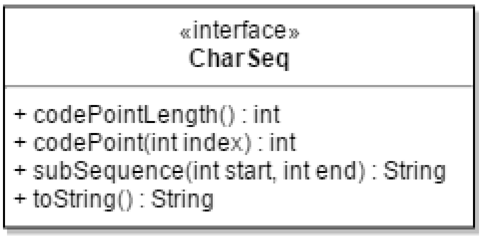
moar Implementation Details

The Regular Expressions (Regexes) used in Java, C++, etc. are mostly based on Perl Regexes. They are more powerful than the ones known in language theory and describe a superset of the regular languages. Perl Regexes, however are non deterministic by design, and have therefore to be handled with care in user-facing applications (so that a specific user input can not crash the complete program/website). In this case specific deterministic Regex engines are used, which however don't support backreferences - a feature that can be useful in some applications. Dominik and Markus proposed the notion of deterministic Regexes with backreferences that work with deterministic Memory Occurence Automata (or MOA, these can be thought of as regular automata with variables).

Moar is the first implementation of Regular Expressions based on deterministica MOAs. It aims to support all features provided in Dominik & Markus' paper while also shipping with an API that is inspired by Java's Pattern class. However, we only support a subset of Regexes compared to Java Patterns, in order to simplify the Grammar.

Utility classes/interfaces

CharSeq



The CharSeq interface is an abstraction from the CharSequence interface that is generally used in Java programs and enables us to support other inputs as well (for example input from a byte sequence like in the Lucene module which can be found in the git repository). Moar only works with codePoints instead of chars. This means that we can support UTF-32 and helps us with unicode character duplicates.

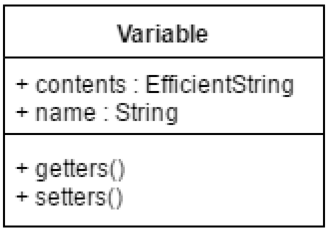
How are MOAs represented in Code?

In moar, a MOA (Java class: com.github.s4ke.moar.Moa) consists of a set of Variables (basic placeholders for the Variables used in the MOA, nothing fancy) and a Graph representation of the automaton (com.github.s4ke.moar.moa.edgegraph.EdgeGraph).

Basic classes

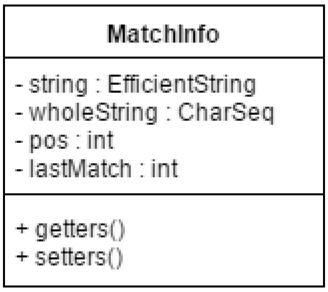
In order to get into how the Graph representation is implemented, we first have to explain two basic data classes: Variable and MatchInfo.

Variable



Objects of this type just holds the current contents of the Variables and is implemented as basic as it gets.

MatchInfo



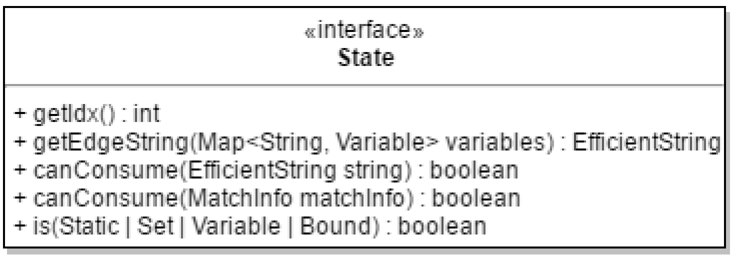
MatchInfo objects are used to hold information about where in the input string the matching process is currently at. The fields are:

- string: what character (or in some cases characters) are currently supposed to be matched. The EfficientString wrapper around character sequences allows us to represent subsequences in an efficient manner by only storing the start and end index together with a reference to the original sequence.
- wholeString: the whole input. The CharSeq interface is used so we can wrap away details of what we are matching. This way we can also support for example Byte-Wise character sequences instead of the default Java Strings.
- pos: the current position in the input string
- lastMatch: the index in the string where the last match ended (otherwise this is set to -1)

The Graph representation

States

We shall now take a look at the nodes of our MOA Graph, the states:



(Note: getIdx() is the equivalent of the markings in a marked alphabet. This is not really necessary for the implementation, but helps us to identify states faster)

As we can see, the State interface has three different methods that look like they are related to identifying what has to be read in order to go to them during evaluation. However not every method can be used for every type of state. But to explain this we have to go over the different State implementations. After that we give the reasoning behind the three different methods.

