Extended LALR(1) Parsing

Wuu Yang

Computer Science Department
National Chiao-Tung University
Taiwan, R.O.C.
Email: wuuyang@cs.nctu.edu.tw

***Abstract*—We identify a class of context-free grammars, called the *extended LALR(1)* (ELALR(1)), which is a superclass of LALR(1) and a subclass of LR(1). Our algorithm is essentially a smarter method for merging similar states in the LR(1) state machines. LALR(1) would merge *every* pair of similar states. In contrast, our algorithm merges a pair of similar states only if no (reduce-reduce) conflicts will be created. Thus, when no conflicts occur, our algorithm produces exactly the same state machines as the LALR(1) algorithm. However, when the LALR(1) algorithm reports conflicts, our algorithm may still produce a (larger) conflict-free state machine. In the extreme case when no states can be merged, our algorithm simply returns the original LR(1) machines. An important characteristic of the ELALR(1) algorithm is that there is no backtracking. On the other hand, the ELALR(1) algorithm does not guarantee the minimum state machines.**

Keywords–context-free grammar; grammar; extended LALR parser; LR parser; LALR parser; parsing

I. INTRODUCTION

Parsing is a basic step in every compiler and interpreter. LR parsers are powerful enough to handle almost all practical programming languages. The downside of LR parsers is the huge table size. This caused the development of several variants, such as LALR parsers, which requires significantly smaller tables at the expense of reduced capability. We identify a class of context-free grammars, called the *extended LALR (1)* (ELALR(1)), which is a superclass of LALR(1) and a subclass of LR(1) [1]. Figure 6 is an example of an ELALR(1) machine.

The core of the LR parsers is a finite state machine. The LALR(1) state machine may be obtained by merging *every* pair of similar states in the LR(1) machine [6]. In case (reduce-reduce) conflicts occur due to merging (note that only reduce-reduce conflicts may occur due to merging similar states) we will have to revert to the larger, original LR(1) machine. A practical advantage of LALR(1) grammars is the much smaller state machines than the original LR(1) machines. However, any conlficts will force the parser to use the larger LR(1) machine. The crux of our approach is to merge only pairs of similar states that do not cause conflicts.

A simple ELALR(1) grammar can be easily created by sequentially composing an LALR(1) grammar and an LR(1) grammar, as follows:

$$
\begin{array}{ll}
P \to U; S & \\
U \to \ldots & \text{(LALR(1))} \\
S \to \ldots & \text{(LR(1) but not LALR(1))}
\end{array}
$$

Assume the subset of production rules starting from the nonterminal U is completely independent of that starting from S. Further assume the U rules form an LALR(1) grammar and the S rules form a LR(1) but not LALR(1) grammar. The sequential composition of the U and S would form an ELALR(1) grammar. For parsing, we need to use the much larger LR(1) state machine.

In this paper, we propose a method that can somehow reduce the state machine of a LR(1)-but-non-LALR(1) grammar because certain parts of the grammar could be LALR(1) and hence these parts can be handled with the smaller LALR(1) state machine. The rest will be parsed with the stadard LR(1) machine.

Our idea is that we start from the LR(1) machine of the grammar and merge as many (similar) states as possible under the constraint that no conflict will be created due to merging. The resulting machine is called the ELALR(1) machine, which can be used in the standard shift/reduce LR parser.

Our algorithm never backtracks. Once the merging of a pair of similar states is committed, the pair of states will remain merged. The algorithm will never undo the merge later. The contribution of this paper is that we arrange the order of merging similar states so that no backtracking is needed. However, our algorithm does not guarantee a smallest ELALR(1) state machine.

