Unifying LL and LR parsing

Wim Pijls
Erasmus University Rotterdam, P.O.Box 1738,
3000 DR Rotterdam, The Netherlands.
wimp@cs.few.eur.nl

Abstract

In parsing theory, LL parsing and LR parsing are regarded to be two distinct methods. In this paper the relation between these methods is clarified.
As shown in literature on parsing theory, for every context-free grammar, a so-called non-deterministic LR(0) automaton can be constructed. Here, we show, that traversing this automaton in a special way is equivalent to LL(1) parsing. This automaton can be transformed into a deterministic LR-automaton. The description of a method to traverse this automaton results into a new formulation of the LR parsing algorithm. Having obtained in this way a relationship between LL and LR parsing, the LL(1) class is characterised, using several LR-classes.

1 Introduction

In the theory of parsing, the two main methods are LL and LR parsing respectively. The LL method is implemented mostly by the recursive descent technique. LR parsing is implemented mainly as a stack algorithm, governed by a so-called action/goto-matrix representing the LR automaton. In a lot of text books on parsing or formal language theory, both methods are explained extensively.
For every context-free grammar, a deterministic LR-automaton can be built. LR parsing is a special way of traversing this automaton. See e.g. [Aho], [Grune] or [Sippu]. Besides, also a non-deterministic LR-automaton can be constructed, see e.g. [Grune] or [Sippu]. (Both kinds of automatons differ from those, which are used to recognize a regular language.) The main result of our paper is the observation that LL(1) parsing is equivalent to a way of traversing the non-deterministic automaton, whereas LR parsing is equivalent is equivalent to traversing the deterministic automaton. This relationship is exploited to derive correspondances between LL(1) and LR grammar classes.

Now, we discuss some preliminaries. 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 (or tokens), P the set of productions, and Z is the start symbol. V^* and V_T^* denote the set of strings consisting of symbols in V and V_T respectively. A greek symbol will denote an element of V^*. We demand that the start symbol Z occurs in only one

production, $Z \rightarrow A\$$, where \$ denotes the end-of-input. For simplicity, we will define a grammar by its productions.
We assume that in each parsing algorithm a global variable *previous* is declared, referring to strings in V_T^*. Furthermore, a procedure *shift* is assumed to be declared which takes the next symbol from the input file and appends this symbol to *previous*. So the string in the variable *previous* consists of all symbols already proceeded. The variable *nextsymbol* refers at each time to the next symbol to be proceeded. Hence the procedure *shift* affects the value of *nextsymbol*.

The following grammar G will be used as running example in this paper.

$$\begin{array}{lcl} Z & \rightarrow & X\$ \\ X & \rightarrow & yY \mid vYw \\ Y & \rightarrow & xX \mid \epsilon \end{array}$$

In Section 2, we recapitulate the non-deterministic automaton. We will show that recursive descent parsing for LL(1) grammars is a special walkthrough of such an automaton. In Section 3 we consider the deterministic automaton. We present a description of traversing such an automaton, whereby we achieve a new formulation of LR parsing. In Section 4, correspondances between some grammar classes are discussed.

2 Non-deterministic automatons and LL(1) parsing

In this section, we recapitulate the notion of a non-deterministic automaton (here abbreviated as NA), as discussed earlier in among others [Grune] and [Sippu]. We will show, that a new parsing algorithm is obtained by traversing the automaton in a special fashion. This new algorithm is equivalent to the recursive descent algorithm.

For every context-free grammar an NA can be constructed. An NA is a directed graph, whereby the nodes and arcs are called states and transitions respectively. An outgoing arc of a state s is also called an *exit* of s. This situation is similar to NFA's, used for regular grammars [Aho]. The transitions in an NA are accompamied by one symbol, which, different from NFA's, may be a non-terminal. In general, each transition is marked by an element of $V_T \cup V_N \cup \{\epsilon\}$. Each state contains an item with the general form $A \rightarrow \alpha \cdot \beta$, where $A \rightarrow \alpha\beta$ is a production. So, there is a one-to-one correspondance between the states and the items, and hence, one state can be identified with one item. An item of the form $A \rightarrow \cdot\epsilon$ is called an ϵ-item. Items with $\beta = \epsilon$ or ϵ-items are called reduce-items. The transitions are determined by the following rules.

- If $\beta = b\gamma$ with $b \in V_T$, there is only one exit, to a state with item $A \rightarrow \alpha b \cdot \gamma$.
- If $\alpha = \epsilon$ and $\beta = B\gamma$, (hence the item has the form $A \rightarrow \cdot B\gamma$), then there are outgoing ϵ-transitions, one for each production $B \rightarrow \delta$, ending at a state with item $B \rightarrow \cdot\delta$. Furthermore, there is also an exit, marked by B, to a state with item $A \rightarrow B \cdot \gamma$.

- A state with a reduce-item has no exits.

For a state s and a symbol $X \in V$, $X \neq \epsilon$, *goto*(s, X) denotes the state at the end of the transition from s, marked by X. Because of the ϵ-transitions, the automaton is called non-deterministic. For grammar G, the resulting NA is depicted in Figure 1.

