Cursors

Vincent Le Maout

Institut Gaspard Monge, Université de Marne La Vallée
Champs-sur-Marne, France
lemaout@univ-mlv.fr

Abstract

The iterator concept is becoming the fundamental abstraction of reusable software and the key to modularity and clean code especially in object-oriented languages like C++ and Java. They serve as accessors to a sequence hiding the implementation details from the algorithm and their encapsulation power allows now true generic programming. The Standard Template Library defines clearly their behavior on simple sequences like linked lists or vectors. In this paper, we define the concept of cursors which can be seen as a generalization of the iterator concept to more complex data structures than sequences, in this case acyclic automata. We show how useful and efficient they can be on applications in C++ based on the Automaton Standard Template Library [VLM98].

1 Introduction

Cursors introduce a software layer between the deterministic finite automaton classes of the Automaton Standard Template Library (ASTL, a C++ automaton library [VLM98]) and the algorithms. Likewise the iterator concept, the cursor concept serves two purposes: making the algorithm undependant from the automaton structure and removing some of the algorithmic responsibilities from the algorithm core. It must be regarded as a generalization of the iterator concept: the main difference is that when iterating on data, a cursor need to know which transition to follow whereas an iterator knows of only one path. Therefore, assigning a traversal algorithm to a cursor makes it an iterator and a cursor can be built from an iterator too.

This obviously implies that these objects have a well-defined, consistent and simple behavior on which to rely on. Moreover, generic programming standards impose tough efficiency constraints one has to comply to.

This paper introduces the concept of cursor on acyclic automata. We first present a few definitions and programming abstractions and then expose the main obstacles we met using ASTL which required introduction of three major models of cursors: forward, stack and depth-first cursors and their applications. We then discuss the issue of applying algorithms and show how easy and straightforward it is to combine them to create more powerful algorithms.

2 Definitions

2.1 Deterministic Finite Automaton

To allow more flexibility, we add to the classical DFA definition a set of *tags*, that is any data needed to apply an algorithm associated to each state.

Let $A(\Sigma, Q, i, F, \delta, T, \tau)$ be a 7-uple defined as follow: Σ is the alphabet, Q a set of states, $i \in Q$ the initial state, $F \subseteq Q$ a set of final (accepting) states, $\Delta \subseteq Q \times \Sigma \times Q$ a set of

transitions, T a set of tags, $\tau \subset Q \times T$ a relation from states to tags.
We distinguish one special state noted 0 and called the null or sink state. For all automaton $A(\Sigma, Q, i, F, \delta, T, \tau)$ we have $0 \in Q$.
We define $P(X)$ as the power-set of a set X and $|X|$ as its number of elements.

2.2 Access Functions and Sink Transitions

To access Δ we define two transition functions δ_1 and δ_2 :

$$\delta_1 \,:\, Q \times \Sigma \cup \{\epsilon\} \to Q$$
$$\forall q \in Q,\, \forall a \in \Sigma \cup \{\epsilon\},\, \delta_1(q,a) = \begin{cases} p \text{ if } (q,a,p) \in \Delta \\ 0 \text{ otherwise} \end{cases}$$

δ_1 retrieves transitions aims given the source state and a letter (possibly the empty word ϵ). In the case of undefined transitions the result is the null (sink) state which make the DFA complete.
A transition verifying the following property is called a *sink transition*:

$$(q,a,p) \in Q \times \Sigma \cup \{\epsilon\} \times Q,\, q = 0 \text{ or } p = 0$$

δ_2 retrieves all outgoing transitions of a source state allowing thus traversal and iteration:

$$\delta_2 \,:\, Q \to P(\Sigma \times Q)$$
$$\forall q \in Q,\, \delta_2(q) = \{(a,p) \in \Sigma \times Q \text{ such that } (q,a,p) \in \Delta\}$$

2.3 Object Properties

- We say that an object `x` is *assignable* iff it defines an operator `=` allowing assignment from an object `y` of the same type. Postcondition is "`x` is a copy of `y`".
- An object is *default constructible* iff no values are needed to initialize it. In C++, it defines a default constructor.
- An object is *equality-comparable* iff it provides a way to compare itself with other objects of the same type. In C++, such an object defines an operator `==` returning a boolean value.
- An object is *less-than-comparable* iff there exist a partial order relation on the objects of its type. In C++, such an object provides an operator `<`.