LR(1) parsers are powerful enough to handle most practical programming languages [8]. The cannonical LR(1) parsers make use of big state machines. LALR(1) parsers [4] are deemed more practical in that much smaller state machines are used. There are several algorithms for computing the LALR machines and lookaheads efficiently [3][5][10]. None of these attempted to parse non-LALR grammars. The classical parser generator *yacc* [7] is based on the LALR(1) grammars. *Yacc* relies on ad hoc rules, such as the order of productions in the grammar, to resolve conflicts in order to apply an LALR parser to a non-LALR grammar. In contrast, this paper, which addresses ELALR(1) grammars, does not employ ad hoc rules. It is known that every language that admits an LR(k) parser

 ISBN: 978-1-61208-555-5

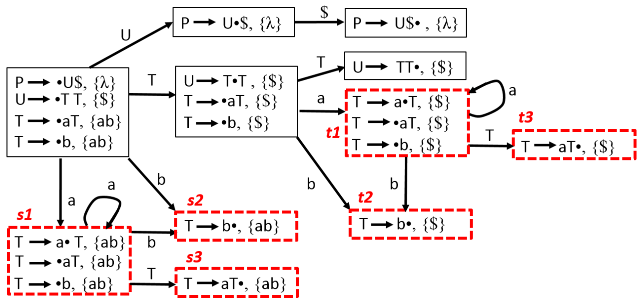

Fig. 1. The LR(1) machine of G_1.

also admits an LALR(1) parser [9]. In order to parse for a non-LALR(1) grammar, there used to be three approaches: (1) use the LR(1) parser; (2) add some ad hoc rules to the LALR(1) parser, similar to what yacc does; and (3) transform the grammar into LALR(1) and then generate a parser. The transformation approach may exponentially increase the number of productions [9] and the transformed grammar is usually unnatural. Our approach provides a fourth alternative: use the extended LALR(1) state machines.

The remainder of this paper is organized as follows. Section 2 will introduce the terminology and explain the extended LALR(1) grammars with examples. Our algorithm is presented in Section 3. Its correctness is also discussed there. Section 4 concludes this paper. In this paper, we are concerned only with one-token lookahead, that is, LR(1), LALR(1), and ELALR(1). We sometimes omit the "(1)" notation if no confusion occurs. However, our method may be extended to ELALR(k).

II. Background and Motivation

A grammar $G = (N, T, P, S)$ consists of a non-empty set of nonterminals N, a non-empty set of terminals T, a non-empty set of production rules P and a special nonterminal S, which is called the start symbol. We assume that $N \cap T = \emptyset$. A production rule has the form

$$A \to \gamma$$

where A is a nonterminal and γ is a (possibly empty) string of nonterminals and terminals. We use the production rules to derive a string of terminals from the start symbol.

LR parsing is based on a deterministic finite state machine, called LR machine. A state in the LR machine is a non-empty set of items. An item has the form $A \to \alpha \bullet \beta, la$, where $A \to \alpha\beta$ is one of the production rules, $\bullet$ indicates a position in the string $\alpha\beta$, and la (the *lookahead* set) is a set of terminals that could follow the nonterminal A in later derivation steps. Two states in the LR machine are *similar* if they have the same number of items and the corresponding items differ only in the lookahead sets. For example, states $s1$ and $t1$ in Figure 1, each of which contains three items, are similar states.

LALR(1) machines are obtained from LR(1) machines by merging *every* pair of similar states. For example, Figure 2 is the LALR(1) machine obtained from the LR(1) machine in

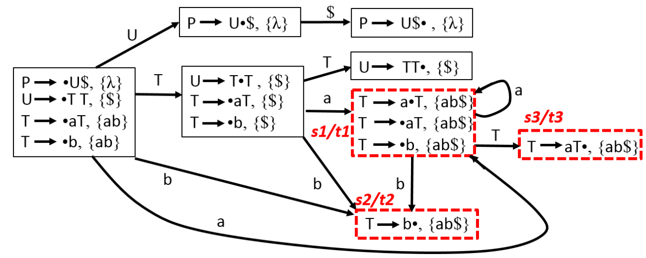

Fig. 2. The LALR(1) machine of G_1.

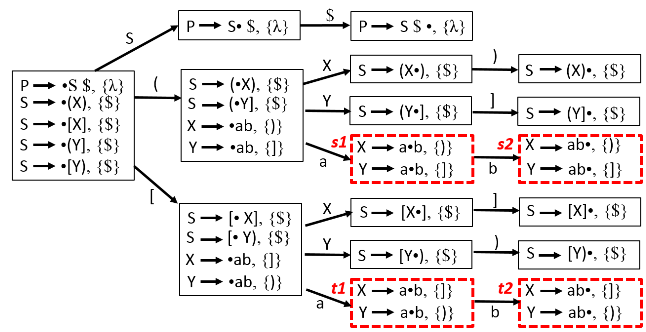

Fig. 3. The LR(1) machine of G_2.

