LR, LC and LL parsing, some new points of view

Wim Pijls
Erasmus University Rotterdam, P.O.Box 1738, 3000 DR Rotterdam, The Netherlands. e-mail: pijls@few.eur.nl

Abstract

In parsing theory, LL parsing and LR parsing are regarded as two distinct methods. In this paper the relation between these methods is clarified. As shown in the literature on parsing theory, we can construct an LR automaton for every grammar. This automaton has a counterpart, the so-called non-deterministic LR automaton. We show, that traversing the non-deterministic LR automaton is equivalent to LL(1) parsing. Viewing the two parsing methods as automaton traversing procedures, new elegant recursive descriptions are obtained, equivalent to existing descriptions in literature.

Having obtained a relationship between LL and LR parsing, the LL(1) class is characterised in various ways, using LR classes.

Keywords: Compiler construction, Formal languages, Parsing theory

1 Introduction

Over the years, many parsing algorithms have been developed. One usually distinguishes between tabular and non-tabular methods. In the latter group, the classical methods are LL, LC and LR parsing. LL parsing is mainly implemented by the recursive descent algorithm, which allows an elegant and understandable description. LR parsing is formulated as a stack manipulating process, which is difficult to grasp for a novice. LC parsing has a similar drawback.

In general, stack manipulating procedures emerge when recursive procedures are transformed into iterative procedures. Conversely, algorithms manipulating stacks may often be formulated in a recursive fashion. This latter property is exploited in this paper. LR and LC parsing are transformed from algorithms which explicitly deal with stacks, into recursive algorithms. Hereby, LR and LC parsing take a compact and elegant shape, similar to the recursive descent description for LL. The new LR parser also resembles a recursive ascent parser[8, 14].

LR parsing may be viewed as a process traversing the LR automaton in a particular way. Apart from an LR automaton also a so-called non-deterministic automaton is known. The counterpart of the LR traversal process for the non-deterministic automaton is constructed. The resulting process turns out to be equivalent to LL parsing.

Although a new algorithm is actually not presented in this paper, the way of formulating existing parsing algorithms gives an unifying view on those algorithms. The new versions being compact and transparent descriptions are relevant, amongst others, for educational purposes. The new theory on the three main parsing methods (LR, LC and LL) is discussed in Sections 2, 3 and 4 respectively. Section 5 compares the set LL(1) with several LR sets.

Now, some preliminaries are discussed. We consider context-free grammars \{V_N, V_T, P, Z\}, where \(V_N\) is the set of non-terminals, \(V_T\) the set of terminals, \(P\) the set of productions, and \(Z\) is the start symbol. \(V\) denotes the entire set of symbols \(V_T \cup V_N\). Greek characters denote strings consisting of symbols in \(V\), except the character \(\epsilon\) which denotes the empty string. We assume that start symbol \(Z\) occurs in just one production \(Z \rightarrow A\$\), where $ denotes the end-of-input. As a running example
in this paper, we use the following grammar $G$ consisting of the productions:

$$
Z \to X$, \\
X \to yY | vYw, \\
Y \to e | xX.
$$

The procedures in this paper include a global variable $nextinput$ referring to the next input symbol to be processed. When the main call to a parse procedure is invoked, $nextinput$ is assumed to contain the first character of the input file. The procedure $shift$ reads the next symbol of the input file and hence assigns a new value to $nextinput$.

An item is a well-known notion in parsing theory. An item is defined as a production with a dot somewhere in the right-hand side. We define some special type of items. A $reduce$ item (for instance $A \to \alpha$·) is an item, in which the dot is at the end of the item. A $null$ item (for instance $A \to \cdot$) has just a dot in its right hand side. We define the following functions working on items:

- $reduce(T)$ is a boolean function, which returns $true$ if $T$ is a reduce item, and $false$ otherwise;
- $lhs(T)$ returns the left-hand-side (the non-terminal to the left of the arrow) of $T$;
- $nextsym(T)$ gives the first symbol beyond the dot; if $T$ has the form $A \to \alpha \cdot B\beta$ for example, then $nextsym(T)$ is equal to $B$;
- $forward(T)$ returns a new item, in which the dot is moved one symbol ahead; if $T$ has the form $A \to \alpha \cdot B\beta$ for example, then $forward(T)$ equals $A \to \alpha B \cdot \beta$;
- $back(T)$ is the reverse operation of $forward(T)$.

