On Directly Constructing LR(k) Parsers Without Chain Reductions

Wilf R. LaLonde
Department of Systems Engineering
Carleton University
Ottawa, Canada. K1S 5B6

ABSTRACT

A chain production is a production of the form A->M where A is a nonterminal and M is either a terminal or nonterminal. Pager in [Pag5] has presented an algorithm which removes all chain reductions from LR(1) parsers _after_ they have been constructed.

In this paper, we present an algorithm for directly constructing LR(k) parsers with arbitrary subsets of the chain productions, called the _useless_ chain productions, optimized out. If this subset is empty, the algorithm is a standard one [And 1, Kn]. If this subset consists of all chain productions, the result is a parser with all chain reductions optimized away. The algorithm, as in [Pag5], also eliminates from the parsers all nonterminals which occur as the left part of useless chain productions. This latter optimization along with the chain reduction optimization significantly decreases the storage space and execution times of the parsers.

This provides an efficient solution of the open problem posed by Aho and Ullman [A&U2] for all LR(k) grammars.

INTRODUCTION

In recent years, much attention has been focused on LR(k) techniques [Kor, DeR1, DeR2, Pag1 to Pag4, A&U2, A&U3, And1, And2, Jo1, LaL]. Since LR(k) parsers are efficient and capable of parsing a large subset of the context free languages, much of the effort has been devoted to decreasing its table storage space and increasing its parsing speed. One approach for doing this has been to eliminate useless chain reductions from the parsers [A&U2, A&U3, And1, Pag5]. However, all of the techniques used above eliminate the chain reductions from the parsers after they have been constructed.

In this paper, we present a technique for directly constructing LR(k) parsers in such a way that reductions involving useless chain productions are eliminated. In addition, all references to the nonterminals occurring as the left part of these productions are also eliminated. In practice, these two optimization significantly reduces the parser storage space and also significantly increases the parsing speed.

BACKGROUND

In this section, we introduce the basic notions necessary for the paper.

A _context free grammar_ (_grammar_ for short) is a four-tuple $G = (N,T,P,S)$ where N and T are finite disjoint sets of _nonterminals_ and _terminals_ respectively, S in N is the _goal symbol_, and P is a finite set of _productions_ of the form A->w where A, the _left part_, is in N and w, the _right part_, is in $(N \cup T)^*$. The vocabulary is $N \cup T$. For parsing purposes, we reserve $\vdash$ and $\dashv$, the _left_ and _right endmarker_, as symbols distinct from any used in the vocabulary of our grammars. We also assume the productions are numbered 1,2,...,p in some order. We abbreviate productions A->w1, A->w2,...,A->wn by A->w1|...|wn.

Conventions: Let G = (N,T,P,S) be a grammar.
(1) A,B,C denote nonterminals in N.
(2) a,b,c denote terminals in T.
(3) L,M,N denote nonterminals or terminals.
(4) u,v denote strings in T* and w,x,y,z strings in $(N \cup T)^*$.
(5) e denotes the empty string.

A production in P of the form A->M where M is in $N \cup T$ is called a _chain_ production. A _useful_ chain production is one so declared by a user, otherwise, is _useless_. For example, if a user has semantic actions associated with all productions, he may declare all the chain productions to be useful. Alternatively, he may declare only a subset to be useful -- perhaps those in which the right part is a terminal, etc.

If A0->A1, A1->A2, ..., An-1->An is a sequence of useless chain productions, we call A0 a _proper ancestor_ of An and An a _proper descendant_ of A0. An _ancestor_ (or _descendant_) of a symbol M in $N \cup T$ is either M itself or a proper ancestor (or descendant) of M. A symbol without proper descendants is called a _leaf_ (_leaves_ in plural). A graph which displays this relationship is called an ancestor graph.

Example

Consider grammar G1 consisting of the four productions E->E+T | T, T->T*a | a. The ancestor graph of G1, where all chain productions have been declared useless, is shown in figure 1(a). The ancestor graph of G1, where only chain production E->T has been declared useless, is shown in figure 1(b). //

A grammar G = (N,T,P,S) is reduced if for each M in N∪T, there is some derivation of the form S=>*xMy=>*w where w is in T*. We will assume that all our grammars are reduced.