Static (Basic) States

A static State is the code representation of basic character only states. These states can only contain singular characters.

Set States

The theoretical model has no need for these states as it doesn't support character classes. But we want to support these in our Regexes and without creating unnecessarily big numbers of basic states to represent them, which would also mean a high memory footpring. We give a short explanation of character classes in Regexes later, but for now, we just think of them as a Set/range of allowed characters.

In our implementation we represent basic Sets (not often needed, mainly for internal things) via the Java Collection Set and Ranges (like [a-ce-z]) by a Google Guava TreeRangeSet which is space efficient as it only stores the ranges of allowed characters instead of all of them. It also already comes with a negation implementation which helps us with negative sets (e.g.: to allow everything but a's).

In our implementation we use `canConsume(EfficientString string) : boolean` for Static and Set states to check whether the current character can be consumed. The implementation for this is just a basic equality (or containment for the Set state) check.

Variable States

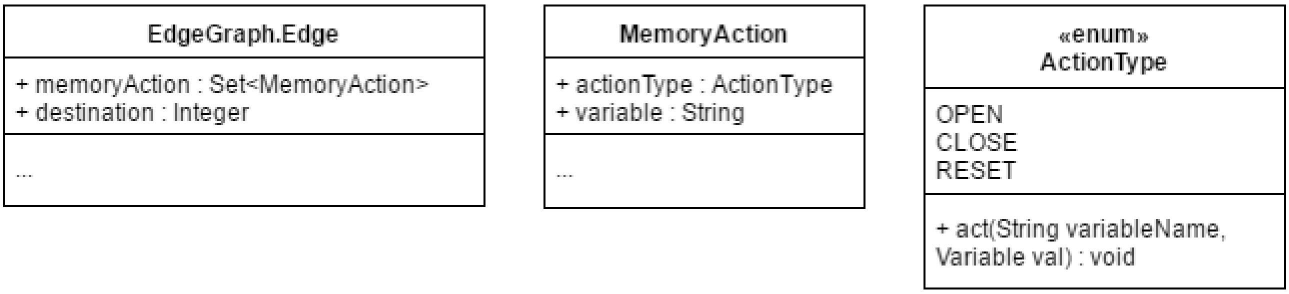
Variable States are the code representation of backreferences and only need to have the variable name as a member. As the variables can contain input longer than one character we don't have a separate "canConsume" method. In our matching algorithm we first get the character sequence we have to match in order to traverse into a variable state via `getEdgeString(Map<String, Variable>) : EfficientString` and then check for equality in the remaining input.

Bound States

Bound states are used to represent boundary checks in Regexes. We can think of them as basic checks like "am I at the beginning of the input?" or "am I at the end of the input?" (the ^ and the $ anchor in typical regex implementations, respectively). For these we use the `canConsume(MatchInfo matchInfo) : boolean` method during matching.

