

The *ParserGenerator* is the class responsible for combining the ParserFactories in various ways and in the end outputting other factories, so they are ready to be used and parse a given file. Our first approach is a naive approach using brute force to combine the factories. The *BruteForceGenerator* makes use of the ParserFactories and combines them by brute force, generating each possible combination.

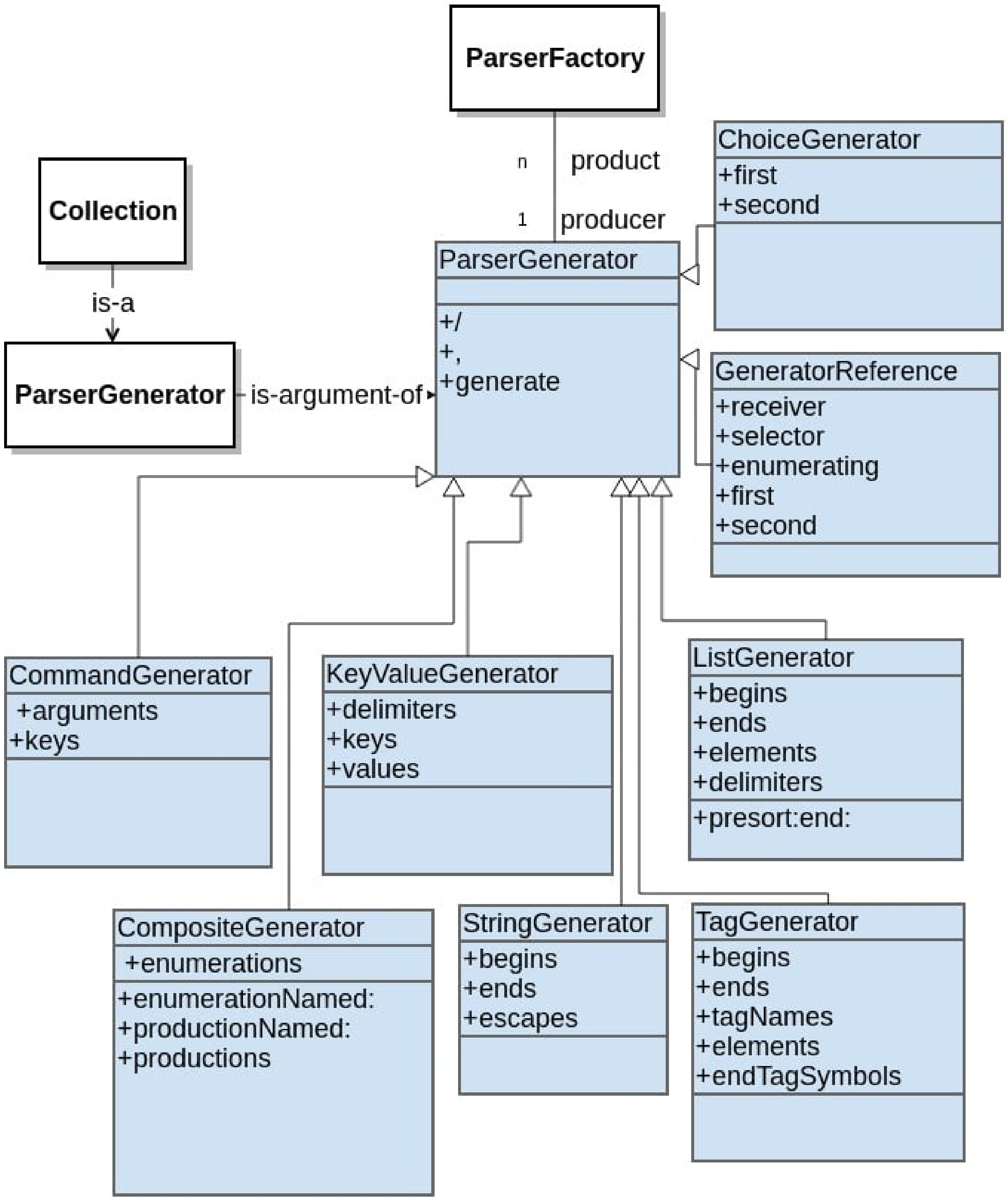


Figure 4.3: Overview of the ParserGenerator implementation

We used this approach because it is fairly simple and serves well for a proof-of-concept. In the future we intend to develop different versions of the ParserGenerator which combine the factories in different ways *e.g.,* with machine-learning algorithms.

We have five main generators that mix together different ParserFactories: StringGenerator, CommandGenerator, ListGenerator, KeyValueGenerator and TagGenerator. Figure 4.3 presents an overview of the implementation. There are special cases of the *ParserGenerators*: *ChoiceGenerator, SequenceGenerator, GeneratorReference and the CompositeGenerator* (explained in the subsection 4.3.1).

Everything that understands the message generate is a ParserGenerator, and will be accepted as an argument. Implementors of generate are ParserGenerators and collections of ParserFactories. The ParserGenerator should always return a collection of ParserFactories. However, the current implementation does not always satisfy this requirement. As in the previous case, some corner cases have not been identified early enough in the design process. It currently returns a collection of PPParsers. The reason it should return a collection of ParserFactories is that a Parser should never be reused (explained in subsubsection 3.2.1.1).

An example of a generator:

```
list
    ˆ (BruteForceListGenerator new)
        begins: #($< $[);
        ends: #($> $]);
        delimiters: #($, $;);
        elements: keyValuePair /  {NilParserFactory new} ;
        yourself
```

Listing 4.6: Creation of a BruteForceListGenerator object with Pharo

The `begins:, ends:  and delimiters:` methods from Listing 4.6 all accept as argument a collection of characters, which equals a collection of ParserFactories because a character understands generateParser. The elements consist of the keyValuePair, which is another BruteForceGenerator and the NilParserFactory which is a ParserFactory.

The most important method of the ParserGenerator is the generate method. Each BruteForceGenerator has the generate method, which then calls generate for every single element and builds the parser for this element. In the above example 2*2*2*2 = 16 parsers will be generated.

## 4.3.1 Special Generators

There are also special cases of the ParserGenerators: *ChoiceGenerator, GeneratorReference and the CompositeGenerator*

- **The CompositeGenerator** is responsible for creating custom combinations from the single generators. An overview of all the CompositeGenerators can be found in Appendix A.1.
- **The ChoiceGenerator** is responsible to handle the choice operator / from PetitParser. The elements rule from Listing 4.6 is an example.
- **GeneratorReference** is responsible for handling nested structures, like lists of lists.

### 4.3.2 ParserGeneratorExecutor

The *ParserGeneratorExecutor* is the final product. It executes the whole process in which the parsers are created and then used to recognize the structure of a given file. It can be called like this: `ParserGeneratorExecutor parse: AString.` or: `ParserGeneratorExecutor parseWithOrder: AString.`

The ParserGeneratorExecutor returns a collection of result *Nodes* (see section 4.4). If parseWithOrder is called, the results are returned in a specific order, where the best of the succeeding parsers is at the start of the collection.
**Determining the validity of a parser:** We used a naive approach where we give the different structure elements a weight *e.g.,* an element recognized by a parser that was built by a TagParserFactory has a weight of 13, a string has a weight of 2 and things that are parsed by the NilParserFactory are weighted with 0 *etc.* This means each time a tag-structure is parsed, we add 13 to the total weight and each time a string- structure is parsed we add 2 to the total weight. Example:

```
<a>
  <b>c</b>
</a>
```

The tags with the names `a` and `b` are both weighted with 13, and `c` has a weight of 0, it is considered water. Therefore the example has a weight of 26. This means, the more elements a parser recognizes, the higher his weight will be. A parser which is incorrect will most likely not recognize many elements and parse most of the content as water. We then put the parser with the highest weight on the top. However this does not work 100% well currently.

