Construction of Efficient Generalized LR Parsers

Miguel A. Alonso[1], David Cabrero[2], and Manuel Vilares[1]

[1] Departamento de Computación
Facultad de Informática, Universidad de La Coruña
Campus de Elviña s/n, 15071 La Coruña, Spain
[2] Centro de Investigacións Lingüísticas e Literarias Ramón Piñeiro
Estrada Santiago-Noia km 3, A Barcia,
15896 Santiago de Compostela, Spain

Abstract. We show how LR parsers for the analysis of arbitrary context-free grammars can be derived from classical Earley's parsing algorithm. The result is a Generalized LR parsing algorithm working at complexity $\mathcal{O}(n^3)$ in the worst case, which is achieved by the use of dynamic programming to represent the non-deterministic evolution of the stack instead of graph-structured stack representations, as has often been the case in previous approaches. The algorithm behaves better in practical cases, achieving linear complexity on LR grammars. Experimental results show the performance of our proposal.

1 Introduction

LR parsing, one of the strongest and most efficient class of parsing strategies for context-free grammars (CFGs), is a two fold process: first, the grammar is compiled into a finite-state machine called *LR automaton*, which has two associated tables of actions and gotos [1]; second, a push-down automata with the LR automaton as finite-state engine is used to parse strings according to the grammar. The efficiency of the parsing relies on the implementation of this run-time phase, which is the center of our discussion in this article.

The class of grammars that can be deterministically analyzed using LR parsing with k lookahead symbols are called LR(k) grammars. They are useful for describing programming languages, but they are very limited when they are used for other purposes, for example parsing of natural languages. If we consider LR parsing tables in which each entry can contain several actions, we obtain non-deterministic LR parsing, often known as *generalized LR parsing.* A kind of generalized LR parsing was proposed by Tomita in [13]. He uses a *graph-structured stack* instead of a single stack in order to deal with multiple parses of a single sentence. Tomita's algorithm has problems with *cyclic* and *hidden left recursive* constructions. Moreover, the complexity of Tomita's algorithm with respect to the input string is $\mathcal{O}(n^{p+1})$, where p is the length of the longer right-hand side of a rule.

Several enhancements to the original Tomita's algorithm have been proposed. Rekers [9] has modified the original algorithm to overcome its limitations, but maintaining at least the original complexity. Other authors prefer to transform the grammar in order to reduce the space and time bounds [10]. Some approaches also use transformations in the construction of the LR automaton and in these the treatment of cyclicity is even more complex and so is often avoided [7].

In recent years there has been a great interest in knowing how one parsing algorithm can be derived from another [7, 12, 6]. Usually, the start point for this comparison is Earley's algorithm [3]. We propose a generalized LR(1) and LALR(1) parsing algorithm for arbitrary context-free grammars which is derived, in a natural way, from the well known Earley's algorithm, preserving cubic time complexity in the worst case but performing better in the average case and attaining linear complexity in the case of LR grammars.

1.1 Parsing Schemata

We will describe parsing algorithms using *Parsing Schemata*, a framework for high-level description of parsing algorithms [12]. An interesting application of this framework is the analysis of the relations between different parsing algorithms by studying the formal relations between their underlying parsing schemata.

Given a *context-free grammar* $G = (V_N, V_T, P, S)$, where V_N is a finite set of non-terminal symbols, V_T is a finite set of terminal symbols, P is a finite set of productions $A \rightarrow \alpha$ and $S \in V_N$ is the start symbol or axiom of the grammar, a *parsing system* for G and string $a_1 \dots a_n$ is a triple $\langle \mathcal{I}, \mathcal{H}, \mathcal{D} \rangle$, with $\mathcal{I}$ a set of *items* which represent intermediate parse results, $\mathcal{H}$ an initial set of items that encodes the sentence to be parsed, and $\mathcal{D}$ a set of *deduction steps* that allow to derive new items from already know items. Deduction steps are of the form $\eta_1, \dots, \eta_k \vdash \xi$, meaning that if all antecedents η_i of a deduction step are present, then the consequent ξ should be generated by the parser. A set $\mathcal{F} \subseteq \mathcal{I}$ of *final items* represent the recognizing of a sentence.

A parsing schemata is a parsing system parameterized by a context-free grammar and a sentence.

We can see a relation between parsing schemata and the *grammatical deduction systems* proposed in [11], where items are called *formula schemata*, deduction steps are *inference rules*, hypothesis are *axioms* and final items are *goal formulas*.

A parsing schemata can be generalized from another one using the following transformations [12]:

– *Item refinement*, breaking single items into multiple items.
– *Step refinement*, decomposing a single deduction step in a sequence of steps.
– *Extension* of a schema by considering a larger class of grammars.

In order to decrease the number of items and deduction steps in a parsing schemata, we can apply the following kinds of filtering:

– *Static filtering*, in which redundant parts are simply discarded.
– *Dynamic filtering*, using context information to determine the validity of items.
– *Step contraction*, in which a sequence of deduction steps is replaced by a single one.

1.2 Notation

Given a CFG $G = (V_N, V_T, P, S)$, we will write $A, B \ldots$ for elements in V_N, $a, b \ldots$ for elements in V_T, $X, Y \ldots$ for elements in V and $\alpha, \beta \ldots$ for elements in V^*, where $V = V_N \cup V_T$. The relation $\Rightarrow$ on $V^* \times V^*$ is defined by $\alpha \Rightarrow \beta$ if there are $\alpha', \alpha'', A, \gamma$ such that $\alpha = \alpha' A \alpha''$, $\beta = \alpha' \gamma \alpha''$ and $A \rightarrow \gamma \in P$ exists. We can suffix each element in a rule r, thus $A_{r,0} \rightarrow A_{r,1} A_{r,2} \ldots A_{r,n_r}$.

The set of items in a parsing schemata corresponding to a given parsing algorithm $\mathcal{A}$ is called $\mathcal{I}_{\mathcal{A}}$, the set of hypotheses $\mathcal{H}_{\mathcal{A}}$, the set of final items $\mathcal{F}_{\mathcal{A}}$ and the set of deduction steps is called $\mathcal{D}_{\mathcal{A}}$.

2 From Earley to LR in Dynamic Programming

In this section we show how a dynamic programming version of LR and LALR parsing algorithms can be derived from Earley's algorithm. As a first step, we describe the latter and we show its relation with LR(0). Next, by adding lookahead we obtain LR(1). Then, the predictive phase is compiled in a finite state automaton and implicit binarization is used to obtain a generalized LR and LALR parsing algorithm with cubic worst case complexity. From this point, minor changes are needed to obtain a version working on a push-down automata implemented in dynamic programming.

2.1 Earley Parsing

An Earley parser [3] for a grammar $\mathcal{G}$ and input $a_1 \ldots a_n$ constructs a sequence of $n + 1$ sets of items. Each item has the form $[A \rightarrow \alpha.\beta, i, j]$, where $A \rightarrow \alpha.\beta$ indicates that the α part of rule $A \rightarrow \alpha\beta \in P$ has been recognized, j indicates the item is in item set j and i is a *back-pointer* to the item set in which we began to recognize the current rule. It holds $0 \leq i \leq j$.