Figure 1 by merging three pairs of similar states: $s1$ and $t1$; $s2$ and $t2$; and $s3$ and $t3$. (Remember two states are similar if they have the same items, except that the lookahdead sets might differ. To *merge two similar states*, we use the same items in the original states, except that the lookahead set of an item is the union of the lookahead sets of the two corresponding items in the original states.) The exact construction of LR and LALR machines from a context-free grammar is discussed in most compiler textbooks, such as [2][6].

Consider grammar G_1:

R1 $P \to U\$$
R2 $U \to TT$
R3 $T \to aT$
R4 $T \to b$

The LR(1) machine of G_1 is shown in Figure 1. States $s1$ and $t1$ are similar states. So are states $s2$ and $t2$ and states $s3$ and $t3$. The three pairs of states can be merged. The resulting machine is shown in Figure 2. Since there is no conflict in the resulting machine, G_1 is LALR(1).

Consider grammar G_2:

R5 $P \to S\$$
R6 $S \to (X)$
R7 $S \to [X]$
R8 $S \to (Y]$
R9 $S \to [Y)$
R10 $X \to ab$
R11 $Y \to ab$

The LR(1) machine of G_2 is shown in Figure 3. Since there is no conflict in Figure 3, grammar G_2 is LR(1).

 ISBN: 978-1-61208-555-5

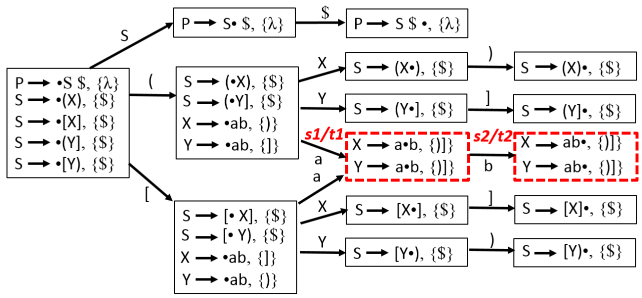

Fig. 4. The LALR(1) machine of G_2, which contains reduce-reduce conflicts.

There are two pairs of similar states in Figure 3: states $s1$ and $t1$; states $s2$ and $t2$. Figure 4 shows the resulting LALR(1) machine by merging the two pairs of similar states. Note that there are two reduce-reduce conflicts in state $s2/t2$ in Figure 4. Therefore, states $s2$ and $t2$ should not be merged. Furthermore, states $s1$ and $t1$ should not be merged either because LR machines are deterministic.

It is easy to combine G_1 and G_2 into a single grammar, in which some, but not all, pairs of similar states may be merged without creating conflicts. For instance, consider grammar G_3 below, which is a sequential combination of G_1 and G_2. Prodcution rules `R1` and `R6` are combined as a single new rule.

R1	$P \to US\$$
R2	$U \to TT$
R3	$T \to aT$
R4	$T \to b$
R6	$S \to (X)$
R7	$S \to [X]$
R8	$S \to (Y]$
R9	$S \to [Y)$
R10	$X \to ab$
R11	$Y \to ab$

Grammar G_3 is clearly not LALR(1). Hence the larger LR(1) machine must be used in parsing. However, it is still possible to reduce the size of LR machine for G_3. Figure 5 shows the LR(1) machine of G_3. There are five pairs of similar states: states $s1$ and $t1$; $s2$ and $t2$; $s3$ and $t3$; $s4$ and $t4$; and $s5$ and $t5$; However, states $s5$ and $t5$ cannot be merged due to a potential conflict. Furthermore, states $s4$ and $t4$ cannot be merged because LR machines must be deterministic. The other three pairs of similar states may be safely merged, creating a machine smaller than the standard LR(1) machine. The resulting machine is shown in Figure 6, which is called the *extended LALR(1) machine* of G_3.