Edges

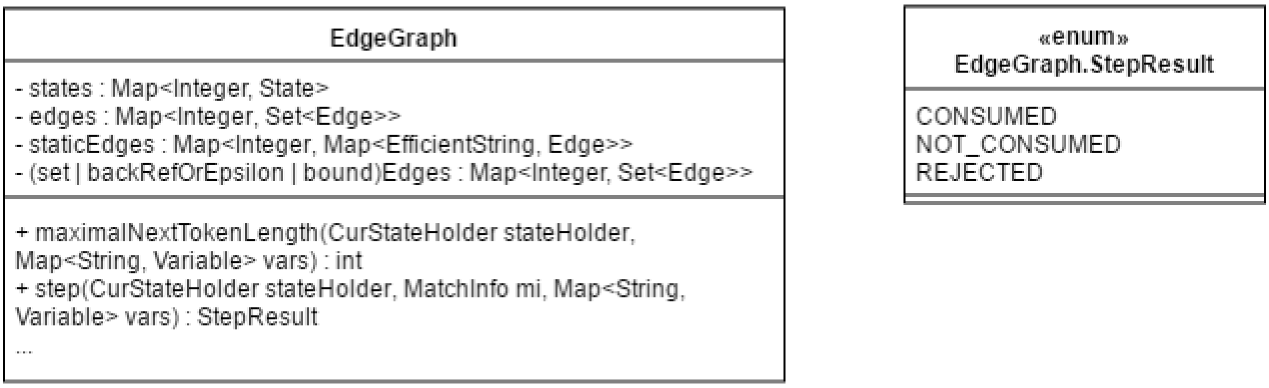
The edges of the MOAs are represented by a simple Edge object in the EdgeGraph:



Every Edge has a set of MemoryActions that determine just like in the theoretical model how the variable state(s) should change if the edge is used.

EdgeGraph

The EdgeGraph is the basic in-code representation of the MOA Graph:



It stores all the states (field: states) with their corresponding Edges (field: edges) which are duplicated into the other *Edges fields for more convenient and faster access during evaluation.

Note: CurStepHolder is an abstraction enables the EdgeGraph to be reused as it doesn't need to store the current state.

It's most prominent methods are:

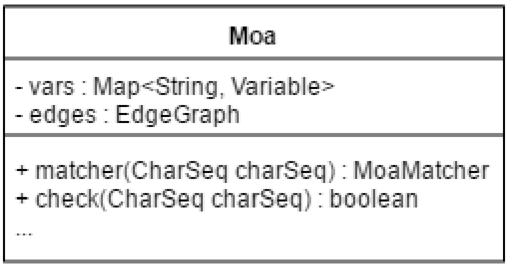
- `maximalNextTokenLength(CurStateHolder stateHolder, Map<String, Variable> vars) : int`

  This method is used to compute the length of the next character sequence ("token") to be read. This is primarily needed because of the fact that Variable States can have tokens of any length and because we can have epsilon edges to the sink (just like in the theoretical model, the MOA always has a source and a sink, these are special unique State objects which we talk about in the Regex chapter that also explains how the MOAs are meant to be created). If there are only transitions to Basic or SetStates this returns 1.

- `step(CurStateHolder stateHolder, MatchInfo mi, Map<String, Variable> vars) : StepResult`

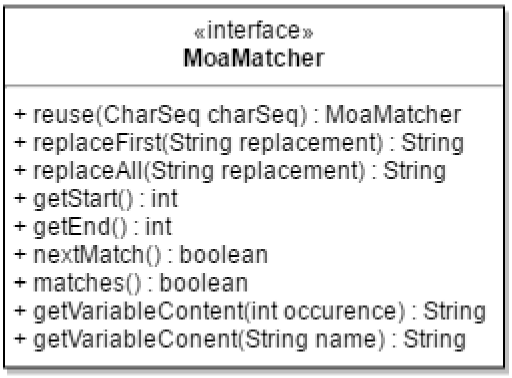
  The EdgeGraph tries to perform a step with the current step represented in the stateHolder object with the given info of the MatchInfo object (position, current "token", see above) and the current variable state. Its possible return values are CONSUMED (success), NOT_CONSUMED (the current token was not consumed, this is used for boundary checks which do not consume anything) and REJECTED (no valid transition could be found).

The Moa class



The Moa class serves as an intermediary entry point for matching. Essentially it could be directly used, but is a bit rough on the edges (therefore it is wrapped into the MoaPattern class, which is explained in the Regexes & Pattern API chapter). Its two most prominent methods are `matcher(CharSeq charSeq) : MoaMatcher` which creates a MoaMatcher object on the given input and the `check(CharSeq charSeq) : boolean` method which immediately checks the input for a complete match.

