Increasing the Applicability of LR Parsing

Mark-Jan Nederhof [*] Janos J. Sarbo

University of Nijmegen, Department of Computer Science
Toernooiveld, 6525 ED Nijmegen, The Netherlands
E-mail: {markjan,janos}@cs.kun.nl

Abstract

In this paper we discuss a phenomenon present in some context-free grammars, called *hidden left recursion*. We show that ordinary LR parsing according to hidden left-recursive grammars is not possible and we indicate a range of solutions to this problem. One of these solutions is a new parsing technique, which is a variant of traditional LR parsing. This new parsing technique can be used both with and without lookahead and the nondeterminism can be realized using backtracking or using a graph-structured stack.

1 Introduction

The class of LR parsing strategies constitutes one of the strongest and most efficient classes of parsing strategies for context-free grammars. LR parsing is commonly used in compilers as well as in systems for the processing of natural language.

Deterministic LR parsing with lookahead of k symbols is possible for LR(k) grammars. Deterministic parsing according to grammars which are not LR(k) can in some cases be achieved with some disambiguating techniques. (Important progress in this field has been reported by Thorup (1994).) However, these techniques are not powerful enough to handle practical grammars for e.g. natural languages.

If we consider LR parsing tables in which an entry may contain multiple actions, then we obtain nondeterministic LR parsing. We will refer to realizations of nondeterministic LR parsing as *generalized* LR parsing. The most straightforward way to obtain generalized LR parsing is by using backtracking (Nilsson, 1986).

A more efficient kind of generalized LR parsing has been proposed by Tomita (1986). The essence of this approach is that multiple parses are processed simultaneously. Where possible, the computation processing two or more parses is shared. This is accomplished by using a *graph-structured stack*.

Although generalized LR parsing can handle a large class of grammars, there is one phenomenon which it cannot deal with, viz. *hidden left recursion*. Hidden left recursion, defined at the end of this section, occurs very often in grammars for natural languages.

A solution for handling hidden left-recursive grammars using Tomita's algorithm was proposed by Nozohoor-Farshi (1989). In that paper, the ordinary acyclic graph-structured stack is generalized to allow cycles. The resulting parsing technique is largely equivalent to a parsing technique which follows from a construction defined earlier by Lang (1974), which makes use of a parse matrix. As a consequence, termination of the parsing process is always guaranteed. This means that hidden left-recursive grammars and even cyclic grammars can be handled.

However, cyclic graph-structured stacks may complicate garbage collection and cannot be realized using memo-functions (Leermakers et al., 1992). Tomita's algorithm furthermore becomes very complicated in the case of augmented context-free grammars (e.g. attribute grammar, affix grammar, definite clause gram-

[*] Supported by the Dutch Organisation for Scientific Research (NWO), under grant 00-62-518

mar, etc.). In this case, different subparses almost always have different attribute values (or affix values, variable instantiations, etc.) and therefore sharing of the computation of context-free parsing would obstruct the correct computation of these values (Nederhof — Sarbo, 1993).

In this paper we discuss an alternative approach to adapting the (generalized) LR parsing technique to hidden left-recursive grammars.

Our approach can be roughly described as follows. Reductions with epsilon rules are no longer performed. Instead, a reduction with some non-epsilon rule does not have to pop all the members in the right-hand side off the stack; only those which do not derive the empty string must be popped, for others it is optional. The definition of the closure function for sets of items is changed accordingly. Our approach requires the inspection of the parse stack upon each reduction in order to avoid incorrect parses.

The structure of this paper is as follows. In the next section we give an introduction to the problem of LR parsing according to hidden left-recursive grammars. We give two naive ways of solving this problem by first transforming the grammar before constructing the (nondeterministic) LR automaton. (These methods are naive because the transformations lead to larger grammars and therefore to much larger LR automata.) We then show how the first of these transformations can be incorporated into the construction of LR automata, which results in parsers with a fewer number of states. We also outline an approach of adapting the LR technique to cyclic grammars.

In Section 3 we prove the correctness of our new parsing technique, called ϵ-LR parsing. Efficient generation of ϵ-LR parsers is discussed in Section 4. We conclude in Section 5 by giving some results on the comparison between the number of states of various LR and ϵ-LR parsers.

We would like to stress beforehand that grammars with nontrivial hidden left recursion can never be handled using deterministic LR parsing (Section 2.5), so that most of the discussion in this paper is not applicable to deterministic LR parsing. We therefore, contrary to custom, use the term "LR parsing" for *generalized* LR parsing, which can at will be realized using backtracking (possibly in combination with memo-functions) or using *acyclic* graph-structured stacks. Wherever we deal with *deterministic* LR parsing, this will be indicated explicitly.

The notation used in the sequel is for the most part standard and is summarized below.

A context-free grammar $G = (T, N, P, S)$ consists of two finite disjoint sets N and T of nonterminals and terminals, respectively, a start symbol $S \in N$, and a finite set of rules P. Every rule has the form $A \rightarrow \alpha$, where the left-hand side (lhs) A is an element from N and the right-hand side (rhs) α is an element from V^*, where V denotes $(N \cup T)$. P can also be seen as a relation on $N \times V^*$.

We use symbols $A, B, C, \ldots$ to range over N, symbols X, Y, Z to range over V, symbols $\alpha, \beta, \gamma, \ldots$ to range over V^*, and $v, w, x, \ldots$ to range over T^*. We let ϵ denote the empty string. A rule of the form $A \rightarrow \epsilon$ is called an *epsilon rule.*

The relation P is extended to a relation $\overset{G}{\rightarrow}$ on $V^* \times V^*$ as usual. We write $\rightarrow$ for $\overset{G}{\rightarrow}$ when G is obvious. The transitive closure of $\rightarrow$ is denoted by $\rightarrow^+$ and the reflexive and transitive closure is denoted by $\rightarrow^*$.

We define: $B \angle A$ if and only if $A \rightarrow B\alpha$ for some α. The transitive closure of $\angle$ is denoted by $\angle^+$.

We distinguish between two cases of left recursion. The most simple case, which we call *plain left recursion*, occurs if there is a nonterminal A such that $A \angle^+ A$. The other case, which we call *hidden left recursion*, occurs if $A \rightarrow B\alpha$, $B \rightarrow^* \epsilon$, and $\alpha \rightarrow^* A\beta$, for some A, B, α, and β; the left recursion is "hidden" by the empty-generating nonterminal. (An equivalent definition of hidden left recursion is due to Leermakers et al. (1992).)

A grammar is said to be *cyclic* if $A \rightarrow^+ A$ for some nonterminal A.

A nonterminal A is said to be *nullable* if $A \rightarrow^* \epsilon$. A nonterminal A is called a *predicate* if it is nullable and $A \rightarrow^* v$ only for $v = \epsilon$.[1]

[1] The term "predicate" seems to have been used in this sense for the first time by Koster (1971).

We call a nonterminal A *reachable* if $S \rightarrow^* \alpha A \beta$ for some α and β. We call a grammar *reduced* if every nonterminal is reachable and derives some terminal string. Where we give a transformation between context-free grammars, we tacitly assume that the input grammars are reduced and for these grammars the output grammars are guaranteed also to be reduced.

