AutomatonJS — Programming Language for automata

Raphaël Jakse (raphael dot jakse at gmail dot com)

July 15, 2013

Contents

Introduction

AutomatonJS is a dedicated language. Its goal is easing the task of writing algorithms working on automata in a pedagogic and intuitive way. Of course, intuition is a rather personal element and two different persons can have different feelings on what they find intuitive or not, so trying to design an intuitive language that will please everybody might be a waste of time. Therefore, instead of just being intuitive, defining more precise directions to follow when designing the language should be a good start.

So far, these directions were considered:

1. Do not reinvent everything. Instead, try to aggregate wonderful ideas to conceive a wonderful language. This has several advantage:
   (a) With a good probability, what exists was already deeply though by other people we can probably trust, so the idea is well-tried, ready for usage. By using other people ideas when they exist give them credit on their work and save your time for completely new, unexplored ideas.
   (b) If a way of doing things is already spread, people don't have to learn something new. While learning new things is great, be forced to learn a new programming language to do a specific task can make people give up the usage of the tool.
2. Close to the mathematics. Scientific students and scientists are in the target of the language and math is (should be, at least!) a familiar language through these people. Mathematical language is precise, clear and concise and algorithms on automaton are often described with mathematical notation.
3. Be explicit. No weird symbols, no weird idioms. Unfortunately, mathematical symbols can be difficult to handle while programming. However, ASCII art to represent them is not desirable. The code should be readable for newcomers and gessing sense of expressions of the language should be easy and reliable. |_| and /\ for the union and the intersection operators can be hard to understand. "union" and "inter" are more likely to be understood at the first glance.

In "do not reinvent everything", we have "do not reinvent a new language". More precisely, AutomatonJS is not really a new language: choice to base AutomatonJS on a already existant language was made, with these advantages in mind:

- Real dedicated languages, by design, are limited. They have features their authors thought of and as time flies, successful dedicated languages tend to be generalist. Lets have a full-featured language which already has its community and its ecosystem.
- Less work for more. An already existent language already has well-tried tools like interpreters, (JIT-)compilers, debuggers, editors, etc. and these tools have been made powerful by years.
- If the community of the base language is already large, people will find documentation and help more easily and less people will have to learn a new language.

The choice of the base language was taken with these points in mind:

- relatively familiar syntax (close to C's) for most people
- Great community
- Great tools
- Web-compatible, that is to say, easy to run in a browser.
- Strong type handling, which is better for both pedagogic and debugging concerns.

Based of these points, a language comes rapidly in mind: Javascript. Javascript is the only one language to run in browser natively and present browsers competitions imply constant enhancement of the language and the tools running it (engines, interpreters, JIT compilers, debuggers), their speed included. Its C-like syntax is also a great point.

In fact, pure Javascript in its ECMAScript 6 version (still to be released) is very close to what seems to be needed to play with automata comfortably. What is still to add to the language is an Automaton class, a decent set handling and a better type handling: that's what is AutomatonJS for. Learning AutomatonJS is learning pure Javascript plus the few additions of AutomatonJS to Javascript. The base language of AutomatonJS is Javascript in its strict mode[1].

Given that, the goal is to stay very close to Javascript and to add just what we need. However, except Firefox, browsers of nowadays still don't support ECMAScript 6. Unfortunately, unlike ECMAScript 6, older versions of ECMAScript are really lacking things. As a result, work has been done to support some ECMAScript 6 points even in non-ecmascript 6 browsers.

In this documentation, differences between what is done when coding in pure Javascript and what should be done in AutomatonJS instead will be highlighted. This includes usage of some ECMAScript 6 subtleties. This documentation is not there to learn how to program. It is here to present the most important features of Javascript that should be known to program in AutomatonJS and to cover additions of AutomatonJS to Javascript and best practices to follow when programming in AutomatonJS.

Let's go.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode

1 Statements in Javascript

Statements, in Javascript, are instructions or expression ended by a semicolon, or control flow statements (which don't end with a semicolon). Empty statements are also allowed, so e.g. writing a semicolon right after an if statement is valid.

The semicolon is often optional, so a end of line can replace entirely a semicolon. To see when the semicolon can be omitted, you can look at the Section 7.9 of the ECMAScript 5.1 language specification[2].