Parsing begin with a set of initial items $[S \rightarrow .\alpha, 0, 0]$, where $S \rightarrow \alpha \in P$, and successively applies the three operations *scanner*, *predictor* and *completer* until new items cannot be generated.

A scanner operation can be applied if there exists an item $[A \rightarrow \alpha.a\beta, i, j]$ and $a_{j+1} = a$, generating the item $[A \rightarrow \alpha a.\beta, i, j + 1]$. A predictor operation is applied when an item $[A \rightarrow \alpha.B\beta, i, j]$ exists, generating an item $[B \rightarrow .\gamma, j, j]$ for each $B \rightarrow \gamma \in P$. This operation represents the predictive descendent phase of the algorithm. A completer operation can be applied if two items $[A \rightarrow \alpha.B\beta, i, k]$ and $[B \rightarrow \gamma., k, j]$ exist, and produces a new item $[A \rightarrow \alpha B.\beta, i, j]$, which represent that $a_{k+1} \ldots a_j$ can be reduced to B and therefore, as we know that α reduces the input $a_{i+1} \ldots a_k$, we can ensure that αB reduces $a_{i+1} \ldots a_j$.

If a final item $[S \rightarrow \alpha., 0, n]$ has been generated, the input sentence belongs to the language generated by the grammar.

The parsing schemata corresponding to Earley's algorithm is the following [12]:

$$\mathcal{I}_{\text{Earley}} = \left\{ [A \rightarrow \alpha.\beta, i, j] \mid A \rightarrow \alpha\beta \in P,\ 0 \leq i \leq j \right\}$$

$$\mathcal{H}_{\text{Earley}} = \left\{ [a, i, i+1] \mid a = a_i \right\}$$

$$\mathcal{D}^{\text{Init}}_{\text{Earley}} = \left\{ \vdash [S \rightarrow .\alpha, 0, 0] \right\}$$

$$\mathcal{D}^{\text{Scan}}_{\text{Earley}} = \left\{ [A \rightarrow \alpha.a\beta, i, j], [a, j, j+1] \vdash [A \rightarrow \alpha a.\beta, i, j+1] \right\}$$

$$\mathcal{D}^{\text{Pred}}_{\text{Earley}} = \left\{ [A \rightarrow \alpha.B\beta, i, j] \vdash [B \rightarrow .\gamma, j, j] \right\}$$

$$\mathcal{D}^{\text{Comp}}_{\text{Earley}} = \left\{ [A \rightarrow \alpha.B\beta, i, k], [B \rightarrow \gamma., k, j] \vdash [A \rightarrow \alpha B.\beta, i, j] \right\}$$

$$\mathcal{D}_{\text{Earley}} = \mathcal{D}^{\text{Init}}_{\text{Earley}} \cup \mathcal{D}^{\text{Scan}}_{\text{Earley}} \cup \mathcal{D}^{\text{Pred}}_{\text{Earley}} \cup \mathcal{D}^{\text{Comp}}_{\text{Earley}}$$

$$\mathcal{F}_{\text{Earley}} = \left\{ [S \rightarrow \alpha., 0, n] \right\}$$

In Fig. 1 we show how Earley's algorithm proceed, using arcs to represent the part of the input string an item recognizes. We suppose the grammar includes the following two rules:

$$\begin{array}{l} A \rightarrow \alpha B \gamma \\ B \rightarrow abc \end{array}$$

and the item $[A \rightarrow \alpha.B\beta, i, k]$ was already generated. Thus, $\alpha \overset{*}{\Rightarrow} a_i \ldots A_{k-1}$, which is represented in Fig. 1 by the first thin arc. At this moment, items containing rules of B are predicted. In our case the only item predicted is $[B \rightarrow .abc, k, k]$, which is represented by a dashed arc in Fig. 1. The three following arcs represent the application of three Scan steps in order to recognize terminals a, b and c. The items generated are $[B \rightarrow a.bc, k, k+1]$, $[B \rightarrow ab.c, k, k+2]$ and $[B \rightarrow abc., k, j]$, respectively. This last item and $[A \rightarrow \alpha.B\beta, i, k]$ are the antecedents of a Comp step generating $[A \rightarrow \alpha B.\beta, i, j]$, which is represented by the thick arc.

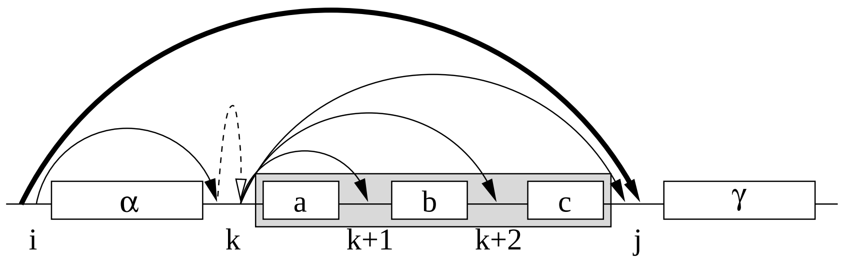

Fig. 1. Graphic representation of Earley's algorithm

2.2 From Earley to LR(0)

The previous parsing schemata corresponds to "uncompiling" an LR(0) parser: while LR(0) parsers *compile* the Pred steps into an finite state control, an Earley parser use *run-time* items. What Pred really does is compute the closure function in run-time[1]. In this sense, the Scan and Comp steps correspond to shift and reduce operations of classical LR parsers.

In order to obtain a version closer to LR stack computations, we can make several minor changes in the Scan and Comp steps, involving a slightly different use of indexes: the components i and j of an item $[A \rightarrow \alpha.\beta, i, j]$ will now represent the part of the input string recognized by the element $X \in V$ suffix of α. Completer step must now have m elements as antecedents, where m is the length of the right hand side of the rule to be reduced. In the new LR(0) parsing schemata, the Scan and Comp steps are called Shift and Reduce because they show a close relation to shift and reduce operations of LR parsers:

$$\mathcal{I}_{\mathrm{LR}(0)} = \mathcal{I}_{\mathrm{Earley}}$$

$$\mathcal{H}_{\mathrm{LR}(0)} = \mathcal{H}_{\mathrm{Earley}}$$

$$\mathcal{D}^{\mathrm{Init}}_{\mathrm{LR}(0)} = \mathcal{D}^{\mathrm{Init}}_{\mathrm{Earley}}$$

$$\mathcal{D}^{\mathrm{Shift}}_{\mathrm{LR}(0)} = \left\{ [A \rightarrow \alpha.a\beta, i, j], [a, j, j+1] \vdash [A \rightarrow \alpha a.\beta, j, j+1] \right\}$$

$$\mathcal{D}^{\mathrm{Pred}}_{\mathrm{LR}(0)} = \mathcal{D}^{\mathrm{Pred}}_{\mathrm{Earley}}$$

$$\mathcal{D}^{\mathrm{Reduce}}_{\mathrm{LR}(0)} = \left\{ \begin{array}{l} [B \rightarrow X_1 X_2 \cdots X_m., j_{m-1}, j_m], \ldots, [B \rightarrow X_1.X_2 \cdots X_m, j_0, j_1], \\ [A \rightarrow \alpha.B\beta, i, j_0] \\ \vdash [A \rightarrow \alpha B.\beta, j_0, j_m] \end{array} \right\}$$

