AN EFFICIENT *LALR*(1) AND *LR*(1) LOOKAHEAD SET ALGORITHM

PAULO S. L. M. BARRETO[†]

Abstract. The most popular method for computing *LALR*(1) lookaheads is not the most efficient one, mainly (as we conjecture) due to certain deficiencies in the description of core algorithms. Similarly, full *LR*(1) parser generators are not as common as their *LALR*(1) counterparts, despite their usefulness regarding conventional semantic routine attachment and attribute grammar compilation. We address these two issues by describing an efficient algorithm to compute either *LALR*(1) or *LR*(1) lookahead sets without requiring any modification of the underlying transitive closure algorithm.

1. Introduction

Most compiler building tools employ *LL*(1) or *LALR*(1) parser generation algorithms. Modern methods for generating *LALR*(1) lookahead sets are almost invariably based on computing the transitive closure of certain relations and associated set-valued functions, a problem for which efficient algorithms are known, notably the one by Eve and Kurki-Suonio [5].

Among these methods, the most popular is undoubtedly DeRemer and Pennello's [4], although the algorithm by Park, Choe and Chang [14] (henceforth called simply PCC) is arguably more efficient, and another by Ives [8] is claimed to be even more so. Recent research into the subject includes the method of Anzai [1].

A natural question to ask is why the more efficient algorithms are not as popular as DeRemer and Pennello's. We conjecture that the main reasons are:

- *Clarity of presentation.* DeRemer and Pennello provide a very clear and complete exposition of the problem, the theory underlying the proposed solution, and the implementation steps to be followed. On the other hand, the description of PCC is heavily theoretical, and Ives's paper is very brief and somewhat sketchy.
- *Lack of an efficient* Path/ctx *algorithm.* Both PCC and Ives define and use a new function between nonterminals, called Path by PCC and ctx by Ives, that associates certain sets of terminals to pairs of nonterminals. The concept of Path/ctx is essential in their methods for lookahead computation. However, Ives does not provide any algorithm for computation of ctx, and the method employed by PCC for Path is only superficially described but even so requires a nontrivial change in Eve and Kurki-Suonio's closure algorithm; besides, no formal proof of correctness is given for the modified algorithm. To be sure, the definition of ctx is slightly different from that of Path, as it includes a special symbol Λ in $\mathrm{ctx}(A, B)$ to flag that B figures in a right derivation from A in any number of steps; this is, however, unnecessary, as the unit derivation relationship can be indicated explicitly.

Key words and phrases. Compilers, Parser generator, Lookahead set algorithm.

[†]The author was supported by the Brazilian National Council for Scientific and Technological Development (CNPq) under research productivity grant 312005/2006-7 and universal grant 485317/2007-9.

We also notice that, in many circumstances, it might be convenient to resort to the full power of *LR*(1) parsers. A typical situation is semantic routine invocation, which usually takes place at nonterminal reductions. If a semantic routine must be activated somewhere within a production (not at its end), the usually adopted strategy is to introduce into the grammar a new nonterminal that only generates the empty string and serves exclusively as placeholder for the semantic routine call. This technique may be implicitly applied to implement certain classes of attribute grammars like the *ECLR* class described in [15]. Unfortunately, this procedure may destroy the *LALR*(k) property for any k, a feature clearly undesirable in compiler construction tools [2]. The use of full *LR*(1) items in parser automaton states may also be attractive to remove parser conflicts arising in the compilation of regular right-part grammars [12]. Syntax error repair methods like that of [11] also benefit from finer *LR*(1) lookahead sets.

Although the canonical *LR*(1) construction [10] may be too expensive for practical applications due to the large number of states in the generated parser, an efficient algorithm was proposed in [13] and refined in [6] and [16] to build equivalent but state-minimized *LR*(1) automata. These methods are based on iterative generation and merging of compatible *LR*(1) states, and are still inefficient in the sense that the transitive closure of a state (the *completer* operation) must be computed repeatedly.

Our contributions in this paper are the following. We address the *LALR*(1) issues by describing a new algorithm for Path/ctx computation that does not require any changes in the underlying transitive closure algorithm; rather, we simply construct a different relation graph and show that the new construction is formally correct. Besides, using a lemma by Park, Choe and Chang [14, lemma 3.1], we show how to use the Path/ctx function to eliminate the need for the repeated *completer* computation, thus removing this inefficiency from the full *LR*(1) generation algorithm.