Definition. A grammar is *ELALR(1)* if and only if at least a pair of similar states in its LR(1) machine may be merged without creating conflicts.

In other words, a grammar is ELALR(1) if and only if it has a conflict-free state machine that is smaller than the grammar's LR(1) machine.

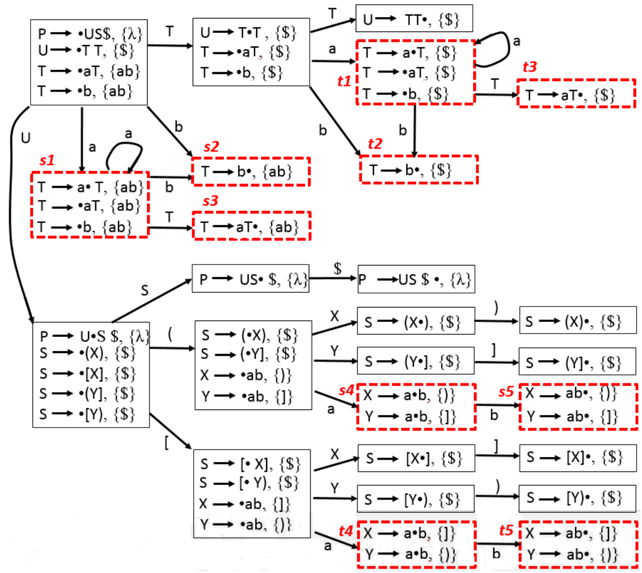

Fig. 5. The LR(1) machine of G_3.

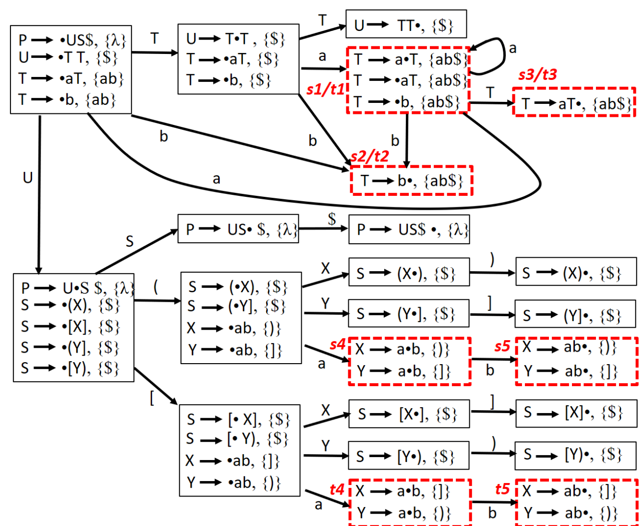

Fig. 6. The ELALR(1) machine of G_3.

In determining which pairs of similar states may be safely merged, trial-and-error is a straightforward method. However, we can do better.

We will start from a few observations and facts. First note that the LR(1) as well as the LALR(1) machines are all deterministic.

Theorem 1: Merging two similar states in LR(1) machines can possibly create reduce-reduce conflicts, but never shift-reduce conflicts.

For example, in Figure 4, there are reduce-reduce conflicts in the merged state $s2/t2$.

Lemma 2: Consider the fragment of an LR machine in Figure 7. If states $s1$ and $t1$ are similar, then (1) states $s1$ and $t1$ have the same number of successor states; and (2) their

 ISBN: 978-1-61208-555-5

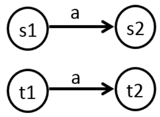

Fig. 7. A pair of similar states in an LR(1) machine. States $s1$ and $t1$ can be merged only if states $s2$ and $t2$ are merged. There will be an edge $(s1, t1) \to^a (s2, t2)$ in the similarity graph.