We define k:x as the first k symbols of x if |x|> k[1] and as x otherwise. If X is a set of strings in (N∪T)*, we generalize k:X as {k:x| x is in X}. We further define $FIRST^G_K(x)$ as k:{w in T*| $x=>_G{}^*w$}. We drop G when no ambiguity arises.

Example

If G is a grammar with productions S->aSbS and S->e, then FIRST1(S) = {e,a}, FIRST1(b) = {b}, and FIRST1(Sb) = {a,b}. //

An augmented grammar associated with a grammar G = (N,T,P,S) is a grammar G' = (N∪{S'},T,P∪{S'->S},S') where S' is a new nonterminal not in N∪T.

An LR(k) parsing machine M for grammar G = (N,T,P,S) based on the set Pu of useless productions of G is a finite state machine (FSM) with transition symbols of the form "X" or "X if U" where

(1) X, an action of M, is an element of
 (a) N∪T (a shift action),
 (b) P-Pu (a reduce action), or
 (c) {Accept} (an accept action), and
(2) U, a lookahead set for X, is a subset of T* such that each w in U satisfies |w|≤k.

A reduce action of the form A->w where A->w is the ith production of G is also represented by #i, a #-symbol (pronounced number symbol) of G.

We represent a parsing machine by its transition graph. As it turns out, an LR(k) parsing machine has exactly one final state and this final state has no successors. For convenience, we therefore leave the edges leading to this final state unterminated.

A state of the parsing machine is inadequate if it contains at least two actions one of which is a reduce action; otherwise it is adequate. Informally, a state is inadequate if "lookahead" is required to resolve between the actions of the state. This is not required for a state which is adequate.

1. |x| stands for the length of x.

Example

The transition graph of an LR(1) parsing machine M1 for G1 is shown in figure 3(a). States 0, 2, 4, 5, and 7 are adequate; all others are inadequate. //

The following algorithm interprets LR(k) parsing machines.

Algorithm 1 An LR(k) parsing machine interpreter.

Input An LR(k) parsing machine M for grammer G = (N,T,P,S) and input string w in T*.

Output A sequence of productions in P possibly followed by the word error.

Method

(1) [Initialize]
Let I0 be the initial state of M and let ⊢ and ⊣ be the left and right endmarker.[2] Begin with current state p = I0, stack s = (⊢,I0), and current input string u = w ⊣. Iteratively perform step (2).

(2) [Perform one parse step]
Let u = au' and v = k:u. Perform one of (3), (4), or (5) depending on which applies.[3] If none applies, output error and stop.

(3) [Shift Action]
There exists a "Shift a" action in p with v in its lookahead set (if this set exists). Stack (a,q) where q is the a-successor of p, set p = q, and set u = u'.

(4) [Reduce Action]
There exists a "Reduce A->w" action in p with v in its lookahead set (if this set exists). Output A->w, unstack |w| pairs from s leaving (M,q) as the top pair, stack (A,r)[4] where r is the B-successor of q and B is an arbitrary descendant of A which is a leaf, and set p = r.

(5) [Accept Action]
There exists an "Accept" action in p with v in its lookahead set (if this set exists). If v ≠ ⊣ , output error. In any case, stop.

2. Neither endmarker must be a member of N∪T.

3. The advantage of LR(k) parsing is that the choice is unique. Note also that the lookahead information is not always needed.

4. Of course, q must contain a "Shift B" action. Also, it is well-known that only the states are required for parsing purposes.

Example

The result of applying algorithm 1 to M1 and a+a*a is shown in figure 2(b). //

The rest of the section contains the basic definitions used in the construction technique of the next section.