However this ordering approach is very experimental and not always exact (see chapter 5).

### 4.3.3 Pruning parsers

A problem with the brute force approach is that a large number of parsers is generated. Many of them make no sense, or are highly unlikely to match any file format. Therefore we implemented some steps to prune the parsers and remove those that are almost certainly not useful. To achieve this, we did the following things:

- **A lexer** takes the input file and extracts all signs from it that are neither letters nor digits. The idea behind this is to get a list of all characters that might be used in any of the factories *e.g.,* for an opening, a closing or a delimiter.
- The so called **ArgumentsCreator** has the task to further limit the creation, by comparing the list of inputs against a fixed list of known beginnings, ends and delimiters. We make a few assumptions for this. For example: We know that no list will probably ever start with a `)` and end with a `)`, or that `]` will most likely not be a delimiter. So in the end we assure that for example beginning and end brackets, always match. This process further reduces the number of parsers created.

This process reduces the amount of unnecessarily created parsers greatly, and therefore improves performance.

## 4.4 Output form

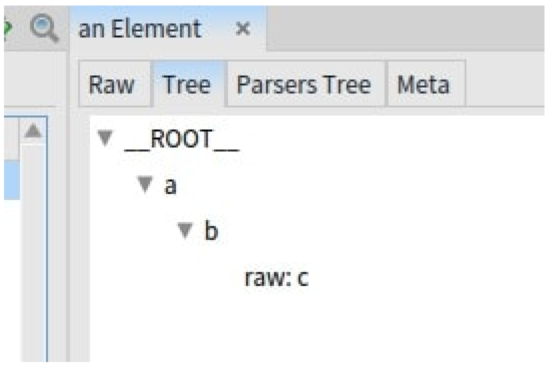
As shown in Figure 4.4 we intended our output to be this tree-like structure. To achieve this, we created the Node class. The parsers built by the ParserFactories will always return an output in the form of a Node. We created the parent class: Node and the following subclasses:

- **RawText** is used for all kind of Text outputs and also unknown outputs (seas). A RawText has a content, which contains all the information.
- **Comment** is used to represent all comments.
- **Element** is used for most of the outputs that are not classifiable as any of the other Node types. Elements have names, and children *e.g.,* XML: A name is the name of a Tag, and its children are all the nested Tags.

With the *Node* class, we can build a containment list which represents the structure of a given source file. If we take a small XML example:

```
<a>
  <b>c</b>
</a>
```

The output in Pharo looks like this:



```
▼ __ROOT__
    ▼ a
        ▼ b
            raw: c
```

Figure 4.4: A containment list structure representation of an XML example in Pharo

## 4.5 Problems/ Challenges

A challenge was posed by nested structures. Because it could easily happen that if we have a construct that is made of lists of lists, it could be that the parser went into an infinite loop and kept calling itself. This could be due to the fact that PetitParser does not handle left recursion automatically. This means if a left recursive rule is generated and used by one of the generated parsers this leads into an infinite loop. One way to fix this is to implement a detection for left-recursive grammars.

# 5 Validation

In this chapter we discuss the experiments we did to validate our approach (see Appendix B.1 for more details on the implementation). We validate our approach by comparing the found structure elements of the auto-generated parser (achieved result) against the structure elements found by the correct parser (correct result). To compare the achieved result with the correct result, we built a simple representation in the form of a set of containment rules, which namely tells us which block is contained in which block. The set of containment rules is achieved by using the nodes explained in section 4.4. To illustrate this take the following example:

```
<a>
  <b>content</b>
</a>
<d>content</d>
```

The correct parser produces the following set of containment rules(CSCR). This is the ground truth:

```
1.  __ROOT__ → a
2.  a → b
3.  b → content
4.  __ROOT__ → d
5.  d → content
```

Let's assume that the generated parser produces the following set of containment rules (GSCR):

```
1.  __ROOT__ → a
2.  a → b
3.  b → content
4.  __ROOT__ → d
5.  d → false
6.  a → d
```

The numbers on the left are the line numbers and are only used for reference reasons.

## 5.1 Calculating precision, recall and f-measure

To calculate the precision, recall and f-measure[1], we need to know what the *true positives(TP), false positives(FP) and false negatives(FN)* are. To get the TP, FP and FN we compare all the elements of the generated parser containment list against the containment list of the correct parser.

- **The true positives** are the relevant selected items. In our example above the elements 1,2,3 and 4 of GSCR, are also contained in CSCR. Therefore the number of true positives is 4.
- **The false positives** are the wrongly selected items. In this example the false positives would be the elements 5 and 6 of GSCR, because they are not in CSCR.
- **The false negatives** are the items that would be correct, but the program failed to identify them. In our example the false negatives would be element 5 of CSCR. This is the only element that the generated parser did not recognize.

The precision is calculated like this:
`true positives / (true positives + false positives).`
The recall is calculated like this:
`true positives / (true positives + false negatives).`

We are in our case not able to measure the true negatives(TN). The true negatives are everything that is rejected correctly. However we only have correct items in an infinite sea of incorrect items. Basically everything that is in a given file is correct, because we want to recognize the whole file. We do not want to recognize anything incorrect. And everything that is not in the given file would be the TN. This is why we measure precision, recall and f-measure instead of accuracy.

[1] `https://en.wikipedia.org/wiki/F1_score`

## 5.2 Experiment

This section explains how the experiment has been executed

### 5.2.1 Description

In the experiment we use Crassula for the automatic recognition of structure on XML, CSS and JSON files. Therefore we took 10 different files from each type. A link to all the files used can be found in Appendix B.3. The explanation on how to reproduce the experiment can be found in Appendix B.4. We measure the following:

- Precision
- Recall
- F-measure
- The position of the parser in the results-list. The position is determined by giving the structure elements a certain weight (see subsection 4.3.2).
- The execution time needed.

We report the results of the best parser that is generated for a specific file. The best parser is the parser with the highest f-measure.

We split the experiment in two parts

**1. Individual parsing**: Each file will be parsed individually by the ParserGeneratorExecutor.

**2. Pair parsing**: The files will be parsed in pairs. We take a small reference file, of the corresponding format, which contains all the needed syntax elements and the file we actually want to parse, which is of the same format[2]. In Appendix B.2 you can find the files we used as base files.

The idea is that the ParserFactory is generated from the first file in the list, which is the short reference file. This factory blueprint will then be reused for the other file in the list, which is in our case the file we want to extract the structure from. We expect that in most cases the factory that is created from the small file can create the parser for the small file, as well as for the big file, because the arguments we use to build the factory should be the same for both files. In theory this should perform much faster, because the part that takes most of the time is the failing parsers (parsers that do not succeed). And therefore if the first file is a very short one, the failing parsers will not take too much time and we can limit it to a set of only valid parsers that we can reuse.