2 Hidden left recursion and LR parsing

The simplest nontrivial case of hidden left recursion is the grammar G_1 given by the following rules.

$$\begin{array}{lcl} A & \rightarrow & BAc \\ A & \rightarrow & a \\ B & \rightarrow & b \\ B & \rightarrow & \epsilon \end{array}$$

In this grammar, nonterminal A is left-recursive. This fact is hidden by the presence of a nullable nonterminal B in the rule $A \rightarrow BAc$. Note that this grammar is ambiguous, as illustrated in Figure 1. This is typically so in the case where the one or more nullable nonterminals which hide the left recursion are not all predicates.

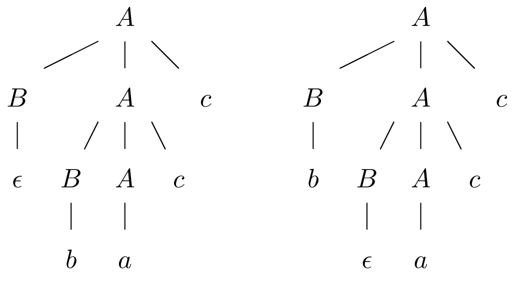

Figure 1: Two parse trees with the same yield, showing ambiguity of G_1.

2.1 Generalized LR parsing and hidden left recursion

We now discuss informally how (generalized) LR parsing fails to terminate for the above grammar. We assume that the reader is familiar with the construction of (nondeterministic) LR(0) automata. Our terminology is taken from Aho et al. (1986).

A pictorial representation of the LR(0) parsing table for G_1 is given in Figure 2. LR parsing of any input w may result in many sequences of parsing steps, one of which is illustrated by the following sequence of configurations.

Stack contents	Inp.	Action
Q_0	w	red($B \rightarrow \epsilon$)
$Q_0\ B\ Q_1$	w	red($B \rightarrow \epsilon$)
$Q_0\ B\ Q_1\ B\ Q_1$	w	red($B \rightarrow \epsilon$)
$Q_0\ B\ Q_1\ B\ Q_1\ B\ Q_1$	w	red($B \rightarrow \epsilon$)
⋮	⋮	⋮

The sequence of parsing steps illustrated above does not terminate. We can find a non-terminating sequence of parsing steps for the LR(0) automaton for every hidden left-recursive grammar. In fact, this is even so for the LR(k), LALR(k), and SLR(k) automata, for any k. Hidden left recursion has been identified by Soisalon-Soininen — Tarhio (1988) as one of two sources, together with cyclicity, of the looping of LR parsers.

Various other parsing techniques, such as left-corner parsing (Nederhof, 1993a) and cancellation parsing (Nederhof, 1993b), also suffer from this deficiency.

2.2 Eliminating epsilon rules

We first discuss a method to allow LR parsing for hidden left-recursive grammars by simply performing a source to source transformation on grammars to eliminate the rules of which the right-hand sides only derive the empty string. To preserve the language, for each rule containing an occurrence of a nullable nonterminal a copy must be added without that occurrence. Following Aho — Ullman (1972), this transformation, called ϵ-*elim*, is described below. The input grammar is called G.

1. Let G_0 be G.

2. Remove from G_0 all rules defining predicates in G and remove all occurrences of these predicates from the rules in G_0.

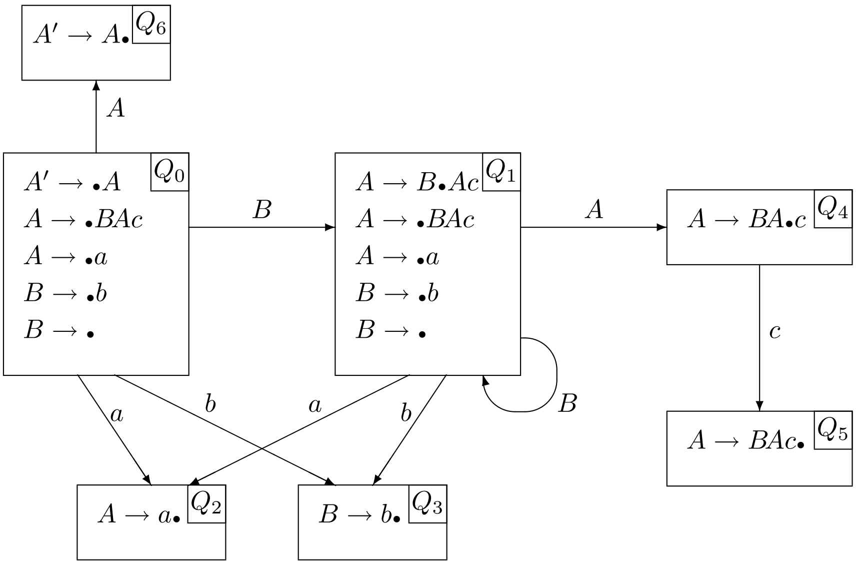

Figure 2: The LR(0) automaton for G_1.

3. Replace every rule of the form $A \to \alpha_0 B_1 \alpha_1 B_2 \ldots B_m \alpha_m$ in G_0, $m \geq 0$, where the members which are nullable in G are exactly $B_1, \ldots, B_m$, by the set of rules of the form $A \to \alpha_0 \beta_1 \alpha_1 \beta_2 \ldots \beta_m \alpha_m$, where β_i is either B_i or ϵ and $\alpha_0 \beta_1 \alpha_1 \beta_2 \ldots \beta_m \alpha_m \neq \epsilon$. Note that this set of rules is empty if $m = 0$ and $\alpha_0 = \epsilon$, in which case the original rule is just eliminated from G_0.

4. If S is nullable in G, then add the rules $S^\dagger \to S$ and $S^\dagger \to \epsilon$ to G_0 and make $S^\dagger$ the new start symbol of G_0. (In the pathological case that S is a predicate in G, $S^\dagger \to S$ should of course not be added to G_0.)

5. Let ϵ-*elim*(G) be G_0.

Note that for every rule $A \to \alpha$ such that α contains k occurrences of nullable non-predicates, the transformed grammar may contain 2^k rules.

In this paper, an expression of the form $[B]$ in a rhs indicates that the member B has been eliminated by the transformation. It is for reasons of clarity that we write this expression instead of just leaving B out.

An item of the form $A \to [\alpha_0] X_1 [\alpha_1] \ldots [\alpha_{i-1}] \bullet X_i \ldots X_m [\alpha_m]$ is said to be *derived* from the *basic* item $A \to \alpha_0 X_1 \alpha_1 \ldots \alpha_{i-1} \bullet X_i \ldots X_m \alpha_m$.[2] According to the convention mentioned above, $A \to \alpha_0 X_1 \alpha_1 \ldots X_m \alpha_m$ is a rule in G, and $A \to X_1 \ldots X_m$ is a rule in ϵ-*elim*(G). The item of the form $S^\dagger \to \bullet$ which may be introduced by ϵ-*elim* will be regarded as the derived item $S^\dagger \to [S]\bullet$.

Example 2.1 Let the grammar G_2 be defined by the rules

$$\begin{array}{lcl} A & \to & BCD \\ B & \to & \epsilon \\ B & \to & b \\ C & \to & \epsilon \\ D & \to & \epsilon \\ D & \to & d \end{array}$$

Step 2 of ϵ-*elim* removes the rule $C \to \epsilon$ defining the only predicate C. Also the occurrence of C in $A \to BCD$ is removed, i.e. this rule is replaced by $A \to B[C]D$.

