PatternJS — regular expressions for trees

Mihai Bazon

September 18, 2011

Contents

1 pattern.js -- regular expressions for trees

This is an implementation of a pattern matching engine similar in power to regular expressions but applicable to trees (array of arrays, in this case) rather than strings.

Based on ideas in an excellent paper by Russ Cox[1], we compile search queries to a home-made bytecode and provide a specialized VM to run the bytecode. The VM can

[1] http://swtch.com/rsc/regexp/regexp2.html

only say if a tree matches a query or not; to search, we apply the query to all the subtrees. Speed was not a goal for my implementation[2], but it's pretty fast by my tests.

2 Syntax by examples

This library is useful when you need to search something in an array and the search is more complicated than what indexOf can handle. In other words, when the search is even moderately complex, you need this tool.

For example, suppose you're looking for the sequence 3, 4, 5 in the following array:

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

You would like to know at which index it starts. You'll usually need to know where it ends too, but this is a simple expression that doesn't involve repetitions, so the end index is trivial to get—it's `START + LENGTH`.

With PatternJS we can use the following code:

```
var $ = require("pattern");
var haystack = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ];
var needle = $.compile(3, 4, 5);
$.search(haystack, needle, function onmatch(m){
    console.log(m.$node);              // == haystack in this case
    console.log(m.$start);             // 2
    console.log(m.$end);               // 5
    console.log(m.$match().content()); // [ 3, 4, 5 ]
});
```

The above code shows the basic API to this library. It exports two main functions: `compile` and `search`. The first takes a search query and returns the compiled bytecode. The `search` function takes the array to search, the bytecode to run, and an "onmatch" function. This function takes a single argument that contains various information about the match. Above I showed the following 4 properties:

- `$node` — the node that matched
- `$start` — the position where match starts
- `$end` — the position where the match ends
- `$match` — this returns a `Match` object. I'll describe them later. For now you can notice that it has a content() method, which returns the node content. In the case above, `$match().content()` is the same as `$node.slice($start, $end)`.

2.1 Repetition (`MANY`, `MORE`)

The above example is already a bit more than `indexOf` can handle, but as promised, this library is as powerful as regular expressions. Here's how we can handle repetition. Suppose we want to find blocks of one or more "a"-s in an array:

[2] Russ's article provides many optimization hints; for now, my implementation is the just simplest recursive version.

```
var a = [ 1, 2, "a", 3, 4, "a", "a", "a", "b", "a", "a", "c" ];
var pat = $.compile($.MORE("a"));
$.search(a, pat, function(m){
    // log where it starts, where it ends, and contents
    console.log(m.$start, m.$end, m.$match().content());
});
```

We get the following output:

```
2 3 [ 'a' ]
5 8 [ 'a', 'a', 'a' ]
9 11 [ 'a', 'a' ]
```

We can also search for repetitive expressions consisting of more than one node. For example, finding consecutive occurrences of 2, 3, 4 in an array:

```
var a = [ 1, 2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 3, 4, 5 ];
var pat = $.compile($.MORE(2, 3, 4));
$.search(a, pat, function(m){
    console.log(m.$start, m.$end, m.$match().content());
});
// ==> 1 13 [ 2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 3, 4 ]
```

That's right, there's a single contiguous match—the sequence 2, 3, 4 is repeated 4 times, starting at index 1 and ending at index 13.

