

# A Polynomial-time Regular Expressions Implementation \*

Juan Pedro Alves Lopes , Maria Alice Silveira de Brito ,  
Paulo Eustáquio Duarte Pinto

<sup>1</sup>IME/DICC, Universidade do Estado do Rio de Janeiro (UERJ)  
Av. São Francisco Xavier, 524, Maracanã – 20550-013 – Rio de Janeiro – RJ

me@juanlopes.net, malice@ime.uerj.br, pauloedp@ime.uerj.br

**Abstract.** *Regular expressions are a notation to define regular languages in terms of simple composable operations. They are equivalent to finite automata in expressive power. In practice, however, modern regular expressions implementations diverge from the original theory. Most changes are made to allow greater expressive power. This convenience comes at the cost of making language membership a harder problem than it could be. In many modern languages, the regex language membership is a NP-complete problem. Besides, the way they are implemented sometimes causes expressions that could be processed in linear time to take exponential time. This fact may be seen as a security risk for many applications that use regular expressions. In this work, we suggest a simple implementation (based on Thompson’s Construction Algorithm) that has superior worst-case performance than many popular implementations. We also introduce a notation for the automata created by this algorithm that makes the adopted implementation easier to understand.*

## 1. Introduction

Regular expressions are a common tool in most modern languages and development platforms. Its use is widespread in the industry for many purposes, e.g. pattern recognition, extraction of tokens for formal language parsing or even user input sanitization for security purposes.

Its extensive use originates from a strong theoretical base in the finite automata field, introduced in the 40’s by McCulloch and Pitts [McCulloch and Pitts 1943] and formalized later by Kleene [Kleene 1956].

Regular expressions and finite automata share a close relationship. They are equivalent in expressive power, and the conversion of the former to the latter is trivial. Ken Thompson demonstrated this in his implementation in the late 60’s [Thompson 1968], while developing the *QED* text editor. The same regular expressions were later ported to the well known *ed* and *grep*, all part of the Unix operating system.

Since then, as the implementations evolved, many features have been added to practical regular expressions that diverged from the original language theory. While originally regular expressions strictly describes regular languages, the currently most widespread implementation (*PCRE*) is able to recognize not only any context-free language, but also some context sensitive ones [Popov 2012].

---

\*This work is partially supported by FAPERJ

One of the most notable consequences of this evolution consists in the recognition of language strings described by those expressions, a problem with linear solution originally, having exponential solutions in a large number of modern implementations, including several of the mostly used ones. Perhaps the most dangerous feature in this regard are the *backreferences*, that makes membership a NP-complete problem [Câmpeanu and Santeau 2007]. However, even expressions that could be recognized with finite automata, due to some particularities of implementation, may take exponential time in the worst case [Cox 2007].

Languages like Python, Java and Ruby are widely used in industry, but their standard libraries implement vulnerable regular expressions. These implementations make the use of regular expressions potentially unsafe in otherwise trivial situations. A malicious user may be able to force the execution of an expression with exponential time to perform a denial-of-service attack on a web server.

For example, allowing user input to be interpreted as regular expressions, a malicious user could inject the expression  $(a|a)^+b$ , which given its exponential complexity in vulnerable implementations, would use 100% of CPU for a virtually infinite time if matched against the string *aaa...aaa*. Such attacks may cause instability and unavailability of the web service.

The prevention of this type of attack is often not trivial, as in the case of Java, where commonly used methods, such as *replaceAll* and *split* – of the class *String* – are implemented with regular expressions which are vulnerable to this kind of attack.

Even in cases where a malicious user does not have access to write regular expressions, a programming error can leave the system vulnerable to DoS attacks. This problem becomes critical with the frequent sharing of regexes in online repositories that contain many vulnerable expressions [Kirrage et al. 2013, Weidman 2010].

This work has two main objectives. The first is to demonstrate through tests and benchmarks the fundamental problems in implementations of regular expressions in several modern languages. The second objective is to achieve a minimalist implementation of regular expressions, using the theory described by Kleene and the method proposed by Thompson to construct and simulate the automaton.