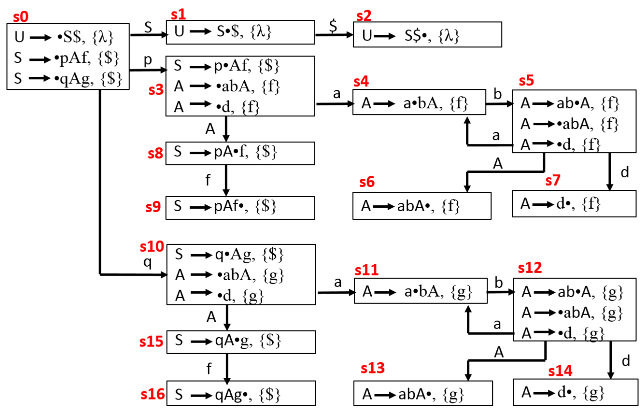

Fig. 8. The LR(1) machine for the example grammar.

corresponding successor states are also similar, *i.e.*, states $s2$ and $t2$ are also similar.

Proof. This is due to the construction of the LR machine. Q.E.D.

Lemma 3: Consider the fragment of an LR machine in Figure 7. Assume states $s1$ and $t1$ are similar (and hence states $s2$ and $t2$ are also similar). States $s1$ and $t1$ can be merged only if states $s2$ and $t2$ are merged.

Proof. This lemma is due to the fact that LR/LALR machines must be deterministic. Q.E.D.

Corollary 4: Consider The fragment of an LR machine in Figure 7. Assume states $s1$ and $t1$ are similar (and hence states $s2$ and $t2$ are also similar). If states $s2$ and $t2$ are not merged, then states $s1$ and $t1$ cannot be merged.

Due to Corollary 4, we should try to merge $s2$ and $t2$ *before* we try to merge $s1$ and $t1$. In general, when similar states in an LR(1) machine are merged, the order of merging had better be from leaves to root. However, the LR(1) machine may contain cycles and is not a tree in general.

Our algorithm, presented in the next section, will take care of these details. A directed cycle in a directed graph, such as states $s4$ and $s5$ in Figure 8, is called a *strongly connected component* (scc). According to Corollary 4, all states in an scc in the LR(1) machine must be merged with their respective similar states simultaneously or none should. In Figure 8, either both pairs of similar states $(s4, s11)$ and $(s5, s12)$ are merged or no pair should be merged. Due to this restriction, we use *aggregates*, which are sets of pairs of similar states, to represent two separate sccs that might be merged.

III. Algorithm

In this section, we explain our algorithm for constructing the finite state machine for extended LALR(1) grammars. We will use the following grammar G_4 to illustrate our algorithm. Figure 8 is the LR(1) machine for this grammar.

R1	$U \to S\$$
R2	$S \to pAf$
R3	$S \to qAg$
R4	$A \to abA$
R5	$A \to d$

We will start from an LR(1) grammar. Draw the LR(1) state machine of the grammar. Since the grammar is LR(1), there is no conflict in the LR(1) machine.

Then the strongly connected components in the LR(1) machine are identified. In Figure 8, states $s4$ and $s5$ form a strongly connected component (scc). So do states $s11$ and $s12$. Strongly connected components in the LR(1) machine can be traced to (direct or indirect) recursive production rules in the grammar. In Figure 8, the scc is due to the recursive rule $A \to abA$.

We then find all pairs of similar states. In Figure 8, there are four pairs of similar states: $(s4, s11)$, $(s5, s12)$, $(s6, s13)$, and $(s7, s14)$.

We then build the *similarity* graph. The similarity graph for the example grammar is shown in Figure 9(a). Initially, the similarity graph contains only vertices; edges are gradually inserted into the graph. Each vertex in the similarity graph denotes a pair of similar states taken from the LR(1) machine. During the construction of the similarity graph, we may add vertices of the form (s, s) (*i.e.*, a pair of identical states), which have no outgoing edges. For each pair of similar states (s_1, t_1), either (1) s_1 and t_1 are exactly the same state, or (2) neither has a successor state, or (3) they have the same number of successor states. Each successor state of s_1 corrresponds to exactly one successor state of t_1. Note that the corresponding successors of s_1 and t_1 are s_2 and t_2 if the edges $s_1 \to^u s_2$ and $t_1 \to^u t_2$ carry the same label, which is u in this case.