2 LR parsing

The basic concept underlying LR parsing is an LR automaton. For our grammar $G$, the LR(0) automaton is depicted in Figure 1. We recall some facts on deterministic automata. Every state contains a set of items. In this set, so-called core items can be distinguished. In the start state $s_0$, there is by definition one core item: $Z \to X$$. In the other states, an item is a core item, if the dot is not the first character directly before the arrow. A null-item in a state, if any, is always derived from another item in the same state and is always assumed to be a non-core-item. It follows that any reduce item is either a core item or a null-item. In general, there are multiple core items in one state.

LR parsing is a particular way of traversing the LR automaton. The traversal procedure is given by the following code:

**procedure** $LRparse$($in$ $s$; $out$ $success$; boolean; $T$; item);

**select**

if $s$ contains a reduce item $T'$ then ($\uparrow$)

$T$ := $T'$;

$s$ := goto($s$, $nextinput$);

if $s$ contains a core item then

$T$ := back($T'$);

end if

**otherwise** $success$ := $false$;

while $success$ and $T$ is not a core item in $s$ do

$A$ := $lhs(T)$;

$s'$ := goto($s$, $A$);

$LRparse(s'$, $success$, $T'$);

if $success$ then $T$ := back($T'$);

end while

**end**
The meaning of $go to(s, X)$ with $s$ a state in the LR automaton and $X \in V$ is self-explanatory. The select statement in the code is borrowed from the Ada language, with a slightly different meaning however. This statement makes the algorithm non-deterministic. The select statement means, that one of the if branches must be chosen, provided that the related guard is true. In the case that no guard is true, the otherwise branch must be chosen. Since both guards may be true at a time (shift/reduce conflict), the selection of a branch is non-deterministic. Another source of non-determinism is the guard itself in (\ref{select}), because a reduce item needs not to be unique in $s$ (reduce/reduce conflict).

To decrease the number of conflicts, we may enhance the items in the LR automaton with so-called look-ahead symbols, also called context symbols. So, a set $LA_s(T)$ of terminal symbol is appended to each item $T$ in a state $s$. Two special cases of enhanced LR(0) automata are the SLR(1) and LALR(1) automaton respectively. See the literature on parsing theory for details. The set of context symbols can be taken into account by changing the line marked by (\ref{select}) into the following new line:

$$\textbf{if } s \text{ contains a reduce item } T' \text{ and nextinput } \in \text{ LA}_s(T') \text{ then }$$

As a result of this addition, the number of conflicts decreases and the procedure $LRparse$ becomes deterministic for a greater set of grammars.

The specification of $LRparse$ is given below, provided that the procedure is deterministic for the grammar under consideration. In the specification, the string previous consists of all input symbols, which have been processed (or more precisely: shifted) and inputfile denotes the symbols to be processed yet.
**Specification** of the call \( \text{LRparse}(s, \ \text{success}, \ T) \):

*pre*: for every core item \( A \rightarrow \alpha \cdot \beta \) in \( s \), \( \alpha \) matches a tail string of \( \text{previous} \);

*post*: if \( \text{success}=\text{true} \), then a core item \( A \rightarrow \alpha \cdot \beta \) in \( s \) is stored into \( T \) and a string \( r \) such that \( \beta \Rightarrow r \) has been moved from \( \text{inputfile} \) to \( \text{previous} \);

if \( \text{success}=\text{false} \), then \( s \) does not contain any core item \( A \rightarrow \alpha \cdot \beta \), such that \( \beta \Rightarrow r \) with \( r \) a start string of \( \text{inputfile} \).

