Monoid machines: a $O(\log n)$ parser for regular languages

Armando B. Matos

2006

Abstract

A new method for parsing regular languages is presented; for each regular language L a monoid $(S, \cdot)$ is defined; in particular, every letter a of the alphabet is mapped into an element $f(a)$ of S (in general S contains elements that are not images of letters). Parsing a word $x = ab\cdots c$ consists essentially in (i) finding a transition monoid associated with the language ($O(1)$ time) and (ii) compute the monoid product $x' = f(a) \cdot f(b) \cdots f(c)$. We show that this method can be applied to every regular language but not (if only finite monoids are allowed) to the class of context-free languages. Consider a finite automaton A recognizing L; the elements of the monoid S_L, which include the set $\{f(a) : a \in \Sigma\}$, correspond to *functions* from the set of states of A into the set of states of A. In general the cardinality of the monoid is exponential on the number of states of the minimum deterministic automaton recognizing the language.

Monoid associativity allows an efficient, non-deterministic and highly parallelizable parsing algorithm that consists in repeatedly replacing an arbitrary pair of adjacent monoid elements by their product. We present an extremely fast ($O(\log n)$ time) parallel parsing algorithm.

1 Introduction and examples

The relationship between monoids and regular languages has been extensively studied. However, most regular language recognizing mechanisms mentioned in the literature, such as finite automata, linear grammars ([HU69]) and k-algebras (see Section 2.2 of [Büc89]) are inherently sequential. By contrast we describe a monoid model for the parallel recognition of regular languages; the corresponding parsing algorithm is elegant, non-deterministic, and highly parallelizable. The parsing can proceed in parallel all along the word to be

analyzed; in particular, there is no right or left preference. The algorithm consists essentially in two parts: (i) a pre-processing phase and (ii) repeatedly replace two consecutive group elements by their composition. This contrasts with all other methods known by the author.

We begin by defining a "monoid machine" and characterize its parsing method.

Definition 1 *Let $(S, \cdot)$ be a finite monoid, Σ a finite alphabet, and let $f : \Sigma \to S$ and $g : S \to \{0, 1\}$ be respectively the* input *and* output *functions. The function f is naturally extended as an homomorphism, $f : \Sigma^\star \to S$ A* monoid machine *is a tuple $(S, \cdot, \Sigma, f, g)$. The monoid machine $(S, \cdot, \Sigma, f, g)$ recognizes the language L defined on alphabet Σ if* ⋆⋆

$$\forall x \in \Sigma^\star \;\; g(f(x)) = 1 \text{ iff } x \in L$$

The monoid machine is essentially based on a transition monoid associated with the language L.

The analysis of a word by the monoid machine $(S, \cdot, \Sigma, f, g)$ is described by the following algorithm.

```
Input:  word x
Output:  1 if x ∈ L, 0 otherwise
   Compute y = f(a_1) · f(a_2) ··· f(a_n)
   While |y| ≥ 2 do
      Select two adjacent elements α and β of y        (⋆)
      Replace αβ by α · β
   Output g(y)
```

The computation is not deterministic (the only non-deterministic step is line $(\star)$) and it can be easily implemented in parallel. Let n be the length of the word being analyzed. It is easy to see that (Section 4), if the number of processors is unlimited, the processing time can be $O(\log n)$, while in a DFA recognition, n steps are needed.

The notation used in this paper is fairly standard, see [HU69]. Chap 4?

The rest of this paper is organized as follows. Next section contains examples of the monoids associated with regular languages; the discussion of Example 2.4 is more detailed. In Section 3 we prove that every regular language can be parsed by a monoid machine, in Section 4 we present (non-deterministic) sequential and a parallel programs for the parsing of regular languages and in Section 5 contains some conclusions. ¡

2 Examples

We now present some examples of language recognition by monoid machines. In each case it is important to check that we have indeed a monoid, i.e. that the operation is associative. In these examples we assume for simplicity that *f is the identity function.* In the last example the construction of the monoid machine is exemplified.