[1] In LR(0), the closure of a state st proceed as follow: for each LR(0) item $[A \rightarrow \alpha.B\beta]$ in st and production $B \rightarrow \gamma$, it includes $[B \rightarrow .\gamma]$ in st until no more elements can be included in st.

$$\mathcal{D}_{\mathrm{LR}(0)} = \mathcal{D}^{\mathrm{Init}}_{\mathrm{LR}(0)} \cup \mathcal{D}^{\mathrm{Shift}}_{\mathrm{LR}(0)} \cup \mathcal{D}^{\mathrm{Pred}}_{\mathrm{LR}(0)} \cup \mathcal{D}^{\mathrm{Reduce}}_{\mathrm{LR}(0)}$$

$$\mathcal{F}_{\mathrm{LR}(0)} = \mathcal{F}_{\mathrm{Earley}}$$

In Fig. 2 we show how LR(0) parsers proceed for the same case of Fig. 1. We suppose the item $[A \rightarrow \alpha.B\beta, ?, k]$ was already generated[2]. At this point, the item $[B \rightarrow .abc, k, k]$ is predicted. Shifts of the terminals a, b y c generate the items $[B \rightarrow a.bc, k, k+1]$, $[B \rightarrow ab.c, k+1, k+2]$ and $[B \rightarrow abc., k+2, j]$, respectively, which are represented by three thin arcs in Fig. 2. The difference of behavior between Earley and LR(0) parsing algorithms is due to the different coverage of Scan and Shift deduction steps, as can be observed comparing Fig. 1 and Fig. 2. Following the parsing process, the reduction of rule $B \rightarrow abc$ is performed by grouping the items corresponding to the recognition of each element of the right-hand side of that rule, generating the item $[A \rightarrow \alpha B.\beta, k, j]$, which is represented by a thick arc in Fig. 2.

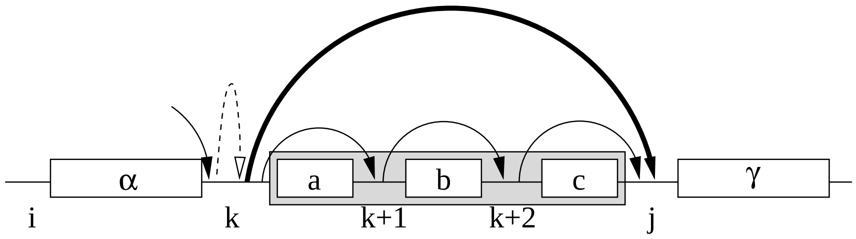

Fig. 2. Graphic representation of LR(0) algorithm

2.3 LR Parsing with Lookahead

In order to obtain a parsing schemata for LR(1), we must introduce the notion of lookahead into items, as is done in the classical construction of finite state control for LR(1) parsers [1]. The items in LR parsing schemata can be obtained by *item refinement* of LR(0) items if we consider that in Earley and LR(0) parsing schemata, each item represents a set of items having the same dotted rule and indexes but probably different lookahead:

$$\mathcal{I}_{\mathrm{LR}} = \left\{ [A \rightarrow \alpha.\beta, b, i, j] \mid A \rightarrow \alpha\beta \in P,\ \ b \in V_T,\ \ 0 \leq i \leq j \right\}$$

where b represents the lookahead. The parsing schemata, in which lookahead is set in $\mathcal{D}^{\mathrm{Pred}}_{\mathrm{LR}}$ and checked in $\mathcal{D}^{\mathrm{Reduce}}_{\mathrm{LR}}$, by means of a *dynamic filter*, is the following:

[2] The first index can not be known from the information in Fig. 2.

$$\mathcal{H}_{\text{LR}} = \mathcal{H}_{\text{Earley}}$$

$$\mathcal{D}_{\text{LR}}^{\text{Init}} = \{\vdash [S \to .\alpha, \$, 0, 0]\}$$

$$\mathcal{D}_{\text{LR}}^{\text{Shift}} = \left\{ [A \to \alpha.a\beta, b, i, j], [a, j, j+1] \vdash [A \to \alpha a.\beta, b, j, j+1] \right\}$$

$$\mathcal{D}_{\text{LR}}^{\text{Pred}} = \left\{ [A \to \alpha.B\beta, b, i, j] \vdash [B \to .\gamma, b', j, j] \mid b' = \text{first}(\beta b) \right\}$$

$$\mathcal{D}_{\text{LR}}^{\text{Reduce}} = \left\{ \begin{array}{l} [B \to X_1 X_2 \cdots X_m., b', j_{m-1}, j_m], \ldots, [B \to X_1.X_2 \cdots X_m, b', j_0, j_1], \\ [A \to \alpha.B\beta, b, i, j_0] \\ \vdash [A \to \alpha B.\beta, b, j_0, j_m] \mid b' \in \text{first}(\beta b),\ \exists [b', j, j+1] \in \mathcal{H}_{\text{LR}} \end{array} \right\}$$

$$\mathcal{D}_{\text{LR}} = \mathcal{D}_{\text{LR}}^{\text{Init}} \cup \mathcal{D}_{\text{LR}}^{\text{Shift}} \cup \mathcal{D}_{\text{LR}}^{\text{Pred}} \cup \mathcal{D}_{\text{LR}}^{\text{Reduce}}$$

$$\mathcal{F}_{\text{LR}} = \left\{ [S \to \alpha., \$, 0, n] \right\}$$

where "first" is defined as follows:

Definition 1. An element $a \in V_T$ is in first(X), where $X \in V$, if $X = a$ or $X \to \varepsilon \in P$ and $a = \varepsilon$ or $X \to Y_1 \cdots Y_i \cdots Y_m \in P$ and $a \in \text{first}(Y_i)$ and $\forall_{j=1}^{i-1}\ \varepsilon \in \text{first}(Y_j)$.
The extension to first(α), where $\alpha = X_1 \cdots X_i \cdots X_n \in V$, is straightforward: $a \in \text{first}(\alpha)$ if $a \in \text{first}(X_1) \cup \cdots \cup \text{first}(X_i)$ and $\varepsilon \notin \text{first}(X_i)$ and $\forall_{j=1}^{i-1}\ \varepsilon \in \text{first}(X_j)$. If $\alpha \overset{*}{\Rightarrow} \varepsilon$ then $\varepsilon \in \text{first}(\alpha)$.

In Fig. 3 we show how LR(1) parsers proceed. The only difference with respect to Fig. 2 is that b' is computed as a valid lookahead in Pred step and it is compared with the first element in γ when the reduction of rule $B \to abc$ is performed.