Statements can be grouped at any time with curly brackets. Thus, this is valid :

```
1 hello();
2
3 {
4     action(1);
5     action(2);
6     action(3);
7 }
8
9 hello(2);
```

Curly brackets introducing a new scope, in browsers that support the let keyword natively, these two following codes are equivalent:

```
1 let(i=2, d="hello") {
2     console.log(d);
3     ++i;
4 }
5 // i and d are undefined everywhere
```

```
1 {
2     let i=2, d="hello";
3     console.log(d);
4     ++i;
5 }
6 // i and d are undefined in browser supporting let declarations
```

Although the semicolon can be optional, I strongly advise you to always write it unless you really know what you are doing. This is particularly true in AutomatonJS, which sometimes rely on semicolons to make the difference between a set literal and a block of instructions.

2 Declaring variables

2.1 The var statement

In Javascript, declaring variables is done with the var keyword :

```
1 var v = "Hello world!";
```

While this is possible is AutomatonJS, this is not advised. The let keyword is preferred.

[2] http://www.ecma-international.org/ecma-262/5.1/#sec-7.9

2.2 The let statement

```
let v = "Hello world!";
```

The let keyword, when natively supported by the browser, acts differently than var. The var keyword define the variable at the function level :

```
let enstein = "a scientist";
if(enstein) {
   var firstName = "albert"
}
else {
   var firstName = "roger";
}
alert(firstName); // shows "albert".
```

On the contrary, let works at block-level:

```
let enstein = "a scientist";
if(enstein) {
   let firstName = "albert";
}
else {
   let firstName = "roger";
}
alert(firstName); // ERROR, firstName is not defined.
```

2.2.1 Current limitation

In browsers that not support let natively, let acts like var. That is to say, the previous code shows "albert" instead of issuing an error. As of Jully 2013, Firefox is the only browser to show the right behavior.

2.3 The let expression

The let expression allows you to declare a variable inside a block.

```
let f = 5;

let (i=2, d=3) {
   alert(i * d * f); // shows 30
   d = 0;
   f = 9;
   alert(d); // shows 0;
}
// f is still defined and is equal to 9
// but i and d are not.
```

2.3.1 Current limitation

In browsers that don't support the let expression natively, you cannot define things inside the let expression and have it defined outside when going out the let block :

```
let f = 5;

let (i=2, d=3) {
  var g = 2;
```

```
5 }
6
7 alert(g); // g is not defined in Chromium, but is equal to 2 in Firefox.
```

Anyway, don't use the var keyword. Use let and don't define variable inside a block to have it define outside. Instead, write:

```
1 let f = 5;
2 let g; // that where g should be defined anyway
3 let (i=2, d=3) {
4   g = 2;
5 }
6
7 alert(g); // this shows 2 everywhere.
```

2.4 Typed declaration

This is a AutomatonJS specific way of defining variables and is the recommended way of doing things as it forces the developer not to change the type of variables.

The syntax is the following :

```
1 variable_name : Type;
2 variable_name : Type = initial_value;
3 variable_name : Set of Type;
```

Where Type is one of these case-insensitive keywords (keyword inside the same item list are equivalent):

- String
- Integer or int
- Number or float
- Boolean or bool
- Set
- Automaton
- Object (native Javascript Object)
- List, Table, Array
- Function

Additionally, Type can be any defined Javascript class. In this case, Type is case-sensitive.

2.4.1 Examples :

```
1 i : Integer;         // i is an integer initialized to 0;
2 j : Integer = 2;     // j is an integer initialized to 2;
3 d : Number;          // d is a number initialized to 0;
4 l : List;            // l is a Javascript native Array.
5 s : String;          // s is an empty string
6 A : Automaton;       // A is an Automaton
7 B : Set;             // B is a set which can contain anything
8 C : Set of Integer; // C is a Set restricted to integers.
9 D : Set of Set;      // D is a Set of Set
```

Typed declarations define variable in the block, not in the function, when let is natively supported by the browser. They behave exactly like the AutomatonJS let keyword.

2.4.2 Current limitations

The limitations are the same as for let : the scope of definition is wrong in browsers which don't support the let keyword natively.

Moreover, constraint on sets of sets can't be fully writen with a typed declaration:

```
1 s : Set of Set of Integer; // this won't work
```

2.5 The const statement

The const statement lets you define constants. It does exactly the same as the let keyword, but forbids the modification of the defined variable.

```
1 const bestYear = 1993;
2
3 bestYear = 1992; // TypeError: s is read-only
```

2.5.1 Current limitations

const defines constants at function level, like var, where it should define them at block level, like let:

```
1 if(true) {
2   const g = 2;
3 }
4 return g; // returns 2 instead of raising an error about g being undefined.
```

3 Control flow statements

Control flow statements let you structure your programs. They allow the execution or the repetition of a set of instructions under certain conditions and change the order of execution of the code in a logical way.

