Efficient Parsing Using Recursive Transition Networks with Output

Article · August 2009

DOI: 10.1007/978-3-642-04235-5_17 · Source: OAI

2 authors:

Javier Sastre
Taiger
8 PUBLICATIONS **20** CITATIONS
SEE PROFILE

Mikel L. Forcada
University of Alicante
150 PUBLICATIONS **1,175** CITATIONS
SEE PROFILE

Efficient parsing using recursive transition networks with output

Javier M. Sastre[*†], **Mikel L. Forcada**[†]

[*]Institut Gaspard-Monge, Université Paris Est,
F-77454 Marne-la-Vallée Cedex 2, France

[†]Grup Transducens, Departament de Llenguatges i Sistemes Informàtics, Universitat d'Alacant,
E-03071 Alacant, Spain.

Abstract

We describe here two efficient parsing algorithms for natural language texts based on an extension of recursive transition networks (RTN) called recursive transition networks with string output (RTNSO). RTNSO-based grammars may be semiautomatically built from samples of a manually built syntactic lexicon. Efficient parsing algorithms are needed to minimize the temporal cost associated to the size of the resulting networks. We focus our algorithms on the RTNSO formalism due to its simplicity which facilitates the manual construction and maintenance of RTNSO-based linguistic data as well as their exploitation.

1. Introduction

This paper describes two efficient parsing algorithms for natural language texts which use recursive transition networks (Woods, 1970) (RTNs) generated from a large and manually built syntactic lexicon such as the one defined in (Gross, 1975). Both algorithms are easily defined based on the formal definition of RTN with string output[1] (RTNSOs) given in section 2., which corresponds to a special type of pushdown letter transducer (a pushdown automaton with output), and the formal definition of the set of outputs (such as parses or translations) associated to an input string given in section 3. A first algorithm for the efficient computation of that set is given in section 4., paying special attention to null closure due to its crucial role in the algorithm. An explanation of how to modify the preceding algorithm in order to obtain a more complex but also more efficient Earley-like algorithm is given in section 5. Comparative results are given in section 6. Concluding remarks close the paper.

1.1. Parsing with RTNs and lexicon-grammar

Lexicon-grammar is a systematic method for the analysis and the representation of the elementary sentence structures of a natural language (Gross, 1996) which has produced, over the last 30 years, a large and fine-grain linguistic resource[2] describing syntactic and semantic properties of French simple sentences (among other languages) for 13375 simple verbs, 10325 predicative nouns (e.g.: *pride* as in *to be proud of*) and 39297 idiomatic expressions playing the role of predicative element (e.g.: *to wear the pants*). There exists a technique to semiautomatically build a RTN-based grammar from samples of the lexicon-grammar for automatic parsing (Roche, 1993). Although the RTN formalism is not the most expressive, its simplicity has permitted its practical application to the analysis of specialized domain corpora (Nakamura, 2004) with RTN-based computer tools (INTEX (Silberztein, 1993), Unitex (Paumier, 2006), Outilex (Blanc et al., 2006)). However, the temporal cost of applying a large RTN to a large corpus may be too high for certain applications. We expect to reduce it by employing efficient parsing algorithms.

[1]For instance, the result of parsing as a tagged input.

[2]Partially available through the HOOP interface (Sastre, 2006) at `http://hoop.univ-mlv.fr`

2. Recursive transition networks with string output

A non-deterministic RTNSO $R = (Q, \Sigma, \Gamma, \delta, Q_I, F)$ is made of a finite set of states $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ (which will also be used as the alphabet for the pushdown stack), a finite input alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$, a finite output alphabet $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_{|\Gamma|}\}$, $Q_I \subseteq Q$ the set of initial states, $F \subseteq Q$ the set of acceptance states, and a transition function

$$\delta : Q \times ((\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})) \to \mathcal{P}(Q \times (Q \cup \{\lambda\})) \quad (1)$$

where $\mathcal{P}(\cdot)$ represents the set of all subsets of a given set and $\lambda \in Q^*$ is an empty sequence of states. As to the possible types of transitions:

- Transitions of the form $\delta(q, (\sigma, \gamma))$ with $\sigma \in \Sigma$ and $\gamma \in \Gamma$ read one input symbol and write an output symbol; those of the form $\delta(q, (\sigma, \varepsilon))$ read a symbol but do not write anything, those of the form $\delta(q, (\varepsilon, \gamma))$ do not read an input symbol but write an output symbol, and those of the form $\delta(q, (\varepsilon, \varepsilon))$ neither read nor write symbols.