2.4 Iterators

Quoting from SGI Standard Template Library reference documentation :

> Iterators are a generalization of pointers: they are objects that point to other objects. As the name suggests, iterators are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element.

Iterators are:

1. *default constructible.*

2. *assignable* (infix operator =).

3. *singular* if none of the following properties is true. An iterator with a singular value only guarantees assignment operation.

4. *incrementable* if applying ++ operator leads to a well-defined position.

5. *dereferenceable* if pointed object can be safely retrieved (prefix operator *).

6. *equality-comparable* (infix operator ==).

Iterators constitutes the link between the algorithm and the underlying data structure: they provide the sufficient level of encapsulation to make processing undependant from the data. Schéma

2.5 Range

A valid *range* `[x,y)` where `x` and `y` are iterators represents a set of positions from `x` to `y` with the following properties :

1. `[x,y)` refers to all positions between `x` and `y` but not including `y` which is called the *end-of-range* iterator (also denoted as a *past-the-end* iterator).

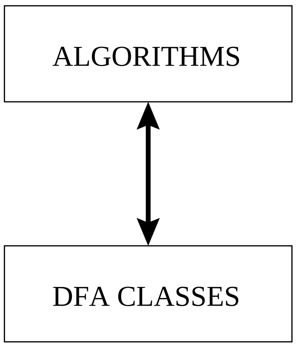
2. All iterators in the range except `y` are incrementable and dereferenceable.

3. All iterators included `y` are equality comparable.

4. A finite sequence of incrementations of `x` leads to position `y`.

3 Weaknesses of the Two-Layer Model

ASTL is structured around a "two-layer" model with algorithms directly lying on the automaton data structure:



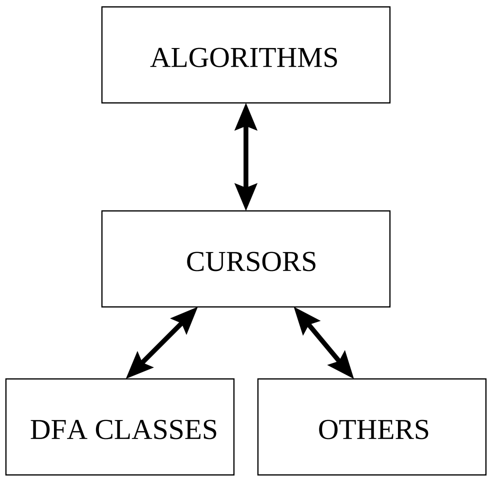
This unfortunately induces limits to the algorithm application spectrum and shortcomings regarding mainly four aspects:

1. A much too strong coupling between the algorithms and the automaton classes reduces genericity and impose strictly complying input data: a new algorithm version has to be written for more exotic data or an entire new automaton class has to be designed (which is not always possible due to combinatorial problems for instance but is surely time-taking and error prone).This yields multiple instances of the same code which contradicts generic programming purpose. Moreover, an intermediary layer allows for data hiding and algorithm application to other structures than automata (see Sect. 4.4.2).

2. Common parts shared by many algorithms should be written only once and be reused as is. This means moving intensively used functionalities to a third part other than algorithm core or automaton. The best example is the iteration over transitions of a DFA: all algorithms have an iteration policy and they can be reduced to a few ones, depth-first, breadth-first and a couple of others. A cursor is the place to implement those common parts (see Sect. 6 about depth-first cursor).

3. The algorithm must not imposes behavior that should be externally decided of: for example, writing a DFA union algorithm should not involve decisions about applying it on-the-fly, by lazy construction or by copy. The decision should be taken at utilization time and not at design time (see how to apply an algorithm in Sect. 7).

4. Reusing and combining algorithms should be a straightforward operation requiring no extra code. Hard coded algorithms prevent such flexibility but cursor adapters allows it.



4 Forward Cursor

Using a cursor is a way of entirely hiding the processed data structure which need not to actually be an automaton any more (see application 4.4.2). Moreover, it allows to draw functionalities from the algorithm to the cursor layer leading to clearer and lighter algorithms (see algorithm language in sect. 6.5).