The implementation proposed in this work does not include certain features which are common most modern *flavors* of regular expressions. Some of these features can be implemented without sacrificing execution efficiency; whereas others may introduce some complexity, while still running in polynomial time. Some features, however, can not be implemented without making the algorithm exponential in the worst case.

We intend to show in this paper that the problem of recognizing strings in the languages denoted by regular expressions can be efficiently solved even with the simplest implementations, provided that its theory is observed.

## 2. Automaton Construction and Simulation

Given the synergy between regular expressions and finite automata and considering the ease of dealing with each of them in specific situations, it is useful to be able to convert between them. Ken Thompson [Thompson 1968] described a method for constructing automata from regular expressions.

The method consists in building the automaton recursively by analyzing the syntax tree of the expression, a procedure that needs to be done before start to build the automaton. In the original paper by Thompson, this step was done by building the expression in *infix* notation. However, in order to allow extensibility of expression format, we will define the full grammar of regular expressions that are accepted.

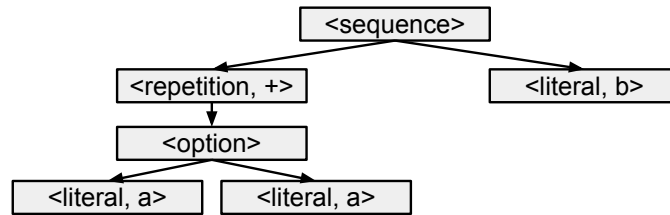
## 2.1. Syntactic Analysis

The first step in the construction of the automaton is to parse the expression, building the abstract syntax tree (AST). A simple generative grammar can be defined – using the EBNF (*Extended Backus-Naur Form*) – that represents the language formed by all regular expressions.

```
<start>      ::= [ <option> ]
<option>     ::= <sequence> { "|" <sequence> }*
<sequence>   ::= <repetition> { <repetition> }*
<repetition> ::= <primary> { "*" | "?" | "+" }*
<primary>    ::= <literal> | "(" <start> ")"
```

It is not defined, but <literal> is any character except any of \*?+|().

In this way, the expression can be easily analyzed. For example, the expression  $(a|a)^+b$  would be evaluated as the tree depicted in Figure 1.



**Figure 1. AST for the expression  $(a|a)^+b$ . In this case, we omit the trivial nodes, e.g. *option* nodes with a single *sequence* inside, *repetition* nodes without any repetition operator, etc.**

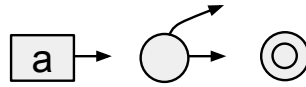
Using the EBNF, the tree can be produced by several parser generator tools present in many languages. Notable examples include the YACC (*Yet Another Compiler Compiler*), the JavaCC (*Java Compiler Compiler*) and ANTLR (*ANother Tool for Language Recognition*).

With this AST, conversion to finite automaton becomes rather trivial, as will be discussed in the next sections.

## 2.2. Notation

We will introduce an unusual notation to represent the automaton, that will make the construction easier. For convenience purposes, every state  $q_i \in Q$  will be of one of three possible types:

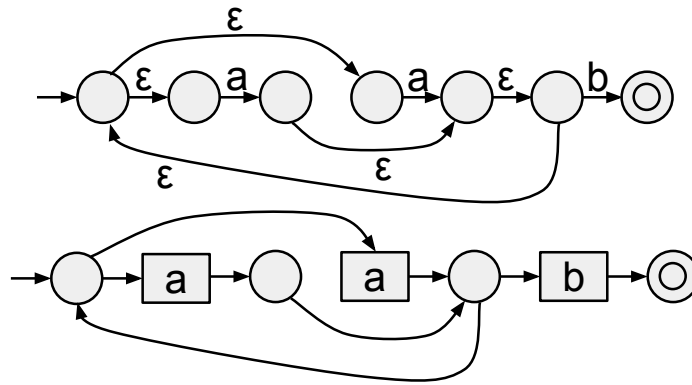
1. CONSUME state: has as single transition that consumes a single character from the input;
2. JUMP state: has one or more  $\epsilon$  transitions to distinct states;
3. MATCH state: an accept state with no transitions.



