A New Approach to the Construction of Generalized LR Parsing Algorithms[*]

Miguel A. Alonso and **David Cabrero**[†] and **Manuel Vilares**
Departamento de Computación
Facultad de Informática, Universidad de La Coruña
Campus de Elviña s/n, 15071 La Coruña, Spain
{alonso,cabrero,vilares}@dc.fi.udc.es

Abstract

We present a Generalized LR parsing algorithm for unrestricted context-free grammars working in complexity $\mathcal{O}(n^3)$. It differs from previous approaches in the use of dynamic programming techniques to cope with the non determinism, instead of a graph-structured stack. The steps for deriving our algorithm from classical Earley's parsing algorithm are shown.

1 Introduction

LR parsing strategies can analyze LR grammars, which are deterministic. 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*, which can analyze non-deterministic context-free grammars. It this context, some mechanism is needed in order to represent the non-deterministic evolution of the stack. Tomita (Tomita 86) have proposed an algorithm based on a *graph-structured stack* but it has problems with *cyclic* and *hidden left recursive* constructions. Rekers (Rekers 92) has modified the original algorithm to overcome its limitations, but maintaining its original complexity $\mathcal{O}(n^{p+1})$, where p is the length of the longer right-hand side of a rule. Space and time bounds can be reduced transforming the form of the grammar (Sheil 76). 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 (Nederhof & Sarbo 93).

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 (Earley 70), preserving cubic time complexity in the worst case.

We will describe parsing algorithms using *Parsing Schemata*, a framework for high-level description of parsing algorithms (Sikkel 97). An interesting application of this framework is the analysis of the relations between different parsing algorithms. Recently, several authors have shown how one parsing algorithm can be derived from another considering Earley's algorithm as starting point (Nederhof & Sarbo 93; McLean & Horspool 96). Using parsing schemata, algorithms are related by studying the formal relations between their underlying parsing schemata (Sikkel 97).

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 schema* is a parsing system parameterized by a context-free grammar and a sentence.

A parsing schema can be generalized from another one using the following transformations (Sikkel 97):

- *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.

[*] This work was partially supported by project XUGA10505B96 of Xunta de Galicia and by *Acción integrada* HF96-36 of Government of Spain.

[†] Currently at Centro de Investigacións Lingüísticas e Literarias "Ramón Piñeiro", Estrada Santiago-Noia km 3, A Barcia, 15896 Santiago de Compostela, Spain.

In order to decrease the number of items and deduction steps in a parsing schema, 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.

1.1 Notation

Given a CFG $G = (V_n, V_T, P, S)$, we will write $A, B \dots$ for elements in V_N, $a, b \dots$ for elements in V_T, $X, Y \dots$ for elements in $V_N \cup V_T$ and $\alpha, \beta \dots$ for elements in V^*. 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 $A_{r,0} \rightarrow A_{r,1} A_{r,2} \dots A_{r,n_r}$.

The set of items in a parsing schema 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 Relating Earley and LR Parsing Algorithms

Given a grammar $\mathcal{G}$ and a input string $a_1 \dots a_n$, an Earley parser (Earley 70) for $\mathcal{G}$ constructs a sequence of $n+1$ set of items. Each item $[A \rightarrow \alpha.\beta, i, j]$ indicates that $\alpha \overset{*}{\Rightarrow} a_{i+1} \dots a_j$, with $0 \leq i \leq j$, and that symbol A has been predicted in the item set i.

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

scanner operation can be applied if there exists an item $[A \rightarrow \alpha.a\beta, i, j]$ and $a_{j+1} = a$, yielding the item $[A \rightarrow \alpha a.\beta, i, j+1]$.

predictor operation, which represents the predictive descendent phase of the algorithm, can be applied when an item $[A \rightarrow \alpha.B\beta, i, j]$ exists, generating an item $[B \rightarrow .\gamma, j, j]$ for each $B \rightarrow \gamma \in P$.

