RETROSPECTIVE:

Efficient Computation of LALR(1) Look-Ahead Sets

Thomas J. Pennello

ARC International
Santa Cruz, CA 95060
tom.pennello@arc.com

Frank DeRemer

8 South Circle
Santa Cruz, CA 95060
fderemer@alum.mit.edu

We are honored to join the authors of a distinguished set of papers in writing a retrospective of our paper on LALR(1) lookahead set computation.

Our original motivation dates back to the very definition of LALR by Frank [Der 69]. Frank became a professor at the University of California at Santa Cruz (UCSC) when its Board of Studies in Information Sciences was young. At Santa Cruz he created the Translator Writing System (TWS) in XPL. The TWS initially generated parsers for SLR(1) grammars, and was later expanded to LALR(1) -- or so it was thought.

Tom joined the graduate program at UCSC in 1974. Visiting professor David Watt from the University of Glasgow showed us his unique approach to computing lookahead sets using relations. Watt's approach helped us see the error in both Watt's result and that of the XPL TWS -- it was NQLALR(1) (defined in our paper), not LALR(1).

We defined the correct relations that obtained true LALR(1) and it simply remained to compute the lookahead sets efficiently. This involved the transitive closure of the relations. The relations generally had cycles, and Martin Chaney, a fellow graduate student of Tom's, pointed out that collapsing the "maximal cycles" into single "fat nodes" results in a directed acyclic graph. Tom turned to the strongly-connected component algorithm of Tarjan to form DAGs from the LALR relations. The transitive closure is then simple and linear in the size of the DAG's nodes and edges.

Tom did verifying research on a Burroughs B5700, where the XPL TWS had been translated into XALGOL. Later, Tom translated the TWS into variants of Pascal, and in that form it was put into service by major corporations and is still used today by MetaWare Incorporated, the company we founded (first as a partnership) in 1979 (subsequently merged with UK company ARC International in October 1999). Thus, the first LALR(1) implementation of our paper has survived essentially intact and is still in use more than 20 years later.

The first significant follow-on work to our paper was publishing its content in ACM TOPLAS [Der 82], where we provided "bonus material" consisting in the first effective technique for debugging non-LALR grammars. The technique arose from the insight into the grammars that the relations provided. Debugging messages generated by the TWS proved invaluable in intensive summer courses on compiler construction that we conducted for many years, and in commercial development at MetaWare. Before the debugging messages, students resorted to staring at the LR(0) DPDA and the grammar until insight came. In another implementation of our debugging technique [Gro 91] the author states "you can more or less locate the problem immediately".

In the early 80s we used the TWS in a commercial application, replacing a recursive-descent compiler for COBOL written in assembly language for a Wang Laboratories' IBM 370 clone. The authors of the original compiler complained that the LR interpreter was too slow. This inspired Tom's follow-on work published in PLDI that produced LR parsers 10 times faster than the usual [Pen 86]; this silenced the Wang critics. This work was later referred to as "pioneering work on functional LR parsing" [Spe 95]. These faster parsers use the hardware run-time stack as the parser's stack. The Tarjan algorithm again proves critical to avoid checking for stack overflow upon entry into each state: the strongly-connected components of the LR(0) DPDA are "broken" by removing nodes such that the SCCs become acyclic; stack checks are inserted at the removed nodes. This paper inspired the same stack-checking method in a scanner generator [Bum 93].

Berkeley YACC and its cousin Bison also adopted our LALR algorithm. In Berkeley YACC's acknowledge-ment [Cor ND] one reads

> Berkeley Yacc is based on the excellent algorithm for computing LALR(1) lookaheads developed by Tom Pennello and Frank DeRemer. The algorithm is described in their almost impenetrable article in TOPLAS 4,4.

We are sad the author found our paper difficult! We tried hard to make it penetrable – even elucidating.

Other tools have adopted our technique, such as an LALR(1) parser generator written in Scheme [Bou 96], a parser generator for Java [Joh 01], and the "parser generator LALR" from the University of Karlsruhe [Gro 91]. Especially with the popularity of Berkeley YACC and Bison, it's quite possible that the majority of grammars these days are "DeRemed" (as one student nicknamed LALR parser generation) by our technique.

We are generally very pleased with our work's reception. In addition to the Berkeley YACC citation, Paul Johnson [Joh 01] refers to it as "the canonical reference to the LALR1 construction algorithm", and John Grout [Gro 96] advises a beginner to "go back to the source... a classic paper ..." (we hope the beginner can cope with impenetrable articles).

In a most unusual reference, an assertion in our paper later shown wrong by others is cited as an argument for the freedom to publish algorithms such as DeCSS [App 00]. We are happy to have goofed and fortified Appel's argument.

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

Our experience with Tarjan's algorithm in our LALR work and in compiler optimizer work at MetaWare leads us to believe that all computer science students should know the algorithm, even if they haven't the slightest interest in parsing. Our Pascal implementation of Tarjan's algorithm used nested procedures and then later CLU's iterators (after we added them to our MetaWare Professional Pascal compiler, using nested procedures to implement them). CLU's iterators are extensively used in programming at MetaWare and we recommend their study as well (and wish they were a feature of languages other than our own).

One final tidbit we offer: don't publish conjectures. If you couldn't figure out how to prove your conjecture, your punishment is to referee papers that do!

References

[Der 69] Franklin L. DeRemer. Practical translators for LR(k) languages. Ph.D. dissertation, Dep. Electrical Engineering, Massachusetts Institute of Technology, Cambridge, 1969.

[Der 82] Frank DeRemer and Thomas Pennello. Efficient Computation of LALR(1) Look-Ahead sets. ACM Transactions on Programming Languages and Systems October 1982 Volume 4 Issue 4. http://portal.acm.org/citation.cfm?id=357187

[Spe 95] Michael Sperber and Peter Thiemann. The essence of LR parsing. http://portal.acm.org/citation.cfm?id=215579

[Pen 86] Thomas Pennello. Very fast LR parsing. Proceedings of the SIGPLAN symposium on Compiler construction July 1986. http://portal.acm.org/citation.cfm?id=13326

[Cor ND] Robert Corbett. Berkeley YACC acknowledgement. ND=No date known. http://troi.lincom-asg.com/~rjamison/byacc/ACKNOWLEDGEMEN. See also http://sources.isc.org/devel/tools/byacc-1.9.tar.gz for the source to Berkeley YACC, which contains the acknowledgement in file ACKNOWLEDGEMENTS.

[Bou 96] Dominique Boucher. An LALR(1) parser generator written in Scheme. http://www.iro.umontreal.ca/~boucherd/Lalr/documentation/lalr.html

[Bum 93] Peter Bumbulis and Donald Cowan. RE2C: a more versatile scanner generator. ACM Letters on Programming Languages and Systems (LOPLAS) Volume 2 , Issue 1-4 March-Dec. 1993. http://portal.acm.org/citation.cfm?id=176487

[Joh 01] Paul Cody Johnson. Syntacs Translation Toolkit User Manual. http://www.inxar.org/syntacs/inxar/syntacs/docs/userman/userman_toc.html

[Gro 96] John Grout, email response. http://compilers.iecc.com/comparch/article/96-11-088

[App 00] Andrew W. Appel. DECLARATION OF ANDREW W. APPEL IN OPPOSITION TO PLAINTIFF'S MOTION. http://www.cs.princeton.edu/~appel/papers/corley.html

[Gro 91] Josef Grosch email concerning the "parser generator LALR". http://compilers.iecc.com/comparch/article/91-02-018. See also http://www.info.uni-karlsruhe.de/~uebprakt/WS9899/documentation/lalr.ps

[Lis 77] Abstraction mechanisms in CLU. Communications of the ACM Volume 20 , Issue 8 (Aug 1977). http://portal.acm.org/citation.cfm?id=359789

Efficient Computation of LALR(1) Look-ahead Sets

by
Frank DeRemer
Thomas J. Pennello

Information Sciences
University of California
Santa Cruz, CA. 95064

ABSTRACT

We define two relations that capture the essential structure of the problem of computing LALR(1) look-ahead sets, and present an efficient algorithm to compute the sets in time linear in the size of the relations. In particular, for a PASCAL grammar, our algorithm performs less than 20% of the set unions performed by a popular compiler-compiler (YACC).