Let G = (N,T,P,S) be a grammar and let Pu be the set of useless chain productions of G. An LR(k) item for G is a pair <A->x.y#i,u> where A->xy is the ith production of P and $|u| \le k$. Symbol #i is called a #-symbol (pronounced number symbol). If the dot is to the left of symbol M, the LR(k) item is said to be an M-item (#-item if M is a #-symbol).

If <A->x.y#i,u> is an LR(k) M-item of G, the action (and lookahead set) associated with it is either (1), (2), or (3) below according to whether M is nonterminal B, terminal a, or #-symbol #i.

(1) "Shift B",
(2) "Shift a if V" where V = FIRSTk(yu),
(3) "Reduce A->w if {u}" where A->w is the ith production of P.

We never associate a lookahead set with nonterminal shift actions. Two actions are inconsistent if they are distinct and yet have non-disjoint lookahead sets; otherwise, they are consistent. Thus two actions, one of which is of type (1), must always be consistent. Two actions of the form "Shift a if {u,...}" and "Reduce A->w if {u,...}" are inconsistent. So are "Reduce A->x if {u,...}" and "Reduce B->y if {u,...}" for A->x≠B->y.

If there exists actions associated with distinct LR(k) items which are inconsistent, the LR(k) items are also termed inconsistent; otherwise, they are termed consistent. A set of LR(k) items is consistent if every pair of items is consistent; otherwise, it is inconsistent.

The Algorithm for Constructing LR(k) Parsers With Chain Production Optimizations

Let G be a grammar and let Pu be the set of useless chain productions of G. We define relations $\rightarrow$ and $\downarrow$, the transition successor and immediate successor relations respectively of G based on Pu below.

Let I be the set of LR(k) items of G exclusive of those associated with elements of Pu, and let M be a leaf of G.

$\xrightarrow{M}$ = {(<A->x.Ly#i,u>, <A->xL.y#i,u>) in IxI| L is an ancestor of M (which includes M)}.

$\downarrow$ = {(<A->x.By#i,u>, <C->.z#j,v>) in IxI| C is a descendant of B (which includes B) and v is in FIRSTk(yu)}

Relation $\xrightarrow{M}$ is the transition M-successor of G based on Pu. Relation $\rightarrow$ is the union of all $\xrightarrow{M}$ such that M is a leaf of G.

Example

Consider grammar G1 of figure 1 where the productions have been numbered from 1 to 4. Suppose all chain productions are declared useless. We compute the set P0 $\xrightarrow{+}\downarrow^{*}$[5] where P0 = {<E->E.+T#1,+>}. We can write

P1 = P0$\xrightarrow{+}$ = {<E->E+.T#1,+>}

P2 = P1 $\downarrow$ = {<T->.T*P#3,+>} since T is a descendant of T and + is in FIRST 1(e+) = FIRST1(+). Note that <T->.a#4,+> is not added since T->a is a useless chain production.

P3 = P2$\downarrow$ = {<T->.T*P#3,*>} since * is in FIRST1(*P+).

P4 = P3$\downarrow$= P3. Thus the process is finished.

Therefore Q = {<E->E.+T#1,+>}$\xrightarrow{+}\downarrow^{*}$ = {<E->E+.T#1,+>, <T->.T*P#3,+>, <T->.T*P#3,*>}. Notice that Q $\xrightarrow{T}\downarrow^{*}$ is undefined since T is not a leaf of G. However, Q $\xrightarrow{a}\downarrow^{*}$ is defined.

Q^0 = Q $\xrightarrow{a}$ = {<E->E+T.#1,+>, <T->T.*P#3,+>, <T->T.*P#3,*>} since T is an ancestor of a. It can be shown that $Q^0\downarrow$ is empty. Hence Q $\xrightarrow{a}\downarrow^{*}$ = Q^0. //

Algorithm 2 Construction of LR(k) Parsing Machines.

Input Grammar G, integer k, and set Pu of useless chain productions of G.

Output The LR(k) parsing machine C for G based on Pu.

Method Perform the following steps in succession.

(1) [Initialize]
Let G' = (N',T,P',S') be the augmented grammar for G = (N,T,P,S,) and number the productions so that S'->S is the 0th production of G'. Let $\rightarrow$ and $\downarrow$ be the transition and immediate successor relations of G' based on Pu.