[2]File for json: `http://json.org/example.html`. File for CSS, constructed according to: `http://w3schools.com/css`. XML file, constructed according to W3C XML-Specs

## 5.3 The results

We ran the experiment on three file types: XML, CSS and JSON. The files XML1/ CSS1/ JSON1, are the shortest and XML10/ CSS10 /JSON10 are the longest (see Appendix B.3 for file sizes).

| Single parse | | | | | |
|---|---|---|---|---|---|
| File | precision | recall | f-measure | Position of best parser | Time (seconds) |
| XML1 | 1 | 1 | 1 | 1 | 1.67 |
| XML2 | 1 | 1 | 1 | 1 | 3.4 |
| XML3 | 1 | 1 | 1 | 1 | 1.16 |
| XML4 | 1 | 1 | 1 | 1 | 4.44 |
| XML5 | 1 | 1 | 1 | 1 | 100 |
| XML6 | 1 | 1 | 1 | 1 | 24 |
| XML7 | 1 | 1 | 1 | 1 | 702 |
| XML8 | 1 | 1 | 1 | 5 | 298 |
| XML9 | 1 | 1 | 1 | 1 | 2086 |
| XML10 | 1 | 1 | 1 | 1 | 5781 |

Table 5.1: Results of the XML single- experiment measured in precision, recall, f-measure and execution time. The position of the best parser in the results list is added to see how well the ordering works. File sizes can be found in Appendix B.3.

| Pair parse | | | | | |
|---|---|---|---|---|---|
| File | precision | recall | f-measure | Position of best parser | Time (seconds) |
| XML1 | 1 | 1 | 1 | 1 | 3.56 |
| XML2 | 1 | 1 | 1 | 1 | 3.31 |
| XML3 | 1 | 1 | 1 | 1 | 5.78 |
| XML4 | 1 | 1 | 1 | 1 | 6.31 |
| XML5 | 1 | 1 | 1 | 1 | 6.81 |
| XML6 | 1 | 1 | 1 | 1 | 32 |
| XML7 | 1 | 1 | 1 | 1 | 26 |
| XML8 | 1 | 1 | 1 | 1 | 16 |
| XML9 | 1 | 1 | 1 | 1 | 302 |
| XML10 | 1 | 1 | 1 | 1 | 1211 |

Table 5.2: Results of the XML pair- experiment measured in precision, recall, f-measure and execution time. The position of the best parser in the results list is added to see how well the ordering works. File sizes can be found in Appendix B.3.

| Single parse | | | | | |
|---|---|---|---|---|---|
| File | precision | recall | f-measure | Position of best parser | Time (seconds) |
| CSS1 | 1 | 1 | 1 | 12 | 1.98 |
| CSS2 | 1 | 1 | 1 | 26 | 13 |
| CSS3 | 1 | 1 | 1 | 12 | 1.72 |
| CSS4 | 1 | 1 | 1 | 24 | 3.66 |
| CSS5 | 1 | 1 | 1 | 6 | 156 |
| CSS6 | 1 | 1 | 1 | 37 | 67 |
| CSS7 | 1 | 1 | 1 | 5 | 989 |
| CSS8 | 0.572 | 0.4 | 0.471 | 13 | 375 |
| CSS9 | 1 | 1 | 1 | 2 | 563 |
| CSS10 | 0.32 | 0.31 | 0.31 | 12 | 13633 |

Table 5.3: Results of the CSS single- experiment measured in precision, recall, f-measure and execution time. The position of the best parser in the results list is added to see how well the ordering works. File sizes can be found in Appendix B.3.

| Pair parse | | | | | |
|---|---|---|---|---|---|
| File | precision | recall | f-measure | Position of best parser | Time (seconds) |
| CSS1 | 1 | 1 | 1 | 3 | 1.29 |
| CSS2 | 1 | 1 | 1 | 3 | 3.31 |
| CSS3 | 1 | 1 | 1 | 3 | 2.61 |
| CSS4 | 1 | 1 | 1 | 3 | 2.76 |
| CSS5 | 1 | 1 | 1 | 3 | 4.74 |
| CSS6 | 1 | 1 | 1 | 3 | 4.81 |
| CSS7 | 1 | 1 | 1 | 3 | 8.07 |
| CSS8 | 0.572 | 0.4 | 0.471 | 3 | 25 |
| CSS9 | 1 | 1 | 1 | 3 | 261 |
| CSS10 | 0.32 | 0.31 | 0.31 | 3 | 310 |

Table 5.4: Results of the CSS pair- experiment measured in precision, recall, f-measure and execution time. The position of the best parser in the results list is added to see how well the ordering works. File sizes can be found in Appendix B.3.

| Single parse | | | | | |
|---|---|---|---|---|---|
| File | precision | recall | f-measure | Position of best parser | Time (seconds) |
| JSON1 | 1 | 1 | 1 | 108 | 1.67 |
| JSON2 | 1 | 1 | 1 | 43 | 3.4 |
| JSON3 | 1 | 1 | 1 | 29 | 1.16 |
| JSON4 | 1 | 1 | 1 | 54 | 4.4 |
| JSON5 | 0.55 | 0.78 | 0.64 | 4 | 100 |
| JSON6 | 1 | 1 | 1 | 46 | 24 |
| JSON7 | 1 | 1 | 1 | 51 | 702 |
| JSON8 | 1 | 1 | 1 | 46 | 298 |
| JSON9 | 0.83 | 0.87 | 0.85 | 34 | 2086 |
| JSON10 | 1 | 1 | 1 | 44 | 5781 |

Table 5.5: Results of the JSON single- experiment measured in precision, recall, f-measure and execution time. The position of the best parser in the results list is added to see how well the ordering works. File sizes can be found in Appendix B.3.

| Pair parse | | | | | |
|---|---|---|---|---|---|
| File | precision | recall | f-measure | Position of best parser | Time (seconds) |
| JSON1 | 1 | 1 | 1 | 25 | 28 |
| JSON2 | 1 | 1 | 1 | 24 | 29 |
| JSON3 | 1 | 1 | 1 | 25 | 28 |
| JSON4 | 1 | 1 | 1 | 37 | 31 |
| JSON5 | 0 | 0 | 0 | - | - |
| JSON6 | 1 | 1 | 1 | 25 | 31 |
| JSON7 | 1 | 1 | 1 | 25 | 30 |
| JSON8 | 1 | 1 | 1 | 25 | 45 |
| JSON9 | 0 | 0 | 0 | - | - |
| JSON10 | 1 | 1 | 1 | 30 | 128 |

Table 5.6: Results of the JSON pair- experiment measured in precision, recall, f-measure and execution time. The position of the best parser in the results list is added to see how well the ordering works. File sizes can be found in Appendix B.3.

| Summary single parse | | | | |
|---|---|---|---|---|
| File type | precision | recall | f-measure | Position of best parser |
| JSON | 0.94 | 0.94 | 0.94 | 46 |
| XML | 1 | 1 | 1 | 1 |
| CSS | 0.89 | 0.87 | 0.88 | 15 |

Table 5.7: Summary of the single parse experiment

| Summary pair parse | | | | |
|---|---|---|---|---|
| File type | precision | recall | f-measure | Position of best parser |
| JSON | 0.8 | 0.8 | 0.8 | 27 |
| XML | 1 | 1 | 1 | 1 |
| CSS | 0.89 | 0.87 | 0.88 | 3 |

Table 5.8: Summary of the pair parse experiment

**Observations concerning precision, recall and f-measure:**
The approach works very well for most files and is able to recognize and output the structure. However one problem that manifested itself during the experiments is that the implementation of the NilParserFactory comes with some problems. Let's take the following example:

```
i {
  animation-duration: 3s;
  animation-iteration-count: infinite;
  animation-timing-function: ease-in-out; }


i:nth-child {
  transform: rotate;
  animation-delay: 29032258065s; }
```

Listing 5.1: A CSS example on which the correct parser fails and only recognizes the first rule-set

In the example from Listing 5.1 the correct parser fails to recognize the second rule-set[3]. We are not sure why it fails, however we suspect one of the following:

- It could be a problem that we somewhere unintentionally reuse a bounded sea parser and do not create a specific one (see subsubsection 3.2.1.1).

[3]https://www.w3schools.com/css/css_syntax.asp

- The other option is that if we encounter a construct that looks like an island, inside the before-water, the parser already recognizes it as an island, and stops. That we do not want to happen. The reason could be because the selector `i:nth-child` is recognized as a key-value pair instead of water. However the intention would be to skip everything of the selector as water and stop when the list island, which starts with {, begins. If re remove the **:** in the selector, the parser will succeed and recognize both rule-sets.