- Only transitions of the form $\delta(q, (\varepsilon, \varepsilon))$ can contain elements of $Q \times Q$ in their result set; that is, only transitions without input and output can push a state onto the stack. These transitions represent a *call*: in addition to performing a transition to a new state, they push a return state onto the stack which will be popped later: $(q_c, q_r) \in \delta(q_s, (\varepsilon, \varepsilon))$ represents a *subroutine jump* to state q_c which pushes the return state q_r onto the stack. The rest of the transitions can only return elements of $Q \times \{\lambda\}$, that is, they do not push anything onto the stack.

- It has to be noted that δ does not define any transitions which pop states from the stack. This is because states can be automatically popped from a nonempty stack everytime the RTNSO reaches an acceptance state in F; the popped state is reached without consuming any input or writing any output. The definition of popping transitions is implicit in the definition of R.

- As a general rule, loops consuming no input but generating output or making calls are not allowed: infinite length parses of natural-language input sequences make no sense and complicate the corresponding parsing algorithms which should avoid falling into infinite loops.

3. Language of translations of a string

In this section we will formally derive the computation of the set of output translations associated to an input string following (Garrido-Alenda et al., 2002).

During the application of a RTNSO, a triplet $(q, z, \pi) \in Q \times \Gamma^* \times Q^*$ represents the fact that a partial output (PO) z has been generated up to the point of reaching state q with a stack π. We call $V \in \mathcal{P}(Q \times \Gamma^* \times Q^*)$ a set of POs (SPO); we define Δ as the PO transition function over a SPO for an input symbol as

$$\Delta : \mathcal{P}(Q \times \Gamma^* \times Q^*) \times \Sigma \to \mathcal{P}(Q \times \Gamma^* \times Q^*) \quad (2)$$

$$\Delta(V, \sigma) = \{(q', zg, \pi) : (q', \lambda) \in \delta(q, (\sigma, g)) \wedge (q, z, \pi) \in V\}, \quad (3)$$

where $q, q', q_c, q_r \in Q$, $z \in \Gamma^*$ is an output string, $\pi \in Q^*$ a stack, $\sigma \in \Sigma$ an input symbol, and $g \in \Gamma \cup \{\varepsilon\}$ an output symbol or an empty output. We define the ε-closure $C_\varepsilon(V)$ of a SPO V,

$$C_\varepsilon : \mathcal{P}(Q \times \Gamma^* \times Q^*) \to \mathcal{P}(Q \times \Gamma^* \times Q^*), \quad (4)$$

as the smallest SPO containing V and every PO directly or indirectly derivable from V through ε-transitions, that is, through zero, one or more transitions without consuming any input symbol. Informally, elements are added to the ε-closure through three different kinds of ε-moves until no more elements are added:

- **Output without input:** adding (q', zg, π) for each (q, z, π) in the closure and for each q' and g such that $(q', \lambda) \in \delta(q, (\varepsilon, g))$;
- **Call or push:** adding $(q_c, z, \pi q_r)$ for each (q, z, π) in the closure and for each q_c and q_r such that $(q_c, q_r) \in \delta(q, (\varepsilon, \varepsilon))$;
- **Return or pop:** adding (q_r, z, π) for each $(q, z, \pi q_r)$ in the closure such that $q \in F$;

An efficient way to compute the ε-closure is described in section 4.

We recursively define

$$\Delta^* : \mathcal{P}(Q \times \Gamma^* \times Q^*) \times \Sigma^* \to \mathcal{P}(Q \times \Gamma^* \times Q^*), \quad (5)$$

the extension of Δ to strings in Σ^*, as follows:

$$\Delta^*(V, \varepsilon) = C_\varepsilon(V) \quad (6)$$

$$\Delta^*(V, z\sigma) = C_\varepsilon(\Delta(\Delta^*(V, z), \sigma)) \quad (7)$$

Finally, we define $\mathcal{T}(\sigma_1 \dots \sigma_l)$ the language of translations of input string $\sigma_1 \dots \sigma_l$ as the set of output strings of the final part of the SPO (having only final states and no pending return-from-call states) reachable from the initial SPO (having every initial state coupled with an empty output string and an empty return stack) through zero, one or more transitions and consuming the whole input string:

$$\mathcal{T}(\sigma_1 \dots \sigma_l) = \{z \in \Gamma^* : (q, z, \lambda) \in \Delta^*(Q_I \times \{\varepsilon\} \times \{\lambda\}, \sigma_1 \dots \sigma_l) \wedge q \in F\}. \quad (8)$$

4. Processing an input string

Based on the formal definition given in the previous section, we propose a breadth-first algorithm decomposed into algorithms 1 (*translate_string*), 2 (*translate_symbol*), and 3 (*closure*) to process an input string for a given (non-deterministic) RTNSO and to generate the set of corresponding output strings based on the equations above.