4.1 Definition

A forward cursor is basically an object pointing to a DFA transition $(q, a, p) \in Q \times \Sigma \cup \{\epsilon\} \times Q$ allowing access and forward moves along it. Likewise iterators they represent positions in the automaton and therefore can be used to define ranges over it.

4.2 Properties

1. A forward cursor is default constructible but has by default a singular value.
2. A forward cursor is assignable.
3. A forward cursor is equality-comparable.
4. A forward cursor is dereferenceable (one can access to source, aim and letter of pointed transition) iff it does not point to a sink transition.
5. A forward cursor is "incrementable" (it may move along the current transition) iff it does not point to a sink transition.

4.3 Interface

`X`	a type which is a model of forward cursor
`x, y`	objects of type `X`
`a`	is a letter. Alphabet is assumed to be an unsigned integral type
$(q, a, p) \in Q \times \Sigma \cup \{\epsilon\} \times Q$	the transition `x` is pointing to

Name	**Expression**	**Semantics**
source state	`x.src();`	return q
aim state	`x.aim();`	return p
letter	`x.letter();`	return a
final source	`x.src_final();`	return `true` if $q \in F$
final aim	`x.aim_final();`	return `true` if $p \in F$
comparison	`(x == y)`	`true` if `x` represents
		the same position as `y`
forward	`x.forward();`	move along current
		transition
forward with letter	`x.forward(a);`	move along transition labeled with `a` return `true` if $\delta_1(q, a) \neq 0$
first transition	`x.first_transition();`	set `x` on the first transition of $\delta_2(q)$ return `true` if $\delta_2(q) \neq \emptyset$
next transition	`x.next_transition();`	move on to the next transition of set $\delta_2(q)$ return `false` if (q, a', p') is a sink transition
find	`x.find(a);`	set `x` on the transition labeled with letter `a` return `true` if $(q, a, \delta_1(q, a))$ is not a sink transition

Writing a cursor is a fairly simple operation. One has to consider three primary actions:

1. What to do when setting the cursor on the first outgoing transition of current state.
2. What to do when moving on to the next outgoing transition.
3. And what to do when moving forward along the currently pointed transition.

Secondarily, one has to figure out which action to take for `find`. This induce the `forward(a)` behavior since it is semantically equivalent to `if (find(a) == true) forward();`

4.4 Applications

We implemented first one forward cursor class with the default behavior. This was enough to write an algorithm testing if a word defined by a range `[first,last)` is in the recognized language of a DFA accessed through a cursor `c`:

```
bool is_in(InputIterator first, InputIterator last, ForwardCursor c) {
  while (first != last && c.forward(*first))
    ++first;
  return first == last && c.src_final();
}
```

Then cursor adapters revealed their true encapsulation and adaptability power. We implemented a dozen of them:

- five set operations featuring union, intersection, difference, symmetrical difference and concatenation.
- a hash cursor, computing a hash value for the recognized word along its path. This algorithm described in [DR92] realizes a bijective mapping between the recognized words of the DFA and integers providing a fast perfect hash function.
- a default transition cursor using a failure forward function making the automaton complete in a space saving way: if requested transition is a sink transition then the cursor moves along the default transition.
- string and C-string cursors giving to a simple word or sequence a cursor interface and therefore a flat automaton look.
- permutation cursor implements in a very simple way a virtual automaton recognizing all permutations of a word.
- a scoring cursor computing an integer value for a matched pattern as the sum of all the scores of the matched sub-expression. This was used in a search engine to evaluate the amount of confidence for each piece of matched text.

In the following three sections, we will take a closer look at three of these adapters, the set operations, the permutation cursor which highlight the cursor encapsulation power and the scoring cursor which shows their evolutivity power.