As noticed before, LR parsing is formulated mostly as a stack manipulating procedure, governed by an action/goto matrix. The new algorithm applied to an LR(0) automaton is equivalent to the familiar LR(0) parsing algorithm in the sense that, for any given input string, the states of the LR(0) automaton under consideration are visited in the same order. The same statement holds when replacing LR(0) with SLR(1) or LALR(1). At any time during an \( \text{LRparse} \) call, the state parameters in the recursion stack make up the states stack occurring in the traditional LR parsing algorithm. The new formulation of LR parsing resembles the recursive ascent parser in [8, 14]. However, in that parser, a separate procedure is defined for each state. Our new algorithm also provides more insight into the meaning of the states: processing a state is completed, only when a core item has been recognized. Hence, processing a state means recognizing a core item.

Apart from the LR(0) automaton, we can also construct automata LR(1) or more generally LR(\( k \)). The above code \( \text{LRparse} \) can also be used for those automata.

### 3 LC parsing

The seminal paper on LC parsing is [15]. A clear description of LC parsing can be found in [11]. A number of sophisticated algorithms has been derived from LC parsing, see amongst others [4, 12, 7]. Since LC parsing uses a pushdown stack, we want to find a recursive formulation. The result is shown below.

**procedure** \( \text{LCparse}(\text{in } \ X:\text{nonterminal}; \ \text{out } \ \text{success}:\text{boolean}) \);

\( T := A \rightarrow \cdot \alpha, \ \text{such that} \)

\( (nextrsym(T) \in V_T \ or \ \alpha = e) \ and (A = X \ or \ A \in \text{Leftcorners}(X)) \); \hspace{5mm} (1)

\( \text{success} := \text{true}; \)

**while** \( \text{success} \) **do** \n
\( T' := T; \)

**select** \n
\( \text{if reduce}(T') \ and \ X = \text{lhs}(T') \ \text{then exit procedure}; \) \hspace{5mm} (2)

\( \text{if reduce}(T') \ \text{then } T := A \rightarrow B \cdot \gamma, \ \text{such that} \)

\( B = \text{lhs}(T') \ and (A = X \ or \ A \in \text{Leftcorners}(X)) \) \hspace{5mm} (3)

\( \text{nextrsym}(T') \in V_T \ and \ \text{nextrsym}(T') = \text{nextrinput} \ \text{then} \)

\( T := \text{forward}(T'); \)

**shift** \n
\( \text{if nextrsym}(T') \in V_N \ \text{then} \)

\( \text{LCparse} (\text{nextrsym}(T'), \ \text{success}); \)

\( \text{if success} \ \text{then } T := \text{forward}(T'); \)

**otherwise** \( \text{success} := \text{false}; \)

The pushdown stack in [11] contains non-terminals and items, corresponding to the non-terminal parameter \( X \) and the local variables \( T \) and \( T' \) of the type \textit{item} in the code of \( \text{LCparse} \). The goal of the procedure \( \text{LCparse}(X, \ \text{success}) \) is to recognize \( X \). If this procedure call ends with \( \text{success}=\text{true} \), then \( X \Rightarrow r \) with \( r \) a start string of \( \text{inputfile} \). To parse the entire inputfile, a call \( \text{LCparse}(Z, \ \text{success}) \) with \( Z \) the start symbol must be performed.
To understand the code of \textit{LCParse}, one needs the notion of \textit{left corner}. A non-terminal \( Y \) is a \textit{direct left corner} of a non-terminal \( X \), if a production \( X \to Y \gamma \) exists. The transitive closure of this relation is also used and is called an \textit{indirect left corner}. The set of all (direct and indirect) left corners of \( X \) is denoted by \( \text{Leftcorners}(X) \).