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 $a_{k+1} \dots a_j$ can be reduced to B and therefore, as we know that α reduces the input $a_{i+1} \dots a_k$, we can ensure that αB reduces $a_{i+1} \dots a_j$.

The input sentence belongs to the language described by the grammar if the final item $[S \rightarrow \alpha., 0, n]$ is generated.

The parsing schema corresponding to Earley's algorithm is the following (Sikkel 97):

$$\mathcal{I}_{Earley} = \{[A \rightarrow \alpha.\beta, i, j]\}$$

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

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

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

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

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

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

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

The previous parsing schema corresponds to "uncompile" 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. In this sense, the *Scan* and *Comp* steps correspond to *Shift* and *Reduce* 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 in α just before the dot. If $\alpha = \varepsilon$ then $i = j$. As a consequence, the 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 $LR(0)$ deduction steps, the *Scan* and

Comp steps are called *Shift* and *Reduce* because they show a close relation to *shift* and *reduce* operations of LR parsers:

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

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

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

3 Using Lookahead: SLR(1) and LR(1)

We can modify the previous parsing schema by adding lookahead in order to mimic the behavior of SLR(1) parsers. For this purpose, we introduce a *dynamic filter* in the *Reduce* step, using the function “follow”, which is defined in relation to a function “first”:

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

Definition 2 *An element $a \in V_T \cup \{\$\}$ is in follow(A), where \$ is an special end-of-input marker not in V_T and $A \in V_N$, if $a = \$$ and A is the start symbol or $A' \rightarrow \alpha A\beta \in P$ and $a \in ($first$(\beta) - \{\varepsilon\})$ or $A' \rightarrow \alpha A \beta \in P$ and $\varepsilon \in$ first(β) and $a \in$ follow(A').*

Checking of lookahead is performed by the condition $\exists[a, j, j+1] \in \mathcal{H}_{SLR} \wedge a \in \text{follow}(B)$ introduced in the *Reduce* deduction step for SLR(1):

$$\mathcal{D}^{Reduce}_{SLR} = \left\{ \begin{array}{l} [B \rightarrow X_1 X_2 \cdots X_m., j_{m-1}, j_m], \\ \vdots \\ [B \rightarrow X_1.X_2 \cdots X_m, j_0, j_1], \\ [A \rightarrow \alpha.B\beta, i, j_0] \\ \vdash [A \rightarrow \alpha B.\beta, j_0, j_m] \mid \\ \exists [a, j, j+1] \in \mathcal{H}_{SLR} \ \wedge \\ a \in \text{follow}(B) \end{array} \right\}$$

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

In order to obtain a parsing schema 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 (Aho & Ullman 72). The items in *LR* parsing schema can be obtained by *item refinement* of *SLR* items if we consider that in *Earley* and *SLR* parsing schema, each item represents a set of items having the same dotted rule and indexes but probably different lookahead:

$$\mathcal{I}_{LR} = \{[A \rightarrow \alpha.\beta, b, i, j]\}$$

where b represents the lookahead. The parsing schema, in which lookahead is set in $\mathcal{D}^{Pred}_{LR}$ and checked in $\mathcal{D}^{Reduce}_{LR}$, is the following:

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

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

$$\mathcal{D}^{Shift}_{LR} = \left\{ \begin{array}{l} [A \rightarrow \alpha.a\beta, b, i, j], [a, j, j+1] \\ \vdash [A \rightarrow \alpha a.\beta, b, j, j+1] \end{array} \right\}$$

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

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

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

$$\mathcal{F}_{LR} = \{[S \rightarrow \alpha., \$, 0, n]\}$$

4 Compiling Predictive Information into Tables

A more compact and efficient algorithm can be obtained by avoiding the computation of *Pred* steps in run-time. Instead, they are compiled, resulting in the construction of states as in classical LR algorithms (Aho & Ullman 72). The items in the new LR^c parsing schema are equivalent to those ones of LR schema, because items $[A \rightarrow \alpha.\beta, b, i, j]$ are simply replaced by $[st, i, j]$, where st is the precomputed state which contains the element $[A \rightarrow \alpha.\beta, b]$. The parsing schema is the following:

$$\mathcal{I}_{LR^c} = \{[st, i, j]\}$$

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

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

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

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

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

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

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

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:

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.