After recapitulating the notion of a non-deterministic automaton, we now introduce some enhancements in order to obtain a deterministic algorithm. The states or items are labeled with *First* and *Follow* sets. For the definition of *First*(β) with $\beta \in V*$, or *Follow*(A) with $A \in V_N$, used in Definition 1, we refer to [Aho, Grune].

Definition 1 *For a state s containing an item T of the form $A \rightarrow \alpha \cdot \beta$, we define First$(s)$ = First(T) = First(β) and Follow(s) = Follow(T) = Follow(A).*

We will define a walk through the automaton. For this walk, the following interpretation holds: if we are in a state with item $A \rightarrow \alpha \cdot \beta$, we are attempting to recognize $A \rightarrow \alpha\beta$, whereby α has already recognized and β is still to be recognized; choosing some ϵ-exit from an item $A \rightarrow \alpha \cdot B\gamma$ to a new state containing $B \rightarrow \cdot\delta$ means that we guess that a string r in the front of remaining input matches δ and consequently B. The fashion in which the automaton has to be walked through, is expressed formally by the procedure *parse*, presented in Figure 2. This procedure has one input parameter s of the *state* type, or alternatively, of the *item* type, because states and items can be identified in a non-deterministic automaton. There is one output parameter, *success*, of the type boolean. In the formal specification of *parse*, we assume that the value of *previous* on entry and on exit is denoted by *previous*$_1$ and *previous*$_2$ respectively. Furthermore, in the specification and in the code of *parse*, the item in s is assumed to be of the form $A \rightarrow \alpha \cdot B\gamma$. The formal specification of *parse* is as follows.

pre: α matches a tail string of *previous*$_1$;

post: if *succes=true*, then a string $r \in V_T^*$ exists, such that *previous*$_1$+r =*previous*$_2$
and $B\gamma \overset{*}{\Rightarrow} r$;
if *success=false* then such a string r does not exist.

The algorithm may be non-deterministic, due to the fact that more than one ϵ-transition can be chosen in some state. The absence of any non-deterministic choice between ϵ-transitions is equivalent to the LL(1)-property.

Theorem 1 *The code of the procedure parse is correct for LL(1) grammars.*

Proof(sketch)
First, we prove partial correctness, i.e. the the procedure meets the specification, provided that every call terminates. Second, termination is proven.
It can be shown easily that the precondition holds for every subcall in the body of *parse*, provided that the precondition is satisfied for the main call and the postcondition is satisfied for each former subcall. Furthermore, it can also be