`MORE` is in fact the equivalent of `+` in ordinary regexps. It matches its argument at least once. There is also a `MANY` combinator which is like `*` in regexps—it matches the argument zero or more times:

```
var a = [ 1, 2, 3, 2, 4, 4, 4, 1, 2, 4, 0, 2, 0, 1 ];
var pat = $.compile(2, $.MANY(4));
$.search(a, pat, function(m){
    console.log(m.$start, m.$end, m.$match().content());
});
```

The above matches a 2 followed by zero or more 4-s. It outputs:

```
1 2 [ 2 ]
3 7 [ 2, 4, 4, 4 ]
8 10 [ 2, 4 ]
11 12 [ 2 ]
```

2.2 Wildcard (`ANYTHING`, `WHATEVER`)

Ordinary regexps provide an easy way to match "anything"—via a single dot character. Or to match a sequence of consecutive "anything"-s (which we call "whatever"), you would use `.*`. PatternJS provides similar features, here are quick examples:

```
// match any sequence of type 2, X, 4
var a = [ 1, 2, 3, 4, 2, 4, 2, 1, 4, 5 ];
var pat = $.compile(2, $.ANYTHING(), 4);
$.search(a, pat, function(m){
    console.log(m.$start, m.$end, m.$match().content());
});
```

outputs:

```
1 4 [ 2, 3, 4 ]
6 9 [ 2, 1, 4 ]
```

2.3 Greedy and non-greedy repetitions (NG suffix)

A similar example with WHATEVER:

```
var a = [ 1, 2, 3, 4, 2, 4, 2, 1, "a", "b", 4, 5 ];
var pat = $.compile(2, $.WHATEVER(), 4);
$.search(a, pat, function(m){
    console.log(m.$start, m.$end, m.$match().content());
});
```

The output is interesting:

```
1 11 [ 2, 3, 4, 2, 4, 2, 1, 'a', 'b', 4 ]
```

Similar to ordinary regexps, the MANY and MORE constructs are "greedy". They match as many characters as possible. For this reason, because the expression that we're looking for is "2 followed by whatever sequence followed by 4" it matches from the first 2 in the array to the last 4.

There are cases, like the above, when we want the repetitive constructs to be "non-greedy". We provide MANYNG, MORENG and WHATEVERNG combinators for this case. By the way, WHATEVER() is equivalent to MANY(ANYTHING()), and WHATEVERNG() is equivalent to MANYNG(ANYTHING())[3].

The non-greedy pattern in the above example would be:

```
var pat = $.compile(2, $.WHATEVERNG(), 4);
```

and the output:

```
1 4 [ 2, 3, 4 ]
4 6 [ 2, 4 ]
6 11 [ 2, 1, 'a', 'b', 4 ]
```

2.4 To be or not to be (MAYBE)

In ordinary regexps this is provided by the question-mark operator; in PatternJS it's called MAYBE. Here's an example that matches sequences of "a", "b", optionally separated by a dash:

```
var a = [ 1, 2, "a", "b", 3, "a", "x", "b", "a", "-", "b", 3, "a", "b" ];
var pat = $.compile("a", $.MAYBE("-"), "b");
$.search(a, pat, function(m){
    console.log(m.$start, m.$end, m.$match().content());
});

// ==>
// 2 4 [ 'a', 'b' ]
// 8 11 [ 'a', '-', 'b' ]
// 12 14 [ 'a', 'b' ]
```

[3] I've nothing to do with the fact that "anything" ends in "ng" though. ;-)

2.5 Alternatives (OR)

You can use OR to provide alternatives at a certain point. Example: match consecutive sequences of 1, 2, or 3:

```
var a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 3, 2, 1, 4, 5, 6, 2 ];
var pat = $.compile($.MORE($.OR(1, 2, 3)));
$.search(a, pat, function(m){
    console.log(m.$start, m.$end, m.$match().content());
});

// ==>
// 0 3 [ 1, 2, 3 ]
// 9 12 [ 3, 2, 1 ]
// 15 16 [ 2 ]
```

Note that OR takes multiple "atoms" and returns a combinator that matches any of them. Passing that to MORE, I got a combinator that matches a sequence of 1, 2 or 3. In a regexp you would do "(1|2|3)+". Because OR returns a single expression, there's an implicit grouping above.

How would you devise an expression matching "a", "b" OR "c", "d" (that is, either an "a" followed by a "b", or a "c" followed by a "d"). You would need one of the combinators in the next section.