Step 3 removes all rules with an empty rhs, viz. $B \to \epsilon$ and $D \to \epsilon$, and replaces

[2] We avoid writing dots in dotted items immediately to the left of eliminated members.

$A \rightarrow B[C]D$ by the set of all rules which result from either eliminating or retaining the nullable members, viz. B and D (C is not a member anymore!), such that the rhs of the resulting rule is not empty. This yields the set of rules

$$\begin{array}{lcl} A & \rightarrow & B[C]D \\ A & \rightarrow & B[CD] \\ A & \rightarrow & [BC]D \end{array}$$

Step 4 adds the rules $A^\dagger \rightarrow A$ and $A^\dagger \rightarrow \epsilon$. The new start symbol is $A^\dagger$.

We have now obtained ϵ-*elim*(G_2), which is defined by

$$\begin{array}{lcl} A^\dagger & \rightarrow & A \\ A^\dagger & \rightarrow & \epsilon \\ A & \rightarrow & B[C]D \\ A & \rightarrow & B[CD] \\ A & \rightarrow & [BC]D \\ B & \rightarrow & b \\ D & \rightarrow & d \end{array} \qquad \Box$$

Note that in the case that ϵ-*elim* introduces a new start symbol $S^\dagger$, there is no need to augment the grammar (i.e. add the rule $S' \rightarrow S^\dagger$ and make S' the new start symbol) for the purpose of constructing the LR automaton. Augmentation is in this case superfluous because the start symbol $S^\dagger$ is not recursive.

In the case of G_1, the transformation yields the following grammar.

$$\begin{array}{lcl} A & \rightarrow & BAc \\ A & \rightarrow & [B]Ac \\ A & \rightarrow & a \\ B & \rightarrow & b \end{array}$$

The LR(0) table for this grammar is represented in Figure 3.

Together with the growing number of rules, the above transformation may also give rise to a growing number of states in the LR(0) automaton. In the above case, the number of states increases from 7 to 8, as indicated by Figures 2 and 3. As G_1 is only a trivial grammar, we may expect that the increase of the number of states for practical grammars is much larger. Tangible results are discussed in Section 5.

2.3 A new parsing algorithm

To reduce the number of states needed for an LR automaton for ϵ-*elim*(G), we incorporate the transformation in the closure function. This requires changing the behaviour of the LR automaton upon reduction.

This approach can in a different way be explained as follows. Items derived from the same basic item by ϵ-*elim* are considered the same. For instance, the items $A \rightarrow BAc\bullet$ and $A \rightarrow [B]Ac\bullet$ in Figure 3 are considered the same because they are derived from the same basic item $A \rightarrow BAc\bullet$.

All items are now represented by the basic item from which they are derived. For instance, both items in Q_5 in Figure 3 are henceforth represented by the single basic item $A \rightarrow BAc\bullet$. The item $A \rightarrow [B]Ac\bullet$ in state Q_7 is now represented by $A \rightarrow BAc\bullet$.

As a result, some pairs of states now consist of identical sets of items and may therefore be merged. For the example in Figure 3, the new collection of states is given in Figure 4. It can be seen that states Q_5 and Q_7 are merged into state $Q_{5/7}$.

In the resulting LR table, it is no longer indicated which derived items are actually represented. Correspondingly, the behaviour of the new automaton is such that upon reduction all possibilities of derived items are to be tried nondeterministically.

For instance, consider the parsing of *bacc* using the LR(0) automaton in Figure 4. The first sequence of parsing steps is without complications:

Stack contents	Inp.	Action
Q_0	*bacc*	shift
$Q_0\ b\ Q_3$	*acc*	red($B \rightarrow b$)
$Q_0\ B\ Q_1$	*acc*	shift
$Q_0\ B\ Q_1\ a\ Q_2$	*cc*	red($A \rightarrow a$)
$Q_0\ B\ Q_1\ A\ Q_4$	*cc*	shift
$Q_0\ B\ Q_1\ A\ Q_4\ c\ Q_{5/7}$	*c*	red(?)

Now there are two ways to perform a reduction with the item $A \rightarrow BAc\bullet$. One way is to assume that B has been eliminated from this rule. In other words, we are dealing with the derived item $A \rightarrow [B]Ac\bullet$. In this case we remove two states and grammar symbols from the stack. The sequence of configurations from here on now begins with

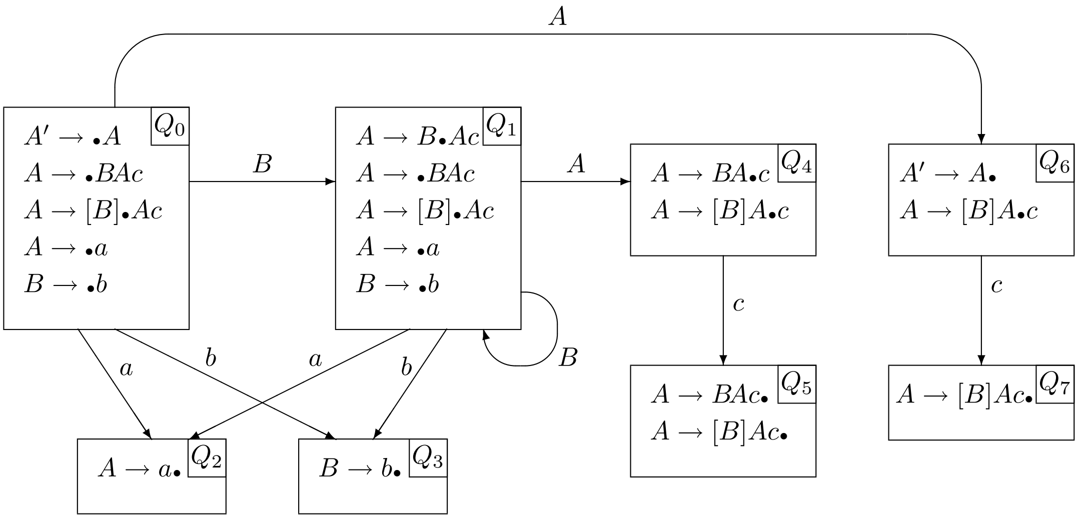

Figure 3: The LR(0) automaton for ϵ-*elim*(G_1).