The algorithm keeps for each prefix of the input string a SPO V, whose elements are triplets of the form (q, z, π) where $q \in Q$ is a state, $z \in \Gamma^*$ is an output string and $\pi \in Q^*$ is a last-in-first-out stack holding states in Q (λ will be used to represent the empty stack), as described in the previous section. In the algorithms, the RTNSO $R = (Q, \Sigma, \Gamma, \delta, Q_I, F)$ is treated as a global variable.

Based on van Noord's *per subset* algorithm (van Noord, 2000), algorithm 3 is an efficient algorithm for the computation of the ε-closure of a given SPO V. To start it copies V into E; then it uses V to store the result of the computation and E to keep a trace of the partial outputs that have not been processed yet. V is iteratively incremented with the new reachable states from an arbitrary partial output of E by one ε-transition. Each time a partial output of E is retrieved ($(q, z, \pi) \leftarrow \mathrm{next}(E)$) to be processed, it is also removed from E. Each time a new partial output is added to V (**if** $(\mathrm{add}(V, (q, z, \pi)))$, it is also added to E ($\mathrm{insert}(E, (q, z, \pi))$) for further processing. The difference between $\mathrm{add}()$ and $\mathrm{insert}()$ is that the former performs a duplicity test before adding the PO to V and returns the boolean result of this test, and the latter adds the PO *blindly* to E and returns nothing.

```
Algorithm 1 translate_string($\sigma_1 \dots \sigma_l$)        ▷ $\mathcal{T}$, eq. (8)
Input: $\sigma_1\sigma_2 \dots \sigma_l$, an input string of length $l$
Output: $T$, the set of translations
 1: $V \leftarrow \mathrm{closure}(Q_I \times \{\varepsilon\} \times \{\lambda\})$        ▷ initial SPO $V_0$
 2: $i \leftarrow 0$
 3: while $V \neq \emptyset \wedge i < l$ do     ▷ $V_{i+1} = C_\varepsilon(\Delta(V_i, \sigma_{i+1}))$
 4:     $V \leftarrow \mathrm{closure}(\mathrm{translate\_symbol}(V, \sigma_{i+1}))$
 5:     $i \leftarrow i + 1$
 6: end while
 7: $T \leftarrow \emptyset$
 8: if $i = l$ then
 9:     for each $(q, z, \lambda) \in V : q \in F$ do
10:         $T \leftarrow T \cup \{z\}$
11:     end for
12: end if
```

As can be seen, the algorithm is quite simple and does not require a complex data structure. At each iteration it

Algorithm 2 translate_symbol(V,σ) ▷ $\Delta(V,\sigma)$, eq. (3)

Input: V, the SPO σ, the symbol
Output: W, the SPO after processing σ

```
1:  $W \leftarrow \emptyset$
2:  for each $(q,z,\pi) \in V$ do
3:      for each $(q',g) : (q',\lambda) \in \delta(q,(\sigma,g))$ do
4:          $W \leftarrow W \cup \{(q',zg,\pi)\}$
5:      end for
6:  end for
```

Algorithm 3 closure(V) ▷ $C_\varepsilon(V)$

Input: V, the SPO whose ε-closure is to be computed
Output: V after computing its ε-closure

```
1:  $E \leftarrow V$                                        ▷ unprocessed PO queue
2:  while $E \neq \emptyset$ do
3:      $(q,z,\pi) \leftarrow \mathrm{next}(E)$
4:      for each $(q',g) : (q',\lambda) \in \delta(q,(\varepsilon,g))$ do     ▷ $\varepsilon$-
5:          if $\mathrm{add}(V,(q',zg,\pi))$ then                  ▷ transitions
6:              $\mathrm{insert}(E,(q',zg,\pi))$
7:          end if
8:      end for
9:      for each $(q_c,q_r) \in \delta(q,(\varepsilon,\varepsilon))$ do
10:         if $\mathrm{add}(V,(q_c,z,\pi q_r))$ then              ▷ push-
11:             $\mathrm{insert}(E,(q_c,z,\pi q_r))$               ▷ transitions
12:         end if
13:     end for
14:     if $\pi = \pi' q_r \wedge q \in F \wedge \mathrm{add}(V,(q_c,z,\pi q_r))$ then
15:         $\mathrm{insert}(E,(q_r,z,\pi'))$                      ▷ pop-transition
16:     end if
17: end while
```