2.6 Grouping expressions (NAMED, GROUP)

In ordinary regexps you can group sub-expressions with parens. That serves two purposes: firstly it can limit action of other operations to only a part of the expression, as in `a(foo|bar)b` — without the parens, the pipe would pick between `afoo` and `barb`; and secondly, you can reference those groups after a match using a convenient shortcut. In Perl for example, you can access each group with $1, $2 etc. (a $ followed by the group index).

GROUP takes any number of expressions and matches when those expressions are found in sequence. It's the solution to the last question in the previous section. To match either "a" followed by "b", or "c" followed by "d":

```
var a = [ 1, 2, "a", 3, "c", "d", 4, "a", "b", 5, "c", "e" ];
var pat = $.compile($.OR($.GROUP("a", "b"),
                         $.GROUP("c", "d")));
$.search(a, pat, function(m){
    console.log(m.$start, m.$end, m.$match().content());
});

// ==>
// 4 6 [ 'c', 'd' ]
// 7 9 [ 'a', 'b' ]
```

NAMED is like GROUP, but the first argument must be a name by which you want to access the match later. For example:

```
var a = [
    "foo", "=", 1,
    "bar", "=", 2,
```

```
    "baz", "=", 3
];
var pat = $.compile(
    $.NAMED("key", $.ANYTHING()),
    "=",
    $.NAMED("val") // missing expression implies $.ANYTHING()
);
$.search(a, pat, function(m){
    console.log(m.key.content(), m.val.content());
});
```

outputs:

```
[ 'foo' ] [ 1 ]
[ 'bar' ] [ 2 ]
[ 'baz' ] [ 3 ]
```

The actual pattern to search for is `ANYTHING = ANYTHING`. (the second one is `$.NAMED("val")`, but lacking any expression, `ANYTHING()` is implied).

The first `ANYTHING` is being assigned the name "key", and the second one "val". As you can see, on a successful match they become properties of the data object "m". Those properties are `Match` objects and they provide some useful methods, one of which is `content()`, which simply returns all the matched content. As you can see above, `content()` returns an array—because expressions can generally match multiple elements.

Here's another example to make this more obvious:

```
var a = [
    "foo", "=", 1,
    "bar", "=", 2,
    "baz", "=", 3
];
var pat = $.compile(
    $.NAMED("def", $.ANYTHING(), "=", $.ANYTHING())
);
$.search(a, pat, function(m){
    console.log(m.def.content());
});

// ==>
// [ 'foo', '=', 1 ]
// [ 'bar', '=', 2 ]
// [ 'baz', '=', 3 ]
```

As you can see, `NAMED` can take more than two arguments—anything following the name is part of the searched expression. `content()` returns the three of them this time.

When you are only interested in the first element of the `content()` array, you can use `first()` as a shortcut for `content()[0]`. There is more to be said about `Match` objects, as you'll see below.

2.7 MAYBE ambiguity?

`NAMED` introduces a funny question when used in conjunction with the `MAYBE` combinator. What does the following output:

```
var a = [ "a", "b", "b", "b", "c" ];
var pat = $.compile("a", $.MAYBE("b"), "b", $.NAMED("f", $.MORE("b"), "c"));
$.search(a, pat, function(m){
    console.log(m.f.content());
});
```

Well, it outputs `[ 'b', 'c' ]`, because even the MAYBE operator is greedy! Being so, it takes as many characters as possible. The first part of the expression could have matched only `[ 'a', 'b' ]`, instead of `[ 'a', 'b', 'b' ]`, but the greedy-ish nature of MAYBE made it take all it could.

For this reason, even though it seems seldom useful, there is a non-greedy version of MAYBE, which, of course, it's called MAYBENG. With the following pattern the result is different:

```
pat = $.compile("a", $.MAYBENG("b"), "b", $.NAMED("f", $.MORE("b"), "c"));
// ==> [ 'b', 'b', 'c' ]
```

2.8 Back references (REF)

Once you gave a name to a sub-expression, you can refer to it in the search query. This part is commonly \N in regular expressions (where N is the index of the parenthesized group).