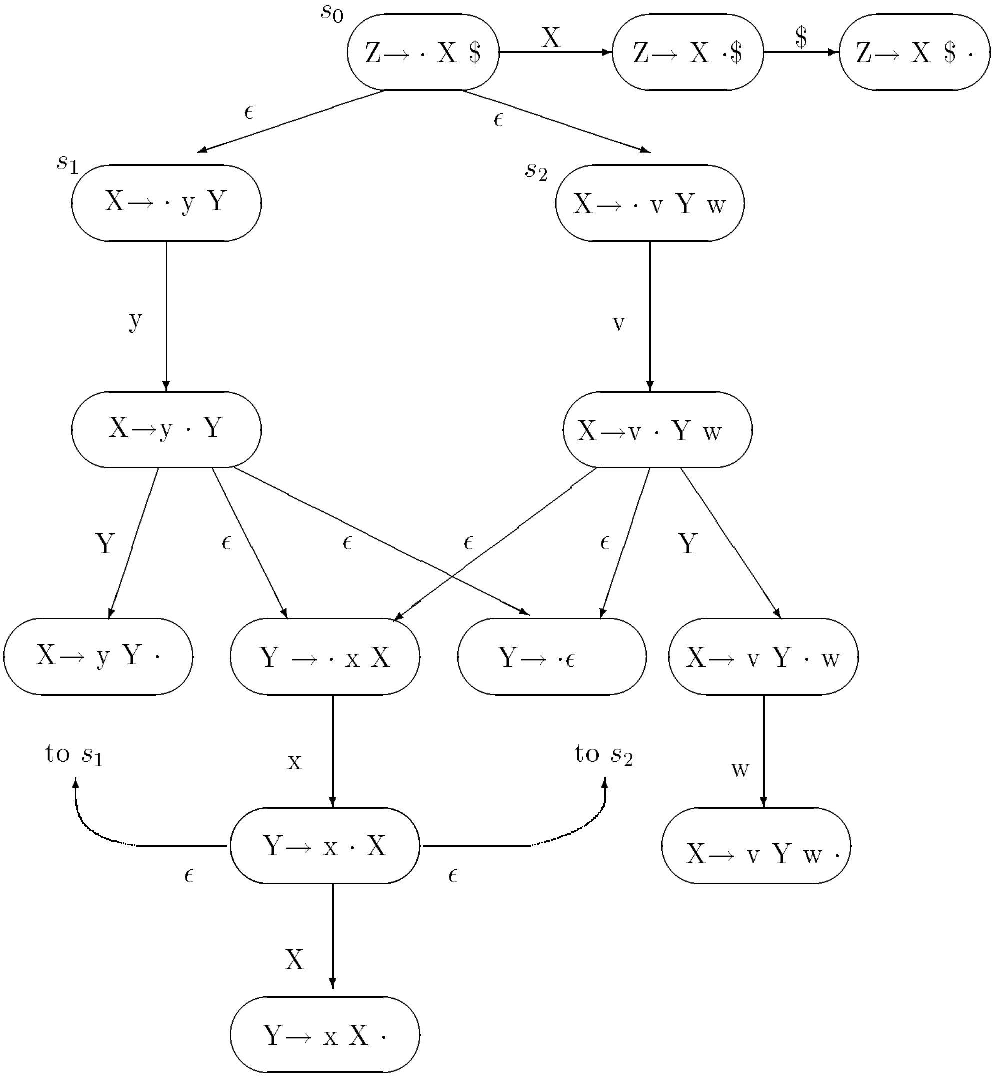

Figure 1: A non-deterministic automaton.

```
procedure parse(in s:state, out success:boolean);
if the item in s is a reduce-item then
     success:=true;
else if $B \in V_T$ and $B$ =nextsymbol then
     [ s':= goto (s, nextsymbol);
       shift;
       parse(s', success);
     ]
else if $B \in V_T$ and $B \neq$ nextsymbol then
     success:=false;
else if $B \in V_N$ then
     [ s':= a new state reached via an $\epsilon$-transition,
            such that nextsymbol in First(s'),
            or $\epsilon$ in First(s') and nextsymbol in Follow(s');
       parse(s', success);
       if success then parse(goto(s, B), success);
     ]
```

Figure 2: Traversing the non-deterministic automaton

shown easily, that the postcondition holds for the main call, if the postcondition is satisfied for every subcall. So, partial correctness is proven.
Now we prove termination. If the execution does not terminate, we have cycling in the the recursion, i.e. states in a cycle of the graph are parameter in a *parse* call repeatingly and no edge in the cycle is marked by a terminal symbol. At least one state in the cycle contains an item of the form $A \rightarrow \cdot\gamma$. It follows that a derivation exists: $A \overset{*}{\Rightarrow} \alpha A \beta$ and $\alpha \overset{*}{\Rightarrow} \epsilon$, where α is a string consisting of the symbols marking the edges of the cycle. Therefore, if the execution does not terminate, the grammar exhibits so-called (hidden) left recursion. Such a grammar gives conflicts and does not belong to LL(1). □

Parsing an input string is equivalent to invoking *parse*(s_0, *success*), where s_0 denotes the state with item $Z \rightarrow \cdot A\$$. It is trivial that the precondition is satisfied for this call. When this call terminates and *success=true*, a string or file matching $A\$$ has been recognized. It follows, that the entire input has been recognized, because the \$-sign is assumed to occur only at the end of an input file or input string. For LL(1)-grammars the parsing algorithm of this section is equivalent to the recursive descent algorithm.

3 Deterministic automatons and LR parsing

The non-deterministic automaton can be transformed into a deterministic one. This transformation is identical to the fashon in which an NFA for scannning regular expressions is converted into a DFA [Aho]. The resulting automaton is a so-called LR automaton, cf. [Aho, Grune, Sippu]. For our grammar G, this LR automaton is shown in Figure 3.

Every state contains a set of items. In such a set, so-called core items can be distinguished, cf. [Aho]. In s_0, the start state, there is by definition one core item: $Z \rightarrow \cdot X\$$. In the other states, those items where the dot is not at the start of the right hand side, are defined to be core items. It follows that every reduce-item is either a core item or an ϵ-item. Notice that an ϵ-item is never a single item in a state. As we will show in the next section, for an LL(1)-grammar, every state in the LR-automaton, contains exactly one core item.

In a state, reduce/reduce or shift/reduce conflicts can occur, see [Aho, Grune, Holub]. To diminish the number of occurrences of such a conflict, additional conditions can be formulated, under which reduction of an item is permitted. According to the strength of these conditions, several classes are distinguished, viz. LR(0), SLR(1) and LALR(1), where LR(0) $\subseteq$ SLR(1) $\subseteq$ LALR(1). A grammar belongs to LR(0), SLR(1) or LALR(1), if no state of the automaton has conflicts, using the LR(0), SLR(1) or LALR(1) condition respectively for permitting reductions.

The parse procedure of the previous section can be transformed into a new procedure, appropriate for deterministic automatons. The new body is shown in Figure 4. We assume that the procedure is applied only to conflict-free automatons, i.e. conditions for permitting reductions are utilized, such that no conflicts are left. There is one extra output parameter T, of the type item. T' is a local variable of the same type. T has an output value only if *success=true* on exit. For a core item T of the form $A \rightarrow \alpha B \cdot \gamma$, *pred*($T$), the predecessor of T, is defined as the item of the form $A \rightarrow \alpha \cdot B\gamma$. Hence *pred*($T$) is not defined for the core item in the start state $Z \rightarrow \cdot A\$$. The value of the call *left-hand-side*(T) with T an item, is a non-terminal to be stored into the local variable N. The meaning of this function is self-explanatory. The formal specification of the call *parse*(s*, succes,* T) is the following.

pre: For every core item $A \rightarrow \alpha \cdot \beta$ in s, α matches a tail string of $previous_1$

post: if *succes=true*, then T is a core item; let T be given by $A \rightarrow \alpha \cdot \beta$; a string $r \in V_T^*$ exists, such that $\beta \overset{*}{\Rightarrow} r$ and $previous_1 + r = previous_2$;
if *success=false*, a core item T and a string r with the above property do not exist.