Obviously, at least one item \( T \) satisfying the required conditions can always be found in statement (1). Notice that neither (1) nor (3) specifies a unique choice for \( T \). Hence, the procedure is non-deterministic. Fortunately, context symbols or look-ahead symbols can be utilized in practice. We will explain how look-ahead symbols can be exploited. In case an item \( T \) with \( \text{nextsym}(T) \in V_T \) is chosen in (1), the execution of the procedure call proceeds only if \( \text{nextsym}(T) = \text{nextinput} \). (The reader is invited to check that, if \( \text{nextsym}(T) \neq \text{nextinput} \), the otherwise branch causes an immediate unsuccessful stop.) In case an item \( T \) with \( \alpha = \epsilon \) is chosen in (1), the procedure will be able to terminate successfully only if \( \text{nextinput} \in \text{Follow}(\gamma) \). Similarly, an item \( T = A \to B \cdot \gamma \) to be chosen in (3) is sensible only if \( \text{nextinput} \in \text{First}(\gamma) \) or, \( \epsilon \in \text{First}(\gamma) \) and \( \text{nextinput} \in \text{Follow}(A) \). Nevertheless, even using context symbols the choice for \( T \) in (1) or (3) may still be non-deterministic.

There is a second source of non-determinism. A collision between the statements (2) and (3) respectively may happen. If this is the case, then \( \text{lhs}(T') = X \) by (2) and, moreover by (3), an item of the form \( A \to X \cdot \gamma \) can be found. From the condition \( A = X \) or \( A \in \text{Leftcorners}(X) \), we conclude that the grammar under consideration is left-recursive. We give an example of such a collision. Let a grammar be given with productions \( V \to vVu \) and \( U \to u \mid Uu \). During execution of a subcall \textit{LCParse}(\( U, \text{success} \)), the first values for \( T \) are equal to \( U \to u \) and \( U \to u \) respectively. As long as \( \text{nextinput} = u \), \( T \) takes alternately the values \( U \to U \cdot u \) (according to (3)) and \( U \to Uu \). Whenever \( T \) equals \( U \to Uu \), the guard in (2) is also true. If look-ahead symbols are considered, this guard is chosen only when \( \text{nextsym} = w \).

4 Non-deterministic automata and LL parsing

Beside a deterministic automaton, also a non-deterministic automaton (here abbreviated as \textit{NA}) can be constructed, cf. [5, 16]. The \textit{NA} for the grammar \( G \) is depicted in Figure 2. An \textit{NA} relates to an LR(0)-automaton as an NFA does to an FSA. The latter types of automata are intended to process regular expressions. Their definitions can be found in the literature on formal language theory or compiler constructions. The transitions in an \textit{NA} are accompanied by one symbol, which is an element of \( V \cup \{ \epsilon \} \). Since there is a one-to-one correspondence between states and items, each \textit{NA} state can be identified with one item. The traversal through the automaton is expressed by the procedure:

\begin{verbatim}
procedure NAParse(in T:item, out success:boolean);
if reduce(T) then success:=true;
else if nextsym(T) \in V_T and nextsym(T) = nextinput then
    T\':=forward(T);
    shift;
    NAParse(T\', success);
end
else if nextsym(T) \in V_N then
    T\':=an item A \to \alpha reached by an \epsilon edge; (*)
    NAParse(T\', success);
    if success then T\':=forward(T\');
    NAParse(T\', success);
else success:=false;
\end{verbatim}

The input parameter is an item, or equivalently a state including an item. For grammar \( G \), an input file is parsed by the call \textit{NAParse}(\( T, \text{success} \)) with \( T \) equal to \( Z \to \cdot X \$ \). The specification of \textit{NAParse} is similar to that of \textit{LCParse}, regarding the single item in each state \( s \) as a core item.
Figure 2: A non-deterministic automaton.

In the lines marked by (1) and (3) in the code of NAParse, there are recursive subcalls to NAParse. Here, we have instances of tail recursion. In general, tail recursion can be eliminated easily (without introducing stacks). It can be replaced by an iterative control structure. When this transformation is applied to NAParse, we obtain the procedure LLparse. The recursive call at (2) is not tail recursive and hence, for this subcall, the recursion is not eliminated.