(2) [Compute the initial state]
Add I0 = {<S'->.S#0,⊣>} $\downarrow^{*}$[6] to the states of C (initially empty) and mark it "unprocessed".

(3) [Compute successor states]
For each "unprocessed" state R of C, mark R "processed", and for each leaf M such that R contains an LR(k) N-item where N is an ancestor of M (also for each #-symbol M), add R$\xrightarrow{M}\downarrow^{*}$ to the states of C (as the M-successor of R) and mark it "unprocessed" if it is not already there. If M is a #-symbol, the successor state will be the empty set. This state is taken as the sole final state of C.

(4) [Introduce an accept action]
Replace transition symbol #0 in state R[7] by
(a) "Accept" if R is adequate, or
(b) "Accept if {⊣}" if R is inadequate.

---

5. If R is a relation and X is a set, XR represents the set {b|(a,b) is in R and a is in X} and XR* represents the smallest set A containing X such that if a is in A, then {a}R is contained in A. The set XR1R2...Rn represents (...((XR1)R2)...)Rn.

6. Symbol ⊣ is the right endmarker not in N'∪T.

7. Only one state of C will contain transition symbol #0.

(5) [Introduce lookahead sets]
For each inadequate state R of C, and each transition symbol M of R where M is either a terminal symbol or #-symbol, add the lookahead set L where L is the union of the lookahead sets associated with the LR(k) M-items of R.

If the set of useless productions is empty, the above algorithm is a variant of Knuth's LR(k) table constructor [Kn, A&Ul]. We will refer to the LR(k) parsing machine for G based on ϕ as the principle LR(k) parsing machine for G.

As constructed, the actions of a parsing machine do not always have associated lookahead sets.[8] However, it is possible to construct the parsing machine in such a way that the associated lookahead set is added to each action. When this is done, we say that the parsing machine has forced lookahead sets.

Example

Figures 3 and 4 contain the incomplete (i.e. without lookahead) LR(1) parsing machines C1 and D1 for G1 based on Pu = {E→T, T→a} and ϕ respectively. The completed LR(1) parsing machine M1' for G1 based on {E→T, T→a} is shown in figure 5(a). The result of applying algorithm 1 to M1' and a+a*a is shown in figure 5(b). It is instructive to compare the number of steps in figures 2(b) and 5(b). //

Conditions For The Algorithm To Succeed

In this section, we show that the parsing machine M with forced lookahead obtained from an LR(k) grammar G with useless chain productions Pu in conjunction with the interpreter i.e. algorithm 1 is in fact a parser for G which outputs an "optimized" bottom up parse of an input string in L(G) i.e. a sequence of productions of G exclusive of those in Pu. We begin with some basic lemmas.

Lemma 1 Let G be an LR(k) grammar, let <A→x.My#p,u> be an LR(k) item in some state R of the principle LR(k) parsing machine for G, and let w be a string in FIRSTk(Myu). There exists in R an LR(k) item either of the form

(1) <B→.#i,w>, or
(2) <B→α.az#i,v> where w is in FIRSTk(azv).

Proof Obvious. //

Lemma 2 Let G be an LR(k) grammar, let C be the principle LR(k) CFSM for G, and let B1, B2, A, and B be vocabulary symbols of G (B may be either a terminal or nonterminal) such that

B1=>*A=>B and B2=>*B

by chain productions where each nonterminal of the derivations (except B) are different.
If A→B is the rth production of G and LR(k) items <A1→x1.B1y1#p,u1> and <A2→x2.B2y2#q,u2> (the latter different from <A→.B#r,u2>) are in the same state R of C, then

F = FIRSTk(y1u1) ∩ FIRSTk(y2u2) = ϕ.

Proof Consider S, the B-successor of R. Suppose the lemma is false. Then there exists a string w in F. Since B1=>*A=>B and w is in FIRSTk(y1u1), <A→.B#r,w> is in R. Therefore <A→B.#r,w> is in S. Now, consider B2.