Theorem 2 *The code of the procedure parse is correct for grammars in LR(0), SLR(1) and LALR(1), in case the reduction criterion for LR(0), SLR(1) and LALR(1) respectively is applied.*

Proof(sketch)
The structure of this proof is identical to that of Theorem 1. Partial correctness, is proven similarly.

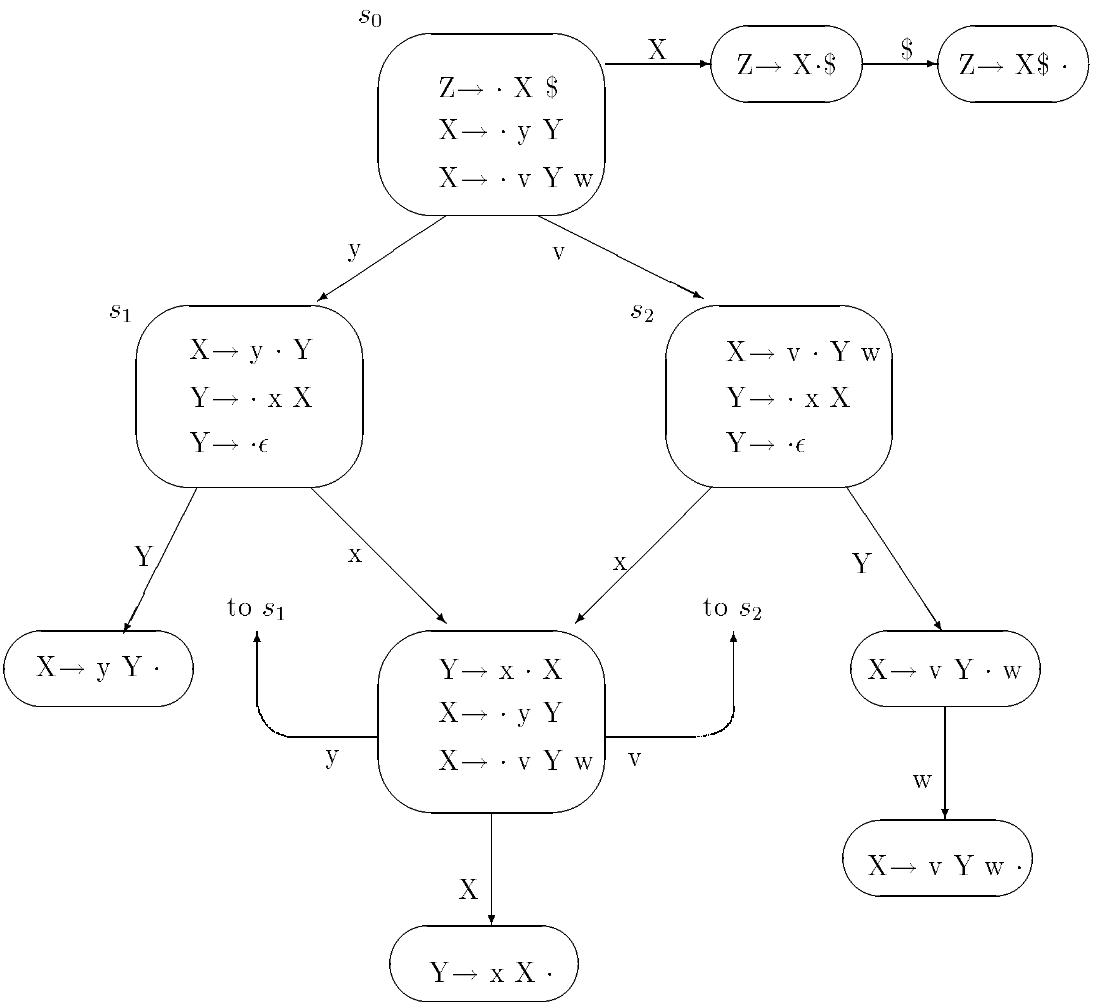

Figure 3: A deterministic automaton

```
procedure parse(in s:state; out success:boolean, T:item);
if reduction of an item T' of s is permitted then
     [ success:=true;
       T:=T';
     ]
else if there is an exit marked with nextsymbol then
     [ s':=goto(s,nextsymbol);
       shift;
       parse(s', success, T');
       if success then T:=pred(T');
     ]
else success:=false;
while success and T is not a core-item in s do
     [ N:=left-hand-side(T);
       parse(goto(s,N), success, T');
       if success then T:=pred(T');
     ]
```