Key Words and Phrases: context-free, LR(k), and LALR(k) grammar, parser, syntactic analysis, look-ahead, digraph, strongly-connected component.

CR Categories: 4.12, 5.23.

Note: Due to the restricted character set of the device on which this text was printed, we use the following conventions:

This	Stands for
x	cross product
6	set membership
u	set union
U	indexed set union, e.g. $\bigcup_i S_i$
C	set inclusion
a	Greek α
b	Greek β
c	Greek γ
d	Greek δ
w	Greek ω
e	Greek ε

Subscripting is often indicated by placing the single subscript to the right of the subscripted element, as in v1, i0, q1, B1, or when the subscript is an expression, it is enclosed in parentheses, as in q(k+1). (+) is used to denote footnotes.

(+) This is a footnote.

1. Introduction.

Since the invention of LALR(1) grammars [DeR 69], the LALR grammar analysis and parsing techniques have been popular as a component of translator writing systems or compiler-compilers. However, DeRemer did not describe how to compute the needed look-ahead sets. Instead, LaLonde was the first to present an algorithm [LLH 71]. Since then LaLonde's algorithm has been published by Anderson, Eve, and Horning [AEH 72, p. 21-22], who also presented their own algorithm (p. 21), and Aho and Ullman have published the one used in the YACC compiler-compiler [A&U 77, p. 238].

Various others have tried their hand at designing such an algorithm, often with the result of implementing a particular subset of LALR(1) that we have dubbed "not quite" LALR(1) or NQLALR(1) [DeR 72, Wat 74, Wet 77]. Subsequently, Watt attempted to repair his original approach [Wat 76], as did Chaney with DeRemer's approach [Alp 76]. Neither of these later attempts was correct, although both were more complex and worked in more cases than did the NQLALR method.

None of the correct LALR(1) algorithms, however, have been nearly as efficient as their NQLALR(1) counterparts. Later we describe the over-simplification that results in the simple, efficient algorithms that are not quite right. Here we simply note that no one heretofore has recognized the essential structure of the problem and provided an algorithm that efficiently exploits that structure, although Watt came close [Wat 76]. That is exactly the purpose of the current paper.

Preview. When a grammar is not LR(0), one or more of the LR(0) parser's states must be "inconsistent", having either a "read/reduce" or "reduce/reduce" conflict, or both. In the former case the parser cannot decide whether to read the next symbol of the input or reduce a phrase on the stack. In the latter case the confusion is between distinct reductions.

Looking ahead at the first symbol of the input may resolve the conflict, and DeRemer defined a grammar to be LALR(1) iff each inconsistent state q can be augmented with look-ahead sets that resolve the conflict and result in a correct, deterministic or "consistent" parser [DeR 69].

More precisely, for each inconsistent state q and possible reduction A -> w in q we let the "look-ahead set for A -> w in q" be denoted by LA(q,A -> w). When the parser is in state q and the symbol at the head of the input is in LA(q,A -> w), it must reduce w to A. Thus the look-ahead sets in q must be mutually disjoint and not contain any of the symbols that could be read from q.

Watt has defined LA(q,A -> w) as

{t ∈ T | S =>+ αAtz and αw accesses q}

[Wat 76]. Intuitively, when the parser is in state q, and αw is on the stack, reduction of w to A is appropriate exactly when the input begins with some terminal t that can follow αA in a rightmost sentential form. It is our purpose to investigate the underlying structure in this definition and to show how to compute LA efficiently.

We decompose the problem into four separate computations. In reverse order of computation they are as follows: LA is computed from "Follow" sets of nonterminal transitions; Follow sets are computed from "Read" sets of nonterminal transitions; Read sets are computed from "Direct Read" sets; and Direct Read sets are computed by inspecting the LR(0) parser.

We define a relation on nonterminal transitions, called "includes", relating the Follow sets, and another, called "reads", relating the Read sets. The Read sets are initialized by inspection of the parser. Then their values are completed by a graph traversal algorithm for finding "strongly connected components" (SCCs), adapted to union the sets appropriately as it searches the digraph induced by the reads relation. If a non-trivial SCC is found, the grammar in question is not LR(k) for any k. Next the Read sets are used as initial values for the Follow sets, which are completed by the adapted SCC algorithm applied to the digraph of the includes relation. This time, if a non-trivial SCC is encountered having a non-empty Read set in it, the grammar is ambiguous. In any case the LALR(1) look-ahead sets are simply unions of appropriate Follow sets.

We now define our terminology, define LALR(1), give theorems relating to look-ahead set computation, present the algorithm, discuss over-simplifications, give statistics for some practical grammars, and conclude.

2. Terminology.

The notions of symbol and string of symbols are assumed here. A vocabulary V is a set of symbols. V* denotes the set of all strings of symbols from V. V+ denotes V* - {ε}, where ε is the empty string. The length of any string α is denoted |α|. The first symbol of a non-empty string α is denoted First α; the string following is denoted Rest α; the last symbol is denoted Last α. As just illustrated we often do not parenthesize arguments to functions when the intent is clear.

If R is a relation, R* denotes the reflexive, transitive closure of R, and R+ denotes the transitive closure. We write $X =_S F(X)$ to mean that X is the smallest set satisfying X = F(X).

CFGs. A context-free grammar is a quadruple G = <T,N,S,P> where T is a finite set of terminal symbols, N is a finite set of nonterminal symbols such that T ∩ N = {} (the empty set), S ∈ N is the start symbol, and P is a finite subset of N x V*, where V = T ∪ N and each member (A, w) is called a production, written A -> w. A is called the left part and w the right part. We require a production S -> S'i for some S' ∈ N and i ∈ T such that i and S appear in no other production.

In this paper we adhere to the following conventions:

S, A, B, C, ...	∈ N
X	∈ V
t, a, b, c ...	∈ T
..., x, y, z	∈ T*
α, β, γ, δ, w	∈ V*

The relation => is pronounced "directly (right) produces" (+) and is defined on V* such that αAy => αwy for all α ∈ V*, y ∈ T*, and A -> w ∈ P. Both =>* and =>+ are pronounced "produces". A nullable nonterminal is one that produces ε. If S =>* α then α is called a sentential form; if α ∈ T* then it is called a sentence. The language L(G) generated by G is the set of sentences, i.e. {α ∈ T* | S =>+ α}.

Let G be a CFG and k ≥ 0. Then G is LR(k) iff S =>* αAy => αwy implies that, if S =>* γ => αwy', then γ = αAy', for all α, γ ∈ V* and y, y' ∈ T* such that Firstk(y') = Firstk(y) [Knu 65]. Here Firstk(y) is that prefix of y of length k, or just y if |y| < k.