2.4 LR(1) and LALR(1) Using Compiled Tables

If we "compile" the computation of Pred steps, as performed the *closure* function in the construction of states in classical LR algorithms [1], we obtain a more compact and efficient algorithm, because the work done in run-time has been reduced by the elimination of Pred steps. The items in the new LR$^{\text{c}}$ parsing schemata are equivalent to those ones of LR schemata, because items $[A \to \alpha.\beta, b, i, j]$ are simply replaced by $[st, i, j]$, where st is the precomputed state which contains the element $[A \to \alpha.\beta, b]$.

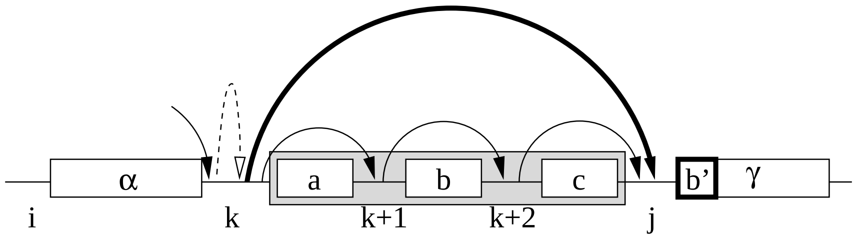

Fig. 3. Graphic representation of LR(1) algorithm

A significative advantage with respect to previous parsing schemata is that we can now differentiate between LR(1) and LALR(1) algorithms simply by choosing the appropriate compiling methods for the finite state control. The parsing schemata is the following:

$$\mathcal{I}_{\mathrm{LR^c}} = \left\{ [st, i, j] \mid st \in \mathcal{S},\ 0 \leq i \leq j \right\}$$

$$\mathcal{H}_{\mathrm{LR^c}} = \mathcal{H}_{\mathrm{Earley}}$$

$$\mathcal{D}^{\mathrm{Init}}_{\mathrm{LR^c}} = \{\vdash [st_0, 0, 0]\}$$

$$\mathcal{D}^{\mathrm{Shift}}_{\mathrm{LR^c}} = \left\{ [st, i, j], [a, j, j+1] \vdash [st', j, j+1] \mid \mathrm{shift}_{st'} \in \mathrm{action}(st, a) \right\}$$

$$\mathcal{D}^{\mathrm{Reduce}}_{\mathrm{LR^c}} = \left\{ \begin{array}{l} [st^m, j_{m-1}, j_m], , \ldots, [st^1, j_0, j_1], [st^0, i, j_0] \vdash [st, j_0, j_m] \mid \\ \exists [a, j, j+1] \in \mathcal{H}_{\mathrm{LR^c}},\ \mathrm{reduce}_r \in \mathrm{action}(st^m, a), \\ st^i \in \mathrm{reveal}(st^{i+1}),\ st \in \mathrm{goto}(st^0, \mathrm{lhs}(r)), \\ m = \mathrm{length}(\mathrm{rhs}(r)) \end{array} \right\}$$

$$\mathcal{D}_{\mathrm{LR^c}} = \mathcal{D}^{\mathrm{Init}}_{\mathrm{LR^c}} \cup \mathcal{D}^{\mathrm{Shift}}_{\mathrm{LR^c}} \cup \mathcal{D}^{\mathrm{Reduce}}_{\mathrm{LR^c}}$$

where $\mathcal{S}$ is the set of states in the LR automaton, $st_0 \in \mathcal{S}$ is the initial state and where $st^i \in \mathrm{reveal}(st^{i+1})$ is equivalent to $st^{i+1} \in \mathrm{goto}(st^i, X)$ if $X \in V_N$ and it is equivalent to $\mathrm{shift}_{st^{i+1}} \in \mathrm{action}(st^i, X)$ if $X \in V_T$. More intuitively, we can say that reveal function traverse the finite state control of the automaton backwards.

$$\mathcal{F}_{\mathrm{LR^c}} = \left\{ [st_f, 0, n] \right\}$$

where st_f is a final state of the LR automaton.

In the preceding, “action” and “goto” refer to the tables that code the behavior of the LR automaton:

- The action table determines what action should be taken for a given state and lookahead. In the case of shift actions, it determines the resulting new state and in the case of reduce actions, the rule which is to be applied for the reduction.
- The goto table determines what will be the state after performing a reduce action. Each entry is accessed using the current state and the non-terminal, which is the left-hand side of the rule to be applied for reduction.

2.5 LR(1) and LALR(1) with Cubic Complexity

The use of the Reduce step increase the complexity of the algorithm to n^{p+1}, where p is the longest right-hand side of rules in P. In order to obtain $\mathcal{O}(n^3)$ complexity in the general case, we can use a implicit binarization of rules [5], splitting each reduction involving m elements in the reduction of $m+1$ rules with at most 2 elements in their right-hand side. Thus, the reduction of a rule

$$A_{r,0} \rightarrow A_{r,1} \dots A_{r,n_r}$$

is equivalently performed as the reduction of the following $n_r + 1$ rules:

$$\begin{array}{l} A_{r,0} \rightarrow \nabla_{r,0} \\ \nabla_{r,0} \rightarrow A_{r,1}\ \nabla_{r,1} \\ \vdots \\ \nabla_{r,n_r-1} \rightarrow A_{r,n_r}\ \nabla_{r,n_r} \\ \nabla_{r,n_r} \rightarrow \varepsilon \end{array}$$

This new treatment of reductions involves a change in the form of the items: a new element, representing a symbol in a rule or a $\nabla_{r,i}$ meaning that elements $A_{r,i+1} \dots A_{r,n_r}$ have been reduced[3], is added. This corresponds to applying *item refinement* to the previous class of items.

With respect to deduction steps, there is a *step refinement* of the Shift step, because we must now differentiate between whether we make the shift of the first symbol in the right hand side of a rule (InitShift) or the shift of other symbols (Shift).

The Reduce step has also been refined into three steps: the selection of the rule to be reduced (Sel), the reduction of the implicit binary rules (Red) and the recognizing of the left-hand symbol of the rule to be reduced (Head).

$$\mathcal{I}_{\mathrm{LR}^3} = \left\{ [A, st, i, j] \cup [\nabla_{r,s}, st, i, j] \mid A \in V_N \cup V_T,\ st \in \mathcal{S},\ 0 \leq i \leq j \right\}$$

$$\mathcal{H}_{\mathrm{LR}^3} = \mathcal{H}_{\mathrm{Earley}}$$

$$\mathcal{D}^{\mathrm{Init}}_{\mathrm{LR}^3} = \left\{ \vdash [-, st_0, 0, 0] \right\}$$

[3] $\nabla_{r,i}$ is equivalent to the dotted rule $A_{r,0} \rightarrow \alpha.\beta$ where $\alpha = A_{r,1} \dots A_{r,i}$ and $\beta = A_{r,i+1} \dots a_{r,n_r}$.