Another issue, which can be seen in the JSON pair parse, is that pair parsing sometimes fails completely. The reason for this is that in some file formats we encounter different starting options *e.g.,* In JSON the file can start with `[` or with {. In the pair parsing the parser will be generated from the minimal reference file, which can either start with one or the other beginning. Therefore the generated parser will not be able to recognize the other beginning.

**Observations concerning execution time:**
As expected the brute force approach of the generator is in general quite slow. However the needed time decreases greatly when doing the pair parsing approach.

We observed that the time does not only depend on the file length, but more on the number of different structure elements found in the file *e.g.,* A JSON file with many key-value pairs will perform slower.

During the experiment we came to the realisation that the part which takes the most time are the failing parsers. This is because some of them can loop infinitely and never terminate. To prevent endless loops we added an execution time limit of 20 seconds per parser.

**Observations concerning the ordering of the results:**
The ordering (see explanation of parseWithorder in subsection 4.3.2) works excellent on XML files, decent on CSS files and worst on JSON files. This is due to the fact that the weighs given to the different elements need to be refined.

### 5.3.1 Conclusion

The experiment shows that the approach works fairly well to automatically extract the structure of a given data serialisation file, if the file is known to us. However testing has to be done towards unknown data serialisation formats. The bounded sea implementation fits the approach well, but still needs some refinement.

The pair parsing approach greatly improves performance, but it has its limitations when a language can start with different structures. *e.g.,* JSON can start with `[` or {. Overall the brute force approach is still very slow.

# 6 Conclusions and future work

We have implemented a parser based approach for structure recognition on data serialisation files. Throughout the process we were able to build parsers for many different formats by using high-level parsing building blocks. The factories are also fairly well suited for a programmer to reduce the amount of work he needs to write a parser. Using the concept of bounded seas, we are able to create a self adapting water for any given island, which enables us to automate the production of the parsers and therefore the structure recognition.

We are able to achieve the automatic recognition by combining our building blocks to create the parsers. To start out, we use a brute force approach of the parser generators. However the brute force approach is not optimal, especially in terms of time. It has great potential for improvement.

To test the approach, we parse 10 files of three different data serialisation formats (XML, CSS and JSON). It performs excellent for XML (F-measure of 1), but struggles with CSS (F-measure of 0.88).

In the future we plan to do the following:

- Further test the BruteForceGenerator on unknown file formats, and analyze how well the approach works.
- Extend the concept of the building blocks to real programing languages and try to recognize structure.
- Improve the ordering of the parsers. Return the best parser first.

- Build more generators that are not based on the brute force approach and therefore may perform more efficient and accurate (*e.g.,* some machine learning algorithms)
- Enable the user to extract the successful parser, and reconstruct its source code, so it can be modified/ reused.

# Acknowledgement

First and foremost I would like to thank Jan Kurš and Haidar Osman for being such awesome supervisors. Jan helped me so much with the implementation work and always kept a close eye on my code. He always took his time to answer my questions without losing patience and did coding sessions with me. Even after he left the university, he still took the time to help me! I was able to learn a tremendous amount about good code and good architecture from him. Haidar met with me countless times to discuss the present and the future of the project and always kept me motivated. His advice and patience during the project was invaluable. I would also like to give special thanks to Professor Oscar Nierstrasz for the opportunity to write my thesis in the Software Composition Group.

# Bibliography

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

[2] 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.

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

[4] Joël Guggisberg. Automatic token classification — an attempt to mine useful information for parsing. Bachelor's thesis, University of Bern, December 2015.

[5] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *Proceedings of ASE 2016 (31st Conference on Automated Software Engineering)*. IEEE Computer Society Press, September 2016. To appear.

[6] Roberto Ierusalimschy. A text pattern-matching tool based on parsing expression grammars. *Software Practice and Experience*, 39(3):221 – 258, March 2009.

[7] Jan Kurš, 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.

[8] Jan Kurš, Mircea Lungu, Rathesan Iyadurai, and Oscar Nierstrasz. Bounded seas. *Computer Languages, Systems & Structures*, 44, Part A:114 – 140, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

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

[10] Oscar Nierstrasz, Markus Kobel, Tudor Gîrba, Michele Lanza, and Horst Bunke. Example-driven reconstruction of software models. In *Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 275–286, Los Alamitos CA, 2007. IEEE Computer Society Press.

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

[12] Cédric Walker. Recognising structural patterns in code — a k-means clustering approach. Bachelor's thesis, University of Bern, August 2016.

# A

## A.1 CompositeGenerator overview

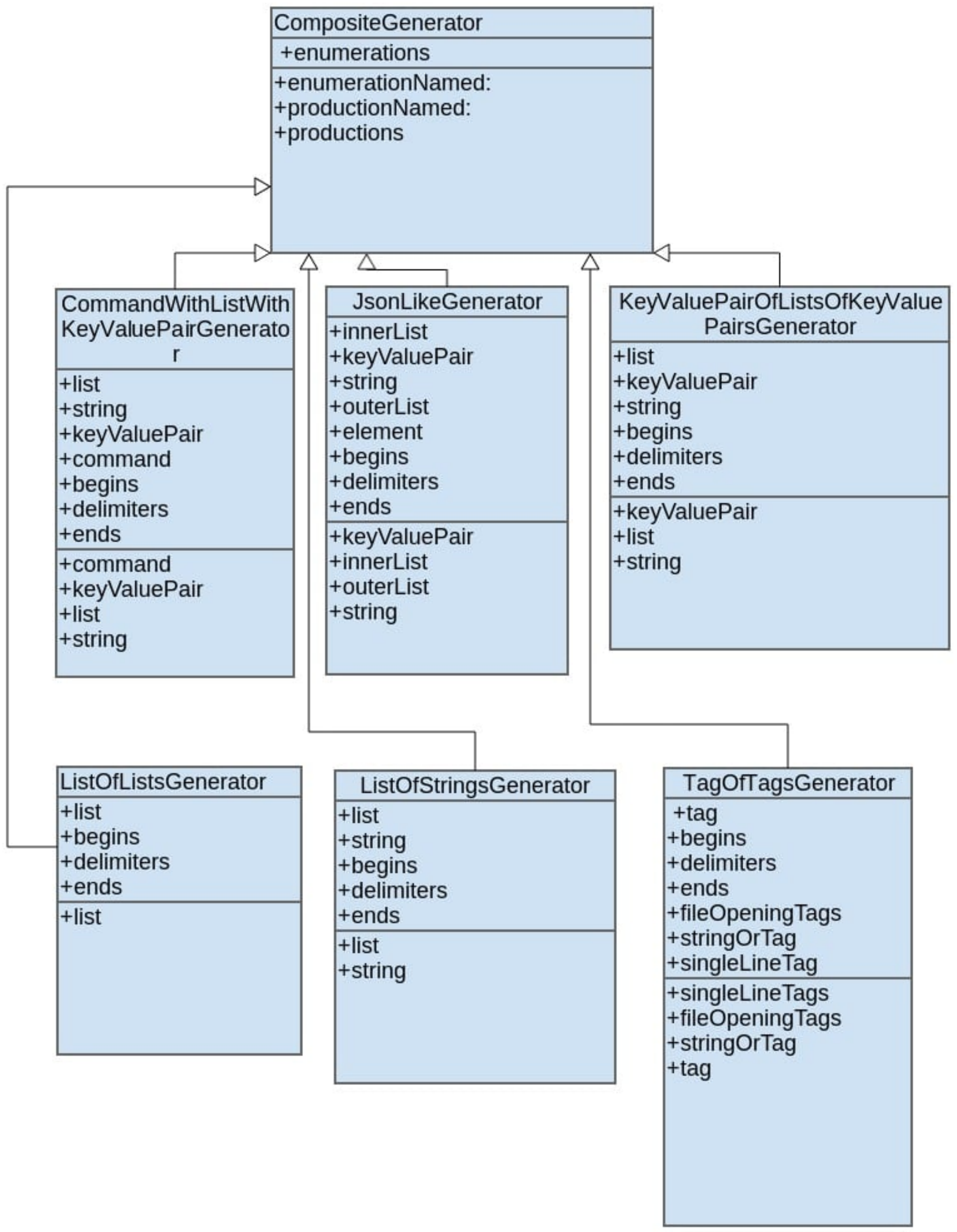


Figure A.1: Overview of the CompositeParserGenerator implementation

# B

## B.1 Validator

To do the validation we built a validator, which takes the result from the generated parser (achieved result) and the correct result from the exact parser (correct result). It compares both results to calculate precision,recall and f-measure.