**Figure 2. Example state representation in the new notation**

Figure 2 shows how the states in this notation may be represented in a diagram.

This representation simplifies visualization by omitting the  $\epsilon$  transition labels and emphasizing the difference between the CONSUME and JUMP states. An example state diagram in the new notation can be seen in Figure 3.



**Figure 3. Same automaton represented in both notations: standard (above) and new (below)**

The diagram in Figure 3 denotes the regular expression  $(a|a)^+b$ . The sequential disposition of the states has some advantages, as it allows direct translation to a program-like representation (as a series of instructions). This automaton, for instance, can be translated as:


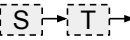
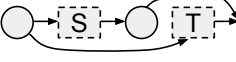
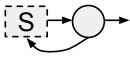
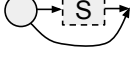
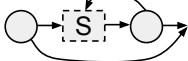
```
0000: JUMP (+1, +3)
0001: CONSUME a
0002: JUMP (+2, )
0003: CONSUME a
0004: JUMP (+1, -4)
0005: CONSUME b
0006: MATCH!
```

The JUMP instructions with more than one label express the non-determinism of the automaton. If a deterministic machine was built to run these instructions, when faced with JUMPS it could proceed to any of the defined addresses.

### 2.3. Construction

The construction of the automaton consists in recursively converting each of the nodes in the AST to a specific automaton following a recipe. Each of these recipes has only one input state and all transitions that do not point to a node within the automaton itself, point to a single output node.

The following table shows the conversion for each node type. For notational purposes, capital letters represent complex automata produced with inner nodes, but they all follow the same construction rule.

Table 1. Automaton construction for each expression type			
Type	Expression	Construction	Instructions
Literal	$a$		CONSUME $a$
Sequence	$ST$		<instructions of $S$ > <instructions of $T$ >
Option	$S T$		JUMP (+1, <size of $S$ >+2) <instructions of $S$ > JUMP (<size of $T$ >+1) <instructions of $T$ >
Kleene+	$S^+$		<instructions of $S$ > JUMP (+1, -<size of $S$ >)
Optional	$S^?$		JUMP (+1, <size of $S$ >+1) <instructions of $S$ >
Kleene*	$S^*$		JUMP (+1, <size of $S$ >+2) <instructions of $S$ > JUMP (+1, -<size of $S$ >)

To convert an automaton using these rules, simply traverse the AST in post-order, initially processing all literals and recursively using them as input to the upper nodes. Figure 4 shows an example of conversion using the expression  $(a|a)^+b$ .

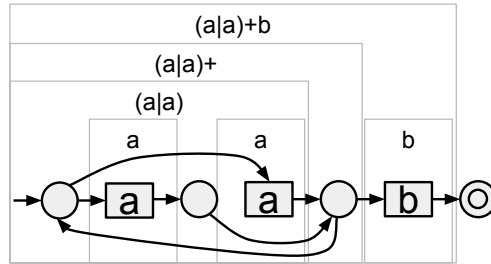


Figure 4. Example of construction for automaton  $(a|a)^+b$

In an optimized implementation, this step can be replaced by machine code generation (as in Thompson's original paper). However, for simplicity we describe some useful representations of the generated automaton.

## 2.4. Simulation

Simulating the execution of a non-deterministic automaton using a deterministic machine is a hard problem in itself. There may be an exponential number of possible paths in the automaton [Rabin and Scott 1959]. This is the exact spot where most implementations fail to find a polynomial solution.

But given that we are not interested in all paths, but only if one of them leads to an accept state, there are some easier options to simulate those automata. Some of them are:

**Backtracking** to try all possible paths, one at a time, rewinding to the last choice in case of failure;

**Backtracking with memoization** same as previous, but memorizing if one state leads to an accept state, given some part of the input;

**Convert to DFA** to eagerly convert the NFA to a deterministic automaton before execution;

**Parallel simulation** to simulate all possible state transitions concurrently, pruning repeated states;

**Lazy DFA conversion** similar to parallel simulation, but memorizing which sets of states have transitions between them, which is the same as lazily converting to a DFA.