The Matcher

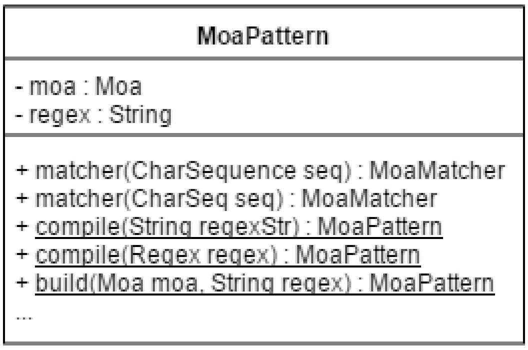


The MoaMatcher interface grants the user more detailed control over how the matching is done. With this interface the user can:

- check for a full-string match with `matches() : boolean`
- check for the next match with `nextMatch() : boolean` (this tries to find the longest sequence in the input that is in the language of the MOA)
- replace the first match of the MOA in the input with `replaceFirst(String replacement) : String` (or all matches with `replaceAll(String replacement) : String`)
- retrieve the variable content via the `getVariableContent(...) : String` methods
- retrieve the start (`getStart() : int`) and the end (`getEnd() : int`) of the last match

It is also meant to be reused (see `reuse(CharSeq charSeq) : MoaMatcher`).

The MoaPattern



This is the main entry class for users and has the same purpose as Java's Pattern class. It doesn't add new functionality over the Moa class, but wraps it into a more beautiful API. It has two separate compile methods that allow for creation from a DSL and from a Regex String. The `build(Moa moa, String regex) : MoaPattern` method is mostly meant for integrators like the JSON export/import module.

Regexes & Pattern API

Our deterministic Regexes can be translated into code in two separate ways:

With a Java DSL:

```
MoaPattern moa = MoaPattern.compile(
                    // DSL style generation of the Regex
                    Regex.reference( "x" )
                        .bind( "y" )
                        .and( Regex.reference( "y" ).and( "a" ).bind( "x" ) )
                        .plus()
                 );
```

This chaining DSL was initially created to be able to create Regexes without any additional Parser. It is an in code representation of a super set of deterministic Regexes (this means for this to be useful in the deterministic context, we have to check for determinism during compilation). Internally it produces a direct representation of the abstract syntax tree of the regex. For a complete list of the supported methods, take a look at the `com.github.s4ke.moar.regex.Regex` class.

Later, we shall explain how this is translated into a MOA.

With a Java Pattern-like Regex string:

```
// building a Regex from a String
MoaPattern moa = MoaPattern.compile("((?<y>\k<x>)(?<x>\k<y>a))+");
```

The parsing of this Regex string is done via ANTLR v4 and internally uses the DSL API (above). The Grammar is the following:

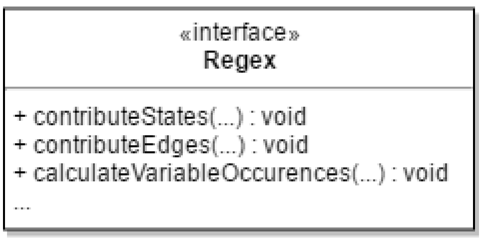
```
grammar Regex;

/**
 * Grammar for parsing Perl/Java-style Regexes
 * after: http://www.cs.sfu.ca/~cameron/Teaching/384/99-3/regexp-plg.html
 * but with left recursion eliminated
 */

regex:
    EOF
    | startBoundary? union endBoundary? EOF;

startBoundary :
    START
    | prevMatch;
prevMatch : ESC 'G';

endBoundary :
    EOS
    | endOfInput;
endOfInput : ESC 'z';

union :
    concatenation
    | union '|' concatenation;

concatenation :
    basicRegex
    | basicRegex concatenation;

basicRegex :
    star
    | plus
    | orEpsilon
    | elementaryRegex;

star :
    elementaryRegex '*';
plus :
    elementaryRegex '+';
orEpsilon:
    elementaryRegex '?';

elementaryRegex :
    backRef
    | group
    | set
    | charOrEscaped
    | stockSets
    | ANY;
```

```
group :
    '(' (capturingGroup | nonCapturingGroup) ')';
capturingGroup : ('?' '<' groupName '>')? union?;
nonCapturingGroup: '?' ':' union?;
groupName : character+;

backRef :
    ESC number
    | ESC 'k' '<' groupName '>';

set :
    positiveSet
    | negativeSet;
positiveSet : '[' setItems ']';
negativeSet : '[^' setItems ']';
setItems :
    setItem
    | setItem setItems;
setItem :
    range
    | charOrEscaped;
range :
    charOrEscaped '-' charOrEscaped;

//should these be handled in the TreeListener?
//if so, we could patch stuff easily without changing
//the grammar
stockSets:
    whiteSpace
    | nonWhiteSpace
    | digit
    | nonDigit
    | wordCharacter
    | nonWordCharacter;
whiteSpace : ESC 's';
nonWhiteSpace : ESC 'S';
digit : ESC 'd';
nonDigit : ESC 'D';
wordCharacter : ESC 'w';
nonWordCharacter : ESC 'W';

charOrEscaped :
    character
    | escapeSeq
    | UTF_32_MARKER utf32 UTF_32_MARKER;
// this odd separation of unused chars and "used chars" is due to ANTLR
// processing in two phases. At first, only the token rules (in CAPS) are
// and then the rules are used. For normal grammars (for programming
languages)
// this is fine, but in our case this means this extra (and ugly) work.
```

```
// Due to this, every single char that is to be matched must be tokenized
// (the ones that are not part of a NAMED token rule are just implicitly made
// into their own token rule). Every "non special" char
// (the ones that are not explicitly mentioned) is therefore tokenized
// into UNUSED_CHAR.
// This approach is by far easier than a hand written parser, though.
character : (UNUSED_CHARS | ZERO | ONE_TO_NINE | 's' | 'S' | 'd' | 'D' | 'w'
| 'W' | 'k' | 'z' | 'G' | ':' | '<' | '>' );
escapeSeq : ESC escapee;
escapee : '[' | ']' | '(' | ')'
     | ESC | ANY | EOS | START | UTF_32_MARKER
     | '*' | '+' | '?'
     | '-' ;
utf32 : (character | escapeSeq)+;

number :  ONE_TO_NINE (ZERO | ONE_TO_NINE)*;

ZERO : '0';
ONE_TO_NINE : [1-9];
ESC : '\\';
ANY : '.';
EOS : '$';
START : '^';
UTF_32_MARKER : '~';

UNUSED_CHARS :
     ~('0' .. '9'
     | '[' | ']' | '(' | ')'
     | '\\' | '.' | '$' | '^'
     | '*' | '+' | '?'
     | ':'
     | 's' | 'S' | 'd' | 'D' | 'w' | 'W' | 'k' | 'z' | 'G'
     | '~');
```

The MoaPattern class behaves in this context similar to Java's native Pattern class and wraps the internal MOA logic for the user. It serves as the entry point to build MoaMatcher objects (similar to Java's Matcher) which encapsulate all the matching logic so that the Pattern itself can be reused as its construction can be expensive.

From Regex to MOA



Now that we know how the Regexes are represented (as an AST), we shall talk about the process that is used to create a MOA from a Regex. This process is internally used by the MoaPattern class to compile the Regex it is given in the `compile(...)` method into a usable Moa.

Accumulate States

In the first phase of the Building process we start at the top of our AST representation of the Regex and let every Regex object contribute the states it has via `contributeStates(Map<String, Variable> variables, Set<State> states, Map<Regex, Map<String, State>> selfRelevant, Supplier<Integer> idxSupplier) : void`.

Trivial Sub-Regexes

For trivial sub Regexes (Primitive, SetRegex, BoundaryRegex, Epsilon) and also for Reference this means that the object has to create a new State with the index supplied by the idxSupplier object (in our case this is a method reference to the getAndIncrement function of a AtomicInteger object, but this can be changed ;)) and add it to the states set (The Reference Regex also creates the Variable it is referring to if it does not exist - this is needed due to the fact that references can occur before the variable is declared) .