If B2≠B, there exists some nonterminal D≠A such that B2=>*D=>B in the above derivation. Therefore there exists an LR(k) item of the form <D→.B#s,w> (for some #s) in R. Therefore <D→B.#s,w> is in S and it is inconsistent with <A→B.#r,w> violating the condition that G is LR(k).

If B2=B, LR(k) item <A2→x2B2.y2#q,u2> is in S. If y2=e, it is inconsistent with <A→B.#r,w>, violating the condition that G is LR(k). If y2≠e, there also exists in S an LR(k) item either of the form <D→.#i,w> or <D→α.az#i,v> where w is in FIRSTk(azv) by lemma 1. But both of these are inconsistent with <A→B.#r,w>. //

A pair <A1→x1.y1#i,u1> and <A2→x2.y2#j,u2> of LR(k) items satisfies the FIRSTk condition if FIRSTk(y1u1) ∩ FIRSTk(y2u2) = ϕ. Two sets S1 and S2 satisfy the FIRSTk condition if every pair of elements p1 and p2 in S1 and S2 respectively satisfy the FIRSTk condition.

Lemma 3 Let G be an LR(k) grammar, let C be the principle LR(k) parsing machine for G, let R1 and R2 be states of C (not necessarily distinct), and let M1 and M2 be transition symbols of R1 and R2 respectively such that M1=>*M and M2=>*M. If each pair of M1- and M2-items in R1 and R2 respectively satisfies the FIRSTk condition, then so does each pair in the M1-successor and M2-successor of R1 and R2 respectively.

Proof If p1 = <A1→x1.M1y1#i,u1> and p2 = <A2→x2.M2y2#j,u2> are arbitrary LR(k) M1- and M2-items in R1 and R2 respectively, then FIRSTk(M1y1u1) ∩ FIRSTk(M2y2u2) = ϕ. Since M1=>*M and M2=>*M, M must generate strings of length less than k. Otherwise, the FIRSTk condition would not hold. Let n be the maximum length of the generated strings. It follows that first of all $FIRST_{k-n}(y1u1) \cap FIRST_{k-n}(y2u2) = \phi$ and therefore $FIRST_k(y1u1) \cap FIRST_k(y2u2) = \phi$ since $1 \leq k-n < k$. //

A proper subset of a set of items is the largest subset which excludes those items associated with useless chain productions.

Lemma 4 Let G be an LR(k) grammar, let C be the principle LR(k) parsing machine for G, and let R1 and R2 be states of C with proper subsets S1 and S2 satisfying the FIRSTk condition. Then S1∪S2 is consistent.

Proof Obvious. //

Theorem 1 Let G be a grammar, let Pu be a set of useless chain productions of G, let C be the principle LR(k) parsing machine for G, and let C' be the LR(k) parsing machine for G based on Pu. If G is LR(k), each state of C' is consistent.

8. This is because they are not always necessary.

Proof Suppose G is LR(k). If Pu is empty, there is nothing to prove since our algorithm becomes the standard one. Suppose Pu is non-empty.

Suppose for induction that state R in C' is the union of the proper subsets of states R1,R2,...,Rn in C pairwise satisfying the FIRSTk condition.

For the basis, the initial state I' of C' consists of the proper subset of the initial state I of C. The above is therefore satisfied trivially.

We wish to show that each successor of R satisfies the above. Suppose R has an M-successor S. If M is a #-symbol, S is empty (trivially satisfying the above condition). Otherwise, S is the union of the proper subsets of the following states in C: the L-successor of Ri (if it exists), $1 \leq i \leq n$, where L is an ancestor of M. If only one such successor exists, we are done since each state of C is consistent. Otherwise, consider any two such distinct successors S1 and S2. Then there exists some ancestors L1 and L2 of M and some p and q where $1 \leq p,q \leq n$ such that S1 is the L1-successor of Rp and S2 is the L2-successor of Rq. If $p \neq q$, the proper subsets of S1 and S2 satisfy the FIRSTk condition by lemma 3. If p=q, then there exists some M' such that L1=>*M'=>*M and L2=>*M'=>*M by chain productions where every nonterminal (except M') in the derivations from L1 and L2 to M' are different. Suppose arbitrarily that L1 is different from M'. Then we can write L1=>*A=>M' and L2=>*M' for some nonterminal A. By lemma 2, respective pairs of M1- and M2-items in the proper subset of Rp=Rq satisfy the FIRSTk condition. By lemma 3, the proper subsets of S1 and S2 satisfy the FIRSTk condition. //