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.

A significative advantage with respect to previous parsing schema 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.

5 Achieving Cubic Complexity

Reduce step in previous parsing schemata increase the complexity of the algorithms to n^{p+1}, where p is the longest right-hand side of rules in P. Implicit binarization of rules (Lang 91) can be applied to achieve $\mathcal{O}(n^3)$ complexity in the general case. Thus, the reduction of a rule

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

can be equivalently performed as the reduction of the following $n_r + 1$ rules with at most 2 elements on their right-hand side:

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

Applying *item refinement* to LR^c items, we obtain the new form of items, with a new element that represent a symbol in a rule or a $\nabla_{r,i}$ meaning that elements $A_{r,i+1} \ldots A_{r,n_r}$ have been reduced. Therefore, $\nabla_{r,i}$ is like a dotted rule $A_{r,0} \rightarrow \alpha.\beta$ where $\alpha = A_{r,1} \ldots A_{r,i}$ and $\beta = A_{r,i+1} \ldots a_{r,n_r}$.

With respect to deduction steps, *Reduce* step must be refined into three steps:

Sel, which select the rule to be reduced.

Red, which reduce each implicit binary rule.

Head, which recognize the left-hand symbol of the rule reduced.

We must also apply *step refinement* to *Shift* deduction steps, obtaining:

InitShift, which is applied when the symbol to be shifted is the first symbol in the right hand side of a rule.

Shift, which is applied in the shift of the other symbols in a rule.

As a result, we obtain the following parsing schema:

$$\mathcal{I}_{LR^3} = \left\{ \ [A, st, i, j] \cup [\nabla_{r,s}, st, i, j] \ \right\}$$

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

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

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

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

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

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

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

$$\begin{aligned} \mathcal{D}_{LR^3} = \ & \mathcal{D}^{Init}_{LR^3} \cup \mathcal{D}^{InitShift}_{LR^3} \cup \mathcal{D}^{Shift}_{LR^3} \cup \\ & \mathcal{D}^{Sel}_{LR^3} \cup \mathcal{D}^{Red}_{LR^3} \cup \mathcal{D}^{Head}_{LR^3} \end{aligned}$$

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

where st_f is a final state and Φ is the axiom of the augmented grammar.

5.1 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 $\mathcal{D}^{Red}_{LR^3}$, 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*[1]. In this case, for the class of *bounded item grammars*[2], the number of items is bounded whichever 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.

5.2 Dynamic Programming

The common framework for parsing described by Lang in (Lang 91) is based on dynamic programming interpretation of push-down automata. In that framework, weakly predictive automata, such as LR, can be interpreted in dynamic programming using items containing only the top element of the stack. The resulting automata are very close to inference systems (Villemonte de la-Clergerie 93, pp. 173–175). In this context, the LR^3 parsing schema can be seen as a dynamic interpretation of LR(1) or LALR(1) parsing algorithms using an inference system based on that kind of items. It can be easily transformed into a set of push-down transitions:

- *Head* deduction steps correspond to SWAP transitions.
- *Init*, *InitShift*, *Shift* and *Sel* steps correspond to PUSH transitions.
- *Red* steps correspond to POP transitions.

The push-down transitions which describe the dynamic programming interpretation of LR(1) or LALR(1) push-down automata are the following:

[1]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.