**procedure** LLparse(in T:item, out success:boolean);

success:=true;

while success do

  if reduce(T) then exit procedure;

  else if nextsym(T) ∈ V₇ and nextsym(T) = nextinput then

    T := forward(T);
    shift;

  else if nextsym(T) ∈ V₈ then

    T' := an item A → ·α with A=nextsym(T); (*)
    LLparse(T', success);

  if success then T := forward(T);

  else success:=false;

Notice that the lines marked by (*) in both procedures have equivalent left-hand sides describing the new value of T'. The transformed procedure is the well-known recursive descent parser, formulated in terms of items. So LL parsing is re-covered.

It appears that notions like *shift* and *reduce* and conflicts between them also occur in LL parsing. In literature, these notions seem to be typical to LR parsing.

The statements marked by (*) in above procedures, where an item A → ·α has to be chosen,
are non-deterministic. We may insert an extra requirement for $T'$ to be chosen in the line marked by $^*$. The most obvious requirement is: $nextinput \in First(\alpha)$. If $\varepsilon \in First(\alpha)$, then also look-ahead symbols should be taken into account. The look-ahead symbols in a non-deterministic automaton are the $Follow$ sets. Consequently, if $\varepsilon \in First(\alpha)$, also the condition $nextinput \in Follow(A)$ has to be checked. A shorthand for the above requirements is: $nextinput \in First(\alpha \cup Follow(A))$. An equivalent way to enforce the check is the following addition at the place of $^*$: choose $T'$ such that the related production $A \rightarrow \alpha \cdot \beta$ is the production for the entries $A$ and $nextinput$ in the $LL(1)$ matrix. Adding the above requirements to the procedures $MAParse$ (and similarly to $Liparse$) makes those procedures deterministic for $LL(1)$ grammars.

5 Comparing $LL(1)$ with $LR$-classes

Given a grammar $G$, $N_G$ denotes the NA (non-deterministic automaton) of $G$ and $D_G$ denotes the deterministic $LR(0)$ automaton, which is the identical to the $SLR(1)$ or the $LALR(1)$ automaton apart from the look-ahead set. In this section we try to characterize $LL(1)$ in terms of $LR(0)$, $LALR(1)$ and $LR(1)$.

We assume, that, for each item $T$ in a state $s$ of $D_G$, the $LALR(1)$ look-ahead set is available. This set was abbreviated as $LA_s(T)$ in Section 2. For the remainder of this section, we need two new notions. Let $T$ denote an item with the form $A \rightarrow \alpha \cdot \beta$ included in in a state $s$ of the $D_G$. The notions $F'(T)$ and $F_s(T)$ are defined as follows:

$$F'(T) = First(\beta \cup Follow(A))$$
$$F_s(T) = First(\beta \cup LA_s(T))$$

We can consider the subgraph in $N_G$ generated by the items of a state $s$ in $D_G$. This subgraph is denoted by $H(s)$. In other words, $H(s)$ with $s$ a state in $D_G$ is the underlying graph of $s$. Since $N_G$ is basically a directed graph, so is $H(s)$. Each node in a graph $H(s)$ can be identified with an item $I$. Suppose a node $I$ has successors $I_1, I_2, \ldots, I_k$ in $H(s)$. Then

$$F_s(I) = \bigcup F_s(I_i) \{ I_i \text{ a successor of } I, 1 \leq i \leq k \}. \quad (1)$$

5.1 An alternate definition for the $LL(1)$ property

For an item $T$ of the form $A \rightarrow \alpha$ the following general equality holds:

$$Follow(A) = \bigcup \{ LA_s(T) \mid s \text{ in } D_G \text{ and } T \text{ in } s \} \quad (2)$$

Let $T_1$ and $T_2$ be two items with the form $A \rightarrow \alpha_1$, and $A \rightarrow \alpha_2$ respectively. Then $T_1$ and $T_2$ are called siblings. Notice that, if an item $T$ of the form $A \rightarrow \alpha$ is included in a state $s$, every item $T'$, such that $T$ and $T'$ are siblings, is included in $s$ as well.