The chosen method for this implementation was the *parallel simulation*. It consists, whenever faced with a decision, in choosing both options simultaneously, and keeping parallel executions.

Initially, it may seem that this leads to a set of up to  $2^n$  states running simultaneously, but given that the automaton has a finite number of states, there are at most  $m$  possible states running at any time (for each input character). This is so because eventually multiple decisions can collide in the same state, thus eliminating the exponential nature of the evaluation. So, for an input with  $n$  characters evaluated against an automaton  $m$  states, the worst case time complexity is  $O(n \times m)$ .

The goal is that, at runtime, one character will be read at a time. Then, before each run it must evaluate all  $\epsilon$  transitions and keeps all flows on hold positioned in CONSUME states.

The main advantage is to have polynomial running time ( $O(n \times m)$ ). Also, it just needs to read the input once and does not require the use of stack. However, it needs additional memory ( $O(m)$ ). And compared to running with a converted DFA it is slower to recognize strings.

### 3. Implementation

In this section we will discuss the implementation of the library *Pyrex* (Python Regular Expressions).

The initial goal was to achieve a truly polynomial regular expressions implementation. After that, the implementation was focused on simplicity. We avoided implementation tricks to improve performance at the expense of readability. All choices were made to reduce implementation complexity. Even so, the algorithm used was efficient enough to have better worst-case performance than most implementations.

We chose Python to implement this project. Python is a virtual machine language, like Java. However, its virtual machine does not have the runtime optimizations that most Java implementations have. Also, its dynamic typing sometimes makes it even slower than other static typed languages. Even so, due to the reduced amount of special characters in its syntax, programs written in Python are usually more readable and simpler.

The entire implementation of Pyrex has 67 lines of code and less than 3KB. This implementation covers regular expression parsing, a rudimentary error handling, automaton construction and simulation.

The project was implemented using only Python 2.7 standard library, but it also works in Python 3.0 and higher.

### 3.1. Features

The features were chosen as the minimum set to make possible to compare Pyrex and other implementations, while being simple enough to be explainable in a few minutes. So, we implemented:

- Literals (e.g. *a*)
- Sequences (e.g. *ab*)
- Options (e.g. *ab|cd*)
- Optionals (e.g. *a?*)
- Kleene\* (e.g. *a\**)
- Kleene+ (e.g. *a<sup>+</sup>*)

These features are very close to those described by Thompson [Thompson 1968] and they make it possible to unambiguously represent regular languages.

### 3.2. Code structure

The code is basically divided into two parts: a *parser* and a *matcher*.

The *parser* is responsible for receiving a string with a regular expression, to syntactically parse it and to return an object (the *matcher*) that is able to recognize strings which are part of the regular language defined.

The *matcher* contains information of the generated automaton and functions that can simulate its execution. It is implemented as a Python class with a *match* method, besides the overloading of the operator `__repr__` of Python, which provides a useful view of the object at debug time.

To use *Pyrex*, one must compile the automaton using the *rex* function, and then evaluate strings using the *match* method of the *Machine* class. An example can be seen below:

```
import pyrex
machine = pyrex.rex('a(ab)+')
print machine.match('aababxx')
```

The *match* function return can be either a tuple with two elements (indexes start and end of the match) or a null reference (*None* in Python). In the example, the return would be a tuple (0, 5), indicating that the match starts at index 0 of the string and ends at index 4. Notice that the tuple denotes an open interval to the right.

### 3.3. Parsing

Parsing is done by the function *rex*. It implements a rudimentary recursive descending parser. The EBNF is described in Section 2.1.

The implementation is simple, it uses an instance of the *deque* class as tokens buffer and recursive calls to implement the rules. For each of the four rules defined the EBNF, there is a nested method in *rex* to represent it.

Error handling is simple but effective. Whenever an unexpected character is encountered in the input, it is reported. One of the mechanisms can be observed in the main parser code below, where after consuming all possible characters, if there are still remaining ones at the input, an exception is thrown.