[2]Which are called *bounded state grammars* in (Earley 70)

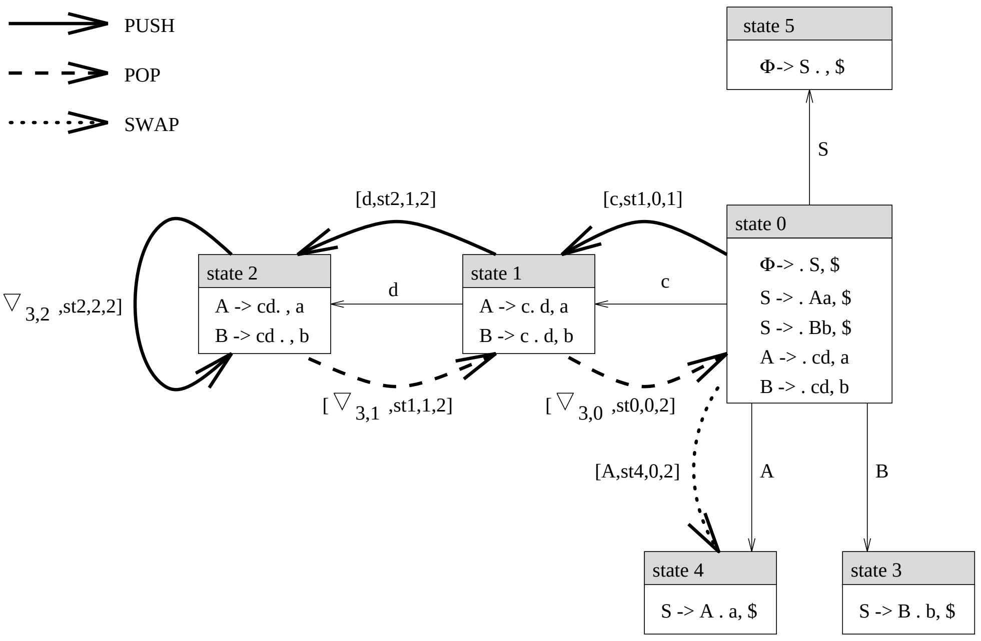

Figure 1: Transitions for the input string cda in the LALR(1) automaton for $\mathcal{G}_1$

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

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

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

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

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

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

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

6 Examples

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 the first purpose, we will use the following grammar $\mathcal{G}_1$, simple but highly instructive:

$$\begin{array}{ll} (0) & \Phi \rightarrow S \\ (1) & S \rightarrow Aa \\ (2) & S \rightarrow Bb \\ (3) & A \rightarrow cd \\ (4) & B \rightarrow cd \end{array}$$

The language generated by $\mathcal{G}_1$ is the set $\{cda, cdb\}$. In Figure 1 we show the LALR(1) automaton for this grammar, with the transitions corresponding to the interpretation of the prefix cd in the input cda. 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[3], but we do not know which of the two rules