**Lemma 1** Let $T_1$ and $T_2$ be two siblings. Then $F'(T_1) \cap F'(T_2) = \emptyset$, iff $F_s(T_1) \cap F_s(T_2) = \emptyset$ in every state $s$ including $T_1$ and $T_2$.

**Proof**

Let $\alpha_1$ and $\alpha_2$ denote the right-hand side of $T_1$ and $T_2$ respectively. Both items have the same left-hand side, say $A$. We distinguish three cases.

If $\varepsilon \notin First(\alpha_1)$ and $\varepsilon \notin First(\alpha_2)$ then we have for every $s$ $F'(T_1) = F_s(T_1)$ and $F'(T_2) = F_s(T_2)$.

If $\varepsilon \notin First(\alpha_1)$, but $\varepsilon \in First(\alpha_2)$, then we have for every $s$ that $F'(T_1) = F_s(T_1) = First(\alpha_1)$, $F'(T_2) = (First(\alpha_2) \cup Follow(A))$ and $F_s(T_2) = (First(\alpha_2) \cup LA_s(T_2))$. Using (2), the lemma is proved.

If $\varepsilon \in First(\alpha_1)$ and $\varepsilon \in First(\alpha_2)$, then $Follow(A)$ is in $F'(T_1)$ as well as in $F'(T_2)$. In general, $LA_s(T_1) = LA_s(T_2)$ for any two siblings $T_1$ and $T_2$ in any state $s$. This set is also included in both
$F_s(T_1)$ and $F_s(T_2)$. We conclude that neither of the intersection sets is empty. □

It is easily seen that the definition of LL(1) amounts to the condition: $F'(T_1) \cap F'(T_2) = \emptyset$ for any two siblings $T_1$ and $T_2$. By Lemma 1, there is another equivalent condition:

$$F_s(T_1) \cap F_s(T_2) = \emptyset$$

for any two siblings $T_1$ and $T_2$ in any state $s$.

We will establish a link between this LL(1) condition and the LALR(1) and LR(0) definition. This will be achieved in Theorems 1 and 4 respectively.

5.2 Some definitions and a property of LALR(1)

An item $T$ in $H(s)$, such that $T$ hasn’t any outgoing edges in $H(s)$, is called an end item of $H(s)$. Notice that, an end item is either a reduce-item or a so-called shift item, which is defined as an item of the form $A \rightarrow \alpha \cdot a \beta$ with $a \in V_T$.

In addition to shift/reduce or reduce/reduce conflicts, we also define shift/shift conflicts. A state $s$ has a shift/shift conflict, if $s$ contains two shift items of the form $A \rightarrow \alpha \cdot a \beta$ and $A' \rightarrow \alpha' \cdot a' \beta'$ with $a \in V_T$. We might say, that a shift/shift conflict does not look like a conflict, because its solution is moved ahead to a next state.

We now come to an important property of LALR(1) grammars, which will be referred to in the subsequent subsections. The absence of both shift/shift conflicts and LALR(1) LALR(1) (reduce/reduce or shift/reduce) conflicts in a state $s$ is equivalent to the relation $F_s(I_1) \cap F_s(I_2) = \emptyset$ for any two end items $I_1$ and $I_2$ in $s$.

5.3 Comparing LL(1) with LALR(1)

Now we will achieve some results on the relation between LL(1) on the one hand and LALR(1) on the other.

**Lemma 2** In any state $s$ in $D_G$ ($G$ a given grammar) the following propositions are equivalent:

a) $s$ has no successor states or its successor states have exactly one core item;

b) $s$ has no shift/shift conflicts and each node of $H(s)$ has at most one incoming edge.

**Proof**

In this proof a node in $H(s)$ will be identified with its item included. The equivalence a) $\iff$ b) is proved by contradiction, i.e., the equivalence $\neg a) \iff \neg b)$ is proved.