In PatternJS it's `REF("name")`. For a quick example, let's try to find all sequences of elements that repeat two or more times:

```
var a = [ 1, 2, 3, 3, 3, 2, 2, 1, 2, 1, 1, 1, 1, 1, 2, 3, 3 ];
var pat = $.compile(
    $.NAMED("a", $.ANYTHING()),
    $.MORE(
        $.REF("a")
    )
);
$.search(a, pat, function(m){
    console.log(m.$start, m.$end, m.$match().content());
    console.log("    A =", m.a.content());
});
```

The output is:

```
2 5 [ 3, 3, 3 ]
    A = [ 3 ]
5 7 [ 2, 2 ]
    A = [ 2 ]
9 14 [ 1, 1, 1, 1, 1 ]
    A = [ 1 ]
15 17 [ 3, 3 ]
    A = [ 3 ]
```

So the expression is:

1. accept *anything*, and *name* it “a”
2. accept one or *more* occurrences of what “a” matched.

Note that because `NAMED` takes any arbitrary expression, `REF` in turn is able to refer back to an arbitrarily complex match. It needs not be a simple literal:

```
var a = [
    1, 2, 3, // noise
    "a", "+", 1,
    4, 5, 6, // noise
    "a", "+", 1,
    "foo", "bar"
];
var pat = $.compile(
    $.NAMED("a", $.ANYTHING(), "+", $.ANYTHING()),
    $.WHATEVER(),
    $.REF("a")
);
$.search(a, pat, function(m){
    console.log(m.$start, m.$end, m.$match().content());
    console.log("    A =", m.a.content());
});

==>

3 12 [ 'a', '+', 1, 4, 5, 6, 'a', '+', 1 ]
    A = [ 'a', '+', 1 ]
```

2.9 Replace nodes

One common feature of regular expression engines is the ability to replace either entirely, or partially, the matched expression with something else. PatternJS provides some features to help with that.

Here's the most basic sample:

```
var a = [ 1, 2, 3, 4, 5 ];
var pat = $.compile(
    2, 3, 4
);
$.search(a, pat, function(m){
    m.$match().replace([ "cut" ]);
});
console.log(a);

// ==>
// [ 1, 'cut', 5 ]
```

More generally, a `Match` node has the following methods:

- `content()` — which you already saw: it returns an array with the elements matched by this sub-expression.
- `first()` — equivalent to `content()[0]`
- `replace(content)` — replace this match, in its parent node, by the given `content`. `content` must be an array, or another `Match` object.

- `swap(obj)` — exchange this match with the given `Match` object. The content of `obj` will be `this.content()`, and `this.content()` will be the content of `obj`.

In another example we're switching two named nodes:

```
var a = [
    "foo", "=", 1,
    "bar", "=", 2,
    "baz", "=", 3
];
var pat = $.compile(
    $.NAMED("key"),
    "=",
    $.NAMED("val")
);
$.search(a, pat, function(m){
    m.key.swap(m.val);
});
console.log(a);

// ==>
// [ 1, '=', 'foo', 2, '=', 'bar', 3, '=', 'baz' ]
```

2.10 Substructure matching

So far we've seen that PatternJS can do on arrays what classical regular expressions can do on strings. You might have noticed that we left out a certain feature of regexps—"character classes". That's the `[a-z]` operator. It's because it doesn't make much sense here: while regexps operate on chars, PatternJS operates on elements of arbitrary types.