## B.2 Base-files for the experiment

```
<?doctype stuff?>
<! other stuff>
<a attribute="value">
    <b x="y">c </b>
</a>
<d att="val">e</d>
```

Listing B.1: XML base file

```
.he .hu {
    random:c4;
    random2:s7 4;
    }
herlu {
    .huf {
        ha:45;
        hn:0.412;
        }
    }
```

Listing B.2: CSS base file

```
{
    "glossary": {
        "title": "example glossary",
        "GlossDiv": {
            "title": "S",
            "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                    "SortAs": "SGML",
                    "GlossTerm": "Standard Generalized Markup
                                   Language",
                    "Acronym": "SGML",
                    "Abbrev": "ISO 8879:1986",
                    "GlossDef": {
                        "para": "A meta-markup language, used
                          to create markup languages such as
                          DocBook.",
                        "GlossSeeAlso": ["GML", "XML"]
                    },
                    "GlossSee": "markup"
                }
            }
        }
    }
}
```

Listing B.3: JSON base file

## B.3 List of files used for the experiment

All the files used can be found on
`https://github.com/Icewater1337/RecognisingStructuralPatterns/tree/master/exampleFiles`. Below is an overview of the files and their length in characters.

| File | File-length |
|---|---|
| XML1 | 430 |
| XML2 | 822 |
| XML3 | 1029 |
| XML4 | 1496 |
| XML5 | 1974 |
| XML6 | 4342 |
| XML7 | 4548 |
| XML8 | 8706 |
| XML9 | 74393 |
| XML10 | 256545 |

Table B.1: XML experiment file sizes in characters

| File | File-length |
|---|---|
| CSS1 | 145 |
| CSS2 | 636 |
| CSS3 | 1005 |
| CSS4 | 1097 |
| CSS5 | 2230 |
| CSS6 | 2431 |
| CSS7 | 4946 |
| CSS8 | 8530 |
| CSS9 | 75080 |
| CSS10 | 126172 |

Table B.2: CSS experiment file sizes in characters

| File | File-length |
|---|---|
| JSON1 | 630 |
| JSON2 | 898 |
| JSON3 | 924 |
| JSON4 | 1238 |
| JSON5 | 1502 |
| JSON6 | 1725 |
| JSON7 | 3558 |
| JSON8 | 4493 |
| JSON9 | 7361 |
| JSON10 | 30143 |

Table B.3: JSON experiment file sizes in characters

## B.4 How to reproduce the experiment

The whole project including the test files can be found in this repository: `https://github.com/Icewater1337/RecognisingStructuralPatterns`. The files are stored in a .txt format.

To reproduce the experiment execute the following steps in order:

1. Download and install Moose `http://www.moosetechnology.org/`.
2. Clone the git repository.
3. Extract the folder *exampleFiles* to your Moose working directory.
4. Open a clean Moose image.
5. Open Monticello Browser.
   (a) Click on +repository
   (b) Choose 'filetree://'
   (c) Select the 'repository' folder of the cloned repository
   (d) load the RecognisingStructuralpatterns.package
6. In the *playground* in Moose, execute
   (a) For single files: TestFilesValidator new singleValidate:'XML'/'CSS'/'JSON'. Depending on which files you want to evaluate.
   (b) For pair parsing: TestFilesValidator new pairValidate:'XML'/'CSS'/'JSON'. Depending on which files you want to evaluate.

# C
# Anleitung zu wissenschaftlichen Arbeiten

This chapter contains a tutorial on how to use Crassula. Our tool Crassula has two main functions:

- Simplify the process of parser building with the ParserFactory.
- Automatic structure recognition of a given data serialisation format with the ParserGenerator.

## C.1 How to build parsers using the ParserFactory

### C.1.1 Installation

1. Clone the repository from github: `https://github.com/Icewater1337/RecognisingStructuralPatterns`
2. Download and install Moose `http://www.moosetechnology.org/` (Pharo 6.0 sources and a Moose 6.1.image file).
3. Open a clean Moose image.
4. Open Monticello Browser.
   (a) Click on +repository
   (b) Choose 'filetree://'

(c) Select the 'repository' folder of the cloned repository

(d) load the RecognisingStructuralpatterns.package

To understand and use the ParserFactory the user has to have basic knowledge of PetitParser. A good explanation can be found in the technical background chapter of this thesis or in the deep into Pharo book [7].

### C.1.2 Overview over the parser factories

With the elements of the ParserFactory we can easily build a parser for a given data serialisation format.

Each implementor of the ParserFactory contains the following methods:

- The **generate** method, which builds the parser for the user, taking a set of arguments.
- The **actionOn:** method which creates the intermediate representation of the parser. It can changed as needed.
- The **generateParser** method which creates the parser and calls actionOn: on it.

In the following subsections we show the important code of each parserFactory and make an example on how to use it.

#### C.1.2.1 StringParserFactory

The core of the StringParserFactory looks like this:

```
generate
    ˆ begin, ((escape,begin) / end negate) star token, end
```

Listing C.1: Implementation of the StringParserFactory in Moose using PetitParser

The StringParserFactory is used to parse strings *e.g.,* "This is a string". As we can see the StringParserFactory needs the following arguments:

- A **begin** parameter, which marks the beginning of the desired string *e.g.,* " or '.
- A **end** parameter, which marks the end of the desired string *e.g.,* " or '. A string mostly starts and ends with the same character.
- An **escape** parameter, which escapes certain characters so they can be used inside a string *e.g.,* \.

The StringParserFactory accepts all input parameters in the form of a PPParser instance.

The code to create a new instance of a StringParserFactory in Pharo:

```
doubleQuoteString:= (StringParserFactory new)
        begins: $" asParser;
        end: $" asParser;
        escape: ('\' asParser / '"'asParser );
        yourself
```

Listing C.2: Creation of a StringParserFactory in Pharo

The above code creates a StringParserFactory object. With the generate method a new instance of a PPParser can be created, or with the generateParser method a new instance of a PPParser can be created where the actionOn: method of the factory will be called in addition.

The default parser built by the StringParserFactory is the doubleQuoteString, which parses for strings that start with double quotes. There are some pre defined parsers for the StringParserFactory:

- The **Identifier**. This parser can be directly generated by calling StringParserFactory identifier. It is used to parse simple identifiers *e.g.,* ident-ifier or iAmAnIdentifier.
- The **IdentifierDigit**. This parser can be directly generated by calling StringParserFactory identifierDigit. It is the same as an identifier, but it can contain digits *e.g.,* id3ntifi3r.
- The **SingleQuoteString**. This parser can be directly generated by calling StringParserFactory singleQuoteString. It generates a parser that can parse strings that start with single quotes *e.g.,*'hello'.