In AutomatonJS, like in Javascript, conditions are surrounded by parenthesis and instructions can be grouped with curly brackets. Like in C, curly brackets can be omitted when a block of instructions is reduced to one instruction.

3.1 The if / else statement

The if statement look like the following:

```
1 if(condition) {
2    // instructions to execute if condition is verified
3 }
```

With the else part:

```
1 if(condition) {
2    // instructions to execute  if condition is verified
3 }
4 else {
5    // instructions to execute  if condition is not verified
6 }
```

There is no elseif keyword. However, Javascript's syntax let you write things like:

```
if(whereToStudy === ujf) {
   res = "right decision";
}
else if(holidays) {
   res = "rest a little";
}
else if(student) {
   res = "stages are great";
}
else {
   res = "unhandled case";
}
```

3.2 Loop statements

3.2.1 The while statement

The while statement looks like the following:

```
while(condition) {
   // instuctions to execute while condition is true
}
```

3.2.2 The do...while statement

The do ... while statement looks like the following:

```
do {
   // instructions executed one time, an then re-executed until condition is false
}
while(condition); // notice the semicolon here
```

3.2.3 The for statement

The for statement looks like this:

```
for(initialisation ; condition ; iterationStatement) {
   // instruction to be executed while condition is true
   // iterationStatement after each iteration
   // initialisation is executed before entering the loop.
   // the scope of the initialisation is that of the body of the loop
}
```

Example:

```
let res = "";
for(let line="I won't write crappy code anymore", i=0, count=100; i < count; ++i) {
   res += line + "\n";
}
// here: line, i and count are not defined
return res;
```

3.2.4 The foreach statement

This is an AutomatonJS-specific structure. It let you iterate over sets and lists in a natural way.

```
res : String;
foreach(e in [2,4,6,8,10]) {
   res += e.toString() + ' ';
}
// here, e is undefined
alert(res); // shows "2 4 6 8 10 "
```

```
res : String = "H–Ways to greet somebody in English : ";
foreach(e in {"hey", "hello", "hi"}) {
   res += e + ' ';
}
return res; // "H–Ways to greet somebody in English : hey hi hello "
```

Warning: do not modify the element of iteration

Do not do that, it will break things or result in unexpected behaviors:

```
s : Set of Set = { [1,2] , [1, 4], [6, 5] };
foreach(e in s) {
   e.push(3);
}
return [s.toString(), s contains [1,2,3]];
```

This code returns a couple : the string representation of the set and a boolean indicating whether [1,2,3] belongs to the set.

[1,2,3] is present in the string representation of the set (and is indeed present in the set), but cannot be found.

```
s : Set of Set = { [1,2] , [1, 4], [6, 5] };
foreach(e in s) {
   e = [e[0], 0];
}
return [s.toString(), s contains [1,0]];
```

Here, the set remains unchanged, as e was a *reference* to the successive elements of s and the assignation operator scratches this references instead of modifying the actual element of the set. This is also true for simple elements as numbers, strings, regular expressions, ... modifying them by an assignation breaks the reference and do not affect the parent.

This works:

```
s : Set of Set = { {1,2} , {1, 4}, {6, 5} };
foreach(e in s) {
   e.add("hello");
}
return [s.toString(), s contains {1, 2, "hello"}];
```

Because sets inside sets tell their parents about their modifications. But be very careful with this, as habits of modifying the element of iteration can rapidly be taken, even when this hurts.

Please, don't modify the set or the list you are iterating on during the loop, this results in undefined behaviors, as in many languages:

```
s : List = [1,2,3,4,5];
r : Set;

foreach(e in l) {
  l.push(e+10);
  r.add(e);
} // infinite loop
return r;
```

Why should this loop be infinite...

```
s : Set = {1,2,3,4,5};
r : Set;

foreach(e in s) {
  s.add(e+10);
  r.add(e);
}
return r; // returns {1,2,3,4,5}
```

...whereas this one should end ?