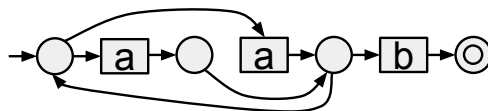
```
e = option()
if tokens:
    raise Exception('Not expected: "{}"'.format(''.join(tokens)))

return Machine(e)
```

### 3.4. Representation

During the parsing, the regular expression is transformed into a representation of the associated automaton. This automaton is represented as an *array* of objects, where each position of the array represents a state of the automaton.

As defined in Section 2.2 (page 3), the notation used in this automaton provides three types of state: CONSUME JUMP and MATCH. CONSUME states are represented by the character they consume. JUMP states are represented by a tuple of integers that inform how many states ahead or behind it should 'skip'. MATCH states do not need representation because they always occur at the end of the automaton (according to this notation).



**Figure 5. Representation of the automaton as a state diagram**

For example, the automaton represented in the Figure 5 is equivalent to the array:

```
[(1, 3), 'a', (2,), 'a', (1, -4), 'b']
```

which can also be represented as instructions

```
0000: JUMP (1, 3)
0001: CONSUME a
0002: JUMP (2,)
0003: CONSUME a
0004: JUMP (1, -4)
0005: CONSUME b
0006: MATCH!
```

### 3.5. Simulation

The simulation is performed by the *matcher* and *match* methods of *Machine* class. The difference between them is that the first returns an iterator that assesses the state of execution of the automaton for each input character. The second just returns the result of the match in the end. In the implementation, the *match* method uses the *matcher* method, as shown in the snippet below:



```
def match(self, string):
    return reduce(lambda _, s: s[1], self.matcher(string), None)
```

The method implemented for the simulation of the automaton is the *parallel simulation*. This method was chosen because it combines a polynomial asymptotic complexity with a reasonable ease of implementation.

As is customary in the implementation of regular expressions, the match is performed in case any substring of the input belongs to the language defined. The parallel simulation method is more advantageous as it requires little change in order to support this mode of execution.

The implementation is based on cycles of use, where a set of initial states simultaneously goes through a transition with the same character  $c$  for a set of subsequent states.

In order to represent the states, we use two lists, A and B, which at all times represent respectively the set of states *of consumption* achieved by the beginning of the evaluation of the current character (A) and the set of states that the evaluation of the current character will reach (B). After the evaluation the lists are swapped.

Each state in these lists comes with the position in the original string where the match began. This information is created when the initial state is placed on the list for each character. It is then copied each time the state is resolved and advances to their successors. In the list, the state is represented by a tuple  $(start, j)$ , where  $start$  is the position in the input where the match began and  $j$  is the index of the current state.

It is important to observe that the same state never enters in list B twice in the same cycle of consumption. This check is performed by the array V, which controls the index of the last character where each state joined the list of states. Initially, this list is populated with values -1.

As only CONSUME states enter the lists of states to participate in the consumption cycle (for obvious reasons), all JUMP states must be assessed before the start of the consumption cycle.

The implementation of the *addnext* method adds states to the following list of consumption (B), recursively solving JUMP states in their respective following CONSUME states. This method also returns the number of times the MATCH state was reached.

```
def addnext(start, i, j):
    if j==self.n: return 1
    if V[j] == i: return 0
    V[j] = i

    if isinstance(self.states[j], tuple):
        return sum(addnext(start, i, j+k) for k in self.states[j])

    B.append((start, j))
    return 0
```

Thus, each consumption cycle is characterized by the steps:

- Start new flow in the initial state with the current character.

- Invert lists of CONSUME, cleaning the list of next (B).
- Evaluate the consumption of the current character adding subsequent states in the next list.

The implementation of these steps can be seen below:

```
def key(a): return (a[1]-a[0], -a[0]) if a else (0, 0)

answer = None
for i, c in enumerate(string):
    addnext(i, i, 0)
    yield i, answer, B

    A, B = B, A
    del B[:]

    for start, j in A:
        if self.states[j] in (None, c) and addnext(start, i+1, j+1):
            answer = max(answer, (start, i+1), key=key)

yield len(string), answer, B
```

Initially a new flow is added starting at character  $i$ . Then the CONSUME lists are reversed. Finally, the CONSUME states are evaluated against the input, the due states advance, in addition to updating the best response, in case some of these states is a MATCH state.