In examples above the elements of the expression or of the array to search were strings or numbers. There is one type which is treated specially: arrays. Put simply, when an element of the search expression is an array, it gets compiled in a subexpression that must match an array at current position. I'm not sure this explanation is clear, but an example should help:

```
var a = [
    1, 2, 3, // noise
    [ "a", [ "b", "c" ] ],            // **1
    [ "a", [ "b", "e" ] ],
    [ "a", [ "b", "d",                // **2
             [ "a", [ "b", "c" ] ]    // **3
           ] ]
];
var pat = $.compile(
    $.NAMED(
        "exp",
        [ "a", [ "b", $.OR("c", "d") ] ]
    )
);
$.search(a, pat, function(m){
    console.log(m.exp.first());
});
```

Note that this time the expression isn't flat anymore. It's an array that starts with "a", followed by an array that starts with "b" and continues with either "c" or "d". The algorithm matches it faithfully. The output is:

```
[ 'a', [ 'b', 'c' ] ]
[ 'a', [ 'b', 'd', [ 'a', [Object] ] ] ]
[ 'a', [ 'b', 'c' ] ]
```

The matched parts are marked with ** above. Note that it matched `["a", ["b", "d"...]]` even though there is stuff following the "d" in the haystack—this is by design: the expressions are not anchored at the right side.

To force that the array finishes at the right side, you can use `END()`:

```
var pat = $.compile(
    $.NAMED(
        "exp",
        [ "a", [ "b", $.OR("c", "d"), $.END() ] ]
    )
);

// and the result with this is ==>
[ 'a', [ 'b', 'c' ] ]
[ 'a', [ 'b', 'c' ] ]
```

3 UglifyJS AST examples

Actually this was the reason why I started this library. UglifyJS[4] generates a complex AST to match the JavaScript program structure. Here is an example:

```
// Program:
function fact(n) {
        if (n == 1) {
                return 1;
        } else {
                return n * fact(n - 1);
        }
}

// AST:
[ 'toplevel',
  [ [ 'defun',
      'fact',
      [ 'n' ],
      [ [ 'if',
          [ 'binary', '==', [ 'name', 'n' ], [ 'num', 1 ] ],
          [ 'block', [ [ 'return', [ 'num', 1 ] ] ] ],
          [ 'block',
            [ [ 'return',
                [ 'binary',
                  '*',
                  [ 'name', 'n' ],
                  [ 'call',
```

[4] https://github.com/mishoo/UglifyJS — a JavaScript parser/compressor toolkit

```
                    [ 'name', 'fact' ],
                    [ [ 'binary', '-', [ 'name', 'n' ], [ 'num', 1 ] ] ]]]]]]]]]]
```

We can observe a few things, for example:

- a function definition looks like: `[ 'defun', NAME, ARGS, BODY ]`, where `NAME` is the function name, `ARGS` is an array of argument names, and `BODY` is an array of statements that the function consists of; statements in this array are full AST-s themselves;
- an `IF` statement looks like: `[ 'if', CONDITION, THEN, ELSE ]`; all the arguments are AST-s themselves;
- a block of statements is `[ 'block', [ STATEMENT, STATEMENT, ... ]]`;
- a reference to a variable is `[ 'name', VARNAME ]`; etc.

I thought having a powerful pattern engine would help simplify some things in UglifyJS, but it's not powerful enough yet. UglifyJS provides extensive features for traversing and transforming the AST (its `ast_walker` API), that can also provide some intimate details like full parent list of the current node, variables in scope, etc.

In any case, PatternJS can be used for various searches/transforms of an UglifyJS AST, and I'll provide some examples below.