A mathematical way of writing some for loops

You can use foreach to write some for loops with a syntax closer to mathematics :

```
for(let i=1; i <= 10; ++i) {
   console.log(i);
}
```

is exactly equivalent to, and is the generated code for:

```
foreach(i in {1,...,10}) {
   console.log(i);
}
```

Note

As comprehensive sets are still not supported by AutomatonJS, {1,...,10} is not a real set literal. As a result, you cannot write things like:

```
s : Set = {1,...,10}; // this should be possible in the future, but is still a
   syntax error for the moment.
```

You can only write {a ,..., b} sets in foreach loops, with a and b being any number or integer variables.

Note

If you prefer using of instead of in, is the foreach construct it's up to you.

Current limitation

Breaking to a specified label doesn't work in browser not natively supporting the for ... of structure. simple break will however work as expected.

3.2.5 The for...in statement

The for ... in statement is a Javascript construction to iterate over indexes of lists or enumerable properties of objects.

```
res : String;
l   : List = [2,4,6,8,10];

for(let i in l) {
   res += i + ': ' + l[i] + '\n';
}
return res;
/* returns :
   "0: 2
    1: 4
    2: 6
    3: 8
    4: 10"
*/
```

Note

- for ... in has not much sense on sets, as their elements have no indexes. Use foreach or for ... of instead.
- if you need the index corresponding to the last iteration, declare the iteration variable before the loop :

```
l   : List = [2,4,6,8,10];
i   : Integer;
for(i in l) {
   // do something here
}
return i; // returns 4
```

3.2.6 The for...of statement

The for ... of statement is a more complicated yet more flexible way to do what foreach does. It is an ECMAScript 6 structure handled by AutomatonJS when browsers do not handle it themselves. The real difference between foreach and for ... of if that the iteration variable must be explicitly declared :

```
res : String;

for(let i of [2,4,6,8]) {
   res += i + ' ';
}
return res;
```

If you need to know what was the last iterated element in the loop, foreach does not really help you. On the contrary, for ... of, while probably less intuitive, let you do this in an elegant way:

```
res : String;

i : integer;
for(i of [2,4,6,8]) {
   res += i + ' ';
}
return i; // returns 8
```

If you want to stay compatible with pure ECMAScript, use for .. of instead of foreach.

Current limitation

Breaking to a specified label doesn't work in browser not natively supporting the for ... of structure. simple break will however work as expected.

3.3 The break statement

break lets you immediately get out of a loop even if the condition is true / if there are still instructions to execute:

```
res : String;
foreach(i in [1,2,3,4,5]) {
   if(i === 3) {
      break;
   }
}
alert('done'); // shows 'done'
return res; // returns "1 2 "
```

3.4 The continue statement

continue lets you jump to the next iteration, or get out of the current loop if no iterations remain, without executing the rest of the current iteration's instructions:

```
res : String;
foreach(i in [1,2,3,4,5]) {
   if(i === 3) {
      continue;
   }
   res += i + ' ';
}
return res; // returns "1 2 4 5 ", note the missing 3.
```

3.5 The return statement

return lets you leave the current function, or program, leaving a value at the same time. You can return everywhere in your program or your function, even in a loop.

```
function hello() {
   while(true) {
      return "world";
   }
}
```

```
6
7 return "hello " + hello() + '.'; // returns "hello world."
```

4 function statements

Functions are fundamental to write and structure good non trivial programs. In Javascript, you can define functions at anytime (e.g. functions can be nested and can be defined in loops or if-else statements). They can take arguments and can call themselves.

Here is an implementation of the factorial:

```
1 function fact(n) {
2   if(n < 2) {
3     return 1;
4   }
5   return n * fact(n-1);
6 }
7
8 return fact(6); // calling fact with 6. returns 720.
```

If a function takes more than one arguments, they are separated with commas.

```
1 function sameParity(a,b) {
2   return a % 2 === b % 2;
3 }
4
5 return sameParity(6, 2); // returns true.
```

Functions have access to the variable of the scope in which they were defined:

```
1 women : Set = {"Marie Curie", "Sophie Germain", "Rita Levi"};
2 function isAWoman(p) {
3   return women contains p;
4 }
5
6 return isAWoman("Marie Curie"); // returns true.
```

5 Literals

In AutomatonJS, literals are the same as Javascript, plus sets. Briefly, we have numbers, strings, objects, arrays, regular expressions and functions.

An important thing to notice is that AutomatonJS supports abbreviated Javascript 1.8 functions.