computes the SPO that can be derived from the precedent one by recognizing the next input symbol, so it suffices to store two SPOs at a time. It firstly fills in the next SPO with the POs directly reachable through one transition consuming the next input symbol, then it completes the SPO with the following reachable POs through one or more ε-transitions, call and return transitions included. Two important limitations of the algorithm are: (a) the impossibility of parsing with RTNs representing left-recursive grammars since the computation of the ε-closure would try to generate a PO with an infinite stack of return states, and (b) the fact that for a SPO V_i having two POs with different stacks and/or outputs from where a call to a same state q_c is performed, the computation of every PO derived from the call to q_c is performed twice although it could be factored (for instance, for the RTNSO of Fig. 1 call to state q_0 is computed $\sum 2^{n+1}$ times for a string $a^n b^n$ when it could be computed $n+1$ times by factoring out common calls). In the next section we show how to modify this algorithm to avoid both limitations.

5. Earley-like RTNSO processing

Finite-state automata can give a compact representation of a set of sequences by factoring out common prefixes and suffixes. RTNs can also factor infixes by defining only a subautomaton for each repeated set of infixes and by using transitions calling the initial state of the corresponding subautomaton each time any infix in the set is to be recognized.

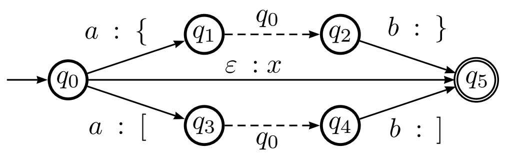


Figure 1: Example of RTNSO. Labels of the form $x : y$ represent an input : output pair (e.g.: $a : \{$ for transition ($\delta(q_0, a, \{) = (q_1, \lambda)$). Dashed transitions represent a call to the state specified by the label (e.g.: transition ($\delta(q_1, \varepsilon, \varepsilon) = (q_2, q_0)$).

However, it is up to the parsing algorithm to detect that the same calling transition is made multiple times at an input point so the called subautomaton is processed only once. Based on Earley's context-free grammar parsing algorithm (Earley, 1970)[3], we show here a modified and more efficient version of the algorithm in section 4. which is able to process left-recursive RTNSOs, and which factores out the computation of infix calls by parallel analyses, as for the RTNSO of Fig. 1.

We exchange the use of a return state stack for a more complex representation of POs, which mainly involves a modification of the ε-closure algorithm: when a call transition to a state q_c is found, a *paused* PO until the completion of the call is generated as well as a new PO starting the called subanalysis from q_c and the current input point, if not already started. Each time a subanalysis is completed, a *resumed* PO is generated for every concatenation of the *paused* POs depending on it and the substring generated by the subanalysis.

5.1. Language of translations through Earley-like processing

POs are represented as 5-tuples $(q, z, q_c, q_h, i) \in (Q \times \Gamma^* \times (Q \cup \lambda) \times Q \times \mathbb{N})$, where q is the current state, q_h the state that initiated this subanalysis, i the input position where this subanalysis began (i representing the point between σ_i and σ_{i+1}, so 0 is the point before the first input symbol), z the output generated from state q_h and input position i up to state q and the current input position, and q_c the start state of the subanalysis which this PO depends on (λ if no subanalysis completion is required, that is, for POs which are not paused but active).

We extend the PO transition function Δ of eq. (3), which corresponds to Earley's "scanner", as follows:

$$\Delta : \mathcal{P}(Q \times \Gamma^* \times (Q \cup \{\lambda\}) \times Q \times \mathbb{N}) \times \Sigma \to \mathcal{P}(Q \times \Gamma^* \times (Q \cup \{\lambda\}) \times Q \times \mathbb{N}) \quad (9)$$

$$\Delta(V,\sigma) = \{(q', zg, \lambda, q_h, i) : (q',\lambda) \in \delta(q,(\sigma,g)) \wedge (q,z,\lambda,q_h,i) \in V\} \quad (10)$$

Note that the function does not apply on POs depending on a subanalysis: they stay paused until the completion of the subanalysis.