4.4.1 Set Operations

Union, intersection, difference, symmetrical difference and concatenation cursors are binary forward cursor adapters. They make use of two underlying forward cursor and provide the same complying interface. They perform on-the-fly all algorithmic operations needed and can be immediately used. The following piece of code check if word `word` is in the recognized language of the intersection of two DFAs `A1` and `A2` :

```
char word[] = "word to check";
DFA A1, A2;
forward_cursor<DFA> c1(A1.initial()), c2(A2.initial());
if (is_in(word, word + 13, intersection_cursor(c1, c2)))
  cout << "ok";
else
  cout << "not found";
```

4.4.2 Permutations Automaton

This example shows a convenient way to hide data structure and to overcome memory space limitations and combinatorial explosion.
The regular expression matcher we developed had to look for n patterns combined in any possible order. The matcher engine had been previously written to search a range `[first,last)` of characters with a cursor `c` on the regular expression DFA:

```
bool match(InputIterator first, InputIterator last, ForwardCursor c) {
  for(; first != last && c.forward(*first); ++first)
     if (c.src_final()) return true;
  return false;
}
```

By simply changing the default cursor to a multiple cursor encapsulating n forward cursors we can look simultaneously for n patterns with the same algorithm. Assigning a unique integer $i \in [1, n]$ to each pattern, the problem comes down to match a word in the DFA recognizing all permutations of the sequence $1, 2, 3, ..., n$.
The first approach is to precompute the automaton for all permutations, but this quickly turns out to be much too space consuming when n reaches 9. The second step is to minimize the automaton which is very efficient: by using a compact data structure, we can shrink the automaton for $n = 8$ to about 20 Ko but the memory usage peak due to minimization and the pretty long processing time remained major drawbacks. Eventually, we created a cursor moving on a virtual automaton simply by keeping tracks of the letters of visited transitions with a bit vector. A vector full of 1 means you have reach an accepting state.

4.4.3 Regular Expression Scoring

The second enhancement to the regular expression matcher was to be able to assign a score to any sub-expression and to sum up the scores when a piece of text matched the general expression. The scores were stored in the states tags at construction stage and once again, nothing had to be changed excepted the cursor: it only had to accumulate each visited state tag along the path.

Example of regular expression defined on words:
`(president[5]|boss[3]|chief[2]) of @{0,3} (company[5]|corporation[4]|Inc)`
That means: look for any of the following words `president`, `boss` or `chief` followed by the word `of` and followed at maximum three words away by `company`, `corporation` or `Inc`. The postfix operator `[n]` assigns scores to sub-expressions and default score is 0. That means that the text: `president of Foobar company` is more interesting to us than the sentence: `chief of a big corporation` because the former gets a score of 10 and the latter a score of 6.

5 Stack Cursor

A stack cursor is in some sense a bidirectional cursor. By keeping track of its previous positions during iteration, it is able to move back.

5.1 Definition

A stack cursor is a forward cursor using a stack to store its path along the automaton allowing thus backward iteration. It is constituted of a stack of forward cursors and all operations apply to the stack top. Its behavior relies on the underlying forward cursor.

5.2 Interface

The interface of stack cursor is nearly the same as the forward cursor: the `forward` action pushes the resulting cursor on to the top and the `backward` action pops it.
Remark that the comparison operator compares not only tops but entire stacks. The reason will be made clear from the depth-first cursor abstraction in Sect. 6.

Stack Cursor Requirements

A stack cursor needs to implement forward cursor requirements plus:

Name	**Expression**	**Semantics**
forward	`x.forward();`	move along current stack top transition and push (p, a', p')
forward with letter	`x.forward(a);`	move along transition labeled with `a` push $(q, a, \delta_1(q, a))$ and return `true` if $\delta_1(q, a) \neq 0$
backward	`x.backward();`	pop. return `false` if resulting stack is empty
comparison	`(x == y)`	return `true` if `x` stack is a copy of `y` stack

From the default stack cursor implementation we designed an adapter called the distance cursor.

5.3 Application : the Distance Cursor

The problem is a typical one which one meet when designing a spelling corrector: given a word w and an editing distance d, find all words w' recognized by an automaton A for which the edition operations consisting of successive letter insertions, deletions and substitutions needed to w' from w does not exceed a score of d. Each operation type is assigned a score and the final distance is the scores sum.
The distance cursor adapts the default stack cursor in two ways:

1. It has the responsibility to hide paths straying too far away from w.
2. It has to adapt the stacking policy to manage the deletion operation: when moving forward, it has to move on to the next letter of w but has to stay on the current transition. That means pushing a copy the current cursor leaving it unchanged.

6 Depth-First Iteration Cursor

6.1 Definition

A depth-first cursor, as the name suggests accesses the transitions of a DFA in a depth-first order. It relies on the stack cursor implementation but belongs to a different concept: a depth-first cursor represents an algorithm stage rather than a mere position in the automaton.