LR parsers. Next we introduce our own formalization of an LR parser, i.e. any one-symbol look-ahead parser, such as an LR(1), LALR(1), or SLR(1) parser [A&J 74]. The generalization to multi-symbol look-ahead is easy, but we have no use for it here. Given some tabular representation

of our "LR1 automaton" defined below and the general LR parsing algorithm to interpret those tables, we have an "LR parser". The particular states, transitions, and look-ahead sets are determined by the grammar in question and the construction technique. For example, the LALR(1) technique produces an "LALR(1) automaton".

An LR1 automaton for a CFG $G=\langle T,N,S,P\rangle$ is a sextuple $LRA(G) = \langle K,V,P,Start,Next,Reduce\rangle$ where K is a finite set of states, V and P are as in G, Start $\in$ K is the start state, Next: K x V -> K is called the transition function, and Reduce: K x V -> 2^P is called the reduce function. We have allowed for nondeterministic or "inconsistent" LR automata; the LALR(1) condition of the next section excludes such cases. A transition is a member of K x V; it is a terminal transition if it is in K x T and a nonterminal transition if K x N. We represent the transition (q,X) by $q \xrightarrow{X}$ p where p = Next(q,X), or $q \xrightarrow{X}$ when p is irrelevant, and we define Accessing_symbol p = X; each state has a unique accessing symbol, except Start which has none.

In figures we represent LR1 automata by state diagrams in which states are connected by transitions. For each state q in which Reduce indicates possible reductions, we list the productions.

A path H is a sequence of states q1, ..., qn such that

$$q0 \xrightarrow{X1} q2 \longrightarrow \cdots \longrightarrow q(n-1) \xrightarrow{Xn} qn.$$

We say that H spells $a = X1\ldots Xn$ and define Spelling H = a and Top H = qn. We denote H by $q0 \xrightarrow{a} qn$, pronounced "q0 goes to qn under a". An alternate notation for H is [q0:a], given the automaton or its state diagram. The concatenation of [q:a] and [q':a'] where Top [q:a] = q' is written [q:a][q':a'] and denotes [q:aa']. We abbreviate [Start:a] to just [a]; thus [] denotes Start alone. We say that a accesses q if Top [a] = q.

A configuration is a member of K+ x V+; its first part is called the state stack and its second the input. The relation $\vdash$ on configurations is pronounced "directly moves to" and is the union of $\vdash_{read}$ and $\vdash_{A\to w}$, for all A -> w $\in$ P. $\vdash_{read}$ is pronounced "reads to": [q:a]tz $\vdash_{read}$ [qa:t]z if Next(Top[q:a],t) is defined. $\vdash_{A\to w}$ is pronounced "reduces w to A in moving to": [q:aw]tz $\vdash_{A\to w}$ [q:aA]tz if A -> w $\in$ Reduce(Top[q:aw],t). $\vdash^*$ and $\vdash^+$ are pronounced "moves to". The language L(LRA(G)) parsed by LRA(G) must be identically L(G), and is {z $\in$ T* | []z $\vdash^+$ [S']}.

A triple (A,a,b) $\in$ N x V* x V* is called an item, written A -> a.b; if b = e, it is a final item. A set of items is called a (parse) table. The set of LR(0) parse tables PT(G) for a CFG G is

$$PT(G) =_S \{Closure\ \{S \to .S'\}\} \cup \{Closure\ IS \mid IS \in Successors\ IS' \text{ for } IS' \in PT(G)\}$$

where

$$Closure\ IS =_S IS \cup \{A \to .w \mid B \to a.Ab \in IS' \text{ and } A \to w \in P\},$$

$$Successors\ IS = \{Succ(IS,X) \mid X \in V\}, \text{ and}$$

$$Succ(IS,X) = \{A \to aX.b \mid A \to a.Xb \in IS\}.$$

An LR(0) automaton for a CFG G is an LR1 automaton LRA(G) such that there exists a bijective function F: K -> PT(G) - {{}} where

$$Start = F^{-1}(Closure\ \{S \to .S'\}),$$

and for all t $\in$ T, X $\in$ V:

$$Next(q,X) = F^{-1}(Closure(Succ(F(q),X))),$$

$$Reduce(q,t) = \{A \to w \mid A \to w. \in F(q)\}.$$

F simply establishes a one-to-one correspondence between tables (except {}, the "trap table") and states, and thus an isomorphism between the parse tables and the parser. Hence, hereafter we elide all occurrences of F and F^{-1}, since context always determines whether q denotes a state or its corresponding parse table. The "LR(0)-ness" of the automaton is evident in that the definition of Reduce(q,t) is independent of t. Hereafter, we often use "parser" rather than "automaton".

It is well known that the LR(0) automaton A is a correct parser for G, i.e. L(A) = L(G); however, in general it is nondeterministic, due to the existence of "inconsistent" states. A state q is inconsistent iff there exists a t $\in$ T such that Next(q,t) is defined and Reduce(q,t) $\neq$ {} (read-reduce conflict), or |Reduce(q,t)| > 1 (reduce-reduce conflict), or both.

As a shorthand notation for a certain sequence of moves, we say that [a]yz $\vdash^*$ [a]bz iff (Top [a],yz) $\vdash^*$ ([Top [a]:b],z). This captures the notion that the parser reads y and reduces it, and possibly the empty string preceding y, to b. We need the vertical bar notation because [a]yz $\vdash^*$ [ab]z does not neces-

---

(+) Usually $\Rightarrow_R$ is used; we drop the R since we always mean "right".

sarily imply that y was reduced to b. For example, consider [aA]txz |- [aAt]xz |- [aA]xz |-* [aAb]z, where y=tx was not reduced to b.

Graphs. A directed graph or digraph is a pair (V',E) where V' is a set of vertices and E is a subset of V' x V', each member of which is called an edge. In this paper V' is always finite. A digraph-path, or simply a path when the context is clear, is a sequence of vertices v1, ..., vn, n > 1, such that (vi,v(i+1)) ∈ E for 1 ≤ i < n; we say there is a path from v1 to vn. A root is a vertex having no paths to it. A directed acyclic graph (DAG) is a digraph in which there is no path from any vertex back to itself. A forest is a DAG in which there is at most one path to each vertex. A tree is a forest having exactly one root.

3. LALR look-ahead sets.

We define "LALR(1) parser" by refining the definition of Reduce for an LR(0) parser. Intuitively, Reduce(q,t) should contain A -> w only if there exists a string aw that accesses q and, when reduced to aA, can lead to a successful parse, given that the input begins with t. First, we define the set of "look-ahead symbols".

Dfn. For an LR(0) parser, LA(q,A->w) = {t ∈ T | [aw]tz $\underset{A\to w}{\vdash}$ [aA]tz |-* [S'1], and aw accesses q}. ∎

Dfn. An LALR(1) parser for a CFG G is like G's LR(0) parser, except that Reduce(q,t) = {A -> w | t ∈ LA(q,A -> w)}. ∎

Dfn. A CFG is LALR(1) iff its LALR(1) parser has no inconsistent states. ∎

The latter defines LALR(1) grammar in terms of LALR(1) parser; a grammar is LALR(1) iff its LALR(1) parser is deterministic. It is desirable to have a definition of LALR(1) grammar that does not involve the parser, but we know of no reasonable way to do this. We do, however, come a little closer in the following theorem, which Watt gave as a definition [Wat 76].