In case (3), we add an edge from the vertex representing the pair (s_1, t_1) to the vertex representing the pair (s_2, t_2). This edge is denoted as $(s_1, t_1) \to^u (s_2, t_2)$ in the similarity graph, which indicates that s_1 and t_1 may be merged only if s_2 and t_2 are merged, according to Lemma 3. (Note that a pair of similar states could be written either as (s, t) or (t, s). In constructing the similarity graph, we had better fix the order of the pair of states. The first time the pair is encountered, the order of s and t is determined from the already constructed part of the similarity graph. When the pair is encountered again, we should use the order determined previously. Otherwise, a part of the similarity graph may be duplicated. Duplication only makes the algorithm spends more time but has no effect on the final result.)

Figure 9(a) is the similarity graph for the LR machine in Figure 8. Note that there may or may not be cycles in a

 ISBN: 978-1-61208-555-5

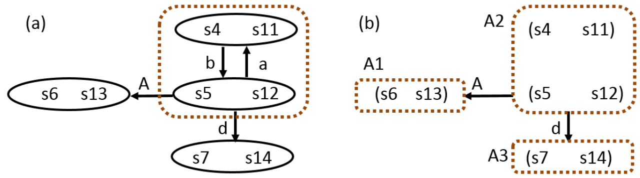

Fig. 9. (a) The similarity graph for Figure 8, in which each vertex represents a pair of similar states in the LR(1) machine. (b) The aggregation graph, in which each node (called an *aggregate*) represents a set of pairs of similar states. The aggregation graph must be acyclic.

similarity graph. Since each vertex in the similarity graph represents a pair of states, we can switch the order of every pair of states, that is, from (s, t) to (t, s), resulting in an isomorphic graph.

Now consider the similarity graph. If vertices (*i.e.*, pairs of states), say $(p_1, q_1), (p_2, q_2), \ldots, (p_h, q_h)$, form a strongly connected component in the similarity graph (this implies that the states $p_1, p_2, \ldots, p_h$ form a strongly connected component in the LR(1) machine. So do the states $q_1, q_2, \ldots, q_h$), then mark these adjacent vertices as an *aggregate*. Finally, the pair of similar states which does not belong to any aggregates will form an aggregate by itself. We may say that the set of all pairs of similar states are partitioned into aggregates. In Figure 9(a), the two pairs of similar states $(s4, s11)$ and $(s5, s12)$ form an aggregate. Each of the remaining two pairs forms an aggregate by itself. These three aggregates, which are labeled *A1*, *A2*, and *A3*, constitute the *aggregation graph*, which is shown in Figure 9(b). Note that each pair of similar states (*i.e.*, a vertex) belongs to exactly one aggregate. We say one aggregate, say A_i, is a *successor* of another aggregate, say A_j, if there is an edge $A_j \rightarrow A_i$ in the aggregation graph. In Figure 9(b), $A1$ and $A3$ are successors of $A1$.

An aggregate is a set of pairs of similar states. These states are closely related to the strongly connected components in the LR(1) machine. According to Lemma 3, every pair of similar states in an aggregate must be merged if any pair of similar states in the same aggregate are merged or none will be merged.

Note that the resulting aggregation graph is acyclic. Thus, we can find a reverse topological order of the aggregates in the aggregation graph. For each aggregate A in the reverse topological order, first check if any of the successor aggregates of A is marked as *unmergable*. If so, then mark this aggregate A also as *unmergable*. Otherwise every pair of similar states in A are merged. If any conflicts occur due to the merge, then undo the merge and mark the aggregate A as *unmergable*. On the other hand, when no conflicts occur, this means that all the pairs of similar states in A can be safely merged. That is, the merge is comitted. We will proceed with the next aggregate in the reverse topological order.

In Figure 9(b), a reverse topological order is $A1$, $A3$, and $A2$. So we will merge the pair of similar states in aggregate $A1$ first. Then we attempt to merge the pair of similar states in aggregate $A3$. Finally we attempt to merge the two pairs of similar states in aggregate $A2$. Figures 10 shows the final state machine, which is actually the LALR(1) machine since this grammar is LALR(1).