2.1 Non empty words consisting only of b's

Let $S = \Sigma = \{a, b\}$ and let f be the identity function. Consider[1] the monoid $(S, \cdot)$

$\cdot$	a	b
a	a	a
b	a	b

Define $g(a) = 0$, $g(b) = 1$. The language $b^+ = \{b, bb, bbb \cdots\}$ is recognized by the monoid machine $(S, \cdot, \Sigma, f, g)$.

2.2 Words containing two or more b's

The following machine recognizes the language defined on $\{a, b\}$ consisting of those words that contain at least two b's: $S = \{a, b, b'\}$, f is the identity function, $g(a) = g(b) = 0$, $g(b') = 1$, and the monoid operation is

$\cdot$	a	b	b'
a	a	b	b'
b	b	b'	b'
b'	b'	b'	b'

2.3 Words ending in bb

The following machine recognizes the language defined on $\{a, b\}$ consisting of those words that end with bb

	a	b	b'
a	a	b	b'
b	a	b'	b'
b'	a	b'	b'

The functions f and g are as in the previous example. Notice that the monoid is not commutative.

[1] For simplicity, here and in the following examples, the row and column that correspond to the unit element ε is not represented.

Letter	Value mod 3	original no. of bits mod 2
0	0	1
1	1	1
2	2	1
a	0	0
b	1	0
c	2	0

Figure 1: **Meaning of the monoid symbols used in the multiple of 3 recognizer. For instance, the symbol c means that the corresponding binary number has a value 2 modulus 3 and that its original representation (which may contain 0's at the left) has an even number of bits.**

2.4 Multiples of 3 written in binary

The machine described below recognizes the regular language defined on $\{0, 1\}$ consisting of those words that represent multiples of 3 written in binary, $L = \{11, 110, 1001 \cdots\}$.

Suppose that $\cdots yz \cdots$ is a partial computation; denote also by y and z the corresponding (original) numeric values. From y mod 3 and z mod 3 we can not compute[2] yz mod 3. We need also to know the value of $|z|$ mod 2:

$\|z\|$ mod 2	y mod 3	$y \times 2^{\|z\|}$ mod 3
0	0	0
0	1	1
0	2	2
1	0	0
1	1	2
1	2	1

An example of the last line of the table is $y = 5 = 2 \pmod 3$ and $|z| = 3 = 1 \pmod 2$:

$$5 \times 2^{3 \bmod 2} = 5 \times 2^1 = 2 \times 2^1 = 1 \pmod 3$$

It is now easy to build a monoid machine that recognizes the language. Let $S = \{0, 1, 2, a, b, c\}$ where the meaning of the letters is given in Figure 1. Figure 2 contains the complete table of the monoid. For instance the operation "$b \cdot 2 = 1$" (row b, column 2) means that if we have xy where $x = 1 \pmod 3$, $|x| = 0$

[2]Here yz denotes the concatenation of words y and z and not the product of the corresponding integers.

	0	1	2	a	b	c
0	a	b	c	0	1	2
1	c	a	b	1	2	0
2	b	c	a	2	0	1
a	0	1	2	a	b	c
b	2	0	1	b	c	a
c	1	2	0	c	a	b

Figure 2: **Monoid table of the Example 2.4, "multiples of 3". The first operand is on the first column and the second on the first row; we have for instance $c \cdot 2 = 0$.**

xy in two parts	symbol of first part	symbol of second part	Result
1 010101 $(1, 21)$	1	a	1
10 10101 $(2, 21)$	c	0	1
101 0101 $(5, 5)$	2	a	1
1010 101 $(10, 5)$	b	2	1
10101 01 $(21, 1)$	0	b	1
101010 1 $(42, 1)$	a	1	1

Figure 3: **Various ways to break the computation of** 1010101**. Due to associativity the result is always the same.**