Thm. LA(q,A -> w) = {t ∈ T | S =>+ aAtz and aw accesses q}. ∎

The proof depends essentially on the correctness of the LR(0) parser; i.e., the moves faithfully reflect the derivation.

Our primary goal here is to show how to compute the LA sets. To do so we focus our attention on nonterminal transitions and define "Follow sets" for them.

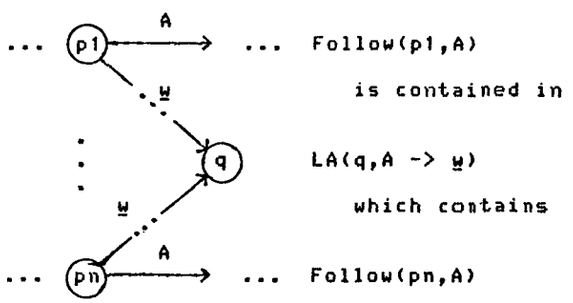
Dfn. For an LR(0) parser with nonterminal transition (p,A), Follow(p,A) = {t ∈ T | [aA]tz |-* [S'1] and a accesses p}. ∎

These are just the terminal symbols that can follow A in a sentential form whose prefix a, preceding A, accesses state p, given the correctness of the LR(0) parser. Stated in terms of derivations, Follow(p,A) = {t ∈ T | S =>+ aAtz and a accesses p}. Thus it is easy to see that each LA set is just the union of some related Follow sets.

Thm Union.

$$LA(q, A \to w) = \bigcup_{(p,A)\ \xrightarrow{w}\ q} Follow(p,A). \ ∎$$

That is, the LA set for production A -> w in state q is the union of the Follow sets for the A-transitions whose source state p has a path spelling w that terminates at q. Intuitively, when the parser reduces A -> w in state q, each such p is a possible top state after w is popped; then the parser must read A in p, all with some terminal t, the first of the input. Diagrammatically,

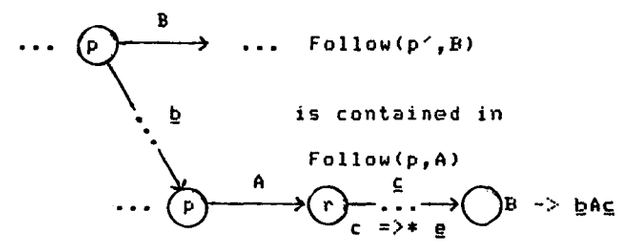


We want to reduce A -> w when in state q, if the next input symbol is in any of the Follow(pi,A) (1 ≤ i ≤ n), i.e. if it can follow A in any of the left contexts "remembered" by states p1 through pn.

The Follow sets are, in turn, related to each other. In particular,

Thm. Follow(p',B) ⊆ Follow(p,A) if $B \to bAc$, c =>* e, and $p' \xrightarrow{b} p$. ∎

Diagrammatically,



This is easy to see since, given some string $\underline{a}$ accessing p′, we have $\underline{ab}$ accessing p, and in an appropriate right context, $\underline{ab}A$ can be reduced first to $\underline{ab}A\underline{c}$ via $\underline{c} \Rightarrow^* \underline{e}$ and then to $\underline{a}B$; thus those symbols that can follow B in the left context remembered by p′ can also follow A in the left context remembered by p. We capture the above inclusion via a relation on nonterminal transitions.

Dfn. (p,A) **includes** (p′,B) iff

$B \to \underline{b}A\underline{c}$, $\underline{c} \Rightarrow^* \underline{e}$, and $p' \xrightarrow{\underline{b}} \cdots \to p$. ∎

Next we observe that the symbols labeling terminal transitions "following" a nonterminal transition (p,A) are obviously in Follow(p,A).

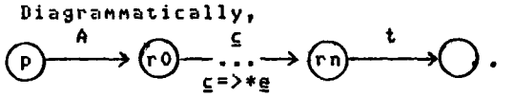
Thm. $Read(p,A) \subseteq Follow(p,A)$. ∎

where

Dfn. For an LR(0) parser with nonterminal transition (p,A),

$Read(p,A) = \{t \in T \mid [\underline{a}A|]tz \vdash^* [\underline{a}A|\underline{c}]tz \vdash [\underline{a}A\underline{c}t]z \vdash^* [S'|]$ and $\underline{a}$ accesses p$\}$. ∎

Diagrammatically,



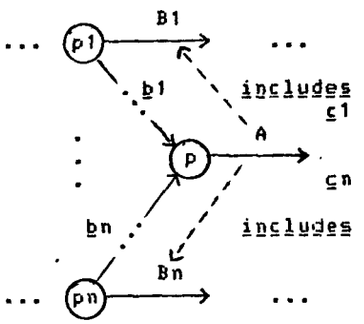
Read(p,A) is the set of terminals that can be read before any phrase including A is reduced. The definition is complicated by the possibility of numerous reductions of the empty string, and nonterminals generating it, viz. $\underline{c} \Rightarrow^* \underline{e}$, before the read move finally occurs. Read(p,A) is just the "direct read symbols" (DR, below) from r0, if $\underline{c} = \underline{e}$ in the above diagram (i.e., n = 0).

Finally, we summarize our observations about the Follow sets, having considered all possible contributions.

Thm Up. $Follow(p,A) =_S$

$$Read(p,A) \cup \bigcup_{(p',B)\ (p,A)\ \underline{includes}\ (p',B)} Follow(p',B) \quad ∎$$

That is, Follow(p,A) is exactly (1) the set of terminals that can be read, via the first read, after reducing a phrase to A in the left context "remembered" by p, before any phrase including A is reduced, unioned with (2) the Follow sets of the nonterminals to which some phrase containing A, followed at most by some nullable nonterminals, can be reduced before reading another symbol, each such nonterminal in the appropriate left context, of course. Diagrammatically, for n such nonterminals, B1 through Bn (not necessarily distinct):



(Bi, Bj; bi, bj; ci, cj not necessarily distinct.)

The dashed arrows indicate the **includes** relation. In a similar manner we can decompose Read sets.

Thm Across.

$$Read(p,A) =_S DR(p,A) \cup \bigcup_{(r,C)\ (p,A)\ \underline{reads}\ (r,C)} Read(r,C). \quad ∎$$

where

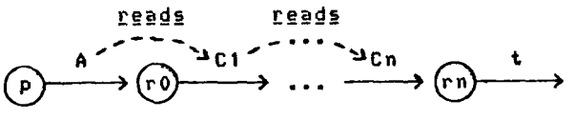
Dfn.

$DR(p,A) = \{t \in T \mid p \xrightarrow{A} r \xrightarrow{t} \}$. ∎

Dfn. (p,A) **reads** (r,C) iff

$p \xrightarrow{A} r \xrightarrow{C}$ and $C \Rightarrow^* \underline{e}$. ∎

The "direct read symbols" (DR) are simply those that label terminal transitions of the successor state r of the (p,A) transition. "Indirect read symbols" arise when nullable nonterminals can follow. Diagrammatically,



Here, (p,A) **reads** (r0,C1) **reads** ... **reads** (r(n-1),Cn); thus, DR(r(n-1),Cn) $\subseteq$ Read(p,A) so that $t \in Read(p,A)$.

In conclusion to this section, we observe that to compute the LA sets, we need the Follow sets, for which we need the Read sets, for which we need the DR sets. The Follow sets are interrelated as described by the **includes** relation, as are the Read sets by the **reads** relation. In the next section we describe the computation of these sets by carrying information along the edges of the graphs induced by the **reads** and **includes** relations. We use a graph traversal algorithm to determine an optimum order of doing so, and simultaneously, to compute the sets.