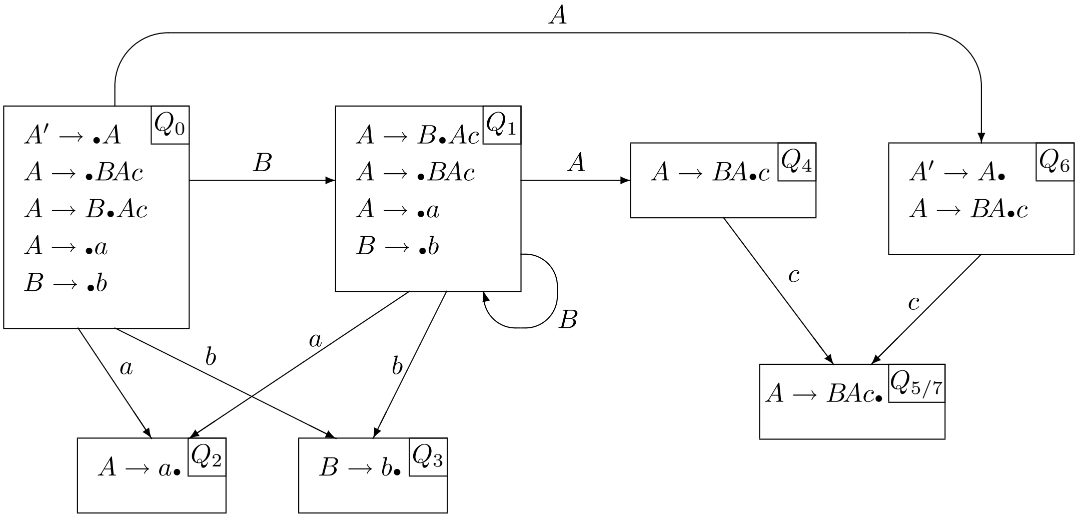

Figure 4: The optimised LR(0) automaton for ϵ-*elim*(G_1) with merged states.

$Q_0\ B\ Q_1\ A\ Q_4\ c\ Q_{5/7}$	c	red$(A \to [B]Ac)$
$Q_0\ B\ Q_1\ A\ Q_4$	c	
⋮	⋮	⋮

The other way to perform reduction is by taking off the stack all the members in the rule $A \to BAc$ and the same number of states, and we obtain

$Q_0\ B\ Q_1\ A\ Q_4\ c\ Q_{5/7}$	c	red$(A \to BAc)$
$Q_0\ A\ Q_6$	c	shift
$Q_0\ A\ Q_6\ c\ Q_{5/7}$		red(?)

We are now at an interesting configuration. We have again the choice between reducing with $A \rightarrow [B]Ac$ or with the unaffected rule $A \rightarrow BAc$. However, it can be established that reduction with $A \rightarrow BAc$ is not possible, because there is no B on the stack to be popped.

At this point, the main difference between traditional LR parsing and the new parsing technique we are developing becomes clear. Whereas for traditional LR parsing, the grammar symbols on the stack have no other purpose except an explanatory one, for our new parsing technique, the investigation of the grammar symbols on the stack is essential for guiding correct parsing steps.

In general, what happens upon reduction is this. Suppose the state on top of the stack contains an item of the form $A \rightarrow \alpha\bullet$, then reduction with this item is performed in the following steps.

1. The parser nondeterministically looks for some sequence of grammar symbols X_1, …, X_m such that there are α_0, …, α_m with

   - $\alpha = \alpha_0 X_1 \alpha_1 \ldots X_m \alpha_m$
   - $\alpha_0 \rightarrow^* \epsilon \wedge \ldots \wedge \alpha_m \rightarrow^* \epsilon$
   - The top-most m grammar symbols on the stack are $X_1, \ldots, X_m$ in that order, i.e. X_1 is deepest in the stack and X_m is on top of the stack.
   - $m = 0 \Rightarrow A = S'$

   In words, α is divided into a part which is on the stack and a part which consists only of nullable nonterminals. The part on the stack should not be empty with the exception of the case where $A \rightarrow \alpha$ is the rule $S' \rightarrow S$.

2. The top-most m symbols and states are popped off the stack.

3. Suppose that the state on top of the stack is Q, then

   - if $A = S'$, then the input is accepted, provided Q is the initial state and the end of the input has been reached; and
   - if $A \neq S'$, then A and subsequently *goto* (Q, A) are pushed onto the stack, provided *goto* (Q, A) is defined (otherwise this step fails).

The way the reduction is handled above corresponds with the reduction with the rule $A \rightarrow [\alpha_0]X_1[\alpha_1]\ldots X_m[\alpha_m]$ in the original LR(0) parser for ϵ-*elim*(G).

Incorporating the transformation ϵ-*elim* into the construction of the LR table can be seen as a restatement of the usual closure function, as follows.

$$\begin{aligned} \mathit{closure}\ (q)\ = \\ & \{B \rightarrow \delta\bullet\theta \mid A \rightarrow \alpha\bullet\beta \in q \wedge \beta \rightarrow^* B\gamma \wedge \\ & \quad B \rightarrow \delta\theta \wedge \\ & \quad \exists v[v \neq \epsilon \wedge \delta\theta \rightarrow^* v] \wedge \\ & \quad \delta \rightarrow^* \epsilon\} \\ \cup \\ & \{A \rightarrow \alpha\delta\bullet\beta \mid A \rightarrow \alpha\bullet\delta\beta \in q \wedge \delta \rightarrow^* \epsilon\} \end{aligned}$$

Note that the expression $\beta \rightarrow^* B\gamma$ allows nonterminals to be rewritten to the empty string. Also note that $\exists v[v \neq \epsilon \wedge \delta\theta \rightarrow^* v]$ excludes rules of which the rhs can only derive ϵ. Efficient calculation of the closure function is investigated in Section 4.1.

Leermakers (1992) proposes similar changes to some functions in the recursive ascent Earley parser in order to allow hidden left recursion. Similar changes were made by Graham et al. (1980) and Leiss (1990) in order to improve the efficiency of Earley parsing. A parsing technique very similar to ours is suggested by Leermakers (1993). A similar idea is also suggested by Sikkel (1990), but without details.

The investigation of the grammar symbols on the stack for the purpose of guiding correct parsing steps is reminiscent of Pager (1970), who proposed a general method for the compression of parsing tables by merging states. If the full stack may be investigated upon reduction, then the need for states in the traditional sense is even completely removed, as shown by Fortes Gálves (1992).[3]

In Section 3 we prove the correctness of the new parsing technique, which we call ϵ-LR parsing.

[3] It is interesting to note that various degrees of simplification of the collection of sets of items are possible. For example, there is an approach half-way between our approach and the one by Fortes, according to which items consist only of the parts which occur normally after the dots. A different way to look at this is as LR parsing for a grammar transformed by transformation τ_{two} from Nederhof — Satta (1994).

2.4 Dealing with cyclic grammars

If needed, ϵ-LR parsing can be further refined to handle cyclic grammars.

The starting-point is again a transformation on grammars, called *C-elim*, which eliminates all *unit rules*, i.e. all rules of the form $A \rightarrow B$. This transformation consists of the following steps.

1. Let G_0 be G.
2. Replace every non-unit rule $A \rightarrow \alpha$ in G_0 by the set of rules of the form $B \rightarrow \alpha$ such that $B \overset{G}{\rightarrow}^* A$ and either $B = S$ or B has an occurrence in the rhs of some non-unit rule.
3. Remove all unit rules from G_0.
4. Let *C-elim*(G) be G_0.

Termination of LR parsing according to *C-elim*$(\epsilon$*-elim*$(G))$ is guaranteed for any G.

If we incorporate *C-elim* into our ϵ-LR parsers, then reduction with $A \rightarrow \alpha$ is performed by the following steps.