(mod 2), $y = 2 \pmod 3$, $|x| = 1 \pmod 2$, then the number represented by xy is associated with b so that

$$xy = 1 \pmod 3 \text{ and } |xy| = 1 \pmod 2$$

That is the case for instance with $x = 1010_2 = 10$ (4 bits), $y = 101_2 = 5$ (3 bits); the binary number $xy = 1010101_2 = 85 = 1 \pmod 3$, $|1010101| = 7 = 1 \pmod 2$. Notice that, by construction, the operation is associative. We could have partitioned xy in the various ways as represented in Figure 3. The output function is defined by $g(0) = g(a) = 1$ and $g(1) = g(2) = g(b) = g(c) = 0$.

2.5 Another example

We consider the finite automata and corresponding transition monoid of [Pin10], Section 1.2.1 ("L'approche algébrique", "Automates déterministes et monöides de transition").

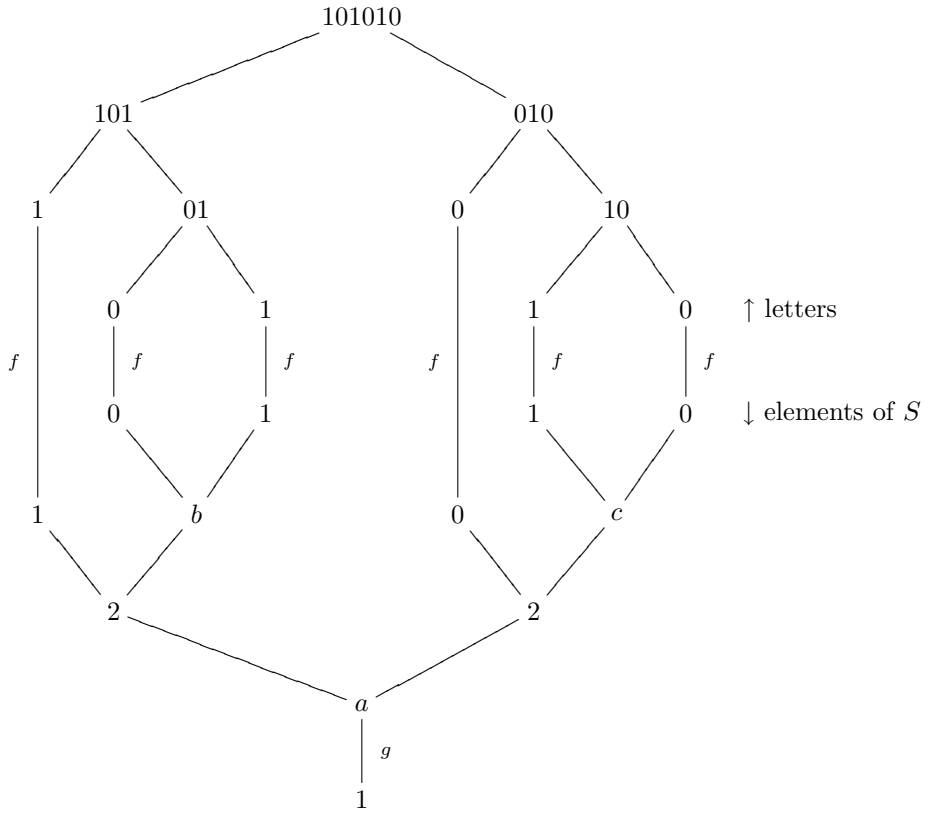


Figure 4: **This computation refers to Example 2.4; see also the monoid table in Figure 2. Top: a possible computation tree showing that $42 = 101010_2$ is a multiple of 3 (because $g(a) = 1$). The (parallel) computation may also be represented by $1\underline{0}\,1\,0\,\underline{1}\,\underline{0} \to \underline{1b}\,\underline{0c} \to \underline{22} \to a$. In the general case, it is easy to see that the minimum number of steps is $\lceil \log n \rceil$ where n is the length of the input word. In this example there are $3 = \lceil \log 6 \rceil$ steps: $1) \to 2)$, $2) \to 3)$ and $3) \to 4)$.**