4. Graph algorithms for LALR computations.

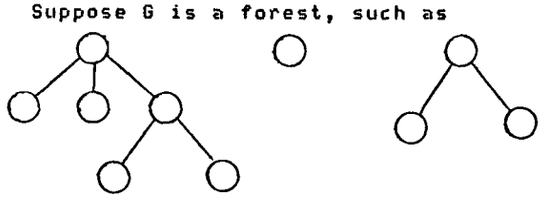
Theorems Up and Across relate Follow, Read, and DR in such a way that, given an appropriate graph traversal algorithm, we can apply it a first time to compute Read from DR, and then apply it a second time to compute Follow from Read. We start by noting that DR is already directly available in the LR(0) parser. Then the two graphs of interest are those induced by the relations reads and includes, respectively. However, let us consider a more general problem first and then return to this specific LALR application.

General case. Let R be a relation on a set X. Let F be a set-valued function such that for all x ∈ X,

Eqn 4.1. $F\ x = {}_S F'\ x \ \cup \bigcup_{xRy} F\ y.$ □

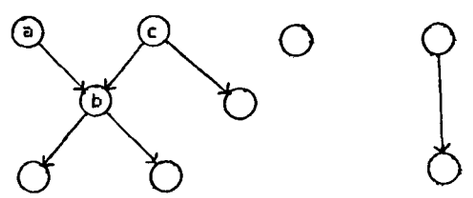
where F′ x is given for all x ∈ X. Let G=(X,R) be the digraph induced by R, i.e. G has vertex set X and (x,y) is an edge iff xRy. Then F x can be efficiently computed from F′ x by traversing G appropriately, as we shall consider first when G is a forest, then a DAG, and finally a general digraph.

Suppose G is a forest, such as



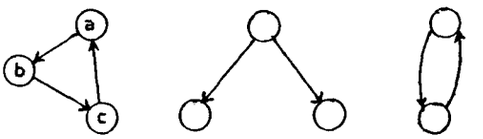
Each leaf x has no y such that xRy: thus F x is simply F′ x. Each non-leaf x is a parent of one or more children; a child of x is a vertex y such that xRy; and F x in this case is F′ x unioned with the F-values of the children of x. Thus, a standard, recursive, tree-traversal algorithm T can be used to compute F in this case, by carrying information from the leaves to the roots.

Now consider a DAG, such as



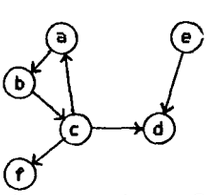
Vertices a and c "share" the child b. Algorithm T correctly computes F for all vertices, but it traverses b and its subtrees twice, once as a subtask of traversing a and again as a subtask of c. An algorithm D, based on T, can avoid such recomputation by marking each vertex on first encounter and never re-traversing marked vertices.

Finally, consider the general case of a digraph with cycles and possibly no roots, e.g.

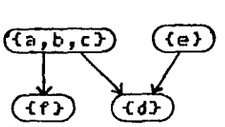


If algorithm D were to start its traversal at vertex a, it would visit a then b then c, incorrectly computing F c = F′ c ∪ F′ a, although it would correctly compute F a and F b. Worse yet, Algorithm T would loop forever. Note that by the definition of F, F a ⊆ F b ⊆ F c ⊆ F a, so F a = F b = F c = F′ a ∪ F′ b ∪ F′ c. A second trip around the cycle would solve the problem for simple loops such as the above, but in the general case we may have loops inside loops, short-cut paths between loops, etc., so that a "second trip" is not so easy to define.

The generalization of such cycles is a strongly connected component (SCC), a maximal subgraph (V",E") such that V" ⊆ V′, E" = E ∩ (V" x V"), and for all distinct vi, vj ∈ V", there is a path from vi to vj (thus, vice versa). It is easy to see, as above, that for V" = {x1, ..., xn} we have F x1 = ... = F xn and that this common value contains F′ x1 ∪ ... ∪ F′ xn. Thus, we could collapse each SCC V" to a single vertex x" with F′ x" = F′ x1 ∪ ... ∪ F′ xn, forming a new digraph G′, apply algorithm D to G′, then distribute the results from G′ to G, e.g. F x1 = ... = F xn = F x". It is well known that, since V" is maximal, G′ will have no cycles, i.e. it will be a DAG. Thus, D will work correctly on G′. Note that each vertex v in G not involved in any cycle will be the only member of a trivial SCC and will thus appear unchanged in G′. For example, the digraph



reduces to the DAG

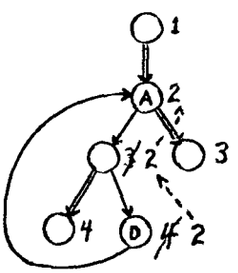


Here we have F f = F′ f, F d = F′ d, F e = F′ e ∪ F′ d, and F a = F b = F c = F {a,b,c} = F′ {a,b,c} ∪ F {f} ∪ F {d} = (F′ a ∪ F′ b ∪ F′ c) ∪ F′ f ∪ F′ d.

In practice we need not actually construct the collapsed graph G′. Rather we can effect the computation of F while finding the SCCs. The following algorithm, Digraph, is an adaptation of one given in the literature [EKS 77]. We first modified the exposition of the algorithm to improve its readability and understandability. Then we added the three statements set off to the right to compute F. Further explication is given below the algorithm.

```
algorithm Digraph:
  input   R, a relation on X, and F', a function from X to sets.
  output  F, a function from X to sets, such that F x satisfies 4.1.
  let     S be an initially empty stack of elements of X
  let     N be an array of zeroes indexed by elements of X
  for     x ∈ X such that N x = 0 do call Traverse(x,1) od
  where recursive Traverse(x,k) =                          # Order Vertices
          call    Push x on S
          assign  N x <- k                                 ; F x <- F' x
          for     y ∈ X   such that xRy                    # Order Edges
            do    if      N y = 0 then call Traverse(y,k+1) fi
                  assign  N x <- Min(N x, N y)             ; F x <- F x ∪ F y
            od
          if      N x = k                                  # An SCC found
            then  repeat  assign N(Top of S) <- Infinity   ; F(Top of S) <- F x
                    until (Pop of S) = x                   # Order Vertices
            fi
          end     Traverse
  end     Digraph
```

The array N serves three purposes: (1) recording unmarked vertices (N x = 0), (2) associating a positive integer value with vertices that are under active consideration (0 < N x < Infinity), and (3) marking with Infinity vertices whose SCCs have already been found. Starting with any unmarked vertex we push it on stack S and mark it with a level number (starting at one) and Traverse its "subtrees" incrementing the level number k as we proceed down the "tree". If ever an edge is encountered that points back up the tree to ancestor A from descendant D (see diagram below), the N-value of D is minimized to that of A, and that minimal value will bubble up the tree all the way back to A, unless a reference back to an ancestor even lower numbered than A is encountered, in which case that smaller value will prevail. Finally, when we have Traversed all "subtrees" and the "root" has not had its N-value reduced, we recognize that root and all vertices above it on the stack S as being an SCC.



Thm. Algorithm Digraph correctly determines SCCs. □

For a proof we refer the reader to that given by Eve and Kurki-Suonio [EKS 77], since ours is but a slight modification of their algorithm.