6.2 Properties

1. A depth-first cursor iterates on transitions in a depth-first order.
2. A depth-first cursor never pushes sink transitions (the underlying stack holds only dereferenceable cursors). When reaching a sink transition, the action taken is conceptually equivalent to a `pop`.
3. It comes from previous property that each non sink automaton transition is reached exactly twice: once in descending stage and then once in ascending stage.
4. A depth-first cursor represents an algorithm stage and consequently can be used to define algorithm applying ranges passed to a function.
5. It is default constructible, assignable, equality-comparable, dereferenceable and incrementable.

6.3 Interface

Name	**Expression**	**Semantics**
source	`x.src();`	return q
aim	`x.aim();`	return p
letter	`x.letter();`	return a
final source	`x.src_final();`	return `true` if $q \in F$
final aim	`x.aim_final();`	return `true` if $p \in F$
forward	`x.forward();`	move on to the next transition in depth-first order return `true` if `x` actually moved forward or `false` if `x` poped.
comparison	`(x == y)`	compares entire stacks.

6.4 Algorithm Depth-first Iteration

To grant more freedom to algorithms users we will use depth-first cursors to define algorithm applying range. Starting from a position in the automaton given by a cursor set to a specified

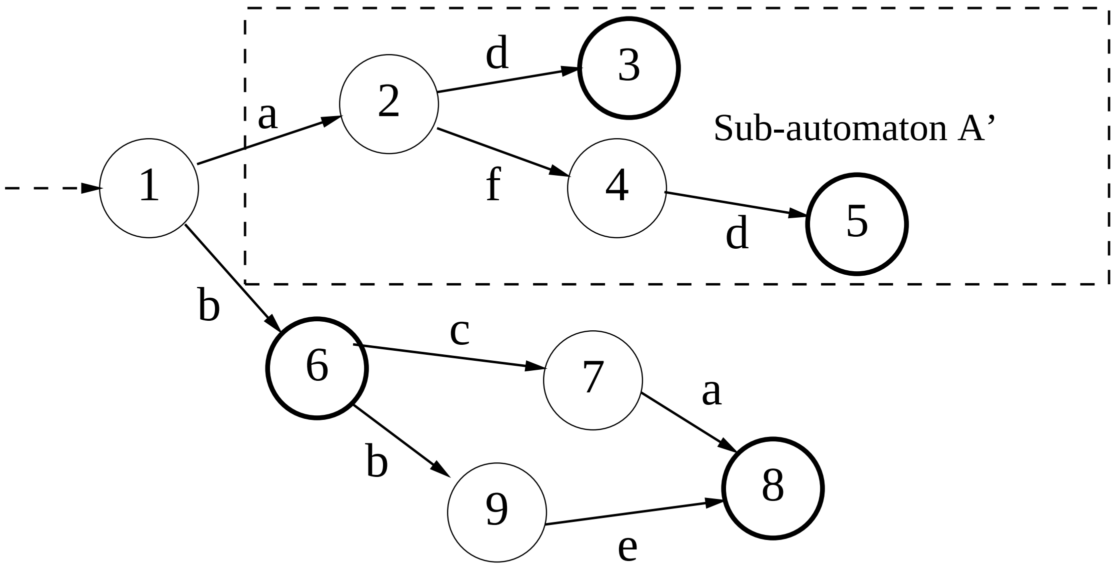


Figure 1: Automaton A

transition, algorithms will apply until a stop condition becomes true, most of the time until the stack is empty. This stop condition can be represented by a cursor too. Remember depth-first cursors are algorithm stages. Consequently, initializing a depth-first cursor with the empty stack makes it a valid end-of-range position and algorithm stops when the beginning cursor reaches the empty stack state.