$$\mathcal{D}^{\text{InitShift}}_{\text{LR}^3} = \left\{ \begin{array}{l} [A, st, i, j] \vdash [A_{r,1}, st', j, j+1] \mid \\ \exists [a, j, j+1] \in \mathcal{H}_{\text{LR}^3},\ A_{r,1} = a,\ \text{shift}_{st'} \in \text{action}(st, a),\ A \in V \end{array} \right\}$$

$$\mathcal{D}^{\text{Shift}}_{\text{LR}^3} = \left\{ \begin{array}{l} [A_{r,s}, st, i, j] \vdash [A_{r,s+1}, st', j, j+1] \mid \\ \exists [a, j, j+1] \in \mathcal{H}_{\text{LR}^3},\ A_{r,s+1} = a,\ \text{shift}_{st'} \in \text{action}(st, a) \end{array} \right\}$$

$$\mathcal{D}^{\text{Sel}}_{\text{LR}^3} = \left\{ \begin{array}{l} [A, st, i, j] \vdash [\nabla_{r,n_r}, st, j, j] \mid \\ \exists [a, j, j+1] \in \mathcal{H}_{\text{LR}^3},\ \text{reduce}_r \in \text{action}(st, a),\ A \in V \end{array} \right\}$$

$$\mathcal{D}^{\text{Red}}_{\text{LR}^3} = \left\{ \begin{array}{l} [\nabla_{r,s}, st, k, j], [A_{r,s}, st, i, k] \vdash [\nabla_{r,s-1}, st', i, j] \mid \\ st' \in \text{reveal}(st) \end{array} \right\}$$

$$\mathcal{D}^{\text{Head}}_{\text{LR}^3} = \left\{ [\nabla_{r,0}, st, i, j] \vdash [A_{r,0}, st', i, j] \mid st' \in \text{goto}(st, A_{r,0}) \right\}$$

$$\mathcal{D}_{\text{LR}^3} = \mathcal{D}^{\text{Init}}_{\text{LR}^3} \cup \mathcal{D}^{\text{InitShift}}_{\text{LR}^3} \cup \mathcal{D}^{\text{Shift}}_{\text{LR}^3} \cup \mathcal{D}^{\text{Sel}}_{\text{LR}^3} \cup \mathcal{D}^{\text{Red}}_{\text{LR}^3} \cup \mathcal{D}^{\text{Head}}_{\text{LR}^3}$$

$$\mathcal{F}_{\text{LR}^3} = \left\{ [\Phi, st_f, 0, n] \right\}$$

where $\mathcal{S}$ is the set of states in the LR automaton, $st_0 \in \mathcal{S}$ is the initial state and Φ is the axiom of the augmented grammar.

In Fig. 4 we show how LR(1) parsers with cubic complexity proceed. Instead of applying a big Reduce step when rule $B \to abc$ must be reduced, the Sel step generates the item $[\nabla_{2,3}, st, j, j]$. Then a Red step is applied, combining this item with the item resulting of the shift of c in order to generate the item $[\nabla_{2,2}, st', k+2, j]$. A new Red step combining this last item with the item resulting of the shift of b is applied in order to generate the item $[\nabla_{2,1}, st'', k+1, j]$. Applying once again a Red step we obtain the item $[\nabla_{2,0}, st''', k, j]$. These items are represented by thick arcs in Fig. 4. The application of a Head step generating the item $[B, st^{iv}, k, j]$ finish the reduction. This last item is not shown in Fig. 4.

Complexity Bounds. As the size of the grammar and the finite-state control of the LR automaton are fixed for a given grammar, we have taken the length n of the input string as parameter of complexity. As items include two indexes to the input string, there are $\mathcal{O}(n^2)$ items. Each deduction step executes a bounded number of steps per item. The worst case is given by $\mathcal{D}^{\text{Red}}_{\text{LR}^3}$, which could combine $\mathcal{O}(n^2)$ items of the form $[\nabla_{r,s}, st, k, j]$ with $\mathcal{O}(n)$ items of the form $[A_{r,s}, st, i, k]$ and therefore this step has $\mathcal{O}(n^3)$ complexity.

As in Earley's algorithm, we can group items in *item sets*[4]. In this case, for the class of *bounded item grammars*[5], the number of items is bounded whichever

[4] An item set can be associated to each position in the input string. Items with fourth component equals to j are in the item set j.

[5] Which are called *bounded state grammars* in [3].

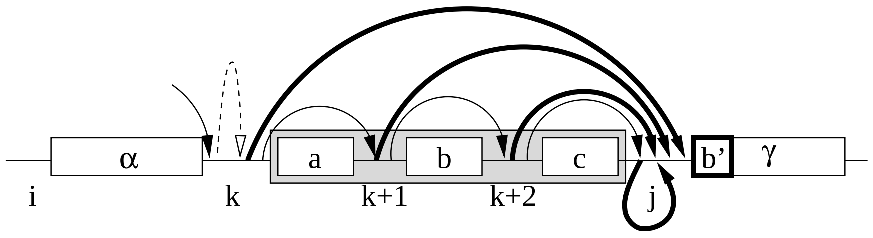

Fig. 4. Graphic representation of LR(1) algorithm with cubic complexity

is the item set, and linear time and space on the length of the input string is attained. This has a practical sense because this class of grammars includes the LR family and, in consequence, linear parsing can be performed when local determinism is present.

2.6 Dynamic Programming Interpretation of LR Push-down Automata

The LR^3 parsing schemata corresponds to a dynamic interpretation of LR(1) or LALR(1) parsing algorithms using an inference system based on S^1 items [16, pp. 173–175]. It can be easily transformed into a set of push-down transitions in order to represent the algorithm into the common framework for parsing described by Lang in [5], which is based on dynamic programming interpretation of Push-Down Automata (PDA). From [16] we know that we can use S^1 items (i.e., items containing only the top element of the stack) in order to obtain a correct interpretation in dynamic programming of weakly predictive automata [16], such as LR.

In Fig. 5 we can observe the stack behavior of the LR algorithm described in Sect. 2.5. In this figure, boxes represent items. Each box contains the first element of the item it represent[6]. Shift and Sel deduction steps *push* a new top element on the stack of items. Red deduction steps *pop* the two items on the top of the stack, placing a new top item. Head deduction steps *swap* the top item by a new top item.

Transitions shown in Fig. 5 exactly correspond with the transitions of PDA described in [5]. In effect, every PDA can be described using the following transitions:

$$\begin{array}{lll} \text{SWAP:} & (B \longmapsto C)(A) = C & \text{such that } B = A \\ \text{PUSH:} & (B \longmapsto CB)(A) = C & \text{such that } B = A \\ \text{POP:} & (BD \longmapsto C)(A, E) = C & \text{such that } (B, D) = (A, E) \end{array}$$

where A, B, C, D and E are items and stacks grow from right to left.

[6] In order to draw a clearer picture, we have not included in boxes the rest of the elements in each item. Also, nabla symbols have only one suffix which indicates the position in rule $B \rightarrow abc$.