### 3.6. *match* Method Return

To recognize words using regular expressions is a *decision* problem. However, it has proven useful over time to recognize not only strings in regular languages, but also any sub-string belonging to it, then returning their location in the source string.

A regular expression *abcd* would return a valid match in a string *zzabcdzz*. This behavior allows multiple solutions to be valid. For example, the same expression could find two different results in the word *abcdabcd*. Indeed, many results could overlap. The expression  $a^*$ , when evaluating the word *aaaa* can lead to 10 different results. The implementation used greatly influences the choice of which results should be returned.

The implementation proposed in this project always favors the longest answers. In case of a tie, the favored answer will be the one that starts leftmost in the input.

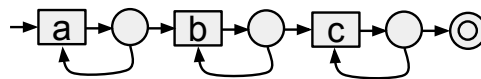
This behavior is consistent with the original article by Thompson and, consequently, with implementation of tools such as *sed* and *grep* of the Unix operating system. On the other hand, it differs from other usual implementations based on Perl, which generally return the first answer found by following a greedy order of evaluation [Cox 2007].

An easy way to verify this is to compare the difference of the match expression  $a^*(b|abc)$  against string *abc*. The default implementation of Python returns the string *ab*. The implementation of this project will find the string *abc*. The difference is that Python's implementation attempts to greedily match the longest string with  $a^*$ , while *Pyrex* can find the longest answer to the entire expression.

### 3.7. Preview

Drawing on the implementation of the *matcher* method it was possible to build a preview function in text mode, showing the step-by-step execution of the algorithm. It aims to facilitate the understanding of the method. It is a command line tool, also written in Python, that uses the *Pyrex* library.

The program shows the steps of the match of any regular expression. For example, if the program is called to visualize the match of the expression  $a^+b^+c^+$  (see the generated automaton in Figure 6) against the string *aabbbcccc*, some steps of the implementation can be seen below:



**Figure 6. Automaton for the expression  $a^+b^+c^+$**

Best answer: <none>

Input: aabbbcccc

^ (3)

```
3 >0000: CONSUME a
    0001: JUMP (1, -1)
0 >0002: CONSUME b
    0003: JUMP (1, -1)
0 >0004: CONSUME c
    0005: JUMP (1, -1)
0006: MATCH!
```

This form of display shows the current status of all flows being executed by the automaton. The number displayed next to each statement is the index of the input where that flow started. In the specific example, there are three flows running:

- The one that starts with each character in the string. (3)
- The one that keeps reading characters b from the input, while there still are. (0)
- The one that is ready to read characters c when they start. (0)

Best answer: aabbbc (0, 6)

Input: aabbbcccc

^ (6)

```
6 >0000: CONSUME a
    0001: JUMP (1, -1)
    0002: CONSUME b
    0003: JUMP (1, -1)
0 >0004: CONSUME c
    0005: JUMP (1, -1)
0006: MATCH!
```

At this time, there is already a match for the string (since there are already characters c read). The best match up to this point is shown on the line *Best answer*. This value is updated at every step. In addition, the following states are active:

**Table 2. Versions of the tested implementations**

Implementation	Environment	Original language
pyrex	Python 2.7.3	Python
re	Python 2.7.3	C
Oniguruma	Ruby 1.9.3	C
java.util.regex	Java 1.6.0_45	Java

- The one that starts with each character in the string. (6)
- The one that keeps reading  $c$  characters at the input while there still are. (0)

The tool uses the matcher method, which returns an iterator that, at each iteration, returns a tuple  $(i, answer, state)$ , where  $i$  is the current index being read at the input,  $answer$  is the best response to date and  $state$  is the current list of states, which is composed of several tuples  $(start, j)$ .

#### 4. Benchmarks

Even though the matching of *backreferences* is an NP-complete problem [Câmpeanu and Santeau 2007], matching regular expressions itself is not. The main objective of this work is to show that even in cases where expressions could be evaluated in polynomial time, it does not happen in a great part of modern implementations.