1. The parser nondeterministically looks for some sequence of grammar symbols X_1, …, X_m such that there are α_0, …, α_m with
   - $\alpha = \alpha_0 X_1 \alpha_1 \ldots X_m \alpha_m$
   - $\alpha_0 \rightarrow^* \epsilon \wedge \ldots \wedge \alpha_m \rightarrow^* \epsilon$
   - The top-most m grammar symbols on the stack are $X_1, \ldots, X_m$.
   - $m = 0 \Rightarrow A = S'$
   - $m = 1 \Rightarrow (X_1 \in T \vee A = S')$
2. The top-most m symbols and states are popped off the stack.
3. Suppose that the state on top of the stack is Q, then
   - if $A = S'$, then the input is accepted, provided Q is the initial state and the end of the input has been reached; and
   - if $A \neq S'$, then the parser nondeterministically looks for some nonterminal B such that $B \rightarrow^* A$ and *goto* (Q, B) is defined, and then B and subsequently *goto* (Q, B) are pushed onto the stack.

Note that the parser which performs reduction in this way, using the parse tables from the ϵ-LR parser, may go into unnecessary dead alleys of length one. The reason is that the goto function is defined in cases where is it obvious that the next step must be a reduction with a rule with a single non-predicate in its rhs, which is of course blocked by the above formulation of reduction. This may be avoided by reformulating the closure function such that rules containing a single non-predicate in their right-hand sides are left out.

How to avoid reductions with unit rules (*unit reductions*) in the case of deterministic LR parsing has been investigated in a number of papers (Heilbrunner, 1985, e.g.,). Our particular technique of avoiding unit reductions is reminiscent of an optimization of Earley's algorithm (Graham et al., 1980).

In the remaining part of this paper, the term "ϵ-LR parsing" will not include the extra extension to ϵ-LR parsing described in this section.

2.5 Applicability of ϵ-LR parsing

In the same way as LR(0) parsing can be refined to SLR(k), LALR(k), and LR(k) parsing ($k > 0$) we can also refine ϵ-LR(0) parsing to ϵ-SLR(k), ϵ-LALR(k), and ϵ-LR(k) parsing. The construction of ϵ-LR tables for these parsing strategies can be adopted from the construction of their LR counterparts in a reasonably straightforward way.

We have shown that ϵ-LR parsing can be used for hidden left-recursive grammars, which cannot be handled using ordinary LR parsing. The variants of ϵ-LR parsing which apply lookahead are useful for making the parsing process more deterministic, i.e. to reduce the number of entries in the parsing table that contain multiple actions.

However, adding lookahead cannot yield completely deterministic parsers in the case of hidden left recursion where at least one of the hiding nonterminals is not a predicate. This is because such a grammar is ambiguous, as discussed earlier. (If all hiding nonterminals are predicates, then we are dealing with a trivial form of hidden left recursion, which can easily be eliminated by eliminating the hiding nonterminals.)

Also in the case of grammars without hidden left recursion, ϵ-LR parsing may have an advantage over ordinary (generalized) LR parsing: the parsing actions corresponding with subtrees of the parse tree which have empty yields are avoided. For these grammars, the application of lookahead may serve to construct deterministic ϵ-LR parsers.

Nederhof (1993a) describes how subtrees which have empty yields can be attached to the complete parse tree without actually parsing the empty string.

2.6 Specific elimination of hidden left recursion

For the sake of completeness, we describe a way of removing hidden left recursion without using epsilon rule elimination. The idea is that we selectively remove occurrences of nullable nonterminals which hide left recursion. In case of a nullable non-predicate A, we replace the occurrence of A by an occurrence of a new nonterminal A'. This A' is constructed so as to derive the same set of strings as A does, with the exception of ϵ.

The transformation, constructing the grammar *HLR-elim*(G) from grammar G, consists of the following steps.

1. Let G_0 be G.
2. For every rule $A \rightarrow B\alpha$ in G_0 which leads to a hidden left-recursive call (i.e. $\alpha \overset{G}{\rightarrow}^* A\beta$ for some β, and $B \overset{G}{\rightarrow}^* \epsilon$), replace the rule by $A \rightarrow \alpha$, and also add $A \rightarrow B'\alpha$ to G_0 provided B is not a predicate in G. Repeat this step until it can no longer be applied.
3. For every new nonterminal A' introduced in G_0 in step 2, or by an earlier iteration of step 3, and for every rule $A \rightarrow \alpha$ in G_0, add to G_0 the rule
   - $A' \rightarrow \alpha$ if not $\alpha \overset{G}{\rightarrow}^* \epsilon$, or rules of the form
   - $A' \rightarrow X_i' X_{i+1} \ldots X_n$ if $\alpha \overset{G}{\rightarrow}^* \epsilon$, where $\alpha = X_1 \ldots X_n$, and X_i is not a predicate.
4. Remove from G_0 all rules $A \rightarrow \alpha$ such that A was rendered unreachable by the elimination of rules in step 2.
5. Let *HLR-elim*(G) be G_0.

Example 2.2 Let the grammar G_3 be defined by

$$\begin{array}{lll} A & \rightarrow & ABAa \\ A & \rightarrow & AAB \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$

The grammar *HLR-elim*(G_3) is given by

$$\begin{array}{lll} A & \rightarrow & Aa \\ A & \rightarrow & A'BAa \\ A & \rightarrow & AB \\ A & \rightarrow & A'AB \\ A & \rightarrow & \epsilon \\ A' & \rightarrow & Aa \\ A' & \rightarrow & A'BAa \\ A' & \rightarrow & A'B \\ A' & \rightarrow & A'AB \\ B & \rightarrow & \epsilon \end{array}$$ □

The transformation *HLR-elim* is very often incorporated in the construction of parsers which can deal with hidden left recursion. An example is the variant of backtrack left-corner parsing as applied in Programmar (Meijer, 1986). See also Nederhof (1993a). A transformation by Graham et al. (1980) is related to *HLR-elim*, but also transforms the grammar into an extended Chomsky normal form.

The size of the grammar resulting from the application of this transformation is much smaller than that in the case of ϵ-*elim*. In fact it is only quadratic in the size of the original grammar.

3 Correctness of ϵ-LR parsing

A formal derivation of ϵ-LR(0) parsing is given by Nederhof (1994). In this section we prove the correctness of ϵ-LR parsing by assuming the correctness of (nondeterministic) LR parsing, which has already been established in literature.

In Section 2.3 we derived the new parsing technique of ϵ-LR parsing. We showed that this kind of parsing is based on traditional LR parsing, with the following differences:

- Instead of using the original grammar G, the transformed grammar ϵ-*elim*(G) is used.
- No distinction is made between items derived from the same basic item. This can be seen as merging states of the LR automaton of ϵ-*elim*(G).
- Because considering derived items as the same leads to a loss of information, a new mechanisms is introduced, which checks upon reduction whether the members of the applied rule are actually on the stack and whether the goto function is defined for the lhs and the state which is on top of the stack after the members are popped.

Because the transformation ϵ-*elim* preserves the language and because we assume the correctness of LR parsing, the correctness of ϵ-LR parsing can be proved by mentioning two points:

- The symbols on the stack and the remaining input together derive the original input, which can be proved by induction on the length of a sequence of parsing steps. This argument shows that no incorrect derivations can be found.
- For every sequence of parsing steps performed by an LR parser (LR(k), SLR(k), etc.) for ϵ-*elim*(G) there is a corresponding sequence of parsing steps performed by the corresponding type of ϵ-LR parser (ϵ-LR(k), ϵ-SLR(k), etc.) for G.