Figure 4: Traversing the deterministic automaton

There are two causes for the procedure to run infinitely. First the recursion cycles, and second, the same item T' is returned twice by a subcall *parse* in the while loop, without shifting any symbol in the mean time. In the first case, there is a derivation $A \overset{*}{\Rightarrow} \alpha A \beta$ with $\alpha \overset{*}{\Rightarrow} \epsilon$, but $\alpha \neq \epsilon$, where α is equal to the string consisting of the symbols along the edges in the cycle. A grammar, which admits such a derivation, is ambiguous and gives conflicts in the aforementioned LR classes. In the second case, a non-terminal N with $N =$*left-hand-side*(T') is recognized twice, without shifting symbols in the mean time. It follows that a derivation of the form $N \overset{*}{\Rightarrow} N$ exists. Grammars with such a derivation are not taken into account. □

The call *parse*$(s_0, success, T)$ is equivalent to the LR parsing algorithm. The state parameters in the recursion stack during execution of this call form actually the states stack of LR parsing. For an LR(1) automaton, the code of Figure 4 is useful too. In the code of Figure 4, reduce has priority in case of a shift/reduce conflict. By interchanging the first two *if*-statements, the priority is interchanged.

4 Characterisations of LL(1), using LR classes

In this section, we will investigate the following problem: *for which LR reduction criterion is a deterministic automaton of a grammar G conflict-free, provided that the NA of G is conflict free.* Hence, we will utilize the LL(1) property in order to characterize some LR classes, because the NA of G is conflict free if an

only if G is in LL(1).
For a given grammar G the non-deterministic automaton is denoted by N_G and the deterministic automaton by D_G. We assume that each item in each state in D_G is enhanced with the so-called LALR(1) *look-ahead set*, cf. [Aho, Grune]. This set is used for LALR(1) grammars to decide whether a reduction is permitted. For illustration, we mention that the LALR(1) look-ahead set of every item in s_1 is equal to $\{\$\}$; the non-core items in s_2 have a look-ahead set equal to $\{w\}$. The LALR(1) look-ahead set of an item I in a state s of D_G is denoted by $LA_s(I)$. It holds in general that $Follow(I) = \bigcup\{LA_s(I) \mid s \text{ in } D_G\}$.

Definition 2 *For every item T in a state s of D_G, a quantity $F_s(T)$, and for every item T in N_G, a quantity $F'(T)$ is defined. Both are subsets of V_T. These quantities are defined in the following way:*

a) if $\epsilon \notin First(T)$, then $F_s(T) = F'(T) = First(T)$;

b) if $\epsilon \in First(T)$, then $F_s(T) = First(T) \cup LA_s(T) \backslash \{\epsilon\}$ and
$F'(T) = First(T) \cup Follow(T) \backslash \{\epsilon\}$.

Note that, if T is a reduce-item, $F_s(T) = LA_s(T)$ and $F'(T) = Follow(T)$. Since $LA_s(T) \subseteq Follow(T)$ for any T in a state s of D_G, also $F_s(T) \subseteq F'(T)$.
A pair of items T_1 and T_2 with the form $A \rightarrow \cdot\alpha_1$ and $A \rightarrow \cdot\alpha_2$ respectively is called a *brother pair*. Two brothers in one state of D_G have the same LA-set.
The absence of any non-determinism in the code of Figure 2 is equivalent to the condition $F'(T_1) \cap F'(T_2) = \emptyset$ for any brother pair T_1 and T_2. Therefore, in the proof of Lemma 1, this condition is used as an alternative of the LL(1) property.

Lemma 1 *A grammar G is in LL(1) if and only if $F_s(T_1) \cap F_s(T_2) = \emptyset$ for any brother pair T_1 and T_2 in any state s of D_G.*

Proof
only-if part
Follows immediately from the property $F'(T_1) \cap F'(T_2) = \emptyset$ and the set inclusion $F_s(T) \subseteq F'(T)$ for any s and any T.

if part (by contradiction)
Assume that there is conflict in an item T of $N(G)$, due to the fact that there are ϵ-transitions to T_1 and T_2 respectively, such that a symbol $a \in F'(T_1) \cap F'(T_2)$. Then T_1 and T_2 are brothers of each other. If $a \in First(T_1) \cap First(T_2)$, then $a \in F_s(T_1) \cap F_s(T_2)$ and a contradiction is obtained. Hence at least one of the two *First* items does not include a. Without loss of generality we assume that $a \notin First(T_1)$. In that case a belongs to $Follow(T_1)$. Due to the general relation $Follow(T_1) = \bigcup\{LA_s(T_1) \mid s \text{ in } D_G\}$, there is a state $\bar{s}$ in D_G, such that $a \in LA_{\bar{s}}(T_1)$. It follows that $a \in F_{\bar{s}}(T_1)$. Since T_2 is a brother of T_1 in $\bar{s}$, also $a \in LA_{\bar{s}}(T_2)$. If $\epsilon \notin First(T_2)$, then $F'(T_2) = F_{\bar{s}}(T_2) = First(T_2)$ and $a \in F_{\bar{s}}(T_2)$. If $\epsilon \in First(T_2)$, then $LA_{\bar{s}}(T_2) \subseteq F_{\bar{s}}(T_2)$ and $a \in F_{\bar{s}}(T_2)$. Again we have $a \in F_{\bar{s}}(T_1) \cap F_{\bar{s}}(T_2)$. Contradiction. □

Let $H(s)$ for a state s in D_G denote the subgraph in N_G of items corresponding to the items of s. An item T in $H(s)$, s a state in D_G, such that T has no outgoing edges in $H(s)$ is called an *end item* of $H(s)$. Hence, an end item is a reduce-item or a so-called *shift* item, i.e. an item of the form $A \rightarrow \alpha \cdot a\beta$ with $a \in V_T$.