Thm. Algorithm Digraph is order |Vertices| + |Edges| of the digraph induced by relation R, i.e. linear in the "size" of R.
Proof: Traverse is called once for each vertex v, due to the immediate marking and the avoidance of re-traversing marked vertices. Inside Traverse, v is pushed on the stack once, and the for-loop body is executed once for each edge from v. The repeat loop executes only intermittently, when an SCC is determined, and simply pops vertices off the stack; ultimately |Vertices| are popped since that many were pushed. Thus, each vertex is pushed once and popped once, and each edge from it is traversed once. □

Cor. Algorithm Digraph performs one set union per edge of relation R, i.e. Fx <- F x ∪ F y. □

In fact, it is possible to reduce the number of such unions inside each non-trivial SCC from the number of edges to the number of vertices in the SCC [EKS 77]. This improvement would become important in "highly connected" SCCs in a grammar with many terminal symbols, i.e. in which set unions become expensive. We did not include the improvement here because it would obscure the essential algorithm. In addition, non-trivial SCCs are infrequent, in practice.

Thm. Algorithm Digraph correctly computes F.
Proof: The theorem is based on the following facts: First, if F x satisfies 4.1, then $F\,x = \bigcup_{x\,R^*\,y} F'\,y$. This is not difficult to show. Second, Digraph implicitly compues R* [EKS 77]. In fact, if F'(x) = {x} for all x ∈ X, then F x = {y ∈ X | x R* y}, the set of all vertices reachable from x in the digraph induced by R. □

Application to LALR. Let the set X in algorithm Digraph be the set of nonterminal transitions of an LR(0) parser. First let F' be DR and R be reads; then the resulting F will be Read; i.e. according to Theorem Across. we will have computed

$$Read(p,A) =_S DR(p,A) \cup \bigcup_{(p,A)\ reads\ (r,C)} Read(r,C).$$

Second let F′ be Read and R be includes; then the resulting F will be Follow, i.e. according to Theorem Up, Follow(p,A) $=_S$

$$\text{Read}(p,A) \cup \bigcup_{(p',B)\ (p,A)\ \text{includes}\ (p',B)} \text{Follow}(p',B).$$

Finally, we compute

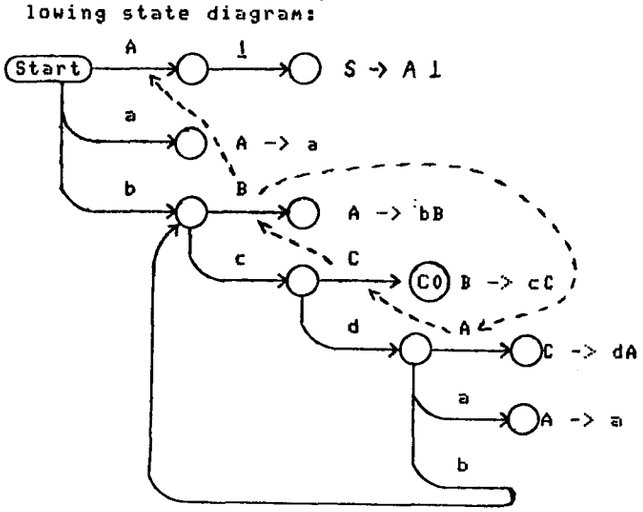
$$\text{LA}(q, A \to w) = \bigcup_{(p,A)\ p \xrightarrow{w} q} \text{Follow}(p,A)$$

according to Theorem Union, and we have the desired LALR(1) look-ahead sets after two applications of algorithm Digraph.

From a relational point of view, t ∈ LA(q,A -> w) iff (q,A -> w) lookback (p,A) includes* (p′,B) reads* (r,C) sees t where (q,A -> w) lookback (p,A) iff $p \xrightarrow{w} q$ and (r,C) sees t iff t ∈ DRS(r,C). Watt proposed this formulation, although with minor errors in the definitions of reads and includes regarding nullable nonterminals [Wat 76]. Watt's proposed bit matrix representations of the sparse relations reads and includes would be wasteful of space and time. We have effectively provided an efficient way to compute R = reads* . sees, then I = includes* . R, and finally LA = lookback . I.

Need for Digraph. Finally, we demonstrate that the generality of algorithm Digraph is needed because both the digraphs induced by includes and reads can, in general, be non-DAGs. In the reads case the existence of a non-trivial SCC implies that we have a grammar that is not LR(k) for any k, and may or may not be ambiguous. In the includes case a non-trivial SCC having a certain property implies an ambiguity.

Consider the LR(0) parser with the following state diagram:

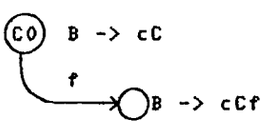


The dashed lines indicate the edges of the digraph induced by the includes relation. The digraph is not a DAG, since we have a cycle, and the corresponding grammar is not only LALR(1) but LR(0). (Adding the production A -> b would make it non-LR(0), but still LALR(1).)

The above example is, however, "dangerously close to being ambiguous", in the sense that, if the Read set were non-empty for any of the A-, B-, or C-transitions involved in the loop, then the grammar would be ambiguous. The generalization of this statement is:

Thm. Let (p,A) be a nonterminal transition that is in a non-trivial SCC of the digraph induced by the includes relation. Then the corresponding grammar is ambiguous if Read(p,A) ≠ {}. □

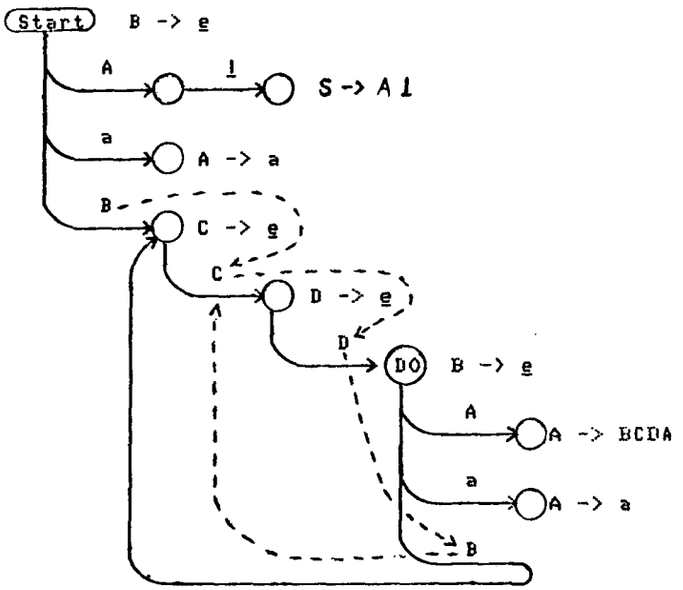
Such an ambiguity can be illustrated in the above parser by adding the production B -> cCf. This changes state C0 to the following:



Thus, f is in DR, Read, and Follow of the C-transition; thus it is in the Follow set of the A-transition in the loop; thus it is in Follow for the B-transition; thus it is in LA(C0,B -> cC) and hence we have a read-reduce conflict since f can also be read from state C0. (The symbol ⊥ is the only other symbol in each of these Follow sets, due to the A-transition from the start state.) The ambiguity is evident in that we now have B =>* cdbB and distinctly B =>* cdbBf, thus essentially the classical "dangling else problem", where f is the else clause and cdb is the if-then clause: B =>* cdbB =>* cdbcdbBf and distinctly B =>* cdbBf =>* cdbcdbBf.