  This proves that ϵ-LR parsing cannot fail to find correct derivations by the assumption that LR parsing according to ϵ-*elim*(G) does not fail to find correct derivations.

In case of ϵ-LR(0) and ϵ-SLR parsing it can also be shown that the set of sequences of parsing steps is isomorphic with the set of sequences of the LR(0) or SLR parsers for ϵ-*elim*(G), and that the corresponding sequences are equally long. It is sufficient to prove that if a reduction can be successfully performed in an ϵ-LR parser, then it can be performed in an LR parser in the corresponding configuration.

For this purpose, suppose that in an ϵ-LR parser some reduction is possible with the item $A \rightarrow \alpha_0 A_1 \alpha_1 \dots A_m \alpha_m \bullet \in Q_m$ such that

- $\alpha_i \rightarrow^* \epsilon$ for $0 \leq i \leq m$,
- the topmost $2m+1$ elements of the stack are $Q_0 A_1 Q_1 \dots A_m Q_m$,
- the goto function for Q_0 and A is defined,
- in the corresponding configuration in the LR parser, the states corresponding with Q_i are called Q'_i.

From the fact that the goto function is defined for Q_0 and A we know that it is also defined for Q'_0 and A and that the item $A \rightarrow [\alpha_0] \bullet A_1 [\alpha_1] \dots A_m [\alpha_m]$ is in Q'_0. This implies that $A \rightarrow [\alpha_0] A_1 [\alpha_1] \dots A_i [\alpha_i] \bullet \dots A_m [\alpha_m]$ is in Q'_i because Q'_i is *goto* (Q'_{i-1}, A_i), for $1 \leq i \leq m$.

Therefore, in the corresponding LR parser a reduction would also take place according to the item $A \rightarrow [\alpha_0] A_1 [\alpha_1] \dots A_m [\alpha_m] \bullet$.

Regrettably, an isomorphism between sequences of parsing steps of ϵ-LR parsers and the corresponding LR parsers is not possible for ϵ-LR(k) and ϵ-LALR(k) parsing, where $k > 0$. This is because merging derived items causes loss of information on the lookahead of items. This causes the parser to be sent up blind alleys which are not considered by the corresponding LR parser.

Because ϵ-LR parsing retains the prefix-correctness of traditional LR parsing (that is, upon incorrect input the parser does not move its input pointer across the first invalid symbol), the blind alleys considered by an ϵ-LR parser but not the corresponding LR parser are of limited length, and therefore unimportant in practical cases.

Theoretically however, the extra blind alleys may be avoided by attaching the lookahead information not to the state on top of the stack before reduction but to the state on top after popping m states and grammar symbols off the stack (m as in Section 2.3). This means that we have lookahead (a set of strings, each of which not longer than k symbols) for each state q and nonterminal A for which *goto* (q, A) is defined.

In the cases we have examined, the number of pairs (q, A) for which *goto* (q, A) is defined is larger than the total number of items $A \rightarrow \alpha\bullet$ in all states (about 4 to 25 times as large), so this idea is not beneficial to the memory requirements of storing lookahead information. In the case of ϵ-LR(k) parsing ($k > 0$), this idea may however lead to a small reduction of the number of states, since some states may become identical after the lookahead information has been moved to other states.

4 Calculation of items

In this section we investigate the special properties of the closure function for ϵ-LR parsing. First we discuss the closure function for ϵ-LR(k) parsing and then the equivalent notion of kernel items in ϵ-LR parsing.

4.1 The closure function for ϵ-LR(k) parsing

If w is a string and k a natural number, then $k : w$ denotes w if the length of w is less than k, and otherwise it denotes the prefix of w of length k. We use lookaheads which may be less than k symbols long to indicate that the end of the string has been reached.

The initial state for ϵ-LR(k) parsing ($k > 0$) is

$$Q_0 = \mathit{closure}\ (\{[S' \rightarrow \bullet S, \epsilon]\})$$

The closure function for ϵ-LR(k) parsing is

$$\begin{aligned}
&\mathit{closure}\ (q)\ = \\
&\quad \{[B \rightarrow \delta\bullet\theta, x] \mid \\
&\qquad [A \rightarrow \alpha\bullet\beta, w] \in q \wedge \beta \rightarrow^* B\gamma \wedge \\
&\qquad B \rightarrow \delta\theta \wedge \\
&\qquad \exists v[v \neq \epsilon \wedge \delta\theta \rightarrow^* v] \wedge \\
&\qquad \delta \rightarrow^* \epsilon \wedge \\
&\qquad \exists y[\gamma \rightarrow^* y \wedge x = k : yw]\} \\
&\quad \cup \\
&\qquad \{[A \rightarrow \alpha\delta\bullet\beta, w] \mid \\
&\qquad\quad [A \rightarrow \alpha\bullet\delta\beta, w] \in q \wedge \delta \rightarrow^* \epsilon\}
\end{aligned}$$

4.2 The determination of smallest representative sets

In traditional LR parsing, items are divided into *kernel* items and *nonkernel* items. Kernel items are $S' \rightarrow \bullet S$ and all items whose dots are not at the left end. The nonkernel items are all the others. (At this stage we abstain from lookahead.)

As we will only be looking in this section at sets of items which are either Q_0 or of the form *goto* (q, X), which result after application of the closure function, we have that the kernel items from a set of items q are a *representative subset* of q. This means that we can

- construct the complete set of items q by applying the closure function to the representative subset, and
- determine whether two sets of items are equal by determining the equality of their representative subsets.

Because the set of kernel items from a set q is in general much smaller than q itself, kernel items are very useful for the efficient generation of LR parsers.

Regrettably, in the case that the grammar contains many epsilon rules, the set of kernel items from a set q may not be much smaller than q itself. In this case therefore, kernel items may not be very useful for generation of ϵ-LR parsers.

Another approach to finding representative subsets for traditional LR parsing can be given in terms of the stages in which the goto function is executed. According to this principle, the representative subset of *goto* (q, X) is

$$K(q, X) = \{A \rightarrow \alpha X \bullet \beta | A \rightarrow \alpha\bullet X\beta \in q\}$$

and other items in *goto* (q, X) are obtained by applying the closure function to $K(q, X)$.

In the case of traditional LR parsing, K computes exactly the kernel items in *goto* (q, X), and therefore the two methods for finding representative subsets are equivalent. That this does not hold for ϵ-LR parsing can be easily seen by investigating the definition of *closure* in Section 2.3: according to the second part

$$\{A \rightarrow \alpha\delta\bullet\beta \mid A \rightarrow \alpha\bullet\delta\beta \in q \wedge \delta \rightarrow^* \epsilon\}$$

in this definition, the dot can be shifted over nullable members and therefore new items can be added whose dots are not at the left end. Therefore, some kernel items may not be in $K(q, X)$.

It turns out that we can also not use K for finding representative subsets in the case of ϵ-LR parsing. The reason is that K does not provide a *well-defined* method to find representative subsets. I.e. for some grammars we can find sets of items q_1 and q_2 and symbols X and Y such that $\mathit{goto}\ (q_1, X) = \mathit{goto}\ (q_2, Y)$ but $K(q_1, X) \neq K(q_2, Y)$.