3.1 Discard redundant block parens

We could easily replace blocks that contain a single statement with the statement, therefore discarding the brackets:

```
var $ = require("pattern");
var U = require("uglify-js");
var sys = require("util");

// sample program (passed through the UglifyJS parser to get the AST)
var ast = U.parser.parse(function a(foo){
    if (foo) {
        bar();
    } else {
        baz();
    }
    if (moo) {
        foo();
        bar();
    }
}.toString());

console.log(sys.inspect(ast, null, null));

var pat = $.compile(
    $.NAMED("block",
            [ "block", [
                $.NAMED("stat", $.ANYTHING()),
                $.END()
            ]])
```

```
);
$.search(ast, pat, function(m){
    m.block.replace(m.stat);
});

console.log(U.uglify.gen_code(ast, { beautify: true }));
```

The AST which is printed first is this:

```
[ 'toplevel',
  [ [ 'defun',
      'a',
      [ 'foo' ],
      [ [ 'if',
          [ 'name', 'foo' ],
          [ 'block',   // ***
            [ [ 'stat', [ 'call', [ 'name', 'bar' ], [] ] ] ] ],
          [ 'block',   // ***
            [ [ 'stat', [ 'call', [ 'name', 'baz' ], [] ] ] ] ],
        [ 'if',
          [ 'name', 'moo' ],
          [ 'block',   // this doesn't match as it has two statements
            [ [ 'stat', [ 'call', [ 'name', 'foo' ], [] ] ],
              [ 'stat', [ 'call', [ 'name', 'bar' ], [] ] ] ],
          undefined ] ] ] ] ]
```

I marked with *** the positions where the pattern will match. The spurious block brackets are dropped and the output is:

```
function a(foo) {
    if (foo) bar(); else baz();
    if (moo) {
        foo();
        bar();
    }
}
```

This is one optimization that UglifyJS does with a lot more code. But we should note that the pattern version is not safe—it should keep parens around `IF` statements without `ELSE`, that are themselves in an `IF` *with* an `ELSE`. It's quite tricky to do with the pattern engine alone.

We could extend it to store parent information and report it in the data object.

```
[ // 1
    "block",
    [ // 2
        [ "stat", ... ] // 3
    ]
]
```

In the above example, a generic pattern engine might note that the parent of the array 3 is the array 2. But that's not the kind of information we're looking for—when dealing with UglifyJS trees we care to know that the parent of the statement is the block (thus, node 1).

3.2 Sample `IF` optimizations

In the following sample we apply some small optimizations to `IF` statements:

- when there is a single statement in both of the branches, convert to conditional;
- when the condition is <= or >=, reverse `THEN` with `ELSE` and change the operator to > or <;

```
var $ = require("pattern");
var U = require("uglify-js");
var sys = require("util");

// sample program (passed through the UglifyJS parser to get the AST)
var ast = U.parser.parse(function a(foo){
    if (foo) {
        bar();
    } else {
        baz();
    }
    if (moo) {
        foo();
        bar();
    }
    if (a <= b) {
        f();
    } else {
        g();
    }
}.toString());

console.log(sys.inspect(ast, null, null));

// helper function to be used with $.CHECK
function NOT_NULL(val){ return val != null };

// another helper: generate an expression that matches either one
// statement or a block containing exactly one statement.
function ONE_STATEMENT(name) {
    return $.OR(
        [ "stat", $.NAMED(name) ],
        [ "block", [
            [ "stat", $.NAMED(name) ],
            $.END()
        ]]
    );
};

// pattern definition
var pat = $.compile(
    $.OR(
        // this finds stuff like if (a<=b) ... else ...
        [ "if", [ "binary",
                  $.NAMED("operator", $.OR("<=", ">=")) ],
          $.NAMED("th", $.CHECK(NOT_NULL)),
```

```
          $.NAMED("el", $.CHECK(NOT_NULL)) ],

        // this finds IFs having exactly one statement on the branches
        $.NAMED("to_conditional",
                [ "if", $.NAMED("co"), ONE_STATEMENT("th"), ONE_STATEMENT("el") ])

    )
);

function onmatch(m) {
    if (m.operator) {
        m.operator.replace([ m.operator.first() == "<=" ? ">" : "<" ]);
        m.th.swap(m.el);
        return m.$start; // *** see below for an explanation of this
    }
    if (m.to_conditional) {
        m.to_conditional.replace([
            [ "stat",
              [ "conditional", m.co.first(), m.th.first(), m.el.first() ]]
        ]);
    }
}

$.search(ast, pat, onmatch);

console.log(U.uglify.gen_code(ast, { beautify: true }));
```