6.5 Application : the Language Algorithm

```
void language(DepthFirstCursor first, DepthFirstCursor last) {
  vector<char> word;
  while (first != last) {                // until end of range
    word.push_back(first.letter());      // push current letter
    if (first.aim_final()) output(word); // if aim is final display word
    while (! first.forward())            // while moving backward
      word.pop_back();                   // pop the last word letter
  }
}
```

Example: display automaton A language (Figure 1)

```
DFA A;

// initialize start of range with state 1:
forward_cursor<DFA> begin(A.initial());

// set the cursor on transition (1,a,2):
begin.first_transition();
```

```
// use empty stack (default value) for end of range:
language(depth_first_cursor(begin), depth_first_cursor());
```

Restricted range: display sub-automaton A' language

```
// initialize start of range with transition (1,a,2):
forward_cursor<DFA> begin(A.initial()), end(A.initial());
begin.first_transition();

// initialize end of range with transition (1,b,6):
end.find('b');
language(depth_first_cursor(begin), depth_first_cursor(end));
```

7 Applying Algorithms

The main benefit of cursors is to allow to decide at the last minute in which way to apply an algorithm rather than at design time. Most of the time, one of the following policies is more adapted to one's need but the algorithm cannot be aware of it. Instead of writing three versions of the code, cursors allow external decisions.

7.1 On-the-fly Processing

Sometimes combinatorial problems get in the way and there is no way to build the desired automaton (Sect. 4.4.2). Sometimes operation is only punctual and there is no need to build an entire automaton as in Sect. 4.4.1 for intersection. Consequently, algorithms impose only constraints on the public interface and possible extra internal processing is left up to the user.

7.2 Lazy Construction

When the ratio of preprocessing time against computing time becomes too large, it is necessary to delay the actual construction or more precisely to incrementally build the resulting automaton. A famous example is the incremental construction of a DFA from a regular expression during the scanning of the text in [ASU86]. It is also particularly interesting in determinizing a NFA.
In this case, construction is made internally by a cursor adapter called the lazy cursor completely encapsulating extra processing. This adapter is passed to the algorithm where a default cursor would be passed in a on-the-fly operation.

7.3 Building by Copy

Whenever actual resulting DFA building is needed, one can either use the `clone` algorithm which makes an exact copy from a cursor into a new DFA or the `copy` algorithm which duplicate the input DFA in ascending stage of the depth-first iteration removing unneeded path leading to non accepting states.

7.4 Algorithm Combining

Cursors offer a simple and straightforward way to extend algorithm power. We saw in Sect. 4.4.1 that set operations are binary operators. It is therefore very easy to combine them to

achieve more sophisticated functionalities. For example, the symmetrical difference defined as $\Delta(A,B) = A \cup B \setminus A \cap B$ is a cursor declared as inheriting from a difference cursor of one union cursor and one intersection cursor. Likewise one can retrieve the language of the difference of two languages concatenation and the language of a DFA within an editing distance from a specified word.
In Sect. 4.4.2 a so called multiple cursor is used to search a text for n pattern simultaneously. This multiple cursor can apply to any algorithm including for example n-ary set operations.

8 Efficiency Issues

Speed tests have been conducted to compare with classical recursive implementations speeds and it turned out that no time was waste because cursors are simple object with very basic abilities: their implementation does not require huge complicated piece of code and most of the time a simple optimal and straightforward solution does the trick. Underlying containers and data structures are few and basic too, these are mostly STL sequential containers, that is, optimized and solid software components.

9 Conclusion

An extended version of this document provides a rigorous detailed description of cursors behavior including time complexities and C++ signatures of methods [1]. Cursors have been successfully implemented in a search engine called word grep [2]. We have overcome most of the problems encountered in the first stage of ASTL development and encouraging results are leading us to consider extending them to stream-based inputs/outputs, cyclic automata and transducers.

References

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986, pp 157-165.

[DR92] D. Revuz. *Dictionnaires et Lexiques - Méthodes et Algorithmes.* PhD. thesis, université Paris VII, 1992, pp. 66-70.

[HU79] J. E. Hopcroft, J. D. Ullman. *Introduction to automata, languages and computation.* Addison-Wesley, 1979.

[MS89] D. R. Musser, A. Stepanov. *Generic Programming.* Lecture Notes in Computer Science 358, Springer-Verlag, 1989.

[SL95] A. Stepanov, M. Lee. *The Standard Template Library.* Hewlett-Packard laboratories, 1995.

[VLM98] Vincent Le Maout. *Tools to Implement Automata, a first step: ASTL.* Lecture Note in Computer Science 1436, Springer-Verlag, 1998, pp 104-108.

[1] ASTL and cursors v1.0 are freely available at `http://www-igm.univ-mlv.fr/~ lemaout`

[2] Enhanced regular expressions defined on words