The remainder of this paper is organized as follows. We define the adopted notation in section 2. On section 3 we briefly review the construction of the minimal *LR*(1) automaton. Section 4 describes the proposed lookahead set algorithm and proves its correctness. On section 5 we report on the results obtained from the proposed algorithm. We conclude in section 6.

2. Notation

Throughout this paper we adopt the following notation.

- Σ is the set of *terminal symbols* and $\mathcal{N}$ is the set of *non-terminal symbols*, with $\Sigma \cap \mathcal{N} = \varnothing$. The *vocabulary* is the set $\mathcal{V} = \Sigma \cup \mathcal{N}$.
- $\mathcal{P}$ is the set of *productions*, which have the form $A \to \alpha$ where $A \in \mathcal{N}$ and $\alpha \in \mathcal{V}^*$.
- $A \underset{\text{rm}}{\longrightarrow} \alpha$ denotes a right derivation of $\alpha \in \mathcal{V}^*$ from $A \in \mathcal{N}$ in one step, and $A \overset{*}{\underset{\text{rm}}{\longrightarrow}} \alpha$ denotes a right derivation in any number of steps. We also use the shorthand $A \overset{*}{\underset{\text{rm}}{\longrightarrow}} B\ldots$, meaning $\exists w \in \Sigma^* : A \overset{*}{\underset{\text{rm}}{\longrightarrow}} Bw$.
- $\forall \alpha \in \mathcal{V}^* : \textsc{first}(\alpha) \equiv \{t \in \Sigma \mid \alpha \overset{*}{\longrightarrow} t\beta, \beta \in \mathcal{V}^*\}$.
- $\forall \rho, \sigma \subseteq \Sigma^* : \rho \oplus \sigma \equiv \textsc{first}(rs) : r \in \rho \wedge s \in \sigma$. That is, $\rho \oplus \sigma$ is simply $\textsc{first}(\rho)$ if $\varepsilon \notin \rho$, or $\textsc{first}(\rho) \cup \textsc{first}(\sigma)$ otherwise (the $\oplus$ operation is called *ε-free* first).
- $\forall A, B \in \mathcal{N} : \text{ctx}(A, B) \equiv \{t \in \Sigma \mid A \overset{*}{\underset{\text{rm}}{\longrightarrow}} Bt\alpha, \alpha \in \mathcal{V}^*\}$. Our notation is only slightly different from that of [8], where $\text{ctx}(A, B)$ contains ε if $A \overset{*}{\underset{\text{rm}}{\longrightarrow}} B$; we prefer to indicate this fact explicitly when needed, and to extend the semantics of $\oplus$ so

that:

$$\mathrm{ctx}(A,B) \oplus \sigma = \begin{cases} \mathrm{ctx}(A,B) \cup \sigma, & \text{if } A \xrightarrow[\mathrm{rm}]{*} B; \\ \mathrm{ctx}(A,B), & \text{otherwise.} \end{cases}$$

3. Construction of the Minimal *LR*(1) Automaton

The first and still the most efficient method to construct the *LR*(1) automaton with minimal number of states for general *LR*(1) grammars was described in [13]. It is an iterative, fixed-point algorithm that partitions the generated states into *visited* and *unvisited* states, starting with a single unvisited state (corresponding to the grammar's start symbol) and proceeding while any unvisited states remain. At each step, an element is removed from the set of unvisited states, its successors are computed, new successor states are marked as unvisited, and the removed state is marked as visited.

The efficiency of this algorithm derives from the possibility of merging distinct states that satisfy certain compatibility criteria; in particular, one can show that one of the criteria described in [6] (called *strong compatibility*) produces the automaton with the minimum possible number of states for the given grammar.

A central aspect of the algorithm is thus the computation of successors of an *LR*(1) state. For each item $[A \to \alpha \cdot X\gamma, u]$ of the state being processed, one must insert into the successor state not only the item $[A \to \alpha X \cdot \gamma, u]$ but also, if X is a nonterminal, all items of form $[N \to Y \cdot \beta, v]$, for each production $N \to Y\beta$ of each nonterminal N such that $X \xrightarrow[\mathrm{rm}]{*} N \ldots$. The lookahead v of one item of this form is given, according to lemma 3.1 of [14], by $v = \mathrm{ctx}(X, N) \oplus \textsc{first}(\gamma u)$.