The output is:

```
function a(foo) {
    foo ? bar() : baz();
    if (moo) {
        foo();
        bar();
    }
    a > b ? g() : f();
}
```

With a relatively simple pattern and `onmatch` function we managed to do some non-trivial transformation to a piece of source code[5]. Note that after handling the first expression in `onmatch` we `return $m.start`. If you return anything from the function, it should be a number and it tells PatternJS where to continue the search. By returning `m.$start` we tell it to re-run the expression at the same position where it found the match. Otherwise it would continue from *after* the expression, leaving the last `if` like this[6]:

```
if (a > b) {
    g();
} else {
    f();
}
```

[5] well, also thanks to the parser and code generator provided by UglifyJS.

[6] for obvious reasons, since the second part of the regexp never had a chance to run.

4 API reference

This section won't be too useful if you didn't go through the examples.

This package exports a few functions and combinators. They are all available both in lower-case and in upper-case (note this doesn't mean they're case insensitive though). I prefer to use upper-case for combinators inside expressions, and lower-case for the compile/search functions.

- `compile(expr)` — to compile an expression into bytecode
- `search(array, expr, onmatch)` — to search a compiled expression on an array

Expression combinators:

- `OR(case1, case2, ...)` — returns an expression that matches any of a few alternate cases;
- `MAYBE(expr, [expr, ...])` — returns an expression that matches `expr` if found, but does not fail if not found;
- `MANY(expr, [expr, ...])` — returns an expression that matches any number of occurrences of `expr` (or zero occurrences);
- `MORE(expr, [expr, ...])` — match at least one occurrence of `expr`;
- `ANYTHING()` — match any expression;
- `WHATEVER()` — equivalent to `MANY(ANYTHING())`;
- `GROUP(expr, [expr, ...])` — returns a single expression that matches a sequence of expressions;
- `NAMED(name, expr, [expr, ...])` — like `GROUP` but gives the new expression a name. If you omit `expr` it defaults to `ANYTHING()`;
- `REF(name)` — returns an expression that matches the same content as matched previously by a `NAMED` group with name `name`;
- `CHECK(predicate)` — matches the current expression if `predicate(expression)` returns non-false;
- `END()` — matches only at the end of the array;
- `MANYNG`, `MORENG`, `MAYBENG`, `WHATEVERNG` — the non-greedy versions for combinators involving backtracking.

Above when I wrote `expr, [expr, ...]` I meant to say that those functions take multiple arguments. The following lines don't match the same expressions:

```
GROUP("a", "b", "c");
GROUP([ "a", "b", "c" ]);
```

I tried to stress this in some examples. The first one matches the sequence anywhere, while the second one asserts that the sequence is at the start of an array.

4.1 The `onmatch` function

The function you pass to `search()` recevies one parameter, let's call it `m`, that can be used to figure out various things about the match:

- `m.$node` — the array where the current match is found;
- `m.$start` — the index in `$node` where the current match is found;
- `m.$end` — the index where the match ends;
- `m.$match()` — returns the current match as a `Match` object (see below);
- additionally, `NAMED` expressions insert properties into this object.

This function may modify `$node`. Some helper API is provided for this by `Match` objects. If it returns anything, it must be a valid index in `$node`, and searching will continue at that position. Without a return value, the algorithm continues searching **after** the current match.

4.2 The `Match` objects

They have the following methods:

- `replace(content)` — replaces the current node with the given `content` (which may be an array or a `Match` object);
- `swap(node)` — swap two `Match` objects — meaning that the content of one replaces the content of the other in the original tree;
- `content()` — returns the content that this node matches, as an array;
- `first()` — returns the first node in the content; same as `content()[0]`.

5 License

PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.