#### C.1.2.2 ListParserFactory

The core of the ListParserFactory looks like this:

```
generate
    | parser |
    parser := begin trim,
    element optional, ((delimiter trim optional, element)
   nonEmpty star),
    end trim

    map: [ :_open :_el :_coll  :_close |
        | retval |
        retval := OrderedCollection new.
        _el isNil ifFalse: [ retval add: _el].
        retval addAll: (_coll collect: #second).
        Array with: 'LIST#', _open start asString with:
    retval
     ].

^ parser
```

Listing C.3: Implementation of the ListParserFactory in Moose using PetitParser

The ListParserFactory is used to parse lists *e.g.,* (1,2,3,4).
As we can see the ListParserFactory needs the following arguments:

- A **begin** parameter, which marks the beginning of the desired list *e.g.,* `(`.
- A **end** parameter, which marks the end of the desired list *e.g.,* `)`.
- A **delimiter** parameter, which separates different list elements *e.g.,* `,` or `;`.
- An **element** parameter, which states what a list element is.

The ListParserFactory accepts all input parameters in the form of a PPParser instance.
The code to create a new instance of a ListParserFactory in Pharo:

```
listParserFactory:= (ListParserFactory new)
        begins: $( asParser;
        end: $) asParser;
        delimiter: $, asParser;
        element: NilParserFactory new generateParser;
        yourself.
```

Listing C.4: Creation of a ListParserFactory in Pharo

The above code creates a ListParserFactory object. With the generate method a new instance of a PPParser can be created, or with the generateParser method a new instance of a PPParser can be created where the actionOn: method of the factory will be called in addition.

#### C.1.2.3 TagParserFactory

The core of the TagParserFactory looks like this:

```
generate
    ˆ (begin,
           tagName,
      end trim) wrapped,
        element nonEmpty star,
    (begin trim,  endTagSymbol,
        tagName,
    end trim) wrapped

    map: [ :_open :_content :_close |
        _open second = _close third
            ifTrue: [ { _open . _content . _close } ]
            ifFalse: [PPFailure message: 'Expected same
   keyword']
    ]
```

Listing C.5: Implementation of the TagParserFactory in Pharo using PetitParser

The TagParserFactory is used to parse tags *e.g.,* <a>content </a >. The last part of the code is used to ensure that the name of the begin tag and the end tag are the same.

As we can see the TagParserFactory needs the following arguments:

- A **begin** parameter, which marks the beginning of the desired tag *e.g.,* <.
- A **end** parameter, which marks the end of the desired tag *e.g.,* >.
- A **tagName** parameter, which is the representation of the tag name *e.g.,* in <a >the `a` is the tag name. The default implementation just parses anything that is not the endTagSymbol.
- An **endTagSymbol** parameter, which helps to mark the end tag *e.g.,* in </a >the `/` is the endTagSymbol.
- An **element** parameter, which represents the content of a tag.

The TagParserFactory accepts all input parameters in the form of a PPParser instance.

The code to create a new isntance of a TagParserFactory in Pharo:

```
lessGreaterThanTag:= (TagParserFactory new)
        begins: $< asParser;
        end: $> asParser;
        endTagSymbol: $/ asParser;
        element: NilParserFactory new generateParser;
        tagName: $> negate star token;
        yourself
```

Listing C.6: Creation of a StringParserFactory in Pharo

The above code creates a TagParserFactory object. With the generate method a new instance of a PPParser can be created, or with the generateParser method a new instance of a PPParser can be created where the actionOn: method of the factory will be called in addition. The created parser for the above example will parse something like this: `<name >anything </ name >`.

#### C.1.2.4 CommandParserFactory

The core of the CommandParserFactory looks like this:

```
generate
    ˆ key,arguments star
```

Listing C.7: Implementation of the CommandParserFactory in Moose using PetitParser

The CommandParserFactory is used to parse commands *e.g.,*
selector {
key:value;
key2:value2; }

As we can see the CommandParserFactory needs the following arguments:

- A key parameter, which represents the name of the command.
- A arguments parameter, which represents the content of the command.

The CommandParserFactory accepts all input parameters in the form of a PPParser instance.

The code to create a new instance of a CommandParserFactory in Pharo:

```
commandParserFactory:= (CommandParserFactory new)
        key: NilParserFactory new generateParser;
        arguments: ListParserFactory new generateParser;
        yourself
```

Listing C.8: Creation of a CommandParserFactory in Pharo

The above code creates a CommandParserFactory object. As we can see the command-ParserFactory can easily be a composite of other factories. Common arguments for it could be a list of elements (in the example a parser created by the ListParserFactory).

With the generate method a new instance of a PPParser can be created, or with the generateParser method a new instance of a PPParser can be created where the actionOn: method of the factory will be called in addition.

#### C.1.2.5 KeyValueParserFactory

The core of the KeyValueParserFactory looks like this:

```
generate
    ˆ key, delimiter trim, value.
```

Listing C.9: Implementation of the KeyValueParserFactory in Moose using PetitParser

The KeyValueParserFactory is used to parse key-value pairs *e.g.,* key: ”value”.
As we can see the KeyValueParserFactory needs the following arguments:

- A **key** parameter, which contains the ’key’ part of the key-value pair *e.g.,* a string.
- A **value** parameter, which contains the ’value’ part of the key-value pair *e.g.,* a string.
- A **delimiter** parameter, which separates the key and the value. *e.g.,* `:`.

The KeyValueParserFactory accepts all input parameters in the form of a PPParser instance.

The code to create a new instance of a KeyValueParserFactory in Pharo:

```
keyValueParserFactory:= (KeyValueParserFactory new)
        key: ’key’ asParser;
        value: ’value’asParser;
        delimiter: $: asParser;
        yourself.
```

Listing C.10: Creation of a KeyValueParserFactory in Pharo

The above code creates a ListParserFactory object. With the generate method a new instance of a PPParser can be created, or with the generateParser method a new instance of a PPParser can be created where the actionOn: method of the factory will be called in addition. The created parser for the above example will parse something like this: `key:value`.

#### C.1.2.6 NilParserFactory

The NilParserFactory is a factory that can be used whenever we want to parse something as water *i.e.,* whenever we want to skip everything until the next element of interest in the parser appears. This is handy when we want to parse for example everything up to the starting point of a list *e.g.,* jajh,.asfj list. In this case we could use the NilParserFactory to parse everything up to the starting point of the list.

### C.1.3 ChoiceParserFactory

Of course we can combine parser factories to create a new ParserFactory. The ChoiceParserFactory handles this case. An example:
combinedFactory := StringParserFactory new / ListParserFactory new.

This will produce a new ParserFactory instance which then can be used to create a parser again.

### C.1.4 How to build a parser using the parser factories

To build a parser with the parser factories we can simply look at a given code and analyze it to see which elements we need to build a parser for the given file:

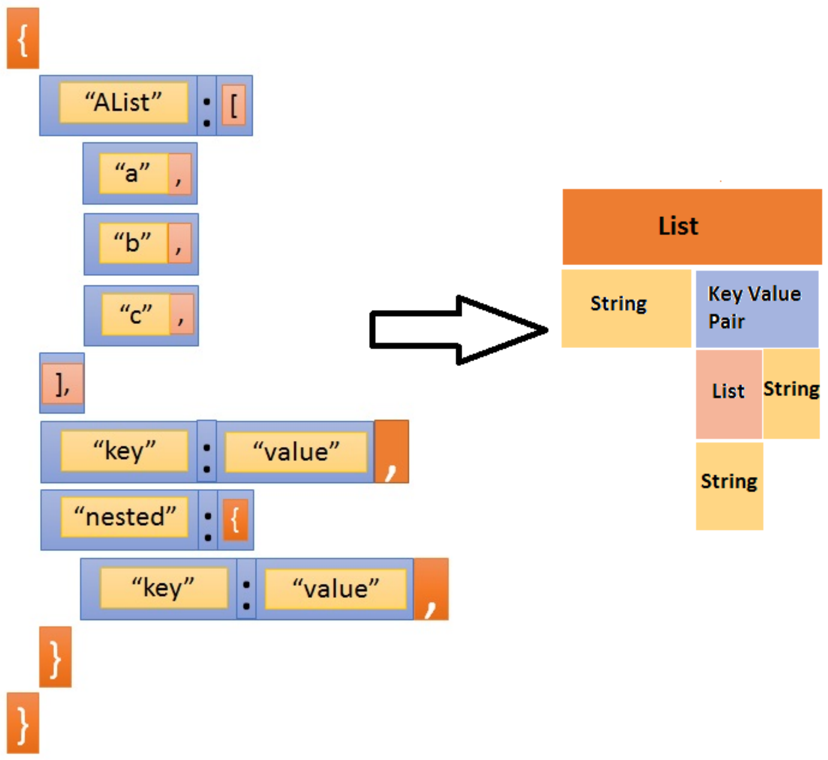


Figure C.1: JSON example with different structural elements colored and labeled

JSON consists of four different elements: *key-value pairs, strings and two different lists*. We called the two different lists *jsonArray* and *jsonObject*. The jsonArray starts with { and the jsonObject starts with `[`.