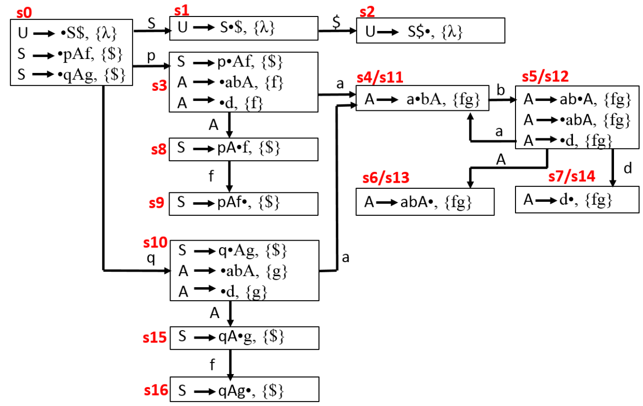

Fig. 10. The state machine after merging four pairs of states. This is actually the LALR(1) state machine.

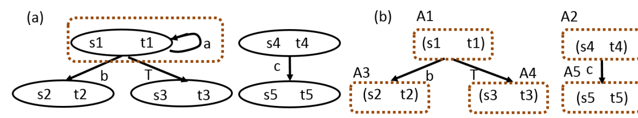

Fig. 11. The similarity graph and the aggregation graph for grammar G_3.

If a grammar is LALR(1), that is, every pair of similar states can be merged without creating conflicts, our algorithm will eventually construct the LALR(1) machine. In the other extreme, if no similar states could be merged without creating conflicts, our algorithm will not merge any states and simply return the original LR(1) machine.

Example. The similarity graph and the aggregation graph for grammar G_3 are shown in Figure 11 (a) and (b), respectively. A topological order is $A3, A4, A1, A5, A2$. The pair of similar states $(s5, t5)$ cannot be merged due to a conflict. Consequently, the pair of similar states $(s4, t4)$ cannot be merged either. The resulting ELALR(1) state machine is shown in Figure 6.

Correctness of the algorithm. There are only finite pairs of similar states in an LR(1) state machine. The transitions among pairs of similar states are also finite. Hence, the similarity graph is finite and can be built in a finite amount of time. The aggregation graph is essentially a reduced similarity graph. Thus, it is also finite and can be built in a finite amount of time.

All the pairs of similar states in the LR(1) machine are partitioned into aggregates. Merging similar states is attempted step by step. In each step, all pairs of an aggregate are merged (if no conflicts occur) or ignored (otherwise). The aggregates are examined in a reverse topological order.

The starting point of our algorithm is the original LR(1) state machine, which is deterministic and satisfies the Viable-Prefix Lemma [6]. Let M be the state machine immediately

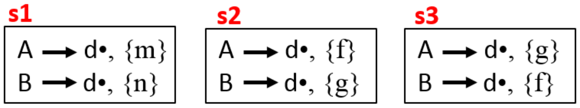

Fig. 12. Three similar states for grammar G_5.

before a step and M' be the one immediately after that step. We may make the following claim:

Claim. If M is deterministic and satisfies the Viable-Prefix Lemma, then M' is also deterministic and satisfies the Viable-Prefix Lemma.

Let AG be the aggregate considered in the current step. If no merging is done in the current step, $M' = M$. If some pairs of similar states are merged in the current step, due to the reverse topological order of merging, all pairs of similar states in all successor aggregates of AG in the aggregation graph have been merged. Therefore, M' is still deterministic. Furthermore, every path in M corresponds to exactly one path in M' and every path in M' is a path in M. If M satisfies the Viable-Prefix Lemma, so does M'.

Note that merging starts from an LR(1) machine, which is deterministic and satisfies the Viable-Prefix Lemma. We conclude that that the state machine eventually produced by our ELALR(1) algorithm is deterministic and satisfies the Viable-Prefix Lemma. Hence we may claim the correctness of our algorithm.

The ELALR(1) machines produced by our algorithm need not be minimum. Consider the following grammar G_5:

R1 $U \to S\$$
R2 $S \to pAf$
R3 $S \to pBg$
R4 $S \to qAg$
R5 $S \to qBf$
R6 $S \to rAm$
R7 $S \to rBn$
R8 $A \to d$
R9 $B \to d$

The LR(1) machine for G_5 will contain three similar states shown in Figure 12. States $s1$ and $s2$ may be safely merged. So do states $s1$ and $s3$. But states $s2$ and $s3$ cannot. It is hard to decide which pair of similar states should be merged in order to achieve the minimum ELALR(1) machine.