If $a \in F_s(T)$ for some symbol a, then a occurs in at least one set $F_s(T')$ with T' a successor item of T in $H(s)$. It follows that a path P exists in $H(s)$ from T to an end item, such that $a \in F(I)$ for each item I in P.

Conversely, if $a \in F_s(T)$, then a is in at least one set $F_s(T')$ with T' a predecessor of T in $H(s)$. Consequentlty, if $a \in F_s(T)$, then there is a path P from a core item to T such that $a \in F(I)$ for each item I in P.

If in a state s $F(I_1) \cap F(I_2) = \emptyset$ for any pair end items I_1 and I_2, then s has no shift/reduce or reduce/reduce conflicts in LALR. In the remainder of this section we pay special attention to states with one core item. In view of this property, beside shift/reduce or reduce/reduce conflicts in LALR, we also define *shift/shift* conflicts. A state s has a shift/shift conflict, if s contains two shift items of the form $A \rightarrow \cdot a\alpha$ and $A' \rightarrow \cdot a\alpha'$ with $a \in V_T$. The occurence of a shift/shift conflict between two shift items I_1 and I_2 is equivalent to the property *First*(I_1)$\cap$*First*(I_2). If a shift/shift conflict occurs, there is no proper conflict, but a successor state of s has at least two core items. The absence of any conflict of any kind for an LALR(1) grammar is equivalent to the property $F_s(I_1) \cap F_s(I_2)) = \emptyset$ for any pair end items I_1 and I_2 in any state s. The absence of any conflict of any kind for a SLR(1) grammar is equivalent to the property $F'(I_1) \cap F'(I_2)) = \emptyset$ for any pair end items I_1 and I_2 in any state s.

Definition 3 *A non-terminal N is called a dummy non-terminal, if there are derivations $A \overset{*}{\Rightarrow} N\alpha$ and $A \overset{*}{\Rightarrow} N\alpha'$, and $N \rightarrow \epsilon$ is the unique production for N.*

Lemma 2 *A grammar G is in LALR(1) and every state s has exactly one core item if and only if G has no dummy terminals and $F_s(T_1) \cap F_s(T_2) = \emptyset$ for every pair brothers T_1 and T_2 in every state s of the LR-automaton.*

Proof

only-if part

If G has dummy non-terminals, i.e., there are derivations $A \overset{*}{\Rightarrow} N\alpha$ and $A \overset{*}{\Rightarrow} N\alpha'$, then a state containing $B \rightarrow \beta \cdot A\beta'$ has an exit, marked by N, to a successor state with core items $A \rightarrow N \cdot \alpha$ and $A \rightarrow N \cdot \alpha'$ Therefore, we conclude that G has no dummy-terminals.

In general, for a state s in D_G without outgoing edges in D_G, each core item is a reduce-item. Since there is exactly one core item, the result holds trivially in any state without outgoing edges, i.e. without successors states.

We prove by contradiction that the result holds in any state s with at least one successor. Assume that $a \in F_s(T_1) \cap F_s(T_2)$ with $a \in V_T$ and T_1 and T_2 a brother pair in a state s with at least one successor. There are paths from T_1 and T_2 to an end item such that $a \in F(I)$ for every item I in each path. If these paths do not meet each other, we have two end items I_1 and I_2 in s, such that $a \in F_s(I_1) \cap F_s(I_2)$. Then an LALR(1) conflict occurs in s or a successor state of s has more than one core item. If the paths meet each other, then consider in each path the last item before the meeting point. These items are called I_1 and I_2. Let A denote the left hand side of the first common item of both paths. Then both I_1 and I_2 have a transition, marked by A, to another state, which consequently has more than one core item. Contradiction.

if part (by contradiction)
In start state s_0, there is one core item.
Assume that a symbol a and two end items I_1 and I_2 in s_0 exist with $a \in F_s(I_1) \cap F_s(I_2)$, or that a transition, marked by a non-terminal P, exist to a successor state of s_0 with at least two core items. In the last case, there are two items I_1 and I_2 in s_0 of the form $A \rightarrow \cdot P\alpha$ and $A' \rightarrow \cdot P\alpha'$ respectively. Since P is not dummy, a symbol $a \in V_T$ exist with $a \in$*First*(P) and hence $a \in F(I_1) \cap I(_2)$. In both cases, there are paths from the core item to I_1 and I_2 respectively, such that $a \in F(I)$ for every item I in each path. The last common item of these paths has two successors I' and I'' with $a \in F_s(I') \cap F_s(I'')$. This contradicts the premiss. We conclude that the result holds for s_0. Since we have proved that every successor state s of s_0 in D_G has one core item, we can prove similarly that the result holds for s. It follows that the result holds for every state in D_G. □