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

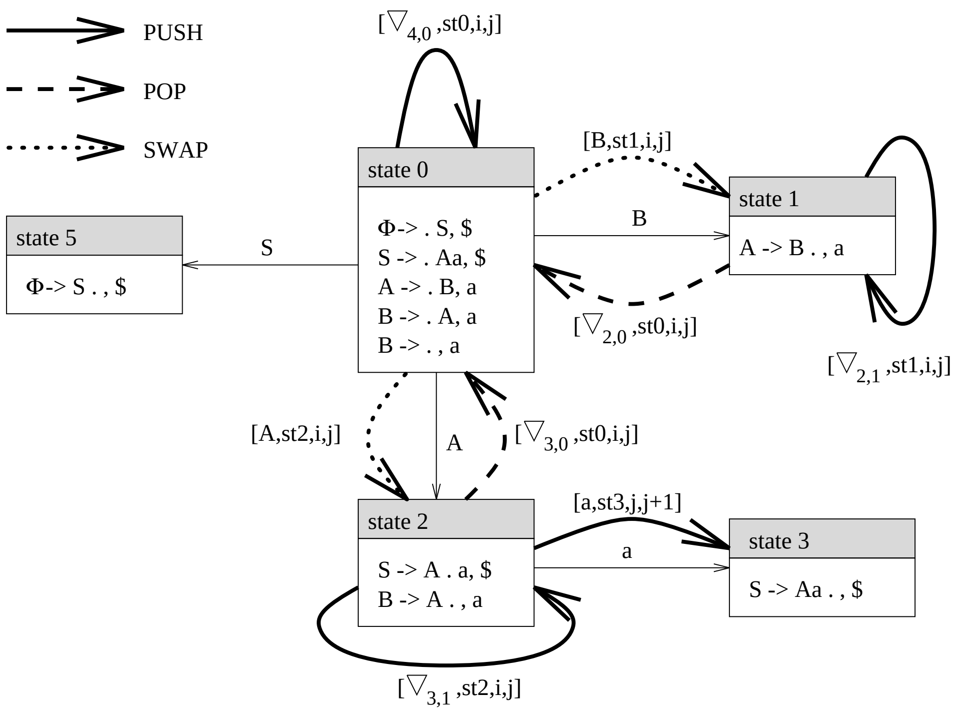

Figure 2: Transitions of a cycle in the LALR(1) automaton for $\mathcal{G}_2$

will be the only correct one. The following action is a *Shift*, pushing the item $[d, st2, 1, 2]$ and changing to state 2. From this state we know that both rules 3 and 4 recognize the input cd, 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 following actions correspond to the reduction by rule 3:

1. *Sel*: Push $[\nabla_{3,2}, st2, 2, 2]$
2. *Red*: Pop $[\nabla_{3,2}, st2, 2, 2]$ and $[d, st2, 1, 2]$ for $[\nabla_{3,1}, st1, 1, 2]$
3. *Red*: Pop $[\nabla_{3,1}, st1, 1, 2]$ and $[c, st1, 0, 1]$ for $[\nabla_{3,0}, st0, 0, 2]$
4. *Head*: Swap $[\nabla 3, 0.st0, 0, 2]$ for $[A, st4, 0, 2]$

To show the analysis of cyclic and recursive grammars, we will use the following grammar $\mathcal{G}_2$, which is again small but instructive:

$$\begin{array}{ll} (0) & \Phi \rightarrow S \\ (1) & S \rightarrow Aa \\ (2) & A \rightarrow B \\ (3) & B \rightarrow A \\ (4) & B \rightarrow \varepsilon \end{array}$$

The language generated by $\mathcal{G}_2$ is $\{a\}$. In Figure 2 you can see the LALR(1) automaton for this grammar, with the transitions corresponding to several cyclic computations.

Starting at state 0, we must reduce rule 4:

1. *Sel*: Push $[\nabla_{4,0}, st0, i, j]$
2. *Head*: Swap $[\nabla_{4,0}, st0, i, j]$ for $[B, st1, i, j]$

As the current state is 1, we can reduce the rule 2, which involves:

1. *Sel*: Push $[\nabla_{2,1}, st1, i, j]$
2. *Red*: Pop $[\nabla_{2,1}, st1, i, j]$ and $[B, st1, i, j]$ for $[\nabla_{2,0}, st0, i, j]$
3. *Head*: Swap $[\nabla_{2,0}, st0, i, j]$ for $[A, st2, i, j]$

Now, we are in state 2 and we can reduce rule 3:

1. *Sel*: Push $[\nabla_{3,1}, st2, i, j]$
2. *Red*: Pop $[\nabla_{3,1}, st2, i, j]$ and $[A, st2, i, j]$ for $[\nabla_{3,0}, st0, i, j]$
3. *Head*: Swap $[\nabla_{3,0}, st0, i, j]$ for $[B, st1, i, j]$

The item resulting from the last transition, $[B, st1, i, j]$, 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 2 and 3 are calculated only once at the first iteration.

To continue the analysis of an input sentence, we must apply a *Shift* in state 2, pushing $[a, st3, j, j+1]$.