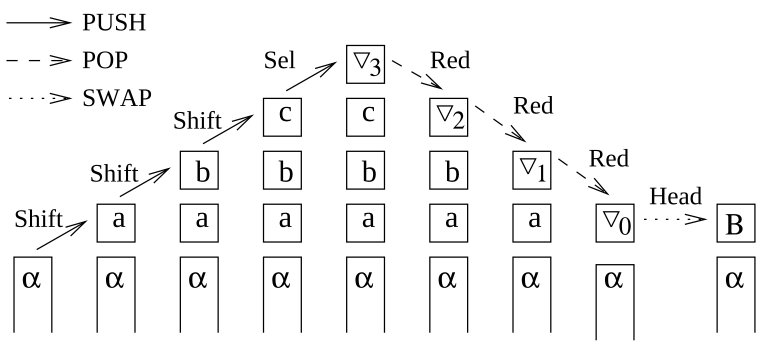

Fig. 5. Graphic representation of stack behavior of LR algorithms

Thus, considering Head deduction steps as SWAP transitions, Init, InitShift, Shift and Sel steps as PUSH transitions and Red steps as POP transitions, we obtain the following set of transitions that describe the dynamic programming interpretation of LR(1) or LALR(1) push-down automata.

$$\mathcal{I}_{\mathrm{LR}^{\mathrm{S1}}} = \left\{ [A, st, i, j] \cup [\nabla_{r,s}, st, i, j] \mid A \in V_N \cup V_T,\ \ st \in \mathcal{S},\ \ 0 \le i \le j \right\}$$

$$\mathcal{H}_{\mathrm{LR}^{\mathrm{S1}}} = \mathcal{H}_{\mathrm{Earley}}$$

$$\mathcal{D}^{\mathrm{Init}}_{\mathrm{LR}^{\mathrm{S1}}} = \left\{ \vdash [-, st_0, 0, 0] \right\}$$

$$\mathcal{D}^{\mathrm{InitShift}}_{\mathrm{LR}^{\mathrm{S1}}} = \left\{ \begin{array}{l} [A, st, i, j] \vdash [A_{r,1}, st', j, j+1] \quad [A, st, i, j] \mid \\ \exists [a, j, j+1] \in \mathcal{H}_{\mathrm{LR}^{\mathrm{S1}}},\ \ A_{r,1} = a,\ \ \mathrm{shift}_{st'} \in \mathrm{action}(st, a),\ \ A \in V \end{array} \right\}$$

$$\mathcal{D}^{\mathrm{Shift}}_{\mathrm{LR}^{\mathrm{S1}}} = \left\{ \begin{array}{l} [A_{r,s}, st, i, j] \vdash [A_{r,s+1}, st', i, j+1] \quad [A_{r,s}, st, i, j] \mid \\ \exists [a, j, j+1] \in \mathcal{H}_{\mathrm{LR}^{\mathrm{S1}}},\ \ A_{r,s+1} = a,\ \ \mathrm{shift}_{st'} \in \mathrm{action}(st, a) \end{array} \right\}$$

$$\mathcal{D}^{\mathrm{Sel}}_{\mathrm{LR}^{\mathrm{S1}}} = \left\{ \begin{array}{l} [A, st, i, j] \vdash [\nabla_{r,n_r}, st, i, j+1] \quad [A_{r,n_r}, st, i, j] \mid \\ \exists [a, j, j+1] \in \mathcal{H}_{\mathrm{LR}^{\mathrm{S1}}},\ \ \mathrm{reduce}_r \in \mathrm{action}(st, a),\ \ A \in V \end{array} \right\}$$

$$\mathcal{D}^{\mathrm{Red}}_{\mathrm{LR}^{\mathrm{S1}}} = \left\{ \begin{array}{l} [\nabla_{r,s}, st, i, k][A_{r,s}, st, k, j] \vdash [\nabla_{r,s-1}, st', i, j] \mid \\ st' \in \mathrm{reveal}(st) \end{array} \right\}$$

$$\mathcal{D}^{\mathrm{Head}}_{\mathrm{LR}^{\mathrm{S1}}} = \left\{ \begin{array}{l} [\nabla_{r,0}, st, i, j] \vdash [A_{r,0}, st', i, j] \mid \\ st' \in \mathrm{goto}(st, A_{r,0}) \end{array} \right\}$$

$$\mathcal{D}_{\mathrm{LR}^{\mathrm{S1}}} = \mathcal{D}^{\mathrm{Init}}_{\mathrm{LR}^{\mathrm{S1}}} \cup \mathcal{D}^{\mathrm{InitShift}}_{\mathrm{LR}^{\mathrm{S1}}} \cup \mathcal{D}^{\mathrm{Shift}}_{\mathrm{LR}^{\mathrm{S1}}} \cup \mathcal{D}^{\mathrm{Sel}}_{\mathrm{LR}^{\mathrm{S1}}} \cup \mathcal{D}^{\mathrm{Red}}_{\mathrm{LR}^{\mathrm{S1}}} \cup \mathcal{D}^{\mathrm{Head}}_{\mathrm{LR}^{\mathrm{S1}}}$$

$$\mathcal{F}_{\mathrm{LR}^3} = \left\{ [\Phi, st_f, 0, n] \right\}$$

where $\mathcal{S}$ is the set of states in the LR automaton, $st_0 \in \mathcal{S}$ is the initial state and Φ is the axiom of the augmented grammar.

3 Example

We are now going to show how a parsing algorithm implementing the LR^{S1} schema use the lookahead to improve performance by avoiding the exploration of useless computations and how it can deal with cyclic and recursive rules. For this purpose, we will use the following grammar $\mathcal{G}$, simple but instructive:

(0) $\Phi \to S$	(1) $S \to Aa$	(5) $S \to Dc$
	(2) $S \to Bb$	(6) $D \to E$
	(3) $A \to cc$	(7) $E \to D$
	(4) $B \to cc$	(8) $E \to \varepsilon$

The language generated by $\mathcal{G}$ is the set $\{cca, ccb, c\}$. Rule 0 represents the augmentation of the grammar [1], rules 1 to 4 generate the stings cca and ccb while rules 5 to 8 describe an unbounded number of derivations for the string c.

In Fig. 6 we show the LALR(1) automaton of $\mathcal{G}$ with the transitions corresponding to the interpretation of the prefix cc in the input cca using rules 1 to 4. Thick arrows, dashed arrows and dotted arrows represent PUSH, POP and SWAP transitions, respectively. In the case of Fig. 6, with computation starting in state 0, the first action is an InitShift, pushing the item $[c, st1, 0, 1]$ and changing to state 1. In this state we know that we are trying to analyze the current part of the input according to rules 3 and 4[7], but we do not know which of the two rules will be the only correct one. The following action is a Shift, pushing the item $[c, st2, 1, 2]$ and changing to state 2. From this state we know that both rules 3 and 4 recognize the input cc, but the lookahead determines that 3 is the correct one. A parser without lookahead would have to explore the two alternatives, discovering the correct one and rejecting the incorrect one some time later. The use of lookahead increase the deterministic domain, allowing better efficiency.