In order to demonstrate this point, some tests were made using the same regular expression in various implementations and comparing their execution times considering the growth of the input string. All tests were run on an Intel Core i7-3770  $4 \times 2 \times 3.4$ GHz with 16GB of RAM. The system was running GNU/Linux Mint 17 with Linux Kernel 3.13.0-24-generic. Versions of environments where benchmarks were run are shown in Table 2.

Each test was run 50 times and the average duration for each input size was recorded. All graphics are in logarithmic scale to facilitate visualization of the large range of values that each implementation represents.

##### 4.1. Benchmark 1: $(a?a)^+b$

In this test the string  $a^n$  was evaluated against the regular expression  $(a?a)^+b$ . The aim is to show the exponential behavior of implementations based on backtracking. The expression chosen forces the choice for each input character about whether or not to use the character  $a$  in the expression  $a?$ .

Figure 7 shows the execution time for the different implementations.

It should be observed that the implementation in Java, which initially has the worst performance, quickly suffers JIT compilation (*Just in Time*), which causes the runtime to drop significantly, while keeping its exponential character.

##### 4.2. Benchmark 2: $a*b$

In this test, the goal was to demonstrate the high constant of Pyrex implementation in cases where the execution time is linear for all implementations.

Figure 8 shows the execution time for the different implementations.