This example can be made arbitrarily more subtle and complex by adding strings of nullable nonterminals to the ends of the various productions and prior to the f in the added production. Additionally changing B -> cC to B -> cCX, e.g., where X -> ε, still produces a read-reduce conflict on symbol f, but now the production involved is X -> ε in state C0. If instead we change B -> cCf to B -> cCXf, then we get a reduce-reduce conflict on symbol f in state C0, since LA(C0,B -> cC) = {f,⊥} and LA(C0,X -> ε) = {f}. The complexity of our proof of the latter theorem reflects such subtleties.

Now consider the LR(0) parser whose state diagram is



Here the dashed lines represent the edges of the reads relation. DR for the two B-transitions and the C-transition is empty, but for the D-transition it is {a}. Thus Read for each is {a} because the B-'s read C-, C- reads D-, and D- reads (the lower) B-transition. Hence we have a read-reduce conflict on symbol a in state D0, since LA(D0,B -> e) contains Follow of the lower B-, which contains Read of the lower B-, which contains a.

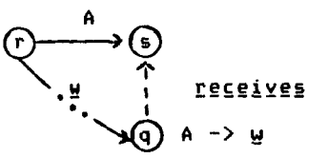
In this particular case the grammar is ambiguous, since we can reduce the empty string to BCD as many times as we want prior to reducing to A the only a in the only string in the language. However, by changing production A -> BCDA to A -> BCDAf we eliminate the ambiguity, while retaining the conflict. Now the grammar is unambiguous, since the number of f's is the number of times we must reduce empty to BCD, but it is still not LR(k) for any k, since the BCDs must be reduced before a is reduced to A but the f's follow the a. In general:

Thm. If the digraph induced by the reads relation contains a non-trivial SCC, then the corresponding grammar is not LR(k) for any k. ∎

Again the proof is complicated by nullable nonterminals, and the example can be arbitrarily complicated by their addition, some of which change the conflict from read-reduce to reduce-reduce.

5. Over-simplifications.

Two "clever ideas" come to mind, each of which is shown below to be inadequate. One is to union the two relations includes and reads so that only one application of Digraph is needed. Another has been invented independently by several researchers [DeR 72], [Wat 74], [Wet 77]. It involves defining another relation receives that is closely related to the union of includes and reads and leads to what we call "not quite LALR(1)" or NQLALR(1) parsers. The basic idea is to relate states rather than transitions. We reason that LA(q,A -> w) must include all symbols that can be looked-ahead-at or read from any "restart" state s such that there is a "look-back" state r with an A-transition to s and path spelling w to q:



If we reduce w to A in q, r may be the top state after popping |w| states. Then we read A and enter s, so any symbol looked-ahead-at or read by s would be in LA(q,A -> w), or really NQLA(q,A -> w). Formally:

Dfn. q receives s iff there is an r ∈ K such that $r \xrightarrow{A} s$ and $r \xrightarrow{w} q$, where A -> w ∈ q. ∎

Dfn. $NQLA(q,A \to w) = \bigcup_{\substack{s \\ q \text{ receives } s}} NQFollow(s)$. ∎

Dfn. $NQFollow(s) =_s$

$NQDR(s) \cup \bigcup_{\substack{s' \\ s \text{ receives } s'}} NQFollow(s')$. ∎

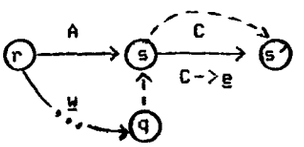
Dfn. $NQDR(s) = \{t \in T \mid s \xrightarrow{t}\}$;
i.e. the same as DR except defined for states rather than transitions. ∎

Thm. $NQFollow(s) = \bigcup_{\substack{s' \\ s \text{ receives}^* s'}} NQDR(s')$. ∎

Cor. $NQLA(q,A \to w) = \bigcup_{\substack{s \\ q \text{ receives}^+ s}} NQDR(s)$. ∎

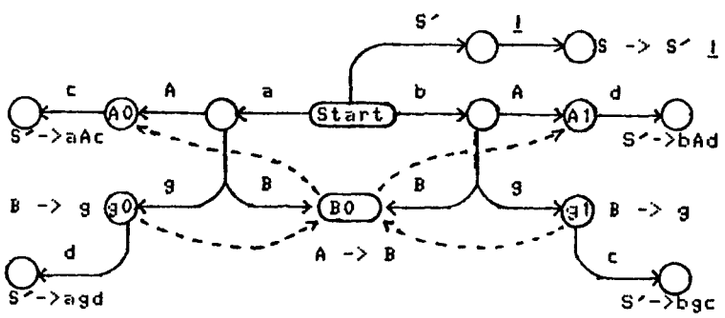
The theorem follows from simply rewriting the definition of NQFollow, and the corollary from back-substituting into the definition of NQLA.

Note that the nullable nonterminals cause no problem here. For example:



The dashed edge from s to s′ comes directly from the definition of receives and obviously serves a purpose similar to the edges of reads. Thus, NQLA(q,A -> w) is just the union of all the NQDRs of the states reachable via the receives relation along one or more edges. Our own IBM 360 implementation [DeR 72] just computes the transitive closure of receives via bit matrix techniques, then unions the related read sets (NQDR) to get the NQLAs.

The inadequacy of NQLALR arises from the fact that inappropriate "paths of reductions" are traced through the parser. In effect, we first consider reducing w to A, landing in state s, but then we consider reductions in s, e.g. B -> xA, without requiring that the A-transition that led us into state s be involved as we leave via reduction. The following LR(0) parser illustrates this point:



The relevant receives edges are indicated by dashed arrows.

In this case both NQLA(g0,B -> g) and NQLA(g1,B -> g) receive both symbols c and d, because the receives relation connects state B0 with both states A0 and A1, which can read c and d, respectively. Hence the grammar is not NQLALR(1) because we have read-reduce conflicts in both states g0 and g1. The grammar is, however, LALR(1) and our correct approach results in LA(g0,B -> g) = {c} and LA(g1,B -> g) = {d}, which are the correct LALR(1) sets.

Finally we consider the simplification of unioning includes and reads, and running the Digraph algorithm once, to simplify the computation. Such a simplification implies that instead of computing

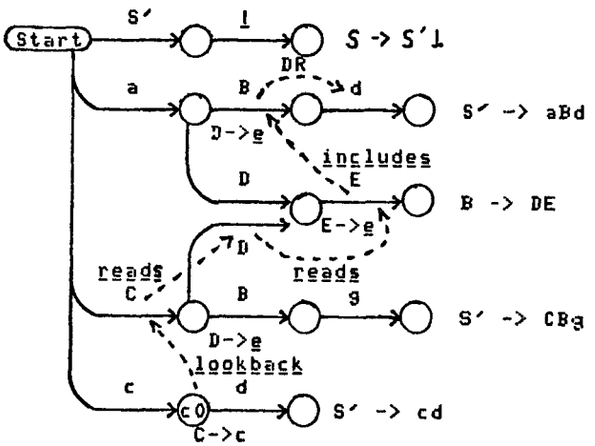
$$Follow(p,A) = \bigcup_{(r,C)\ (p,A)\ includes^*\ .\ reads^*\ (r,C)} DR(r,C)$$

as we have described (+) we instead compute

$$Follow(p,A) = \bigcup_{(p,A)\ (includes \cup reads)^*\ (r,C)} DR(r,C)$$

(+) This formula can be obtained by rewriting the expressions for Follow and Read (see Theorems Up and Across) and back-substituting the rewritten Read in the rewritten Follow).