If p0,p1,...,pn is a sequence of productions of G, the result of removing those productions which are members of Pu is said to be *trimmed according to* Pu.

Theorem 2 Let G be an LR(k) grammar, let Pu be the set of useless chain productions of G, and let C and C' be the LR(k) parsing machines with forced lookahead based on $\emptyset$ and Pu respectively. If algorithm 1 with input C and w outputs
 (1) bottom up parse p0,p1,...,pn or
 (2) a partial bottom up parse followed by *error*,
then algorithm 1 with input C' and w respectively outputs
 (1) p0,p1,...,pn trimmed according to Pu, or
 (2) a partial bottom up parse trimmed according to Pu followed by *error*.

Proof It can be shown by induction that algorithm 1 with C' "simulates" all non-proper actions (those excluding reductions by useless chain productions) of algorithm 1 with C, including error detection. The proof found in Pag5 for LR(1) grammars can be suitably generalized to LR(k) grammars. //

If each state of C' (in the above theorem) is consistent, we claim that algorithm 1 with C' is a "trimmed" parser for G even if G is not LR(k). For instance, if G is S→A|B, A→a, B→a, then each state of the LR(k) parsing machine for G based on {A→a, B→a} is consistent.

If the parsing machine for G does not have forced lookahead, there are cases where the parser (algorithm 1) will fail to detect erroneous strings (though it will work correctly for strings in L(G)). Consider the grammar S→a|b|ac. The LR(1) parsing machine for this grammar based on {S→a, S→b} is shown in figure 6. Algorithm 1 applied to this machine accepts the string acc (among others) provided only that symbol a rather than symbol b be chosen as the arbitrary leaf descendant of S when reducing ac to S. On the other hand, if lookahead is forced, the reduction of ac to S is possible only when the lookahead string is $\dashv$ (thus the error is detected).

References

[A&U1] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Englewood Cliffs, N.J.,1973

[A&U2] A.V. Aho and J.D. Ullman, "Optimization of LR(k) Parsers", J. Computer and System Sciences, Vol. 6, Dec 72, pp. 573-602.

[A&U3] A.V. Aho and J.D. Ullman, "A Technique for Speeding Up LR(k) Parsers", Siam J. Comp., Vol 2, June 73, pp. 106-127.

[And1] T. Anderson, "Syntactical Analysis of LR(k) Languages", Ph.D. Thesis, Univ. of Newcastle Upon Tyne, Computing Lab.

[And2] T. Anderson, J. Eve, and J.J. Horning, "Efficient LR(1) Parsers", Acta Informatica 2 (1973), pp. 12-39.

[DeR1] F.L. DeRemer, "Practical Translators for LR(k) Languages", Tech. Report MAC TR-65, Project MAC, Mass. Inst. of Tech., Cambridge, Oct 69.

[DeR2] F.L. DeRemer, "Simple LR(k) Grammars", CACM, Vol. 14, July 71, pp. 453-460.

[Ear1] J. Earley, "An Efficient Context-Free Parsing Algorithm", Ph.D. Thesis, Carnegie-Mellon Univ., Pittsburgh, PA. also see CACM 13:2, Feb 70, pp. 94-102.

[Jol] M.L. Joliat, "On The Reduced Matrix Representation of LR(k) Parser Tables", Tech. Report CSRG-28, Computer Systems Research Group, Univ. of Toronto, Oct 73.