4. Efficient Computation of ctx

Our algorithm for the efficient computation of $\mathrm{ctx}(A, B)$ is based on the following observations:

Lemma 1.

$$\forall (A \to C\alpha) \in \mathcal{P}, \forall B \in \mathcal{N} : \mathrm{ctx}(C,B) \oplus \textsc{first}(\alpha) \subseteq \mathrm{ctx}(A,B).$$

Proof. Suppose initially that $C \xrightarrow[\mathrm{rm}]{*} Bz$, where $z \in \Sigma^+$ so that $\textsc{first}(z) \subseteq \mathrm{ctx}(C,b)$. But $A \xrightarrow[\mathrm{rm}]{} C\alpha \xrightarrow[\mathrm{rm}]{*} Cw \xrightarrow[\mathrm{rm}]{*} Bzw$, hence $\textsc{first}(z) \subseteq \mathrm{ctx}(A,B)$. Therefore, $\mathrm{ctx}(C,B) = \bigcup_{C \xrightarrow[\mathrm{rm}]{*} Bz} \textsc{first}(z) \subseteq \mathrm{ctx}(A,B)$.

Now suppose that $C \xrightarrow[\mathrm{rm}]{*} B$. In this case, $A \xrightarrow[\mathrm{rm}]{} C\alpha \xrightarrow[\mathrm{rm}]{*} Cw \xrightarrow[\mathrm{rm}]{*} Bw$, where $\alpha \xrightarrow[\mathrm{rm}]{*} w$ e $w \in \Sigma^*$. Thus, $\textsc{first}(\alpha) = \textsc{first}(w) \subseteq \mathrm{ctx}(A,B)$. Therefore, $\mathrm{ctx}(C,B) \oplus \textsc{first}(\alpha) \subseteq \mathrm{ctx}(A,B)$. □

Lemma 2.

$$\mathrm{ctx}(A,B) = \bigcup_{\substack{(A \to C\alpha) \in \mathcal{P} \\ C \xrightarrow[\mathrm{rm}]{*} B\ldots}} \mathrm{ctx}(C,B) \oplus \textsc{first}(\alpha).$$

Proof. It suffices to show that

$$\mathrm{ctx}(A,B) \subseteq \bigcup_{\substack{(A \to C\alpha) \in \mathcal{P} \\ C \xrightarrow[\mathrm{rm}]{*} B\ldots}} \mathrm{ctx}(C,B) \oplus \textsc{first}(\alpha),$$

since the reverse inclusion is a corollary of the previous lemma. We have:

$$\mathrm{ctx}(C, B) \oplus \textsc{first}(\alpha) = \begin{cases} \mathrm{ctx}(C, B) \cup \textsc{first}(\alpha), & \text{if } C \xrightarrow[\text{rm}]{*} B \\ \mathrm{ctx}(C, B), & \text{otherwise.} \end{cases}$$

Therefore,

$$\bigcup_{\substack{(A \to C\alpha) \in \mathcal{P} \\ C \xrightarrow[\text{rm}]{*} B \ldots}} \mathrm{ctx}(C, B) \oplus \textsc{first}(\alpha) \quad = \quad \bigcup_{\substack{(A \to C\alpha) \in \mathcal{P} \\ C \xrightarrow[\text{rm}]{*} B \ldots \\ C \neq A}} \mathrm{ctx}(C, B) \quad \cup \quad \underbrace{\mathrm{ctx}(A, B)}_{\text{if} (A \to A\alpha) \in \mathcal{P}} \quad \cup \quad \bigcup_{\substack{(A \to C\alpha) \in \mathcal{P} \\ C \xrightarrow[\text{rm}]{*} B}} \textsc{first}(\alpha).$$

□

For the main theorem, we need the following definitions:

Definition 1. (A, B) inherits_ctx (C, B) *if, and only if,* $(A \to C\alpha) \in \mathcal{P}$*,* $C \neq A$*, and* B *is a nonterminal such that* $C \xrightarrow[\text{rm}]{*} B \ldots$.