Theorem 3 *A grammar G is in LL(1) and has no dummy non-terminals if and only if G is in LALR(1) and every state s of D_G has exactly one core item.*

Proof
Follows immediately from Lemma's 1 and 2. □

If T is an item in a graph $H(s)$ with s any state in D_G for a grammar G, and the successor items are $T_1, T_2, \ldots, T_k$, then always $F_s(T) \subseteq \bigcup_{i=1}^{i=k} F_s(T_i)$. If moreover $F_s(T_i) \cap F_s(T_j) =$ for any pair brothers T_i and T_j, $1 \leq i, j \leq k$, then $\bigcup_{i=1}^{i=k} F_s(T_i)$ is a partition of $F_s(T)$. It follows that, for a state s with exactly one core item and with the property $F_s(T_1) \cap F_s(T_2) = \emptyset$ for any brother pair T_1, T_2, $H(s)$ is a tree, corresponding to partitioning sets repeatedly. Conversely, if a grammar is in LALR(1) and $H(s)$ is a tree, it can be shown that $F_s(T_1) \cap F_s(T_2) = \emptyset$ for any brother pair T_1, T_2. We conclude that the following statement holds: a grammar G without dummy non-terminals is in LL(1) if and only if G is LALR(1) and $H(s)$ is a tree for any state s of D_G.

Theorem 4 *A grammar G is in LL(1) and has no ϵ-productions if and only if G is in LR(0) and the LR-automaton has exactly one core item in each state.*

Proof
If G is in LL(1) and G has no ϵ-productions, then, by Theorem 3, G is in LALR(0) and the LR-automaton has exactly one core item in each state. If a state has one core item, an ϵ-item must be involved in any shift/reduce or reduce/reduce conflict. Since there are no ϵ-items, G is in LR(0).
If G is in LR(0), then G cannot have ϵ-productions, since every ϵ-item causes a conflict in LR(0). Since G is also in LALR(1) and each state has one core item, G is in LL(1). □

Theorem 5 *A grammar G is in LL(1) and has dummy non-terminals if and only if G is in LR(1) and every state s in the LR(1)-automaton has exactly one core item.*

Proof
If G has one core item in LALR(1), then G has one core item in LR(1). Hence the *only if* part is trivial.
If every LR(1) state has one core item, then every LALR(1) state has one core item. Then in an LR(1) or an LALR(1) state a reduce/reduce conflict can occur only between ϵ-items. Transforming an LR(1) automaton into an LALR(1) automaton may introduce reduce/reduce conflicts, in general. It can be shown easily, that in the particular case that an LR(1) automaton has one core item in every state, a new reduce/reduce conflict between ϵ-items cannot be introduced. (This proof resembles the *if* part proof of Lemma 1 or the proof of the fact that the transition from LR(1) to LALR(1) introduces no shift/reduce conflicts.) Hence an LR(1) grammar with one core item in each state is in LALR(1) and has one core item in each LALR(1) state. Now, the *if* part follows from Theorem 3. □

Let two end items I_1 and I_2 be given in a state of a grammar with $F_s(I_1) \cap F_s(I_2) = \emptyset$. then these items have no LALR(1) conflict. If $\epsilon \in First(I_1)$, then $LA_s(I_1) \subseteq F_s(I_1)$. Since $F_s(I_1) \cap F_s(I_2)) = \emptyset$, for any symbol a with $a \in F_s(I_2)$, also $a \notin LA_s(I_1)$. It is possible that $a \in Follow(I_1)$. In that case we have a conflict in SLR(1). This situation is illustrated by the following grammars G_1 and G_2.

$$\begin{array}{llll@{\qquad}llll}
G_1: & Z & \rightarrow & X\$ & G_2: & Z & \rightarrow & X\$ \\
 & X & \rightarrow & xY \mid x'Vw & & X & \rightarrow & xY \mid x'Vw \\
 & Y & \rightarrow & Vv \mid Ww & & Y & \rightarrow & Vv \mid w \\
 & V & \rightarrow & v' \mid \epsilon & & V & \rightarrow & v' \mid \epsilon \\
 & W & \rightarrow & w' \mid \epsilon & & & &
\end{array}$$

In SLR(1), the state with core item $X \rightarrow x \cdot Y$, has a reduce/reduce conflict in G_1 and a shift/reduce conflict in G_2. Hence these grammars do not belong to SLR(1). If the production $X \rightarrow x'Vw$ is deleted from each grammar, w is no longer in $Follow(V)$ and the conflict is resolved. Since for both G_1 and G_2 each LR state has exactly one core item, both grammars are in LL(1), by Theorem 3. Clearly, the set of SLR(1) grammars with exactly one core item in each state cannot be characterized easily, using LL(1).

Now we focus on grammars with dummy non-terminals. We shall discuss such grammars briefly, because it is always possible to avoid dummy non-terminals, even if we need *markers*. (See [Aho] for the definition of a marker.) Let G be a grammar in LL(1) without dummy non-terminals. Let G' be any grammar obtained by inserting dummy non-terminals in some productions of G. Then G' is in LL(1) if and only if G is in LL(1). The LR automaton of G has exactly one core item in each state. However, this property needs not to be valid for G'.