There is a successor state with core items $C \rightarrow \alpha B : \beta$ and $C' \rightarrow \alpha' B' : \beta'$, $B \in V$, if and only if there are two items in $H(s)$ of the form $C \rightarrow \alpha \cdot \beta$ and $C' \rightarrow \alpha' \cdot \beta'$. $H(s)$ contains two such items, if and only if (in case $B \in V_N$) an item in $H(s)$ with left-hand side $B$ has two incoming edges, or (in case $B \in V_T$) a shift/shift conflict occurs in $H(s)$. □

**Single core grammars.** In the sequel of this section we are concerned with states with exactly one core item. Given a single core grammar $G$, the start state in $D_G$ (=the state containing core item $Z \rightarrow \alpha$ with $Z$ the start symbol of $G$) has exactly one core item. Since every other state of $G$ has exactly one core item as well, we conclude from Lemma 2, that, for every $s$, $s$ has no shift/shift conflicts and each node in $H(s)$ has at most one incoming edge. The fact that each node of $H(s)$ has at most one incoming edges in combination with the single core characteristic of $s$ implies that $H(s)$ has the shape of a tree. As a result of (1) we have in a tree $H(s)$ with root $r$ that the $F_s$-sets of the leaves make up a partition of $F_s(r)$, iff in every inner node $n$ of $H(s)$ the $F_s$-sets of the children of $n$ make up a partition of $F_s(n)$. The children of an item $I$ in a tree $H(s)$ are siblings. Hence, we may also say for a state $s$ with $H(s)$ a tree: $F_s(I_1) \cap F_s(I_2) = \emptyset$ for any two end items $I_1$ and $I_2$, iff $F_s(T_1) \cap F_s(T_2) = \emptyset$ for any two siblings $T_1$ and $T_2$. 

8
Theorem 1 Let a single core grammar $G$ be given. Then $G$ is in LL(1) iff $G$ is in LALR(1).

Proof

if part. Since $s$ doesn’t exhibit any LALR(1) conflicts or shift/shift conflicts, $F_s(I_1) \cap F_s(I_2) = \emptyset$ for any two end items $I_1$ and $I_2$ in every $s$. We argued above for a single core grammar that this property implies that $F_s(T_1) \cap F_s(T_2) = \emptyset$, which is in turn equivalent to the LL(1) characteristic.

only if part. The LL(1) characteristic says that $F_s(T_1) \cap F_s(T_2) = \emptyset$ for any two siblings $T_1$ and $T_2$ in $s$, which is equivalent to $F_s(I_1) \cap F_s(I_2)$ for two given end items $I_1$ and $I_2$ in $s$. Hence $G$ is in LALR(1). □

In the literature on parsing theory, a non-terminal $V$ is called a marker, if $\text{First}(V)$ consists of just $\epsilon$. In addition to the notion of a marker, we introduce the following notion. A marker $A$ is called a disturbing marker, if $A$ occurs in the right hand side of at least two productions. Besides, we define the notion of a semi-marker. A symbol $A \in V_N$ is called a semi-marker, if $\text{First}(M)$ contains $\epsilon$ and at least one terminal.

Lemma 3 If an LL(1) grammar $G$ is not a single core grammar, then $G$ has a disturbing marker.

Proof

Lemma 2 says that there is a state $s$, where proposition b) of lemma 2 does not hold. Let $s$ be chosen, such that proposition b) does hold in every state $s'$ preceding $s$. Then $s$ has one core item. At least one item in $H(s)$, say $I_0$, has two incoming edges. Let $I_0$ be of the form $B \rightarrow \beta$. Since $I_0$ has two incoming edges, $B$ occurs in the right-hand side of at least two productions. Suppose $\text{First}(B) \neq \emptyset$. The $F_s$-set of item $I_0$ and every (direct or indirect) predecessor of $I_0$ in $H(s)$ contains the terminals of $\text{First}(B)$. Since $I_0$ has two incoming edges, there are two paths from the core item in $s$ to $I_0$. At the bifurcation, we have siblings $T_1$ and $T_2$ with $F_s(T_1) \cap F_s(T_2) \neq \emptyset$. Then the LL(1) characteristic is violated. We conclude that $\text{First}(B) = \{\epsilon\}$ and, since $B$ occurs twice in any right-hand sides, $B$ is a disturbing marker. □

As a consequence of Theorem 1 and Lemma 3 the following proposition holds: within the set of grammars without disturbing markers the LL(1) set is equal to the set of single core LALR(1) grammars.