Definition 2. $\mathrm{ctx}_0(A, B) = \bigcup\{\textsc{first}(\alpha) \mid (A \to C\alpha) \in \mathcal{P},\ C \xrightarrow[\text{rm}]{*} B\}$.

Theorem 1. *In a grammar without useless productions or inaccessible symbols,* $\mathrm{ctx}(A, B) = \mathrm{ctx}_0(A, B) \cup \bigcup\{\mathrm{ctx}(C, B) \mid (A, B)$ inherits_ctx $(C, B)\}$.

Proof. We first prove the reverse inclusion:

$$\begin{aligned} & \forall (A \to C\alpha) \in \mathcal{P} \text{ s. t. } C \xrightarrow[\text{rm}]{*} B, \forall w \in \Sigma^* \text{ s. t. } \alpha \xrightarrow[\text{rm}]{*} w : \\ & A \to C\alpha \xrightarrow[\text{rm}]{*} Cw \xrightarrow[\text{rm}]{*} Bw \\ \Rightarrow\quad & \textsc{first}(w) \subseteq \mathrm{ctx}(A, B) \\ \Rightarrow\quad & \textsc{first}(\alpha) = \bigcup\{\textsc{first}(w) \mid \alpha \xrightarrow[\text{rm}]{*} w\} \subseteq \mathrm{ctx}(A, B) \\ \Rightarrow\quad & \mathrm{ctx}_0(A, B) = \bigcup\{\textsc{first}(\alpha) \mid (A \to C\alpha) \in \mathcal{P},\ C \xrightarrow[\text{rm}]{*} B\} \subseteq \mathrm{ctx}(A, B). \end{aligned}$$

Furthermore, we have:

$$\begin{aligned} & \forall (A \to C\alpha) \in \mathcal{P}, \forall t \in \mathrm{ctx}(C, B), \forall w \in \Sigma^* \text{ s. t. } \alpha \xrightarrow[\text{rm}]{*} w, \\ & \exists v \in \Sigma^* : \\ & C \xrightarrow[\text{rm}]{*} Btv \\ \Rightarrow\quad & A \to C\alpha \xrightarrow[\text{rm}]{*} Cw \xrightarrow[\text{rm}]{*} Btvw \\ \Rightarrow\quad & t \in \mathrm{ctx}(A, B) \\ \Rightarrow\quad & \mathrm{ctx}(C, B) \subseteq \mathrm{ctx}(A, B) \\ \Rightarrow\quad & \bigcup\{\mathrm{ctx}(C, B) \mid (A \to C\alpha) \in \mathcal{P},\ C \xrightarrow[\text{rm}]{*} B \ldots\} \subseteq \mathrm{ctx}(A, B) \\ \Rightarrow\quad & \bigcup\{\mathrm{ctx}(C, B) \mid (A, B) \text{ inherits\_ctx } (C, B)\} \subseteq \mathrm{ctx}(A, B). \end{aligned}$$

The forward inclusion is a consequence of the following reasoning. If $\mathrm{ctx}(A, B) = \emptyset$ the inclusion is trivial. Suppose then that $\mathrm{ctx}(A, B) \neq \emptyset$, and let $t \in \mathrm{ctx}(A, B)$. By the definition of ctx, $\exists w \in \Sigma^* : A \xrightarrow[\mathrm{rm}]{*} Btw$. The first production applied in this derivation must begin with a nonterminal, given the presence of B as head of the sentential form. Let $A \to C\alpha$ be this first applied production, so that $\exists z \in \Sigma^* : A \to C\alpha \xrightarrow[\mathrm{rm}]{*} Cz \xrightarrow[\mathrm{rm}]{*} Btw$. There are two possibilities:

(1) $z = tw \Rightarrow C \xrightarrow[\mathrm{rm}]{*} B \wedge \alpha \xrightarrow[\mathrm{rm}]{*} tw$. Thus, $\exists (A \to C\alpha) \in (P) : C \xrightarrow[\mathrm{rm}]{*} B \wedge t \in \textsc{first}(\alpha)$, hence $t \in \mathrm{ctx}_0(A, B)$.