Consider the following example.

$$\begin{array}{lcl} Z & \rightarrow & X\$ \\ X & \rightarrow & Px \mid Py \\ P & \rightarrow & \epsilon \end{array}$$

There is a state with core items $X \rightarrow P \cdot x$ and $X \rightarrow P \cdot y$. In general, when introducing dummy non-terminals, so-called dummy states arise in the LR automaton. A dummy state is a state containing core items $T_1, T_2, \ldots, T_n$ with $A_i \rightarrow \alpha \cdot \beta_i, V_i$ such that $\alpha \overset{*}{\Rightarrow} \epsilon$ is the only derivation for α (i.e. α only matches the empty string). For an LALR(1) grammar, $F_s(T') \cap F_s(T_j'') = \emptyset$, where T' and T'' are core items in any dummy state s. If dummy non-terminals are used, we must replace Lemma 2 with another one. *A grammar G is in LR(1) or LALR(1) and every state s of the LALR- or LR(1)-automaton respectively has one core item or is a dummy state with the additional property that $F_s(T') \cap F_s(T'') = \emptyset$, for any pair core items T' and T'' in s, if and only if $F_s(T_1) \cap F_s(T_2) = \emptyset$ for every brother pair T_1 and T_2 in a state s of the LR-automaton.* Theorems 3 and 5 must be modified accordingly. It is also possible, if dummy non-terminals are used, to redefine the construction of LR-states. This construction can be redefined in the sense that a dummy state that is a successor of a state s, is included entirely in s.

5 Concluding remarks

In section 2 we observed that LL parsing is actually a walk through a non-deterministic automaton. In section 3 we presented a new formulation of the LR parsing algorithm. A recursive formulation of bottom-up parsing has been presented earlier in a different way, and has been called *recursive ascent* parsing; see [Krus], [Penello] and [Roberts]. In section 4 we showed that LL(1) is a particular subclass of LALR(1). Consequently, the recursive descent algorithm for LL(1) grammars can be viewed as a special implementation of the recursive LR algorithm for a particular subclass in LALR(1), viz. the subclass of grammars with exactly one core item in each state. For this subclass of LALR(1), the LR algorithm constructs the syntax tree in a top-down manner.

Since LL(1) is a well-defined special case of LALR(1), there is no longer need of LL(1) parser generators. Because the LR parser, applied to an LL(1) grammar, constructs the syntax tree in a top-down manner, such a parser can deal with an L-attributed definition [Aho] very well.

If the *parse* procedure for LL-parsing is non-deterministic, we must investigate all possible combinations of choices. This can be performed in a depth-first manner, i.e. backtracking, or in a breadth-first manner. For depth-first, it is necessary that the grammar is not left-recursive, i.e., the NA contains no terminal-free cycles. In [Sippu] a general transformation scheme is presented to eliminate left-recursion.
Similarly, if conflicts are not solved, in LR-parsing all possible combinations can

be performed in a depth-first manner, i.e. backtracking, or in a breadth-first manner. Two breadth-first algorithms are the Earley algorithm [Earley] and the Tomita algorithm [Tomita]. These are very similar to each other, as shown in [Sikkel]. For efficiency, the different possibilities, to be investigated, are synchronized by the input of the next character. This feature is applied in the Earley as well as in the Tomita algorithm.

References

[Aho] A.V.Aho, R. Sethi, J.D. Ullman, *Compilers, Principles, Techniques and Tools,* Addison-Wesley, 1986.

[Earley] J.C. Earley, *An efficient context-free parsing algorithm,* Comm. ACM (1970) 13(2) pp. 94-102.

[Grune] D. Grune and C. Jacobs, *Parsing Techniques, a practical guide,* Ellis Horwood 1990.

[Holub] A.I. Holub, *Compiler Design in C,* Prentice Hall.

[Krus] F.E.J. Kruseman Aretz, *On a recursive ascent parser,* Information Processing Letters 1988 (29) pp. 201-206

[Leerm-92a] R. Leermakers, L. Augusteijn, F.E.J. Kruseman Aretz, *A functional LR parser,* Theoretical Computer Science, 1992 (104) pp. 313-323.

[Leerm-92b] R. Leermakers, *Recursive ascent parsing, from Earley to Marcus,* Theoretical Computer Science, 1992 (104) pp. 299-312.

[Penello] T.J. Penello, *Very Fast LR Parsing,* Sigplan Notices, 1986 (21-7) pp.141-151.

[Roberts] G.H. Roberts, *Recursive Ascent, An LR Analog to Recursive Descent,* SIGPLAN Notices, 1988 (23-8) pp. 23-29.

[Sippu] S. Sippu and E. Soisalon-Soininen, *Parsing Theory,* two volumes, Springer Verlag, 1990.

[Sikkel] K. Sikkel, *Cross-Fertilization of Earley Tomita,* Memoranda Informatica 90-69, University of Twente, Department of Computer Science, 1990.

[Tomita] M. Tomita, *Efficient Parsing for Natural Language, A fast algorithm for Practical Systems,* Kluwer Academic Publishers, 1986.