The relation between LALR(1) and LL(1) is illustrated in Figure 3. The larger box represents LALR(1) and the rectangular box consisting of a light gray and a dark gray segment represents LL(1). The dark gray area in the LL(1) box corresponds to the subset of single core grammars and the light gray area is the subset of non-single core grammars. Notice that, by Lemma 3, every grammar in LL(1) in the light gray area has a disturbing marker. By Theorem 1 the dark gray segment also represents the subset of LALR(1) of single core grammars. For illustration, we consider two
LL(1) grammars $G_1$ and $G_2$ each of which has an automaton with a state including at least two core items. $G_1$ belongs to the right lightgray segment inside LALR(1) and $G_2$ belongs to the left lightgray segment outside LALR(1).

$$
G_1: \quad Z \rightarrow Y \$$

$$
Y \rightarrow Xv | Xw \quad X \rightarrow aA | bB
$$

$$
X \rightarrow U \quad A \rightarrow Cc | Dd
$$

$$
U \rightarrow \varepsilon \quad B \rightarrow Cd | Dc
C \rightarrow E
D \rightarrow E
E \rightarrow \varepsilon
$$

In $G_1$, we have $X$ as a disturbing marker, in $G_2$ we have $C$, $D$ and $E$. In $G_2$ a conflict occurs in the state with core items: $C \rightarrow E$ and $D \rightarrow E'$, each with the look-ahead set $\{c,d\}$.

### 5.4 Comparing LL(1) with LR(0) and with LR(1)

LR(0) is a subset of LALR(1). The relationship between LL(1) and LR(0) is depicted in Figure 4. In this figure LL(1) is represented in the same way as in figure 3, i.e., by a rectangular box with a darkgray and a lightgray segment. The following lemma says that the subset of LL(1) grammars without $\varepsilon$-productions is embedded in the intersection of the LR(0) set and the darkgray segment.

**Lemma 4** If a grammar $G$ is in LL(1) and $G$ has no $\varepsilon$-productions, then $G$ is a single core grammar in LR(0).

**Proof**

As a result of Lemma 3, an LL(1) grammar without $\varepsilon$-productions is a single core grammar. In a state with exactly one core item, a reduce item involved in an LR(0) conflict must be a null item. In the absence of $\varepsilon$-productions, there are no null items. □

Lemma 4 has been published before in [3] and [6]. In the same papers, Theorem 1 can be found in a weaker form.

Theorem 1 also applies to LR(1) (replacing LALR(1)). The proof is similar. Adding disturbing markers to an LR(1) grammar doesn’t affect its nature. It follows that every LL(1) grammar is in LR(1). However, for a characterization of LL(1) as an LR(1) subclass, we need to characterize the disturbing markers, that generate states with more than one core item. This is subject to further research.
6 Concluding remarks

The LR algorithm, formulated in literature as a stack manipulating algorithm has been presented in this paper as an elegant and transparent recursive algorithm. Moreover, we have discovered an important insight that is hidden to the reader of the classical stack description: visiting a state aims at recognizing a core item of that state.

The LR parsing is known to be a traversal through an deterministic automaton. The counterpart for a non-deterministic automaton is equivalent to an item-based version of LL parsing. The approach in this paper has also value, when parsing algorithms are implemented by functional programs, cf [10].

To our experience in teaching parsing theory, students appreciate this approach, which presents LL and LR parsing uniformly: as automata traversing processes described recursively.

References