Consider Figure 1.6 of [Pin10]. The deterministic finite automaton $(\Sigma, S, I, F, \delta)$ is defined by

- $\Sigma = \{a, b, c\}$
- $S = \{1, 2, 3\}$, $I = \{1\}$, $F = \{3\}$
- $\delta = \{(1, a, 2), (1, b, 1), (2, a, 2), (2, b, 3), (2, c, 2), (3, a, 2), (3, b, 3), (3, c, 3)\}$

Using the associativity property and the relations in Figure 1.6 of [Pin10], it is possible to reconstruct the operation table of the monoid:

	a	b	c	$\underline{ab}$	$\underline{bc}$	$\underline{ca}$
a	a	$\underline{ab}$	a	$\underline{ab}$	$\underline{ab}$	a
b	a	b	$\underline{bc}$	$\underline{ab}$	$\underline{bc}$	$\underline{ca}$
c	$\underline{ca}$	$\underline{bc}$	c	$\underline{bc}$	$\underline{bc}$	$\underline{ca}$
$\underline{ab}$	a	$\underline{ab}$	$\underline{ab}$	$\underline{ab}$	$\underline{ab}$	a
$\underline{bc}$	$\underline{ca}$	$\underline{bc}$	$\underline{bc}$	$\underline{bc}$	$\underline{bc}$	$\underline{ca}$
$\underline{ca}$	$\underline{ca}$	$\underline{bc}$	$\underline{ca}$	$\underline{bc}$	$\underline{bc}$	$\underline{ca}$

Comments about this table

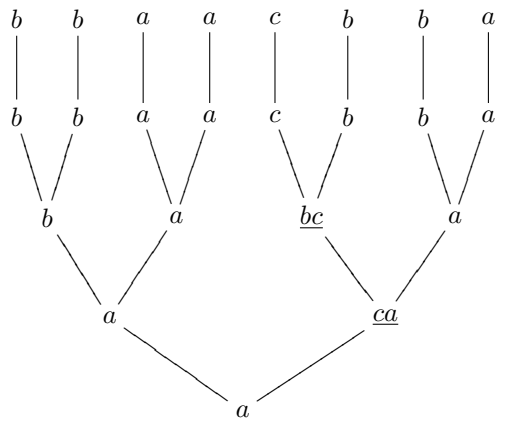
- The monoid set is $\{a, b, c, \underline{ab}, \underline{bc}, \underline{ca}\}$.
- Elements $\underline{ab}$, $\underline{bc}$ abd $\underline{ca}$ are underlined in order to emphasise that they are just names, not monoid products.
- For simplicity the identity element is not represented.

The initial state of the monoid is 1 and $a(1) = 2$, which is not a final state, so that the word *bbaacbba* is not accepted by the automaton.

2.6 Paralel deterministic $O(\log(n)$ parser

1. A DFA $A = \{\Sigma, S, \{s\}, F, \delta\}$ that recognizes the language L is given. Compute the table of the corresponding transition monoid; $O(1)$ time.
2. Given a word $w = a_1 a_2 \dots a_n \in \Sigma^\star$ compute the corresponding monoid in $\lceil \log(n) \rceil$ steps, as illustrated in the following figure for the example

$w = bbaacbba$.



In this diagram the top line consists of letters, while all the other lines contain elements of the monoid (i.e. functions $S \to S$).

3. Let-$\mathcal{M}$ be the monoid corresponding to w. If $\mathcal{M}(s) \in F$ output "$w \in L$", otherwise output "$w \notin L$". In the example, we have $\mathcal{M}(1) = 2 \notin F$ so that $bbaacbba \notin L$.