(2) $z \neq tw \Rightarrow \exists v \in \Sigma^* : w = vz$ (because z is a suffix of tw). Thus, $Cz \xrightarrow[\mathrm{rm}]{*} Btvz$, that is, $\exists (A \to C\alpha) \in \mathcal{P} : C \xrightarrow[\mathrm{rm}]{*} B\ldots \wedge t \in \mathrm{ctx}(C, B)$, hence $t \in \bigcup\{\mathrm{ctx}(C, B) \mid (A, B)\ \mathsf{inherits\_ctx}\ (C, B)\}$.

Remark: there must exist (except possibly if $A = B$) some production $A \to C\alpha$, $C \neq A$; otherwise, $A \xrightarrow[\mathrm{rm}]{*\!\!\!\!/} B\ldots$. In this case, even if the first production applied in the derivation $A \xrightarrow[\mathrm{rm}]{*} Btw$ is of form $A \to A\beta$, it is necessary that $A \to A\beta \xrightarrow[\mathrm{rm}]{*} Au \to C\alpha u \xrightarrow[\mathrm{rm}]{*} Cyu$, which reduces to the above case with $z = yu$. If $A = B \wedge \nexists C : (A \to C\alpha) \in \mathcal{P}$, $C \neq A$, then all productions of the derivation $A \xrightarrow[\mathrm{rm}]{*} Atw$ are of form $A \to A\beta$. In this case, $\forall t \in \mathrm{ctx}(A, A), \exists (A \to A\beta) \in \mathcal{P} : t \in \textsc{first}(\beta)$, and thus $t \in \mathrm{ctx}_0(A, B)$ since $A \xrightarrow[\mathrm{rm}]{*} A$ trivially. □

This theorem makes it possible to apply, *without any modification*, any transitive closure algorithm (but particularly that of [5]) to the relation inherits_ctx, so as to compute the values of $\mathrm{ctx}(A, B)$. These ideas are captured in Algorithm 1, which computes lookahead sets $u \subseteq \Sigma$ for *LR*(1) items of form $[A \to \alpha \cdot \beta, u]$, where $A \in \mathcal{N}$ and $(A \to \alpha\beta) \in \mathcal{P}$. Alternatively, one can use the computed $\mathrm{ctx}(A, B)$ values in an *LALR*(1) parser generator to achieve the efficiency gains reported in [14] and [8] over the results in [4].

5. Experimental results

We have implemented a full *LR*(1) parser generator in Java using our lookahead set algorithm. The generated parsers are table-driven and use a suitable interpreter also written in Java. The parser tables are optimized using comb vector techniques to handle *LR*(0) reduce states and default reductions in *LR*(1) states. No attempt was made to accommodate ambiguous grammars (like the artificial precedence and associativity rules implemented in YACC).

The parser of the generator itself was initially handwritten and then bootstrapped from its own grammar. The generated bytecode is about the same size (11 KiB) as the handwritten recursive-descent parser, but the source code was reduced from 23 KiB to 13 KiB due to the use of tables. On a 2.3 GHz AMD Turion™ 64 X2 platform where the clock granularity is about 16 ms, the bootstrap generation time is 96 ms, with 32 ms corresponding to the construction of parser automaton. Applying the generator to a Java grammar on the same platform, the generation time becomes roughly 480 ms with 176 ms devoted to the parser automaton. Similar experiments with an Oberon grammar result in generation times around 208 ms with 64 ms devoted to the parser automaton. Small grammars (including arithmetic expressions and some designed to illustrate special features like those in [2]) usually take the minimum measurable time of 16 ms for parser automation generation.