The other transitions shown in Fig. 6 correspond to the reduction by rule 3:

Sel:	Push $[\nabla_{3,2}, st2, 2, 2]$
Red:	Pop $[\nabla_{3,2}, st2, 2, 2]$ and $[c, st2, 1, 2]$ for $[\nabla_{3,1}, st1, 1, 2]$
Red:	Pop $[\nabla_{3,1}, st1, 1, 2]$ and $[c, st1, 0, 1]$ for $[\nabla_{3,0}, st0, 0, 2]$
Head:	Swap $[\nabla 3, 0.st0, 0, 2]$ for $[A, st3, 0, 2]$

In Fig. 7 we show the transitions corresponding to several cyclic computations while we try to analyze the first c of the input string cca using rules 5 to 8. Starting at state 0, we may reduce rule 8:

Sel:	Push $[\nabla_{8,0}, st0, 0, 0]$
Head:	Swap $[\nabla_{8,0}, st0, 0, 0]$ for $[E, st9, 0, 0]$

As the current state is 9, we can reduce the rule 6, which involves:

Sel:	Push $[\nabla_{6,1}, st9, 0, 0]$
Red:	Pop $[\nabla_{6,1}, st9, 0, 0]$ and $[E, st9, 0, 0]$ for $[\nabla_{6,0}, st0, 0, 0]$
Head:	Swap $[\nabla_{6,0}, st0, 0, 0]$ for $[D, st7, 0, 0]$

[7] In the sense of Earley parsers, this corresponds to predict the rules 3 and 4.

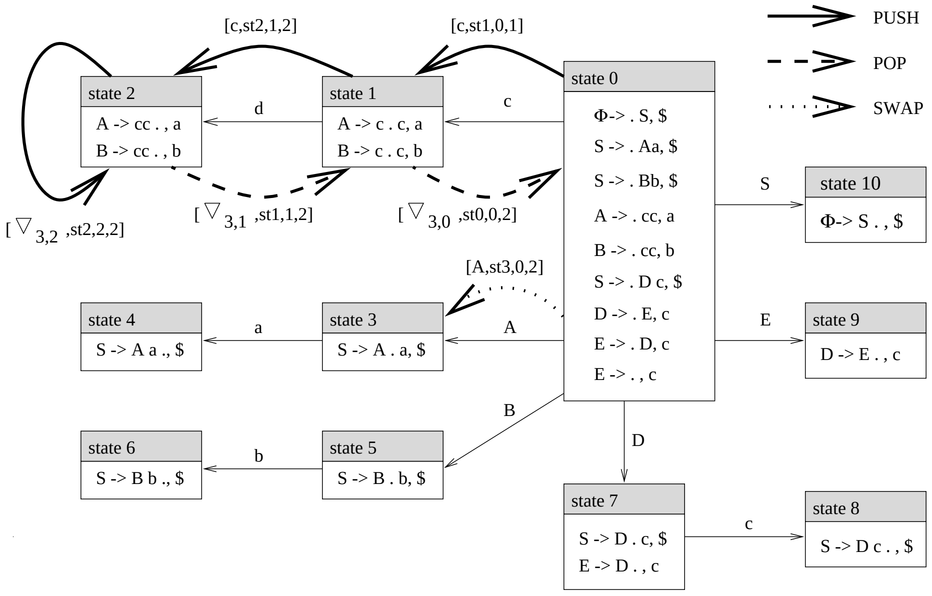

Fig. 6. Transitions for the string *cd* in the LALR(1) automaton of $\mathcal{G}$

Now, we are in state 7 and we can reduce rule 7:

Sel:	Push $[\nabla_{7,1}, st7, 0, 0]$
Red:	Pop $[\nabla_{7,1}, st7, 0, 0]$ and $[D, st7, 0, 0]$ for $[\nabla_{7,0}, st0, 0, 0]$
Head:	Swap $[\nabla_{7,0}, st0, 0, 0]$ for $[E, st9, 0, 0]$

The item resulting from the last transition, $[E, st9, 0, 0]$, had been generated before and therefore we do not need to compute the actions derived from this item again. As a consequence, all items corresponding to the cyclic application of rules 6 and 7 are calculated only once at the first iteration.

To continue the analysis of an input sentence, we must apply a Shift in state 7, pushing $[c, st8, 0, 1]$, but in state 8 we may not apply a reduction because the current lookahead symbol c is not compatible with \$, the lookahead indicated in the element $S \rightarrow Dc., \$$ of state 8.

4 Experimental Results

We have compared an implementation of the proposed parsing algorithm, programmed in Lisp and included in our Ice system, against Bison [2], Glr [9] and Sdf [4], which are to the best of our knowledge some of the most efficient LR-based parsing environments, and also against a implementation of Earley's algorithm included in the Ice system.

Results on deterministic and non deterministic parsing are considered. In both cases, we have used the Pascal syntax as guideline for tests. All tests were

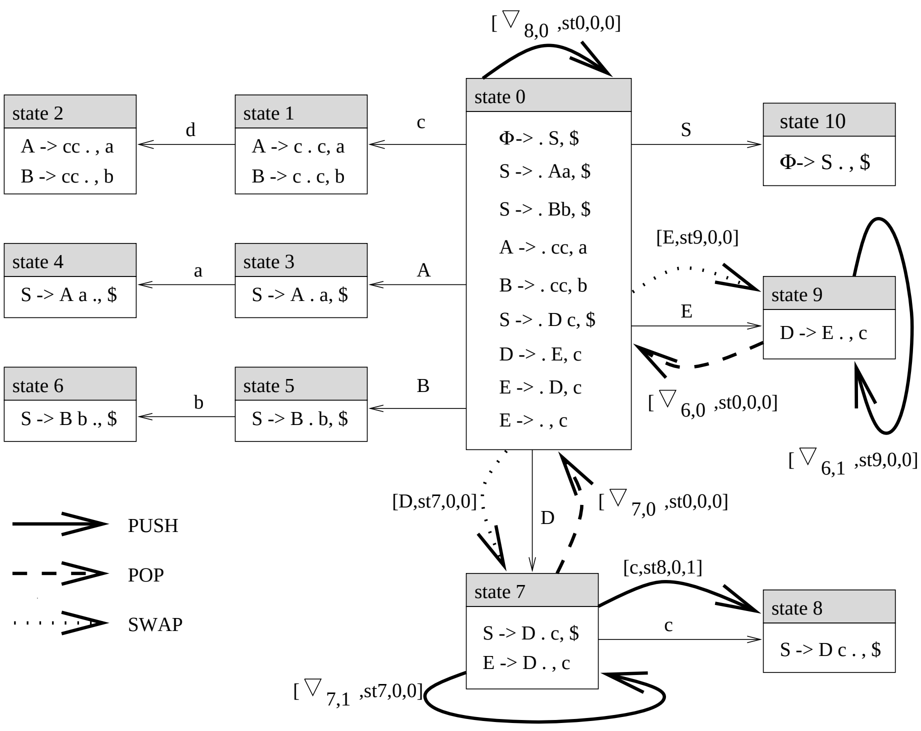

Fig. 7. Transitions corresponding to a cycle in the LALR(1) automaton of $\mathcal{G}$

done using a weakly loaded Sun SPARCstation 10 and lexical time is included because, for the version considered, it is not possible to differentiate lexical and parsing time in GLR and SDF. The time needed to "print" parse trees was not included.