[Knu] D.E. Knuth, "On the Translation of Languages from Left to Right", Inf. Contr. Vol. 8, Oct 65, pp. 607-639.

[Kor] A.J. Korenjak, "A Practical Method for Constructing LR(k) Processors", CACM, Vol. 12, Nov 69, pp. 613-623.

[LaL] W.R. LaLonde, "An Efficient LALR Parser Generator", Tech. Report CSRG-2, Computer Systems Research Group, Univ. of Toronto, Toronto, 1971.

[Pag1] D. Pager, "A Solution to an Open Problem by Knuth", Inf. Contr., Vol. 17, Dec 70, pp. 462-473.

[Pag2] D. Pager, "On Combining Compatible States in LR(k) Parsing", Tech. Report PE 257, Information Sciences Program, Univ. of Hawaii, Honolulu, July 72.

[Pag3] D. Pager, "A Compaction Algorithm for Combining the Symbol-action Lists of an LR(k) Parser", Tech. Report PE 259, Univ. of Hawaii, Honolulu, July 72.

[Pag4] D. Pager, "On the Decremental Approach to Left-to-right Parsers", to appear in Siam J. Computing.

[Pag5] D. Pager, "On Eliminating Unit Productions from LR(k) Parsers", Tech. Report from the Information Sciences Program, Univ. of Hawaii, Honolulu.

List of Figures

(a)

The ancestor graph of G1:

E→E+T|T T→T*a|a

where all chain productions are useless

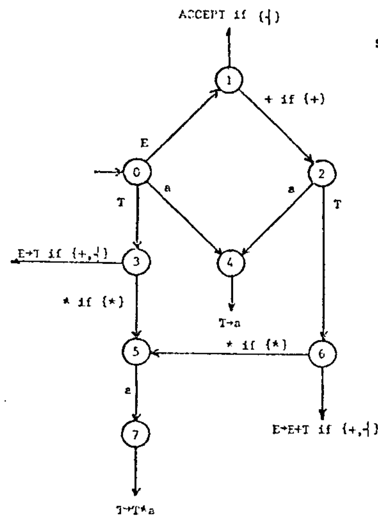
(b)

The ancestor graph of G1:

E→E+T|T T→T*a|a

where only E→T is a useless chain production

FIG. 1



STEP. NO.	STACK	REMAINING INPUT
1	$\vdash^0$	a+a*a⊣
2	$\vdash^0 a^4$	+a*a⊣
3	$\vdash^0 T^3$	+a*a⊣
4	$\vdash^0 E^1$	+a*a⊣
5	$\vdash^0 E^1 +^2$	a*a⊣
6	$\vdash^0 E^1 +^2 a^4$	*a⊣
7	$\vdash^0 E^1 +^2 T^6$	*a⊣
8	$\vdash^0 E^1 +^2 T^6 *^5$	a⊣
9	$\vdash^0 E^1 +^2 T^6 *^5 a^7$	⊣
10	$\vdash^0 E^1 +^2 T^6$	⊣
11	$\vdash^0 E^1$	⊣
12	ACCEPT	⊣

(a)

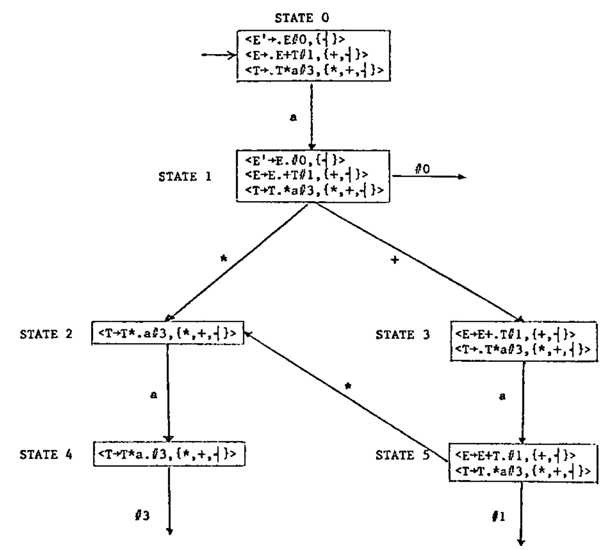
The transition graph of LR(1) parsing machine M1 for G1:

E→E+T|T T→T*a|a

(b)

Using algorithm1 with input M1 and a+a*a

FIG. 2



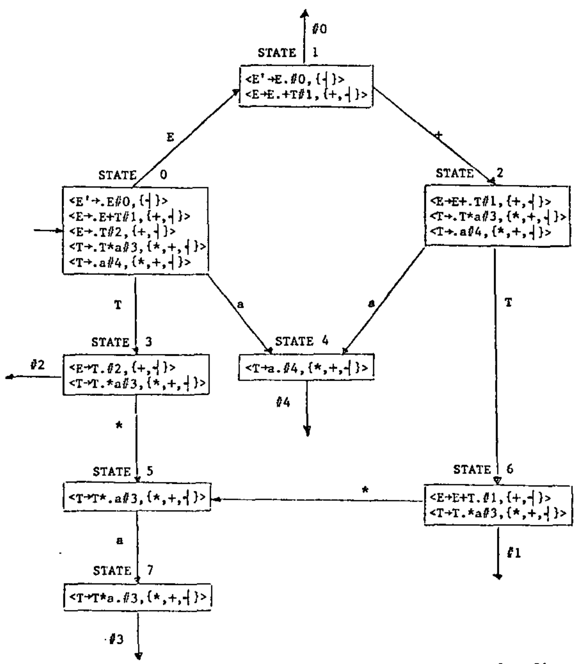
The "incomplete" LR(1) parsing machine for G1 based on

Pu = {E→T, T→a}

G1: E→E+T|T T→T*a|a

Note: Pu is the set of useless chain productions of G1

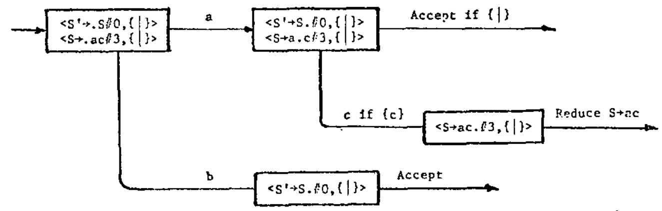
FIG. 3



The "incomplete" principle LR(1) parsing machine for G1

G1: E→E+T|T T→T*a|a

FIG. 4

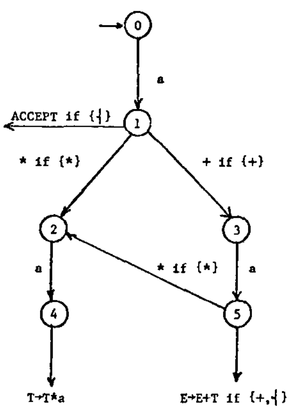


The LR(1) parsing machine for G based on Pu

G: S→a|b|ac

Pu = {S→a, S→b}

Fig. 6



STEP. NO.	STACK	REMAINING INPUT
1	$\vdash^0$	a+a*a⊣
2	$\vdash^0 a^1$	+a*a⊣
3	$\vdash^0 a^1 +^3$	a*a⊣
4	$\vdash^0 a^1 +^3 a^5$	*a⊣
5	$\vdash^0 a^1 +^3 a^5 *^2$	a⊣
6	$\vdash^0 a^1 +^3 a^5 *^2 a^4$	⊣
7	$\vdash^0 a^1 +^3 T^5$	⊣
8	$\vdash^0 E^1$	⊣
9	ACCEPT	⊣

(a)

The LR(1) parsing machine M1' for G1 based on Pu = {E→T, T→a}
G1: E→E+T|T T→T*a|a

(b)

Using algorithm 1 with input M1' and a+a*a. Note: in step(7), action "Reduce T→T*a" is interpreted. Since symbol a is a leaf descendant of T, the T-successor of 3 is taken as the a-successor of 3. Similarly for step(8).

FIG. 5