The solution that we propose is more refined than the methods in traditional LR parsing.

First, we determine the equivalence relation of mutually left-recursive nonterminals, whose classes are denoted by $[A]$. Thus, $[A] = \{B | A \rightarrow^* B\alpha \wedge B \rightarrow^* A\beta\}$.

A nice property of these classes is that $A \rightarrow \bullet\alpha \in q$ and $B \in [A]$ together imply that $B \rightarrow \bullet\beta \in q$ for every rule $B \rightarrow \beta$. Using this fact, we can replace every item $A \rightarrow \bullet\alpha$ in q by $[A]$ without loss of information.

We define the set Z to be the union of the set of all items and the set of equivalence classes of mutually left-recursive nonterminals. The variables $E, E', \ldots$ range over elements from Z.

Our goal is to find a representative set $q' \subseteq Z$ for each set of items q.

First, we define the binary relation *induces* on elements from Z such that

- *induces* (I, J) for items I and J if and only if $I = A \rightarrow \alpha\bullet B\beta$ and $J = A \rightarrow \alpha B\bullet\beta$ and $B \rightarrow^* \epsilon$

- *induces* (I, E) for item I and class E if and only if $I = A \rightarrow \alpha\bullet B\beta$ and $B \in E$

- *induces* (E, E') for classes E and E' if and only if $E \neq E'$ and there are $A \in E$ and $B \in E'$ such that $A \rightarrow \alpha B\beta$ and $\alpha \rightarrow^* \epsilon$

- *induces* (E, I) for class E and item I if and only if there is $A \in E$ such that $I = A \rightarrow \alpha\bullet\beta$ and $\alpha \rightarrow^* \epsilon$

The smallest set $\mathit{repr}\ (q) \subseteq Z$ representing a set of items q can now be determined by the following steps:

1. Determine $q_1 \subseteq Z$ by replacing in q every item $A \rightarrow \bullet\alpha$ by $[A]$.

2. Let q_2 be the subset of q_1 which results from eliminating all items I such that *induces* (E, I) for some equivalence class $E \in q_1$.

3. Determine the set $\mathit{repr}\ (q)$ defined by $\{E \in q_2 | \neg\exists E' \in q_2[\mathit{induces}\ (E', E)\}$.

The reason that no information is lost in step 3 is that the relation *induces* restricted to q_2 is not cyclic.

That $\mathit{repr}\ (q)$ is the smallest set $q' \subseteq Z$ representing q can be formalized by stating that it is the smallest subset q' of Z such that $\mathit{closure}\ (q') = q$, where the definition of *closure* is redefined to

$$\begin{array}{l} \mathit{closure}\ (q) \ = \\ \quad \{B \rightarrow \delta\bullet\theta \mid (A \rightarrow \alpha\bullet\beta \in q \wedge \beta \rightarrow^* B\gamma \vee \\ \quad\quad\quad [A] \in q \wedge A \rightarrow^* B\gamma) \wedge \\ \quad\quad B \rightarrow \delta\theta \wedge \\ \quad\quad \exists v[v \neq \epsilon \wedge \delta\theta \rightarrow^* v] \wedge \\ \quad\quad \delta \rightarrow^* \epsilon\} \\ \quad \cup \\ \quad \{A \rightarrow \alpha\delta\bullet\beta \mid A \rightarrow \alpha\bullet\delta\beta \in q \wedge \delta \rightarrow^* \epsilon\} \end{array}$$

It is evident that the $\mathit{repr}\ (\mathit{goto}\ (q, X))$ must be calculated from the $K(q, X)$ rather than from their closures $\mathit{goto}\ (q, X)$ if efficient parser construction is required. The appropriate restatement of the algorithm calculating *repr* is straightforward.

5 Memory requirements

In this paper we have described three methods of making the (generalized) LR parsing technique applicable to hidden left-recursive grammars:

1. Apply ϵ-*elim* to the grammar before constructing the LR automaton.

2. Apply *HLR-elim* to the grammar before constructing the LR automaton.

3. Construct the ϵ-LR automaton as opposed to the LR automaton.

The last method above is derived from the first one in the sense that an ϵ-LR automaton can be seen as a compressed LR automaton for the transformed grammar ϵ-$elim(G)$. The second method is independent from the other two methods.

To investigate the static memory requirements of these methods, we have determined the number of states of the resulting automata for various grammars.

We first investigate the number of states for three kinds of characteristic grammars:

For every $k \geq 0$ we have the grammar G_1^k defined by the rules

$$\begin{array}{lcl} S & \rightarrow & B_1 \ldots B_k c \\ B_1 & \rightarrow & \epsilon \\ B_1 & \rightarrow & b_1 \\ & \vdots & \\ B_k & \rightarrow & \epsilon \\ B_k & \rightarrow & b_k \end{array}$$

For every $k \geq 1$ we have the grammar G_2^k defined by the rules

$$\begin{array}{lcl} S & \rightarrow & B_1 \ldots B_k S c \\ S & \rightarrow & d \\ B_1 & \rightarrow & \epsilon \\ B_1 & \rightarrow & b_1 \\ & \vdots & \\ B_k & \rightarrow & \epsilon \\ B_k & \rightarrow & b_k \end{array}$$

For every $k \geq 2$ we have the grammar G_3^k defined by the rules

$$\begin{array}{lcl} S & \rightarrow & B_1 \ldots B_k c \\ B_1 & \rightarrow & \epsilon \\ B_1 & \rightarrow & S \\ & \vdots & \\ B_k & \rightarrow & \epsilon \\ B_k & \rightarrow & S \end{array}$$

The grammars of the first group contain no left recursion. The grammars of the second group contain one occurrence of hidden left recursion, and there are k nullable nonterminals hiding the left recursion. The grammars of the third group contain $k-1$ occurrences of hidden left recursion, the j-th one of which is hidden by $j-1$ nullable nonterminals.

Figure 5 shows the numbers of states of various automata for these grammars. It also shows the numbers of states of the LR(0) automata for the original grammars. This kind of automaton does of course not terminate in the case of hidden left recursion, except if the nondeterminism is realized using cyclic graph-structured stacks, against which we raised some objections in Section 1.

These results show that the number of states is always smallest for the ϵ-LR(0) automata. A surprising case is the group of grammars G_3^k, where the number of states of ϵ-LR(0) is 6, regardless of k, whereas the numbers of states of the LR(0) automata for ϵ-$elim(G)$ and *HLR-elim*(G) are exponential and quadratic in k, respectively.

In the above grammars we have found some features which cause a difference in the number of states of the automata constructed by the mentioned four methods. The results suggest that ϵ-LR parsing is more efficient in the number of states for grammars containing more hidden left recursion.

The number of states of LR and ϵ-LR automata is however rather unpredictable, and therefore the above relations between the number of states for the four methods may deviate dramatically from those in the case of practical grammars.

Practical hidden left-recursive grammars do however not yet occur frequently in natural language research. The reason is that they are often considered “ill-designed” (Nozohoor-Farshi, 1989) as they cannot be handled using most parsing techniques.