[3](Woods, 1970) mentions an adaptation of Earley's algorithm for RTNs in (Woods, 1969). However we have not been able to obtain the latter paper.

Let $i, j, k \in \mathbb{N}$ such that $i \leq j \leq k$, we redefine the ε-moves of the ε-closure in section 3. for 5-tuples as follows:

- **Output without input:** analogously to the preceding version, we add $(q', zg, \lambda, q_h, j)$ for each (q, z, λ, q_h, j) in the closure of V_k and for each q' and g such that $(q', \lambda) \in \delta(q, (\varepsilon, g))$;

- **Call or push:** analogously to Earley's "predictor", we add (q_r, z, q_c, q_h, j) and $(q_c, \varepsilon, \lambda, q_c, k)$ for each (q, z, λ, q_h, j) in the closure of V_k and for each q_c and q_r such that $(q_c, q_r) \in \delta(q, (\varepsilon, \varepsilon))$; (q_r, z, q_c, q_h, j) is the new paused PO depending on the subanalysis started by the new PO $(q_c, \varepsilon, \lambda, q_c, k)$;

- **Return or pop:** analogously to Earley's "completer", for each (q, z, λ, q_c, j) such that $q \in F$ (the completed POs) and for each $(q', z', q_c, q_h, i) \in V_j$ (the paused POs depending on the completed one) we add $(q', z'z, \lambda, q_h, i)$ to the closure of V_k (we resume them with the concatenation of the paused PO and the completed PO);

The extension of Δ to strings in Σ^* (eqs. 6 & 7) remains unchanged except for the use of the newly defined Δ and ε-closure.

Finally, the language of translations $\mathcal{T}(\sigma_1 \ldots \sigma_l)$ stays the same except for the adaptation to the new SPO representation and the use of the newly defined functions:

$$\mathcal{T}(\sigma_1 \ldots \sigma_l) = \{z \in \Gamma^* : (q', z, \lambda, q_h, 0) \in \Delta^*(\{(q, \varepsilon, \lambda, q, 0) : q \in Q_I\}, \sigma_1 \ldots \sigma_l) \wedge q' \in F \wedge q_h \in Q_I\}, \quad (11)$$

that is, the outputs of the POs completing ($q' \in F$) an initial state ($q_h \in Q_I$) and not depending on a subanalysis, which are indirectly derived from an initial PO (external calls to the initial states of the RTNSO).

The adaptation of algorithms 1 (*translate_string*) and 2 (*translate_symbol*) is trivial and will not be given here. Algorithm 4 replaces algorithm 3 and is an almost straighforward implementation of the ε-closure for Earley-like processing. As stated in (Earley, 1970), ε-transitions may lead to analyses which may partially reject correct parses. Notice that if a subanalysis starting at an SPO V_i is completed without input consumption (ε-completion), the paused POs to be resumed will belong to the same SPO V_i. If a paused PO depending on the same subanalysis is generated in the same SPO V_i after the ε-completion of the subanalysis, this paused PO will not be resumed by the precedent ε-completion and thus every derived parse will not be returned. Algorithm 4 creates an initially empty set T of ε-translations which it fills with the pairs (q_h, z) extracted from every ε-completed subanalysis (q, z, λ, q_h, j). Each time a paused PO (q_r, z, q_c, q_h, j) is generated, it firstly verifies if call to q_c was already performed and if it led to a ε-completion (there exists at least a ε-translation for state q_c). If so, the paused PO is immediately resumed for each ε-translation in T; if not, call to state q_c is normally performed.

6. Empirical tests

Both algorithms have been programmed using C++ and STL and execution times measured for the RTNSO of Fig. 1 in a Linux Debian platform with a 2.0 GHz Pentium IV Centrino processor and a 2 GB RAM (see Fig. 2(a)). As expected, our first algorithm has an exponential cost even for an acceptor version, that is, without output generation. Although Earley's algorithm cost is n^3, our Earley-based algorithm has also an exponential cost. A pure Earley algorithm (without output generation) would have a linear cost for our example grammar (see Fig. 2(b)). An exponential number of steps are saved due to the factoring of an exponential number of state calls; however, when we resume a set of paused POs with every completion of their common subanalysis, we are computing a cartesian product of the concatenations of every output of every PO with every output of every subanalysis completion. Earley's base algorithm is only a recognizer; it can be easily modified in order to compute the set of parses, but if this set grows exponentially w.r.t. the input length the cost cannot be kept to polynomial time.