In our algorithm, $s1$ and $s2$ will be in one aggregate; $s1$ and $s3$ will be in another; and $s2$ and $s3$ will be in a third aggregate. Exactly which pair is merged depends on the reverse topological order in which the aggregates in the aggregation graph are visited.

For grammar G_5, the resulting state machine after merging states $s1$ and $s2$ has the same size as that after merging states $s1$ and $s3$. From this example, we know that there may not be a unique minimum ELALR(1) machine in general.

An obvious approach to produce a minimum ELALR(1) machine is to try all possible reverse topological orders of aggragates in the aggregation graph. To find the minimum state machines, a naïve algorithm may try all possibilities of merging similar states. However, our algorithm will try all reverse topological orders of aggregates instead of all combinations of pairs of similar states. Since there are fewer reverse topological orders of aggregates than combinations of pairs of similar states, our aggregate-based algorithm should be faster than the pair-of-states-based exhaustive search.

If trying all reverse topological orders is out of the question, we can choose a *plausible* topological order, as follows: We assign a *weight* to each aggregate in the aggragation graph. The weight of an aggregate A is the total number of pairs of similar states in all the aggregates that can reach A in the aggregation graph. For example, in Figure 9, the weights of $A1$ and $A3$ are 3 and the weight of $A2$ is 2. If there are more than one reverse topological order, the one in which un-related aggregates, such as $A1$ and $A3$ in Figure 9, are arranged in decreasing weights is chosen. This implies that we prefer to visit *heavier* aggregates first. This is based on the observation that more pairs of similar states depend on heavier aggregates.

IV. Conclusion

We identify the class of extended LALR grammars and the associated algorithm in this paper. ELALR is located between LR and LALR. Our algorithm is essentially a smarter method for merging similar states in the LR(1) machines. Our algorithm can be extended to ELALR(k), for any k, in a straightforward manner.

Acknowledgement

This work is supported, in part, by Ministry of Science and Technology, Taiwan, R.O.C., under contracts MOST 103-2221-E-009-105-MY3 and MOST 105-2221-E-009-078-MY3.

References

[1] A.V. Aho and S.C. Johnson, "LR Parsing," ACM Computing Surveys, vol. 6, no. 2, June 1974, pp. 99-124.
[2] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools. (2nd Edition) Prentice Hall, New York, 2006.
[3] T. Anderson, J. Eve, and J. Horning, "Efficient LR(1) parsers," Acta Informatica, vol. 2, 1973, pp. 2-39.
[4] F.L. DeRemer, Practical translators for LR(k) languages. Project MAC Tech. Rep. TR-65, MIT, Cambridge, Mass., 1969.
[5] F.L. DeRemer and T. Pennello, "Efficient Computation of LALR(1) LookAhead Sets," ACM Trans. Programming Languages and Systems, vol. 4, no. 4, October 1982, pp. 615-649.
[6] C.N. Fischer, R.K. Cytron, and R.J. LeBlanc, Jr., Crafting A Compiler. Pearson, New York, 2010.
[7] S.C. Johnson, Yacc: Yet Another Compiler-Compiler. Bell Laboratories, Murray Hill, NJ, 1978,
[8] D. E. Knuth, "On the translation of languages from left to right," Information and Control, vol. 8, no. 6, July 1965, pp. 607-639. doi:10.1016/S0019-9958(65)90426-2.
[9] M.D. Mickunas, R.L. Lancaster, V.B. Schneider, "Transforming LR(k) Grammars to LR(1), SLR(1), and (1,1) Bounded Right-Context Grammars," Journal of the ACM, vol. 23, no. 3, July 1976, pp. 511-533. doi:10.1145/321958.321972
[10] D. Pager, "A practical general method for constructing LR(k) parsers," Acta Informatica, vol. 7, no. 3, 1977, pp. 249-268.