Fortunately, we have been able to find a practical grammar which contains enough hidden left recursion to perform a serious comparison. This grammar is the context-free part of the Deltra grammar, developed at the Delft University of Technology (Schoorl — Belder, 1990). After elimination of the occurrences and definitions of all predicates, this grammar contains 846 rules and 281 nonterminals, 120 of which are nullable. Hidden left recursion

Method of construction	G_1^k $(k \geq 0)$	G_2^k $(k \geq 1)$	G_3^k $(k \geq 2)$	G_{Deltra}
LR(0) for G	$2 \cdot k + 3$	$2 \cdot k + 5$	$2 \cdot k + 2$	855
LR(0) for ϵ-*elim*(G)	$2^{k+1} + k + 1$	$3 \cdot 2^k + k + 1$	$2^{k+1} + 2$	1430
LR(0) for *HLR-elim*(G)	$2 \cdot k + 3$	$\frac{1}{2} \cdot k^2 + 4\frac{1}{2} \cdot k + 3$	$\frac{1}{2} \cdot k^2 + 2\frac{1}{2} \cdot k + 1$	1477
ϵ-LR(0) for G	$2 \cdot k + 3$	$k + 6$	6	709

Figure 5: The numbers of states resulting from four different methods of constructing LR and ϵ-LR automata.

occurs in the definitions of 62 nonterminals. Rules are up to 7 members long, the average length being about 1.74 members.

The numbers of states of the automata for this grammar are given in Figure 5. These data suggest that for practical grammars containing much hidden left recursion, the relation between the numbers of states of the four different automata is roughly the same as for the three groups of small grammars G_1^k, G_2^k, and G_3^k: the LR(0) automata for ϵ-*elim*(G) and *HLR-elim*(G) both have a large number of states. (Surprisingly enough, the former has a *smaller* number of states than the latter, although ϵ-*elim*(G) is about 50 % larger than *HLR-elim*(G), measured in the number of symbols.) The ϵ-LR(0) automaton for G has the smallest number of states, even smaller than the number of states of the LR(0) automaton for G.

Although these results are favourable to ϵ-LR parsing as a parsing technique requiring small parsers, not for all practical grammars will ϵ-LR automata be smaller than their traditional LR counterparts. Especially for grammars without left recursion, we have found small increases in the number of states. We consider these grammars not characteristic however because they were developed explicitly for top-down parsing.

Conclusions

We have described a new solution to adapt (generalized) LR parsing to grammars with hidden left recursion. Also LR parsing of cyclic grammars has been discussed. We claim that our solution yields smaller parsers than other solutions, measured in the number of states. This has been corroborated by theoretical data on small grammars and by an empirical test on a practical grammar for a natural language.

Our solution requires the investigation of the parse stack. We feel however that this does not lead to deterioration of the time complexity of parsing: investigation of the stack for each reduction with some rule requires a constant amount of time. This amount of time is linear in the length of that rule, provided investigation of the symbols on the stack is implemented using a finite state automaton.

The results of our research are relevant to realization of generalized LR parsing using backtracking (possibly in combination with memo-functions) or using acyclic graph-structured stacks. Furthermore, various degrees of lookahead may be used.

We hope that our research will convince linguists and computer scientists that hidden left recursion is not an obstacle to efficient LR parsing. This may in the long term simplify the development of grammars, since hidden left recursion does not have to be avoided or eliminated.

Acknowledgements

We received kind help from Job Honig, Theo Vosse, John Carroll, and Hans de Vreught in finding a practical grammar to test our algorithms on. We acknowledge valuable correspondence with Giorgio Satta, René Leermakers and José Fortes.

References

Aho, A.V. — R. Sethi — J.D. Ullman (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Aho, A.V. — J.D. Ullman (1972). *Parsing*, The Theory of Parsing, Translation and Compiling, volume 1. Prentice-Hall.

Fortes Gálves, J. (1992). Generating LR(1) parsers of small size. In: *Compiler Construction, 4th International Conference*, LNCS 641, 16–29, Springer-Verlag.

Graham, S.L. — M.A. Harrison — W.L. Ruzzo (1980). An improved context-free recognizer. *ACM Trans. Prog. Lang. Syst.* 2(3), 415–462.

Heilbrunner, S. (1985). Truly prefix-correct chain-free LR(1) parsers. *Acta Inf.* 22, 499–536.

Koster, C.H.A. (1971). Affix grammars. In: Peck, J.E.L. (Ed): *ALGOL68 Implementation*, 95–109. North Holland Publishing Company.

Lang, B. (1974). Deterministic techniques for efficient non-deterministic parsers. In: *Automata, Languages and Programming, 2nd Colloquium*, LNCS 14, 255–269, Springer-Verlag.

Leermakers, R. (1992). A recursive ascent Earley parser. *Inf. Process. Lett.* 41(2), 87–91.

Leermakers, R. (1993). *The Functional Treatment of Parsing*. Kluwer Academic Publishers.

Leermakers, R. — L. Augusteijn — F.E.J. Kruseman Aretz (1992). A functional LR parser. *Theoretical Comput. Sci.* 104, 313–323.

Leiss, H. (1990). On Kilbury's modification of Earley's algorithm. *ACM Trans. Prog. Lang. Syst.* 12(4), 610–640.

Meijer, H. (1986). *Programmar: A Translator Generator*. PhD thesis, University of Nijmegen.

Nederhof, M.J. (1993a). Generalized left-corner parsing. In: *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, 305–314.

Nederhof, M.J. (1993b). A new top-down parsing algorithm for left-recursive DCGs. In: *Programming Language Implementation and Logic Programming, 5th International Symposium*, LNCS 714, 108–122, Springer-Verlag.

Nederhof, M.J. (1994). *Linguistic Parsing and Program Transformations*. PhD thesis, University of Nijmegen.

Nederhof, M.J. — J.J. Sarbo (1993). Efficient decoration of parse forests. In: Trost, H. (Ed): *Feature Formalisms and Linguistic Ambiguity*, 53–78. Ellis Horwood.

Nederhof, M.J. — G. Satta (1994). An extended theory of head-driven parsing. In: *32nd Annual Meeting of the Association for Computational Linguistics*, 210–217.

Nilsson, U. (1986). AID: An alternative implementation of DCGs. *New Generation Computing* 4, 383–399.

Nozohoor-Farshi, R. (1989). Handling of ill-designed grammars in Tomita's parsing algorithm. In: *International Workshop on Parsing Technologies*, 182–192.

Pager, D. (1970). A solution to an open problem by Knuth. *Inf. and Contr.* 17, 462–473.

Schoorl, J.J. — S. Belder (1990). Computational linguistics at Delft: A status report. Report WTM/TT 90–09, Delft University of Technology, Applied Linguistics Unit.

Sikkel, K. (1990). Cross-fertilization of Earley and Tomita. Memoranda Informatica 90-69, University of Twente.

Sippu, S. — E. Soisalon-Soininen (1990). *Parsing Theory, Vol. II: LR(k) and LL(k) Parsing.* Springer-Verlag.

Soisalon-Soininen, E. — J. Tarhio (1988). Looping LR parsers. *Inf. Process. Lett.* 26(5), 251–253.

Thorup, M. (1994). Controlled grammatic ambiguity. *ACM Trans. Prog. Lang. Syst.* 16(3), 1024–1050.

Tomita, M. (1986). *Efficient Parsing for Natural Language.* Kluwer Academic Publishers.