5.1 Javascript literals

```
1 // strings
2    let str1 = "I'm a string";
3    let str2 = 'I\'m another string';
4    let str3 = 'I end with a new line and contain a \u0253 special character\n';
5    let character = str3[2]; // character equals "e" and is also a string
6
7 // numbers
```

```
8     let i = 0; // zarro
9     let answer = 14.5; // floating point
10    let color  = 0xFF; // hexa
11    let speedOfLight  = 3e10; // 300000000
12    let notANumber = NaN;
13    let inf = Infinity;
14    let negInf = -Infinity;
15
16 // objects
17    let o = {}; // an empty object
18    let idCard = {
19       firstName: "Arthur",
20       lastName:  "Dent",
21       age:32
22    };
23
24    idCard.eyeColors = 'blue';
25
26    let stillAnObject = null;
27
28    alert(typeof stillAnObject); // shows "object"
29 // arrays
30    let a = []; // empy arrays
31    let primeNumbers = [2,3,5,7,11,13];
32    primeNumbers.push(17);
33    let secondValue = primeNumbers[2];
34
35 // regular expressions
36    let r = /(?:http|ftp)s?:\/\/www\.[a-zA-Z]+[a-zA-Z0-9\.]*\/[\s\S]+/g;
37
38 /* functions */
39    // classic function
40
41       let f = function(x) {
42          return 2 * x;
43       };
44
45       alert(f(2)); // 4
46
47    // abbreviated 1-argument function
48
49       let square = x => x*x;
50
51       alert(square(10)); // 100
52
53    let quadraticValue = (a,b,c,x) => a*x*x + b*x + c;
54    alert(quadraticValue(2,3,4,5)); // 69
55
56    let add1 = function(x) x+1;
57
58    alert(add1(2)); // 3
```

For more informations about Javascript literals, you can look at MDN's Values, variables, and literals[3] or the ECMAScript documentation. For regular expressions, you can refer to https://developer.mozilla.org/en-

[3]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Values,_variables,_and_literals

US/docs/Web/JavaScript/Guide/Regular_Expressions[4].

5.2 Set literals

AutomatonJS has built-in support for sets and has a dedicated a dedicated literal to represent it.

A set literal starts with a open curly bracket ({), ends with a close curly bracket (}) and contains one or more values separated with commas.

We could have represented the empty set by something like {} but it would have been ambiguous with empty Javascript objects. Instead, the emptySet keyword represent an empty set.

Examples:

```
let firstPrimeNumbers = {2,3,5,7};
alert(firstPrimeNumbers contains 3); // true

let e = emptySet;
alert(e.card()); // 0

typingEvenEmptySetIsMuchBetter : Set;

IfYouKnowSetOfWhat : Set of Integer;

meals : Set of String = {"breakfast", "lunch", "dinner"};

anything : Set = {14, 38.5, "green"};

singleton = {1};

thisIsNotASet = {}; // this is an object.

// Typed declarations transform objects and lists into sets
ThisIsASet : Set = {};
AnotherSet : Set = [1,2,3,4];
```

6 Operators

AutomatonJS understands every Javascript operator and adds several operators on sets.

Important Note. AutomatonJS changes the signification of the == equality operator. See hereinafter.

6.1 JavaScript operators

6.1.1 Arithmetic operators

Expression	Signification
`a + b`	The sum of `a` and `b`
`a - b`	The difference of `a` by `b`
`a / b`	The division of `a` by `b`
`a * b`	The multiplication of `a` and `b`
`a % b`	`a` modulo `b` (the reminder of the integer division of `a` by `b`)

[4]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

6.1.2 Bitwise operators

Expression	Signification
`a & b`	`a` AND `b`
`a | b`	`a` OR `b`
`a ^ b`	`a` XOR `b`
`a << n`	FIXME
`a >> n`	FIXME
`a <<< n`	FIXME
`a >>> n`	FIXME

6.1.3 Assigning counterparts of these operators

Let op be one of the operators hereinbefore. a op= b means a = a op b.

Example:

```
1 a : integer = 14;
2 a += 3;
3 alert(a); // shows 17
```

Moreover, if a is a number,

- ++a increments a by 1 and returns the new value of a;
- a++ increments a by 1 and returns the old value of a.

6.1.4 Comparison operators

Expression	Signification
`a < b`	true iff `a` greater than and not equal to `b`
`a <= b`	true iff `a` greater than or equal to `b`
`a > b`	true iff `a` lower than and not equal to `b`
`a === b`	`true` iff `a` is exactly the same object than `b`
`a !== b`	`false` iff `a` is exactly the same object than `b`

In AutomatonJS, the meaning of the operators == and != has been changed. The meaning of equality operator is discussed hereinafter.