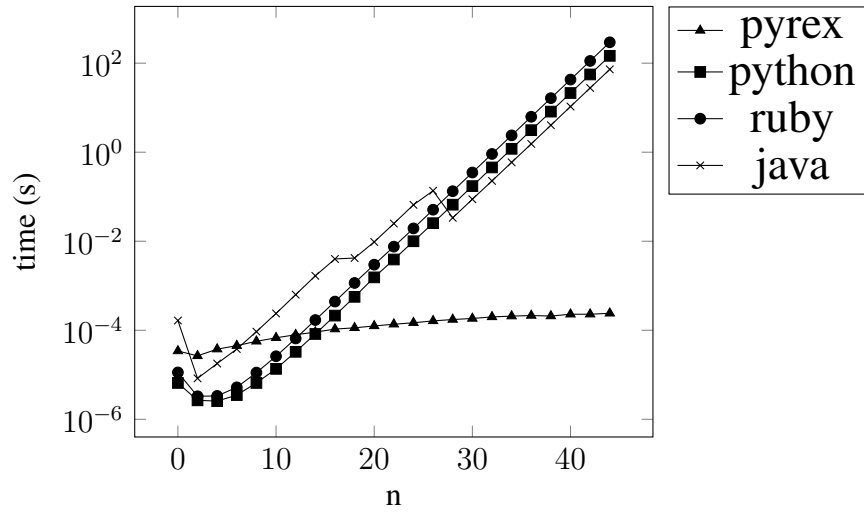


Figure 7. Runtime (in logarithmic scale) to match  $a^n$  against  $(a?a)^+b$

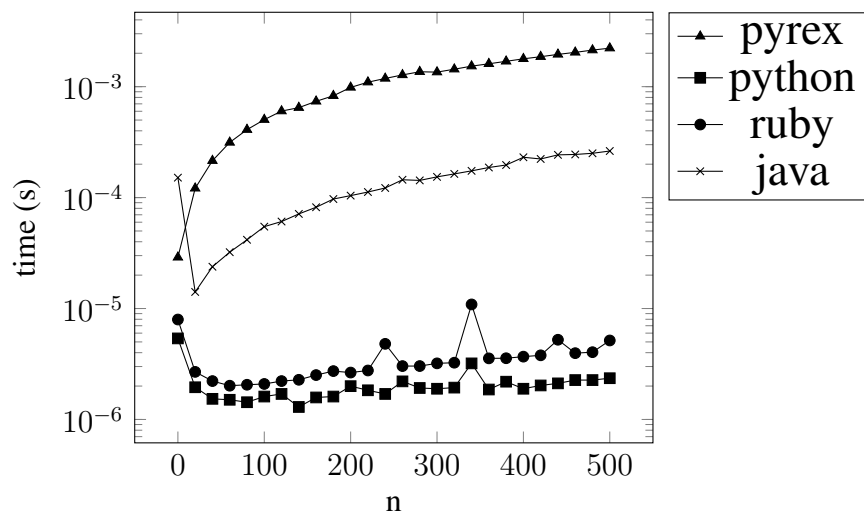


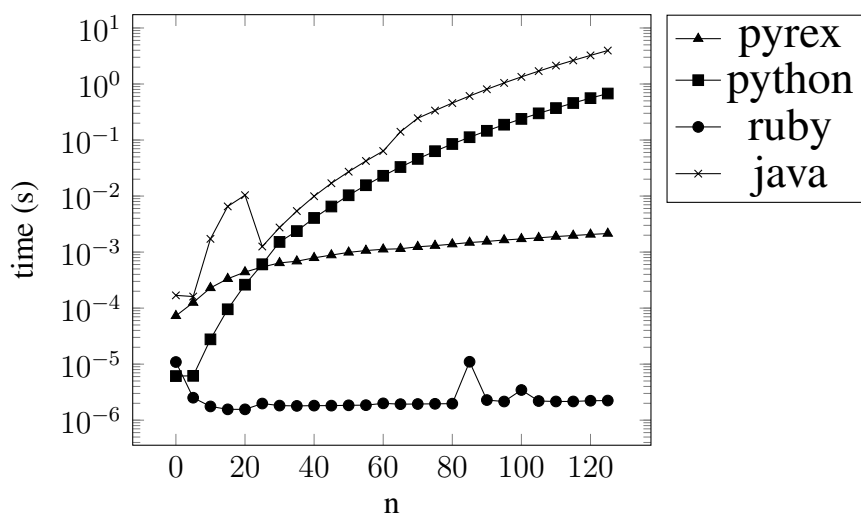
Figure 8. Runtime (in logarithmic scale) to match  $a^n$  against  $a*b$

A clear disadvantage may be observed in implementations in managed code (Pyrex and Java) against other implementations written in C (Ruby and Python) for cases where time complexity is linear.

### 4.3. Benchmark 3: $a^*a^*a^*a^*a^*b$

In this test, the goal was to demonstrate how even in cases where implementations based on backtracking do not generate exponential running time, it is still possible to achieve higher execution time than  $O(n)$ . This expression forces a backtracking for consumption of several repetitions  $a^*$ . The expected running time of implementations based on backtracking is  $O(n^5)$ .

Figure 9 shows the execution time for the different implementations.



**Figure 9.** Runtime (in logarithmic scale) to match  $a^n$  against  $a^*a^*a^*a^*a^*b$

This result also shows how certain static optimizations allow the implementation of Ruby to keep its running time linear while the Python implementation which is also written in C, has a time complexity greater than that of *Pyrex*.

## 5. Conclusion

The main contribution of this work was an implementation of regular expressions using the Thompson Construction Algorithm. A simple method was used to represent the states of non-deterministic finite automata as instructions of an abstract machine that can be deployed on any platform. The implementation was done in Python for simplicity.

The results were quite satisfactory, showing that it is possible to implement regular expressions with polynomial algorithms. The implementation, though not the most efficient for trivial cases, proved to be very useful to demonstrate the well-established theory of regular expressions.

The inability of implementations in languages like Python, Ruby, and Java to recognize them in polynomial time (in the worst case) was also demonstrated, even for some expressions without *backreferences*.

These results confirm the need for parsimony when using regular expressions, especially in software that share hardware resources among multiple users, e.g. websites and other network services.

## References

- Câmpeanu, C. and Santean, N. (2007). On pattern expression languages. *Proceedings AutoMathA*.
- Cox, R. (2007). Regular expression matching can be simple and fast.
- Kirrage, J., Rathnayake, A., and Thielecke, H. (2013). Static analysis for regular expression denial-of-service attacks. *Springer LNCS*, (7873).
- Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. *In Automata Studies, Ann. Math. Stud.*, (34):3–41.
- Mcculloch, W. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147.
- Popov, N. (2012). The true power of regular expressions.
- Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125.
- Thompson, K. (1968). Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.
- Weidman, A. (2010). Regular expression denial of service - redos.