Algorithm 1 Computation of $LR(1)$ lookahead sets

for every nonterminal C **do**
  Compute all nonterminals A satisfying $C \xrightarrow[\text{rm}]{*} A \ldots$
  Compute all nonterminals B satisfying $C \xrightarrow[\text{rm}]{*} B$
end for
▷ *Build relation* inherits_ctx *between pairs of nonterminals:*
for every production $(A \to C\alpha) \in \mathcal{P}$ such that $C \neq A$ **do**
  for every nonterminal B such that $C \xrightarrow[\text{rm}]{*} B \ldots$ **do**
    Set the relation (A, B) inherits_ctx (C, B)
  end for
end for
▷ *Compute* $\text{ctx}_0(A, B)$*:*
Set $\text{ctx}_0(A, B) \leftarrow \varnothing$
for all pairs (A, B) present in the inherits_ctx relation **do**
  for all α such that $(A \to C\alpha) \in \mathcal{P}$ and $C \xrightarrow[\text{rm}]{*} B$ **do**
    Union $\textsc{first}(\alpha)$ into $\text{ctx}_0(A, B)$
  end for
end for
Compute $\text{ctx}(A, B)$ from $\text{ctx}_0(A, B)$ by taking the transitive closure of inherits_ctx
▷ *Compute the lookahead sets on demand during parser generation:*
for each item $[A \to \alpha \cdot X\gamma, u]$ in the state being generated where X is a nonterminal **do**
  for each production $N \to Y\beta$ of each nonterminal N such that $X \xrightarrow[\text{rm}]{*} N \ldots$ **do**
    Let $v \leftarrow \text{ctx}(X, N) \oplus \textsc{first}(\gamma u)$
    Merge into the successor state all items of form $[N \to Y \cdot \beta, v]$
  end for
end for

6. Conclusion

We have presented an efficient algorithm for $LALR(1)$ and $LR(1)$ lookahead computation, conceptually simpler than other existing algorithms and not requiring any alteration in conventional transitive closure algorithms. We have exhibited a formal proof of validity of our method and observed its effectiveness in practice.

Although the basic $LR(1)$ algorithm may be modified to eliminate unit reductions [7], it is still unclear how our proposed algorithm could be used in such a modified parser construction. It is also an open problem how to extend our algorithm to construct $LR(k)$ automata for $k > 1$. We conjecture that this is possible but non-trivial, in a similar fashion to the extension of the state merging algorithm itself [16]. A further line of follow-up research involves error recovery issues as pointed out by [3] and [9], and their interaction with $LR(1)$ lookahead set computation.

References

[1] H. Anzai. Almost boolean algebraic computation of $LALR(1)$ look-ahead set. *Journal of Information Processing*, 14(1):1–15, 1991.

[2] J. C. Beatty. On the relationship between the $LL(1)$ and $LR(1)$ grammars. *Journal of the ACM*, 29(4):1007–1022, 1982.

[3] R. Corchuelo, J. A. Pérez, A. Ruiz, and M. Toro. Repairing syntax errors in *LR* parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):698–710, 2002.
[4] F. DeRemer and T. Pennello. Efficient computation of *LALR*(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, 1982.
[5] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Informatica*, 8:303–314, 1977.
[6] S. Heilbrunner. A parsing automata approach to *LR* theory. *Theoretical Computer Science*, 15:117–157, 1981.
[7] S. Heilbrunner. Truly prefix-correct chain-free *LR*(1) parsers. *Acta Informatica*, 22:499–536, 1985.
[8] F. Ives. Unifying view of recent *LALR*(1) lookahead set algorithms. *SIGPLAN Notices*, 21(7):131–135, 1986.
[9] I.-S. Kim and K.-M. Choe. Error repair with validation in *LR*-based parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(4):451–471, 2001.
[10] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.
[11] B. J. McKenzie, C. Yeatman, and L. de Vere. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems*, 17(4):672–689, 1995.
[12] S. Morimoto and M. Sassa. Yet another generation of *LALR* parsers for regular right part grammars. *Acta Informatica*, 37(9):671–697, 2001.
[13] D. Pager. A practical general method for constructing *LR*(*k*) parsers. *Acta Informatica*, 7:249–268, 1977.
[14] J. C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of *LALR* formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, 1985.
[15] M. Sassa, H. Ishizuka, and I. Nakata. Rie, a compiler generator based on a one-pass-type attribute grammar. *Software - Practice and Experience*, 25(3):229–250, 1995.
[16] L. Schmitz. On the correct elimination of chain productions from *LR* parsers. *International Journal of Computer Mathematics*, 15:99–116, 1984.

This report contains unpublished results obtained by the author in 1988–1989.

Laboratório de Arquitetura e Redes de Computadores (LARC), Departamento de Engenharia de Computação e Sistemas Digitais (PCS), Escola Politécnica, Universidade de São Paulo, Brazil.
E-mail address: pbarreto@larc.usp.br