6.1.5 Boolean operators

Expression	Signification
`!a`	if the boolean evaluation of `a` is `true`, then `false`, otherwise `true`.
`b`	if the boolean evaluation of `a` is true, then `a`, otherwise `b`[5]. This operator is lazy, that is to say if `a` evaluates to `true`, `b` is not evaluated.
`a && b`	if the boolean evaluations of `a` and `b` are both `true`, then `true`, otherwise `false`.

[5] This different in C, where the meaning of the operator is the following: if the boolean evaluation of `a` is `true`, then `true`, otherwise the boolean expression of `b`.

Which expressions evaluates to true / false ?

Expression	Boolean evaluation
`false`	`false`
`""`	`false`
`0`	`false`
`NaN`	`false`
`{}`	`true`
Any non empty string	`true`
Any number not equal to 0	`true`
Any set, even empty	`true`
Infinity, -Infinity	`true`
Any other object	`true`

6.2 Meaning of the equality operators

In Javascript, there are to equality operator: == and === (and their negative variant, != and !==).

Both operators act the same way, except for comparisons between string and numbers. Let a be a string and b a number.

- a === b and b === a both evaluate to false, for any a and b.
- a == b and b == a evaluate to true if the string representation of b is a.

For the other cases, these operators are equivalent:

- if a and b are the same string, a == b
- if a and b are the same number, a == b
- if a and b are the same regular expression, a == b.
- null == null
- true == true
- false == false
- NaN != NaN
- if a and b **point** to the same object, a == b.

However, if a and b are point to distinct objects, a !== b even if a and b are equivalent. For example:

```
1 alert({1,2,3,4,5} === {1,2,3,4,5}); // Shows false, because two distinct objects
    are created.
```

This is an important concern in Javascript, as we need to be able to check the equality between objects, in particular sets. AutomatonJS, as a consequence, changes the meaning of the == operator, to be able to check equality between to distincts objects.

In AutomatonJS, we have:

```
1 alert({1,2,3,4,5} === {1,2,3,4,5}); // false, because two distinct objects are
    created.
2
3 alert({1,2,3,4,5} == {1,2,3,4,5}); // true, like one would expect.
4
5 alert(new Objet == new Object); // true.
```

Whereas new Objet != new Object in pure Javascript.

Note. The === and !== operators remain unchanged.

6.3 Set operators

In addition to the modified == and != operators, AutomatonJS defines new operators. In the following table, A and B refer to two sets, or objects that can be turned into sets like objects or lists. e is any possible value.

Expression	=Signification
union B	The union of A and B
inter B	The intersection of A and B
minus B	The difference of A by B
symDiff B	The symetric difference of A and B
`A contains e` A has e `e belongsTo A`	`true` iff `e` is inside `A`

6.3.1 Assigning counterparts of union, inter and minus

Expression	**Signification**
`A union = B`	• Assign the union of A and B to A. • More efficient than `A = A union B`. • Strictly equivalent to `A.unionInPlace(B);`.
`A inter = B`	• Assign the intersection of A and B to A • More efficient than `A = A inter B`. • Strictly equivalent to `A.interInPlace(B);`.
`A minus = B`	• Assign the difference of A by B to A • More efficient than `A = A minus B`. • Strictly equivalent to `A.minusInPlace(B);`.

Warning. These operator exist for the sake of consistence with Javascript-native assigning operators. Before using assigning counterparts of set operators, please double check you visually prefer them to there equivalent expression. Triple check that your audience won't be lost with these operators if your code targets a particular audience (e.g. students).

Conclusion

This document tries to remain complete on additions of and differences between AutomatonJS to / and Javascript. This is far from covering all the language. For more information, you might want to read things on Javascript keeping differences covered here in mind.

Ideas for the future

- comprehensive sets like {2k+1 | k in N} would be awesome.
- Great ideas can be found here :

- http://koush.com/post/yield-await-v8
- http://mdn.beonex.com/en/JavaScript/New_in_JavaScript/1.7.html, in particular :
  * yeld, to make generators
  * the destructuring assignment
  * array comprehensions (Python-lovers will appreciate)
  * The let statement: http://mdn.beonex.com/en/JavaScript/Reference/Statements/let.html.