2.7 Parallel non deterministic parser

In the example presented in [Pin10], the algebraic properties of the monoid are characterized by the following set R of relations

$$\begin{array}{lll} ab = \underline{ab} & bc = \underline{bc} & ca = \underline{ca} \\ aa = a & bb = b & abc = \underline{ab} \\ ac = a & cb = \underline{bc} & bca = \underline{ca} \\ ba = a & cc = c & cab = \underline{bc} \end{array}$$

Notice that the RHS of each relation is (strictly) shorter than the corresponding LHS.

Then the following non deterministic algorithm is possible

1. While $|w| > 1$ select a contiguous sub-word x of w, such that $x = y$ is a relation. In w replace x by y. Notice that $|w|$ strictly decreases.

2. Finish as in item 3 of the previous algorithm.

We are dealing essentially with the parallel computation of an associative product.

3 Every regular language can be recognized by a monoid machine

As we said earlier, a monoid machine computation is very different from "sequential" models like DFAs, or linear grammars where the input is typically read and processed from the left to the right. The monoid "computation" is non-deterministic (the splitting of the word in parts to be analyzed is irrelevant) and can be easily parallelized. The non-determinism comes from the fact that *any* pair of adjacent symbols can be selected for the next step of the computation.

In this section we prove that every regular language can be recognized by a monoid machine with finite alphabet so that we have another nice characterization of regular languages. The proof of Theorem 1 is important because it describes a method for constructing a monoid machine equivalent to a given regular language; it should also be noticed that the elements of the monoid (which correspond to equivalence classes of words) do not correspond to the states of a recognizing automata but to functions from the set of states into the set of states (see also Example 2.4).

Theorem 1 *A language L not containing the empty word is regular if and only if there is a finite monoid machine $(S, \cdot, \Sigma, f, g)$ that recognizes L.*

Proof. Let us suppose first that L is regular and let $A = (Q, q_0, F, \delta, \Sigma)$ be a complete DFA that recognizes A. We define a finite monoid machine $(S, \cdot, \Sigma, f, g)$ that recognizes L.

Let S be the set of all functions from Q to Q. Obviously S is finite, $|S| = |Q|^{|Q|}$. Let $\alpha, \beta \in S$; define $\alpha \cdot \beta$ as the function $(\alpha \cdot \beta)(q) = \alpha(\beta(q))$. For $a \in \Sigma$ define $f(a)$ as the function that assigns to the state $q \in Q$ the state $\delta(q, a)$ and finally for every $\alpha \in S$ define $g(\alpha) = 1$ as 1 if $\alpha(q_0) \in F$ and $g(\alpha) = 0$ otherwise. It is straightforward to show that this is a monoid machine (the associativity follows from the associativity of function composition) that recognizes the same language as the DFA A. In fact, $(S, \cdot)$ is the transition monoid associated with A, see for instance [Eil74].

We now suppose that some finite monoid machine recognizes L. The computation can in particular proceed from left to right, "consuming" the images by f of the letters of the original word; it is easy to see that this corresponds exactly to the action of a DFA, so that L is regular. ⊣

Observations

– If we want to represent the empty word ε we define a monoid $S' = S \cup \{e\}$,

$e \notin S$, where e is the identity element[3], $a \cdot e = e \cdot a = a$ for every $a \in S'$. Then we define $f(\varepsilon) = e$ and $g(e) = 1$ iff $\varepsilon \in L$.

– In many practical cases it is possible to recognize the language with a a monoid with much less than $|Q|^{|Q|}$ elements.

4 Sequential (non-deterministic) and parallel parsing implementations

We first present the sequential, non-deterministic algorithm.

```
Input:  DFA recognizing L and a word x = a_1 ··· a_n ∈ Σ*
Output:  1 if x ∈ L, 0 otherwise
   From the DFA define (see the proof of Theorem 1):
      - the monoid S and its table,
      - the functions f and g
   Compute the expression x' = f(a_1) · f(a_1) ··· f(a_n) as follows:
   While |x'| ≥ 2
      Select i ∈ {1, 2 ··· , |x'| − 1} (non-deterministic step)
      Replace a_i · a_{i+1} by the corresponding result (use the monoid table)
   Return g(x') (at this stage |x'| = 1)
```

For a fixed language L the algorithm has efficiency $O(n)$.