The two equations are not equivalent in the general case. The following state diagram presents a counter-example:



LA(c0,C -> c) contains Follow of the C-transition, which reads the D-, which reads the E-, which includes the upper B-transition, whose DR set contains d, so we would get a read-reduce conflict in c0. On the other hand, with the correct approach the relevant LA set contains g only.

6. Summary of LALR look-ahead computation.

A complete procedure to compute LALR(1) look-ahead sets is as follows:

A. Compute which nonterminals are nullable.

B. Initialize Read to DR: one set (bit vector of length the number of terminals) for each nonterminal transition, by inspection of the transition's successor state.

C. Compute reads: one list of nonterminal transitions per nonterminal transition, by inspection of the latter transition's successor state.

D. Apply algorithm Digraph to reads to compute Read; if a cycle is detected, announce that the grammar is not LR(k) for any k and print the look-ahead symbols and productions associated with the cycle.

E. Garbage collect the lists of part C if necessary. (Typically little or no space is used compared to includes below, and besides, parts B, C, and D can be executed in parallel so no lists need actually be created.)

F. Compute includes and lookback (+): one list of nonterminal transitions per nonterminal transition and reduction, respectively, by inspection of

(+) For the definition of lookback, see section 4, subheading Application to LALR.

each nonterminal transition and associated production right parts, and considering nullable nonterminals appropriately.

G. Apply algorithm Digraph to *includes* to compute Follow: use the same sets as initialized in part B and completed in part D, both as initial values and as workspace. If a cycle is detected in which a Read set is non-empty, announce that the grammar is ambiguous and print the relevant productions and look-ahead symbols.

H. Union the Follow sets to form the LA sets according to the *lookback* links computed in part F.

I. Check for conflicts; if none, announce that the grammar is LALR(1) -- we have a parser; if some not previously mentioned, print out the relevant symbols and productions.

Linearity. Algorithm Digraph is linear in the size of the relation to which it is applied. For practical grammars, the size (number of edges) of the *includes* relation is about 2 to 3 times the number of nonterminal transitions in the parser. Each nonterminal transition (p,A) has one *includes* edge to it for each production for A that ends in a nonterminal B, or Bc where c =>* e, i.e. usually only two or three at most. The *reads* relation is virtually non-existent in practical cases, so we can ignore it. Thus, for practical LR(0) parsers Digraph is about linear in the number of nonterminal transitions. These statements are substantiated by the following table:

Grammar	Unions	= includes	+ lookback	SCC
PAL	1892	1293	599	35
XPL	876	634	242	5
PASCAL	868	579	289	30

reads	NTX	LR(0)	LA	YACC	Reference
0	555	69	42	9503	[A&U 72]
0	433	40	25	5292	[MHW 70]
7	337	43	26	6096	[Jen 74]

The table lists the total number of Unions performed by our implementation. Note the the number of *lookback* edges listed are only those for reductions in inconsistent LR(0) states. The SCC column records the total number of egdes of the *includes* relation involved in non-trivial SCCs. NTX is the number of nonterminal transitions. The time in seconds for the LR(0) parser construction and the look-ahead (LA) computation is for a Burroughs B5700 with a four microsecond memory. YACC tends to perform about five times as many set unions as does our algorithm.

In the worst case the size of the *includes* relation could be proportional to the square of the number of nonterminal transitions, since the relation could be nearly complete, i.e. xRy for *all* x and y. This worst case is illustrated by the grammar whose productions are {S -> S1 1, Si -> Sj, Si -> t} for $1 \leq i, j \leq n$. Ignoring the path [S1 1], the LR(0) parser for this grammar has n nonterminal transitions, one for each Si, $1 \leq i \leq n$, and each has n *includes* edges to it; i.e. each nonterminal transition has an edge to it from each of the others and from itself! Of course, this example is contrived, highly ambiguous, and has no redeeming virtue from a practical viewpoint.

Comparison with other algorithms. The algorithms of both Aho and Ullman [A&U 77] and Anderson, Eve, and Horning [AEH 72] work by "propagating" look-ahead sets across the edges of a graph. When propagation causes the look-ahead information at a node to be increased, the node is queued up so that it may in turn cause propagation of the new information to other nodes. This is iterated until the queue becomes empty. The order of propagation may not be optimal, in the sense that each edge is traversed only once. This causes the YACC algorithm to perform considerably poorer than ours, as indicated by the table above.

LaLonde's algorithm [LLH 71] is essentially algorithm D (see the beginning of section 4), but to avoid incorrect computation of look-ahead sets (see the example in section 4), no information is retained at the graph's nodes between the computation of the look-ahead sets for distinct reductions. Thus, edges may need to be traversed repeatedly as different look-ahead sets are computed.

In contrast to the above-mentioned algorithms, ours traverses each edge exactly once.

7. Conclusion.

We have defined the two relations *includes* and *reads* which capture the essential structure of the problem of computing LALR(1) look-ahead sets. The look-ahead sets may be computed from information obtained by two successive applications of a graph traversal algorithm applied to the relations. The algorithm is linear in the size of the relation to which it is applied. Thus, barring minor and constant improvements in underlying representations, we suspect that we have the best possible algorithm for this problem. We leave any proof or disproof of this conjecture for future research.

8. Acknowledgements.

Our thanks go to Martin Chaney for an essential insight along the way. Also, the papers of Watt and Wetherell had an inspirational effect on us. But most of all we thank our Lord and Saviour, Jesus Christ, for His inspiration and strength.

Note: There is an error in algorithm Digraph and another re the NQLALR technique. Write to the authors for an errata sheet.

9. References.

[A&U 72] Aho, Alfred V. and Ullman, Jeffrey D. The Theory of Parsing, Translation, and Compiling. Two volumes. Prentice Hall, Princeton, N.J., 1972.

[A&U 77] Aho, Alfred V. and Ullman, Jeffrey D. Principles of Compiler Design. Addison-Wesley, Reading, Mass., 1977.

[AEH 72] Anderson, Tom, Eve, Jim, and Horning, Jim. Efficient LR(1) parsers. Acta Informatica 2 (1973), 12-39.

[Alp 76] Alpern, Bowen, Martin Chaney, Michael Fay, Thomas Pennello, and Rachel Radin. Translator Writing System for the Burroughs B5700. Information Sciences, UC Santa Cruz, Santa Cruz, CA. 95064.

[DeR 69] DeRemer, Frank. Practical translators for LR(k) languages. Ph.D. thesis, Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., 1969.

[DeR 72] DeRemer, Frank. XPL distribution tape containg LALR translator writing system. Information Sciences, UCSC, Santa Cruz, CA. 95064, 1972.

[EKS 77] Eve, Jim, and Kurki-Suonio, R. On computing the transitive closure of a relation. Acta Informatica 8 (1977), 303-314.

[Jen 74] Jensen, Kathleen and Wirth, Niklaus. Pascal User Manual and Report. Second edition, Springer-Verlag, New York, 1974.

[LLH 71] LaLonde, W.R., Lee, E.S., and Horning, J.J. An LALR(k) parser generator. Proc. IFIP Congress 71, North Holland, Amsterdam, 151-153.

[Wat 74] Watt, David A. Personal communication.

[Wat 76] Watt, David A. Personal communication (class notes).

[Wet 77] Wetherell, Charles. A correction to DeRemer's SLR(1) parser constructor algorithm. Unpublished manuscript. Lawrence Livermore Laboratories. Livermore, CA.