We have implemented modified versions of both algorithms which use pointers or handles to the nodes of a trie in order to represent sequences. This allows PO comparisons to avoid expensive string comparisons. During the traversal of a consuming transition, the transition output is efficiently appended to the string since its handle points to the last sequence symbol. Our first algorithm experiences an exponential speed up with the use of tries, but our Earley-based algorithm shows an even worse temporal cost. This is because the completion of a subanalysis involves the concatenation of two sequences, an operation where a trie structure is not so helpful.

7. Conclusion

We have first given here an efficient and simple parsing algorithm for natural language texts with RTNSOs with two limitations: left-recursive RTNSOs cannot be processed and infix calls are not factored. Based on Earley's parsing algorithm for context-free grammars, we have shown how to modify the precedent algorithm in order to suppress both limitations. Finally, we have given some comparative results which show that output generation, rather than being a simple issue, complicates algorithms up to obtaining exponential time costs in spite of the polynomial time of the acceptor-only algorithms.

We are currently studying the use of an RTN-like structure to efficiently build and store the resulting set of translations, to avoid an exponential complexity.

Acknowledgements: This work has been partially supported by the Spanish Government (grant number TIN2006–15071–C03–01), by the Universitat d'Alacant (grant number INV05-40), by the MENRT and by the CNRS. We thank Pr. E. Laporte for his useful comments.

8. References

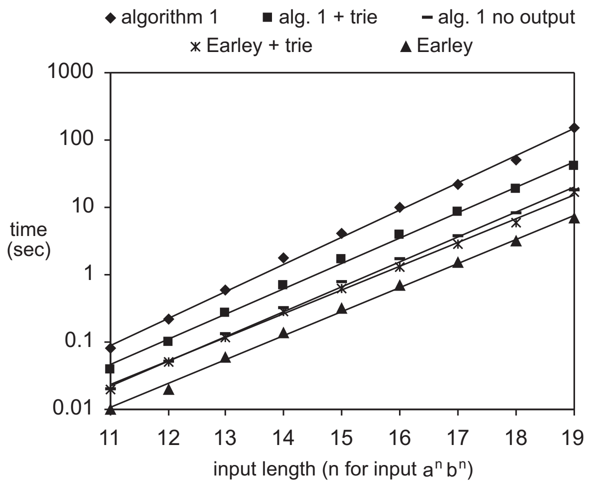
Blanc, Olivier, Matthieu Constant, and Éric Laporte, 2006. Outilex, plate-forme logicielle de traitement de textes écrits. In *Proceedings of TALN'06*. Leuven, Belgium: UCL Presses universitaires de Louvain.

Algorithm 4 closure(V, k) ▷ $C_\varepsilon(V_k)$

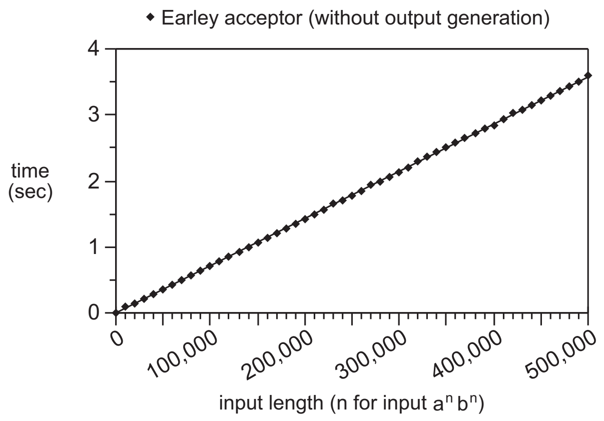
Input: V, the SPO whose ε-closure is to be computed; k, the current input position