The parallel algorithm is similar; the difference is that the replacement of the $a_i \cdot a_{i+1}$ by the result of the operation is made in parallel for $i = 1, 3, \ldots$. We present a recursive version in Figure 5.

To analyze this algorithm let h be height of the computation tree associated with the function `compute`. If $n = |x|$ is a power of 2, say $n = 2^p$, it is easy to see that $h = \log n$; for an arbitrary computation tree (which depends on n), the height h is exactly $\lceil \log n \rceil$.

Corollary 1 *A regular language L can be recognized in parallel in time $O(\log n)$ (where n is the length of the word in consideration) by an unbounded set of parallel elementary finite processors.*

For context-free languages the parsing by finite monoid machines is no longer possible. To show this it suffices to consider the language $\{a^n b^n \ : \ n \geq 0\}$; let $c \in S$ be the symbol resulting of a left to right parsing of the first n symbols

[3]The neutral element e is of course unique, even if S was already a monoid. Notice also that, once $e \notin S$, if $a \cdot b = e$ we must have $a = b = e$.

```
Input:  a DFA recognizing L and a word x = a_1 ··· a_n ∈ Σ*
Output:  1 if x ∈ L, 0 otherwise
   From the DFA compute (see the proof of Theorem 1):
      - the monoid S and its table,
      - the functions f and g
   Compute f(a_1), f(a_2),..., f(a_n)
   Let x' be the monoid product x' = f(a_1) · f(a_2) ··· f(a_n)
   Return g(y) where y = compute(x') (note:  |y| = 1)

compute(x)
Input:  a monoid product x
Output:  the result of the product x (a monoid element)
   n ← |x|
   If n = 1 return(x)
   m ← n/2 (integer division)
   a =compute(x[1 ··· m]) || b =compute(x[m + 1 ··· n])
   return(c) where c = a · b (from the table)
```

Figure 5: **A parallel parser for regular languages. In the function `compute` the following notation is used: (i) $x[i..j]$ denotes the sub-expression of x consisting of the elements with indices i, $i+1$,..., j; (ii) "||" denotes parallel call, that is, "$f(a)$ || $g(b)$" consists of two concurrent processes.**

of $a^n b^n$ (due to associativity any computation order is equivalent). Different values of n must correspond to different values of c but this is not possible because S is finite. The class of regular language is exactly the class of languages whose words only contain a "bounded amount of information" among their subwords and this property corresponds exactly to the existence of an associative (in the algebraic sense) parsing.

5 Conclusions

We described a recognition algorithm for regular languages that works in $O(\log n)$ parallel time where n is the length of the word which is analyzed. Essentially, to each letter of the word being analyzed corresponds a finite function and each basic step of the algorithm is a composition of those functions. These "monoids machines" are inherently non-deterministic: due to associativity, the "computation" of a sequence $a_1 \cdot a_2 \cdots a_n$ can proceed in any order, even in parallel. This contrasts with the usual methods for the recognition of regular languages.

Non-determinism and parallelism are also common properties of DNA and RNA computing (see for instance [Adl94, Amo05]); we think that monoids machines may be an adequate tool for the analysis of some problems in this area.

References

[Adl94] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science, New Series*, 266(5187):1021–1024, 1994.

[Amo05] Martyn Amos. *Theoretical and Experimental DNA Computation*. Springer, 2005.

[Büc89] J. Richard Büchi. Finite automata, their algebras and grammars: Towards a theory of formal expressions. Springer-Verlag, 1989.

[Eil74] Samuel Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, New York, 1974.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, Reading, Massachusetts, 1969.

[Pin10] Jean-Eric Pin. Automates réversibles: combinatoire, algébre et topologie. *Leçons de mathématiques d'aujourd'hui*, 583, 2010.