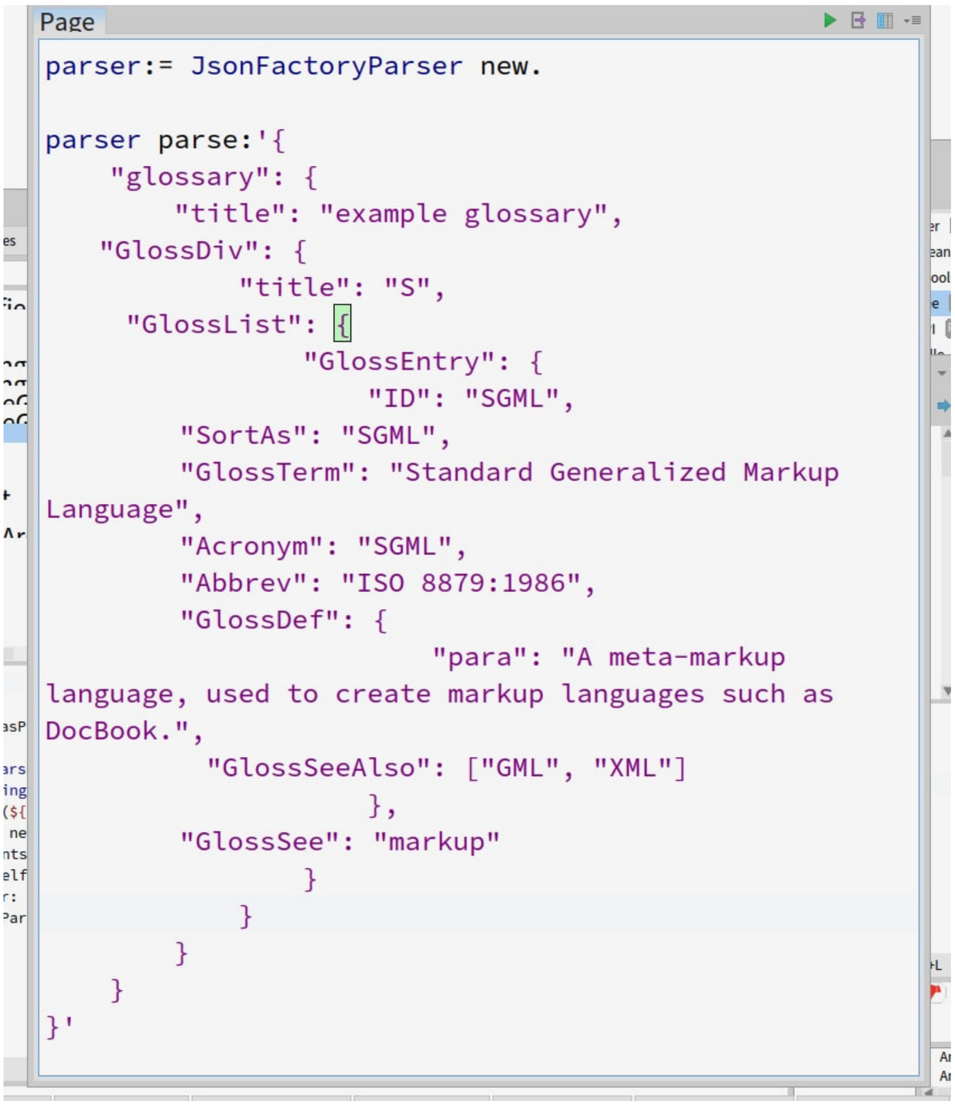
Knowing what the elements are we can now build a parser using the ParserFactories:

```
start
 ^ jsonObject / jsonArray


keywordValuePair
    ^ (KeyValueParserFactory new)
        key: StringParserFactory doubleQuoteString;
        delimiter: $: asParser;
        value: StringParserFactory doubleQuoteString/
   jsonArray /jsonObject / StringParserFactory identifier /
   StringParserFactory identifierDigit;
        generateParser
```

```
jsonObject
    ˆ (ListParserFactory new)
        begins:${ asParser token;
        end:$} asParser;
        delimiter:$, asParser;
        element: keywordValuePair;
        generateParser


jsonArray
    ˆ (ListParserFactory new)
        begins:$[ asParser token;
        end:$] asParser;
        delimiter:$, asParser;
        element: jsonObject /jsonArray / StringParserFactory
   doubleQuoteString / StringParserFactory identifierDigit /
   StringParserFactory identifier ;
        generateParser
```

The start method will be called first when we call `parse:` on the generated parser. Because JSON can start with either `[` or with {, we included jsonObject OR jsonArray in the start method.

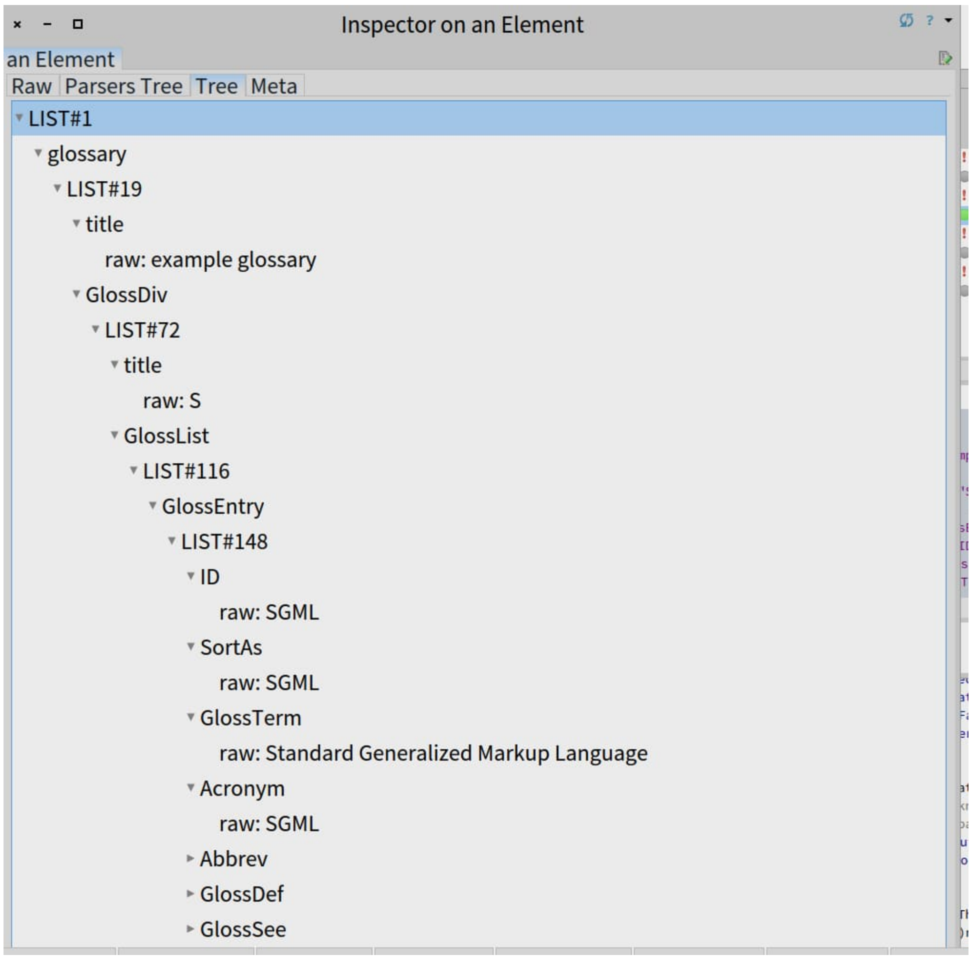
We can use this parser to parse JSON files in Pharo:



```
parser:= JsonFactoryParser new.

parser parse:'{
    "glossary": {
        "title": "example glossary",
   "GlossDiv": {
            "title": "S",
     "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
        "SortAs": "SGML",
        "GlossTerm": "Standard Generalized Markup
Language",
        "Acronym": "SGML",
        "Abbrev": "ISO 8879:1986",
        "GlossDef": {
                        "para": "A meta-markup
language, used to create markup languages such as
DocBook.",
          "GlossSeeAlso": ["GML", "XML"]
                    },
        "GlossSee": "markup"
                }
            }
        }
    }
}'
```