In this method the Regex object also has to store the states that it needs later on in the creation in the selfRelevant Map (currently that's all the states that the object created).

Non Trivial Sub-Regexes

For the other non trivial Regexes (Plus, Concat and Choice, Binding) the implementation is quite basic as they don't require their own states. These just delegate to their own Sub-Regexes.

Accumulate Edges

In the second phase, now that all states are properly known, the edges are added to the moa via `contributeEdges( EdgeGraph edgeGraph, Map<String, Variable> variables, Set<State> states, Map<Regex, Map<String, State>> selfRelevant) : void`. The passed `edgeGraph` already has all the states properly added, so the Regex objects can directly work with the states passed in the `states` set (this is also due to us wanting seperation of concerns during the process of creating the MOA).

In a similar fashion to the accumulation of states the behaviours of the different Regex types differ (by nature of the creation). This is done more straight forward than the accumulation of the states (but uses some of the principles of the delegation to the Sub Regexes for non trivial parents) and every Regex contributes at least one edge in the process (but can be removed by an ancestor Regex).

One important thing is how the sources and sinks of the MOAs are handled, though: If an edge that includes at least one of them is needed, the Regex implementations are required to use the static fields `SRC` and `SNK`of the Moa class (or the EdgeGraph class, they are equivalent) as equality for SRC and SNK is generally checked by equivalence.

Note that the only part of the accumulation of the edges that differs greatly to the one in the paper is how Concat and Plus are built. In Moar this still relies on a older version of the paper that included a special function (see `com.github.s4ke.moar.moa.Moa#f(Set<MemoryAction> a1, Set<MemoryAction> a2) : Set<MemoryAction>`) that recomputed the memory actions. This approach is equivalent to the way this is done in the paper (the paper version should be more easy to understand), but has not been changed in Moar (,yet ?).

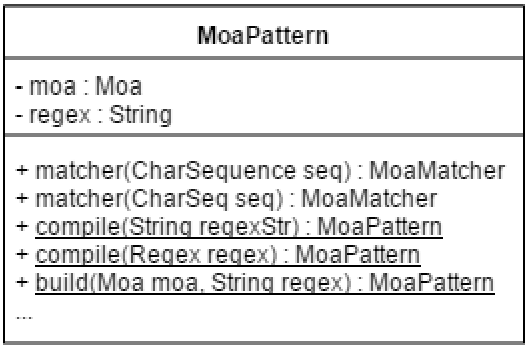
Determinism

As the Regex DSL can produce non deterministic Regexes as well, we have to make sure that we don't produce a MOA which is non deterministic while adding edges. These checks are implicitly done during compilation and a `NotDeterministicException` is thrown if non determinism is detected. Moar tries to help the user by providing the exact position in the MOA where the non determinism was detected (what Edges were tried to be added for which State).

Number variables

For the third phase, as we support References and data retrieval by index of the Binding in our Regexes as well, we have to number all variables. The algorithm is similar to the accumulation of states and edges but is only relevant for the Binding Regex (Plus, Concat and Choice just delegate to their children while all other Sub-Regexes - except for Bindings - have a no-op implementation of their respective `calculateVariableOccurences(Map<String, Variable> variables, Supplier<Integer> varIdxSupplier) : void` method).

Using MoaPatterns



Now that we know how the internals of MoaPatterns work, we can take a quick look into how they are meant to be used.

MoaPattern Creation

This was already discussed when we described how Regexes are represented in code: We use one of the two static compilation methods `MoaPattern#compile(String Regex) : MoaPattern`/ `MoaPattern#compile(Regex regex) : MoaPattern` and then usually **cache the returned object** as the compilation is quite costly.

Accessing the MoaMatcher

The MoaMatcher is easily accessed via the instance methods `MoaPattern#matcher(...)` which either take a CharSequence or an instance of the CharSeq abstraction we talked about earlier in this draft. How the MoaMatcher works is explained in the introductory chapter.