The shared forest can be obtained using the method described in (Lang 91), generating a rule of an *output grammar* in each transition.

7 Conclusion

We have considered Earley's algorithm as a starting point for deriving other well known parsing algorithms, such us Generalized LR. Several intermediate parsing schemata have been used in the path from Earley to LR, applying simple and intuitive transformations in each step. The resulting algorithm can be integrated in the common framework for parsing in dynamic programming proposed by Lang, achieving a $\mathcal{O}(n^3)$ complexity in the worst case

The technique proposed can be extended to deal with grammatical formalisms which have a context-free backbone. As has been shown in (Villemonte de laClergerie 93), there is a straightforward extension of automata-based context-free parsing techniques to Horn-clause analysis. The most common grammatical framework based on Horn-clauses is Definite Clause Grammars (Pereira & Warren 83). We can think of a DCG as a CFG skeleton with attributes associated to grammatical symbols. A implementation of a parser for DCGs based in our specification of LALR(1) has been described in (Vilares Ferro & Alonso Pardo 96).

An interesting facility in natural language parsers is the possibility of dynamically checking the syntax correctness of a text when it has been edited, changing the internal representation of the analysis rather than generating an entirely new one. In this sense, the algorithm presented here can be extended in order to obtain a *full incremental parser*, that is, a parser that can recover parts of older analysis when a new one is tried. With full incrementality we indicate that modifications in every point of the input string are allowed. The implementation of an incremental parser based on our specification of LALR(1) algorithm has been described in (Vilares Ferro & Dion 94).

References

(Aho & Ullman 72) Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1–2. Prentice Hall, 1972.

(Earley 70) J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

(Lang 91) Bernard Lang. Towards a uniform formal framework for parsing. In Masaru Tomita, editor, *Current Issues in Parsing Technology*, pages 153–171. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

(McLean & Horspool 96) Philippe McLean and R. Nigel Horspool. A faster Earley parser. In *Proc. of International Conference on Compiler Construction (CC'96)*, pages 281–293, Linkopen, Sweden, 1996.

(Nederhof & Sarbo 93) Mark-Jan Nederhof and J. J. Sarbo. Increasing the applicability of LR parsing. In *Proc. of Third International Workshop on Parsing Technologies*, pages 187–201, Tilburg (The Netherlands) and Durbuy (Belgium), 1993.

(Pereira & Warren 83) Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proc. of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144. ACL, June 1983.

(Rekers 92) Jan Rekers. *Parsing Generation for Interactive Environments*. Unpublished PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 1992.

(Sheil 76) B. A. Sheil. Observations on context-free grammars. In *Statistical Methods in Linguistics*, pages 71–109, Stockholm, Sweden, 1976.

(Sikkel 97) Klaas Sikkel. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Texts in Theoretical Computer Science — An EATCS Series. Springer-Verlag, Berlin/Heidelberg/New York, 1997.

(Tomita 86) Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston, MA, USA, 1986.

(Vilares Ferro & Alonso Pardo 96) Manuel Vilares Ferro and Miguel Angel Alonso Pardo. An LALR extension for DCGs in dynamic programming. In P. Lucio, M. Martelli, and M. Navarro, editors, *Proc. of APPIA-GULP-PRODE'96 Joint Conference on Declarative Programming*, pages 79–88, San Sebastián, Spain, July 1996.

(Vilares Ferro & Dion 94) Manuel Vilares Ferro and Bernard A. Dion. Efficient incremental parsing for context-free languages. In *Proc. of the 5^{th} IEEE International Conference on Computer Languages*, pages 241–252, Toulouse, France, 1994.

(Villemonte de laClergerie 93) Eric Villemonte de la Clergerie. *Automates à Piles et Programmation Dynamique. DyALog : Une Application à la Programmation en Logique*. Unpublished PhD thesis, Université Paris 7, Paris, France, 1993.