Fig. 8 shows the parsing time for complete Pascal programs using the deterministic grammar. The results for non-deterministic parsing are in Fig. 9. BISON is not considered here because it only works with deterministic grammars. The programs considered are of the form

```
Program P(input, output);
    var a, b: integer;
begin
    a := b{+b}^i
end.
```

in order to reduce the impact of lexical time. The non deterministic grammar contains a rule $Exp \rightarrow Exp{+}Exp$ and therefore the number of ambiguous parses, denoted by C_i, grows exponentially with i, the number of $+$ in the arithmetic expression to be parsed:

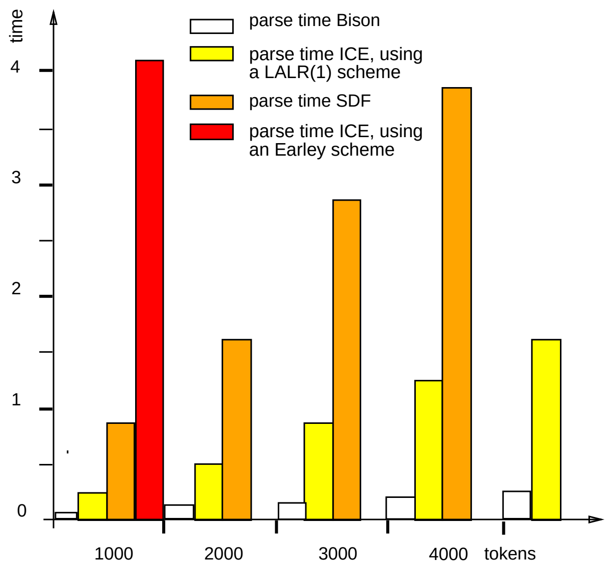

Fig. 8. Results on deterministic parsing

$$C_i = \begin{cases} 1 & \textbf{if } i \in \{0,1\} \\ \begin{pmatrix} 2i \\ i \end{pmatrix} \frac{1}{\text{i+1}} & \textbf{if } i > 1 \end{cases}$$

This test is relevant in natural language processing, a field in which deterministic parsing is not frequent. In fact, the order of ambiguities of the expressions analyzed in the test is the same as in the case of texts written according to the well known grammar of Tomita [13] for prepositional phrases.

5 Conclusion

Earley's algorithm is a good starting point for deriving other and more complex parsing algorithms. In this sense, a Generalized LR algorithm for parsing arbitrary context-free grammars has been derived using several intermediate schemata, applying simple and intuitive transformations in each step. The result is a dynamic programming algorithm which is fully integrated in the common framework for parsing proposed by Lang, with a $\mathcal{O}(n^3)$ complexity in the worst case and with a better behavior in practical cases, as is indicated by several experimental results. This performance is obtained both by avoiding redundant computations and by the high level of sharing attained.

In [14] our specification of the LALR(1) algorithm has been extended to deal with *Definite Clause Grammars* [8] and in [15] to implement full incremental parsing.

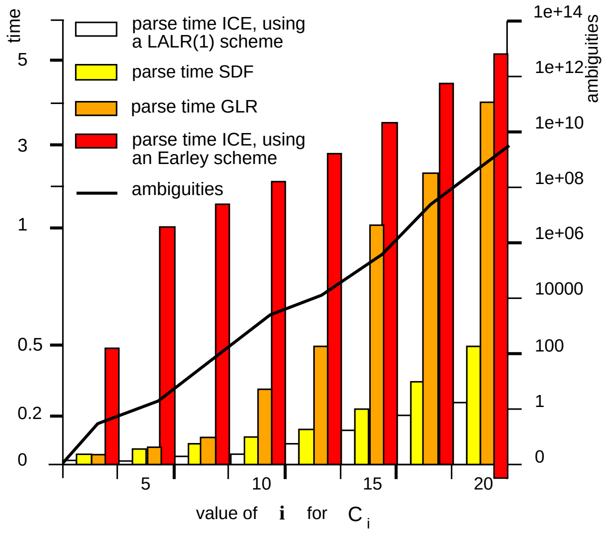

Fig. 9. Results on non deterministic parsing

Acknowledgements

This work has been partially supported by projects XUGA 10505B96 and XUGA 20402B97 of Xunta de Galicia and by *Acción integrada* HF96-36 of the Government of Spain.

References

1. Aho, A. V., Ullman,J. D.: The theory of parsing, translation and compiling. Prentice Hall (1972)
2. Donnelly, C., Stallman, R. M.: Bison reference manual. Free Software Foundation, Inc., 675 Mass Avenue, Cambridge, MA 02139, USA, 1.20 edition (1992)
3. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13** (1970) 94–102
4. Heering, J., Hendriks, P. R. H., Klint, P., Rekers, J.: The syntax definition formalism SDF — reference manual. SIGPLAN Notices **24** (1989) 43–75
5. Lang, B.: Towards a uniform formal framework for parsing. In Tomita, M. (ed.): Current Issues in Parsing Technology. Kluwer Academic Publishers (1991) 153–171
6. McLean, P., Horspool, R. N.: A faster Earley parser. Proc. of International Conference on Compiler Construction (1996) 281–293
7. Nederhof, M.-J., Sarbo, J. J.: Increasing the applicability of LR parsing. Proc. of Third International Workshop on Parsing Technologies (1993) 187–201
8. Pereira, F. C. N., Warren, D. H. D.: Definite Clause Grammars for language analysis — a survey of the formalism and a comparison with Augmented Transition Networks. Artificial Intelligence **13** (1980) 231–278
9. Rekers, J.: Parsing Generation for Interactive Environments. PhD thesis. University of Amsterdam (1992)

10. Sheil, B. A.: Observations on context-free grammars. Proc. of Statistical Methods in Linguistics (1976) 71–109
11. Shieber, S. M., Schabes, Y., Pereira, F. C. N.: Principles and implementation of deductive parsing. Journal of Logic Programming **24** (1995) 3–36
12. Sikkel, K.: Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms. Texts in Theoretical Computer Science — An EATCS Series. Springer-Verlag (1997)
13. Tomita, M.: Efficient Parsing for Natural Language. Kluwer Academic Publishers (1986)
14. Vilares Ferro, M., Alonso Pardo, M. A.: An LALR extension for DCGs in dynamic programming. In Martín Vide, C. (ed.): Mathematical Linguistics, vol. II. John Benjamins Publishing Company (to appear)
15. Vilares Ferro, M., Dion, B. A.: Efficient incremental parsing for context-free languages. Proc. of the 5^{th} IEEE International Conference on Computer Languages (1994) 241–252
16. Villemonte de la Clergerie, E.: Automates à Piles et Programmation Dynamique. DyALog : Une Application à la Programmation en Logique. PhD thesis. Université Paris 7 (1993)