Figure C.2: Parsing a JSON example with the parser that was built using the ParserFactories

This example uses the default Node representation of the output *i.e.*, the output will be represented in the tree form. It looks like this in Pharo:



```
▾ LIST#1
  ▾ glossary
    ▾ LIST#19
      ▾ title
          raw: example glossary
      ▾ GlossDiv
        ▾ LIST#72
          ▾ title
              raw: S
          ▾ GlossList
            ▾ LIST#116
              ▾ GlossEntry
                ▾ LIST#148
                  ▾ ID
                      raw: SGML
                  ▾ SortAs
                      raw: SGML
                  ▾ GlossTerm
                      raw: Standard Generalized Markup Language
                  ▾ Acronym
                      raw: SGML
                  ▸ Abbrev
                  ▸ GlossDef
                  ▸ GlossSee
```

Figure C.3: Structure output of the JSON example represented in Moose

## C.2 ParserGeneratorExecutor tutorial

The usage of the ParserGeneratorExecutor is fairly simple and straight forward. There are the following commands you can use on a file:

- The **parse** method, which parses a file and returns all the possible solutions.
- The **parseWithorder** method, which parses a file using the parse method. The solutions are ordered using different weights for the different elements. This increases the chance that the best parser is more on top of the results list.
- The **parseList** method, which is used to parse a list of files from the same format.

- The **parseListWithorder** method, which is the same as the parseWithOrder method, but using a list of files.
- The **debug** method, which returns a list of debug instances instead of parsed files.

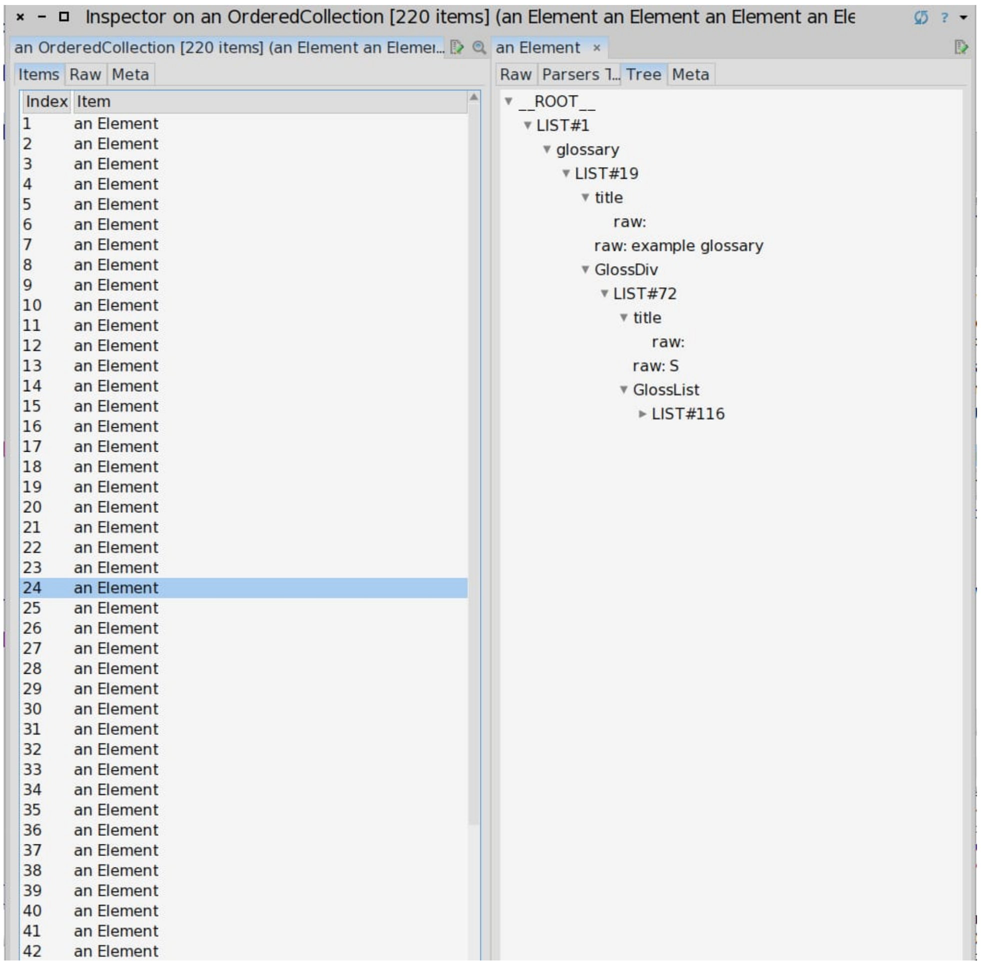
The functions can be executed straight forward:

```
ParserGeneratorExecutor new parseWithOrder: '<a>b</a>'.
```

where <a >b </ a > is an XML example, but can be any given data serialisation format.

When using `parseList` or `parseListWithOrder` the input is a collection of strings, instead of only one string.

The expected output in Pharo when using `parseWithOrder`:



```
Inspector on an OrderedCollection [220 items] (an Element an Element an Element an Ele

an OrderedCollection [220 items] (an Element an Elemei...    an Element

Items Raw Meta                                               Raw Parsers T... Tree Meta

Index Item                                                   ▾ __ROOT__
1     an Element                                               ▾ LIST#1
2     an Element                                                 ▾ glossary
3     an Element                                                   ▾ LIST#19
4     an Element                                                     ▾ title
5     an Element                                                         raw:
6     an Element                                                       raw: example glossary
7     an Element                                                     ▾ GlossDiv
8     an Element                                                       ▾ LIST#72
9     an Element                                                         ▾ title
10    an Element                                                             raw:
11    an Element                                                           raw: S
12    an Element                                                         ▾ GlossList
13    an Element                                                           ▸ LIST#116
14    an Element
15    an Element
16    an Element
17    an Element
18    an Element
19    an Element
20    an Element
21    an Element
22    an Element
23    an Element
24    an Element
25    an Element
26    an Element
27    an Element
28    an Element
29    an Element
30    an Element
31    an Element
32    an Element
33    an Element
34    an Element
35    an Element
36    an Element
37    an Element
38    an Element
39    an Element
40    an Element
41    an Element
42    an Element
```

Figure C.4: ParserGeneratorExecutor parseWithOrder: output example in Pharo