Output: V after computing its ε-closure

```
1:  $T \leftarrow \emptyset$                                         ▷ $\varepsilon$-completion set
2:  $E \leftarrow V$                                         ▷ unprocessed PO queue
3:  while $E \neq \emptyset$ do
4:      $(q, z, \lambda, q_h, j) \leftarrow \mathrm{next}(E)$
5:      for each $(q', g) : (q', \lambda) \in \delta(q, (\varepsilon, g))$ do          ▷ $\varepsilon$-
6:          if $\mathrm{add}(V, (q', zg, \lambda, q_h, j))$ then          ▷ TRANS.
7:              $\mathrm{insert}(E, (q', zg, \lambda, q_h, j))$
8:          end if
9:      end for
10:     for each $(q_c, q_r) \in \delta(q, (\varepsilon, \varepsilon))$ do          ▷ PREDICT.
11:         if $\mathrm{add}(V, (q_r, z, q_c, q_h, j))$ then          ▷ paused
12:             if $\{(g : (q_c, g) \in T\} \neq \emptyset$ then          ▷ $\varepsilon$-com-
13:                 for each $\{g : (q_c, g) \in T\}$ do          ▷ plet.
14:                     if $\mathrm{add}(V, (q_r, zg, \lambda, q_h, j))$ then
15:                         $\mathrm{insert}(E, (q_r, zg, \lambda, q_h, j))$
16:                     end if
17:                 end for
18:             else if $\mathrm{add}(V, (q_c, \varepsilon, \lambda, q_c, k))$ then
19:                 $\mathrm{insert}(E, (q_c, \varepsilon, \lambda, q_c, k))$          ▷ sub-
20:             end if          ▷ analysis
21:         end if
22:     end for
23:     if $q \in F$ then          ▷ COMPLETER
24:         for each $(q', z', q_h, q'_h, i) \in V_j$ do          ▷ paused
25:             if $i = k$ then          ▷ register $\varepsilon$-completion
26:                 $T = T \cup \{(q_h, z)\}$
27:             end if
28:             if $\mathrm{add}(V, (q', z'z, \lambda, q'_h, i))$ then          ▷ resu-
29:                 $\mathrm{insert}(E, (q', z'z, \lambda, q'_h, i))$          ▷ med
30:             end if
31:         end for
32:     end if
33: end while
```



(a)



(b)

Figure 2: Comparative graphics of execution times (in seconds) w.r.t. input length (n) of the different algorithms for the RTNSO of Fig. 1 and input $a^n b^n$. Notice the logarithmic scale of the vertical axis of graphic 2(a): all of its functions are exponential.

Earley, Jay, 1970. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

Garrido-Alenda, Alicia, Mikel L. Forcada, and Rafael C. Carrasco, 2002. Incremental construction and maintenance of morphological analysers based on augmented letter transducers. In *Proceedings of TMI 2002 (Theoretical and Methodological Issues in Machine Translation, Keihanna/Kyoto, Japan, March 2002)*.

Gross, Maurice, 1975. *Méthodes en Syntaxe*. Paris: Hermann.

Gross, Maurice, 1996. *Lexicon Grammar*. Cambridge: Pergamon Press, pages 244–258.

Nakamura, Takuya, 2004. Analyse automatique d'un discours spécialisé au moyen de grammaires locales. In Fairon C. et Dister A. Purnelle G. (ed.), *JADT 2004, International Conference on the Statistical Analysis of Textual Data*. Louvain-la-Neuve: UCL Presses universitaires de Louvain.

Paumier, Sébastien, 2006. *Unitex 1.2 User Manual*. Université de Marne-la-Vallée. `http://www-igm.univ-mlv.fr/~unitex/UnitexManual.pdf`.

Roche, Emmanuel, 1993. *Analyse syntaxique transformationnelle du français par transducteurs et lexique-grammaire*. Ph.D. thesis, Université Paris 7, Paris.

Sastre, Javier M., 2006. HOOP: a Hyper-Object Oriented Platform for the management of linguistic databases. Presentation in Lexis and Grammar Conference, Palermo, Italy, September 6-9. Abstract available at `http://www-igm.univ-mlv.fr/~sastre/publications/sastre06b.zip`.

Silberztein, Max D., 1993. *Dictionnaires électroniques et analyse automatique de textes. Le système INTEX*. Paris: Masson.

van Noord, Gertjan, 2000. Treatment of epsilon moves in subset construction. *Comput. Linguist.*, 26(1):61–76.

Woods, William A., 1969. Augmented transition networks for natural language analysis. Rep. CS-1, Comput. Lab., Harvard U., Cambridge, Mass.

Woods, William A., 1970. Transition network grammars for natural language analysis. *Commun. ACM*, 13(10):591–606.