JSON Serialization

MoaPatterns can be serialized into a human readable format:

```
{
  "regex":"^(?<toast>[a-z]b[^b]\\w)\\k<toast>.$",
  "vars":["toast"],
  "states":[
            {"bound":"^","idx":2},
            {"set":"[a-z]","idx":3},
            {"name":"b","idx":4},
            {"set":"[^b]","idx":5},
            {"set":"\\w","idx":6},
            {"ref":"toast","idx":7},
            {"set":".","idx":8},
            {"bound":"$","idx":9}
  ],
  "edges":[
            {"from":0,"to":2},
            {"from":2,"to":3,"memoryActions":["o(toast)"]},
            {"from":3,"to":4},
            {"from":4,"to":5},
            {"from":5,"to":6},
            {"from":6,"to":7,"memoryActions":["c(toast)"]},
            {"from":7,"to":8},
            {"from":8,"to":9},
            {"from":9,"to":1}
  ]
}
```

Explanation for the JSON fields:

- regex

  This is for documentation purposes only and couldn't be anything else as the Regexes do not have the same expressive power as hand written MOAs.

- vars

  In this field all used vars of the MOA are listed as simple strings.

- states

  In this field the states are listed. "name" represents Basic states, "set" represents Set states, "ref" represents Variable states and "bound" represents boundary states (the list of supported bounds can be found in com.github.s4ke.moar.regex.BoundConstants). The "idx" must be unique and in the range and >= 2 as SRC and SNK are 0 and 1, respectively.

- edges

  In this field we can specify the edges and possible memoryActions (o = open, c = close, r = reset, used like in the example: r(x) - reset the variable x)

Serialization is done via `com.github.s4ke.moar.json.MoarJSONSerializer#toJSON(MoaPattern pattern) : String` and deserialization via `com.github.s4ke.moar.json.MoarJSONSerializer#fromJSON(String json) : MoaPattern`.

This feature is particularly useful as this allows users to not only hand-write MOAs but also enables them to transmit generic MoaPatterns between applications even if no Regex is available. One can also think of an extra application that allows users to create their own MOAs with a GUI which then are exported to this format for later usage.

Trying it out

The moar project also has a separate command line tool to try out the functionality (the cli module). Already compiled binaries of it can be found in the realease section on GitHub (https://github.com/s4ke/moar/releases).

Its usage is as follows:

```
C:\Users\Martin\Downloads>java -jar moar-cli-2016-08-23.jar
usage: moar-cli
 -d           only do determinism check
 -help        prints this dialog
 -ls          treat every line of the input string file as one string
 -m           multiline matching mode (search in string for regex)
 -mf <arg>    file/folder to read the MOA from
 -mo <arg>    folder to export the MOAs to (overwrites if existent)
 -r <arg>     regex to test against
 -rf <arg>    file containint the regexes to test against (multiple regexes
              are separated by one empty line)
 -s <arg>     string to test the MOA/Regex against
 -sf <arg>    file to read the input string(s) from
 -t           trim lines if -ls is set
```

The Lucene Module

Regexes are quite useful if we want to search for data. This is the reason we implemented a proof of concept for a MoaPattern Query for the popular fulltext search engine Lucene in the 'lucene' module. It is currently not as fast as Lucene's own Regex Query type because it does not do optimizations like stopping traversal of the search-tree if it encounters words that will never match. For example all remaining words start with a 'b' if the Regex expects an 'a'. This would require more specific tuning of the moar API for this specific use-case.

It can be used like this:

```
IndexSearcher is = ...
MoaPattern pattern = MoaPattern.compile( "(a*)b\\1" );
MoarQuery tq = new MoarQuery( "tag", pattern );
TopDocs td = searcher.search( query, 10 );
```

Just like with normal Lucene Queries we could now extract the Lucene documents from the TopDocs object. This however, can be looked up in any basic Lucene tutorial.