A New Grammatical Transformation Into LL(k) Form
(Extended Abstract)

Michael Hammer

Massachusetts Institute of Technology*

Introduction

For some time, it has been recognized that left-to-right deterministic top-down parsing has a number of features to recommend it. The logic of such a parser is easily expressed as a one-state pushdown machine, and very flexible translations can readily be performed in conjunction with top-down processing. The major difficulty with this style of parsing is that there are relatively few grammars which satisfy the rather restrictive requirements to admit of top-down parsing (the LL(k) grammars), in comparsion with the grammars that can be parsed deterministically bottom-up (the LR(k) grammars). There has been some research along the lines of trying to apply transformations to non-LL(k) grammars in order to convert them into equivalent LL(k) form [1,2,3]; the most successful approach has been that of Rosenkrantz and Lewis [4]. They define a class of grammars, the LC(k) grammars, which can be parsed in a mixed hybrid of top-down and bottom-up techniques; this class strictly includes the LL(k) grammars, as well as many interesting but non-LL(k) grammars. They then provide a deterministic algorithm for converting any LC(k) grammar into an equivalent LL(k) grammar.

This work is a generalization of, and in the same spirit as, the Lewis and Rosenkrantz program. We investigate a new hybrid parsing method, basically bottom-up in character, but which contains a minimal infusion of top-down ideas. We consider the class of grammars which can be parsed by this method, and observe that it strictly includes the class of LC(k) grammars. Then we exhibit an algorithm for deriving from any such grammar an equivalent LL(k) grammar; this derived grammar is also as "useful" as the original one in directing compilation activities, for it can support translations equivalent to those supportable by the original grammar.

The keystone of this new parsing technique is the notion of making predictions in conjunction with bottom-up parsing. Prediction is a concept usually associated with top-down parsing; the successive making, and eventual fulfillment, of predictions is the essence of LL(k) parsing. In conventional LR(k) parsing, no prediction is made in advance concerning what reductions are going to be performed by the parser, save the lone implicit prediction that the entire input string is going to be reduced to S, the sentence symbol. We add to an LR(k) parser the feature of its being able, based upon its current state and inspection of k symbols of lookahead, to predict that some prefix of the remaining unread input string will, in the normal course of the bottom-up parse, be reduced to some nonterminal A. We say that in this case the parser predicts that it is going to find an A. Having made this prediction, the parse proceeds in no material way differently than it would have, had no prediction been made. The only advantage we shall allow the parser to take of its prescience about its future, is to allow it to delegate authority to a "subroutine", a smaller parsing machine whose goal is not to reduce some input to S (as is the goal of the main machine), but rather to reduce some prefix of the remaining input to A; and having done so, to return control to the main parsing machine. We shall allow this new machine control over its own stack; this stack is created upon call of the sub-parser and destroyed upon its return. We shall allow the sub-parser to call other sub-parsers, or even make recursive calls on itself, with the creation of a new stack attendant upon each call. Thus the model of our generalized bottom-up parser is essentially one of a collection of individual bottom-up parsers, each dedicated to the recognition of a particular nonterminal, and each possibly calling another at appropriate times after inspection of the current lookahead. The working space of each sub-parser is a stack, so the parsing procedure as a whole is effectively utilizing a stack of stacks. This is our model of multiple-stack (MS(k)) parsing machines. They are related to the recursive finite-state machines of Tixer [5] and Lomet [6], but differ from either of those models.

---

*Work reported herein was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number N00014-70-A-0362-0001, and in part by the General Electric Company.

Since each of the sub-parsers is to be essentially a miniature LR(k) parser, our starting point is to define under what circumstances it is possible for an LR(k) machine to predict during the course of a parse that it is going to find an A. We shall define what it means for a non-terminal A to be predictable in state q, and proceed from there to construct MS(k) machines from LR(k) machines.

Preliminary Definition and Notation

Before we introduce our formal model, we review some background terminology.

A context-free grammar is a four-tuple (T,N,P,S) where T is the finite terminal alphabet, N is the finite nonterminal alphabet, P is a finite set of productions of the form $A \to \alpha$ where $A \in N$ and $\alpha \in (N \cup T)^*$, and S in N is the starting symbol. We use V to denote $N \cup T$ and V' to denote $V \cup \{\varepsilon\}$.

Usual notation: Capital Roman letters denote members of N; small Roman letters denote members of T; Greek letters denote members of V*.

For α_1 and α_2 in V*, we write $\alpha_1 \Rightarrow \alpha_2$ if there exist β_1 and β_2 and a production $A \to \gamma$ such that $\alpha_1 = \beta_1 A \beta_2$ and $\alpha_2 = \beta_1 \gamma \beta_2$. We write $\alpha_1 \underset{L}{\Rightarrow} \alpha_2$ if $\beta_1 \in T^*$ and $\alpha_1 \underset{R}{\Rightarrow} \alpha_2$ if $\beta_2 \in T^*$. We let $\overset{*}{\Rightarrow}$, $\overset{*}{\underset{L}{\Rightarrow}}$, and $\overset{*}{\underset{R}{\Rightarrow}}$ denote the reflexive transitive closures of $\Rightarrow$, $\underset{L}{\Rightarrow}$, and $\underset{R}{\Rightarrow}$ respectively.

For any $\alpha \in V^*$ and $k > 0$, $FIRST_k(\alpha) = \{\omega \in T^k | \alpha \overset{*}{\Rightarrow} \omega\tau$ for some $\tau \in T^*\}$, and $FOLLOW_k(\alpha) = \{\omega \in T^k | S \overset{*}{\Rightarrow} \beta\alpha\omega\gamma$ for some β and γ $\}$.

For any context-free grammar G, $L(G) = \{\omega \in T^* | S \overset{*}{\Rightarrow} \omega\}$.

A grammar G is strong LL(k) for some k>0 if and only if given a string ω in T^k and a nonterminal A in N, there is at most one production p in P such that for some ω_1, ω_2, and ω_3 in T*, the following three conditions hold: 1) $S \overset{*}{\Rightarrow} \omega_1 A\omega_3$; 2) $A \overset{*}{\Rightarrow} \omega_2$, where the first production applied is production p; and 3) $\omega = FIRST_k(\omega_2\omega_3)$.

Any strong LL(k) grammar can be deterministically parsed in a top-down manner, by a one-state pushdown machine that uses k symbols of lookahead. The definition states that given a goal A and a lookahead ω, it can be determined which production is to be applied to the goal.

The LR(k) grammars of Knuth [7] are the largest class of grammars for which there exists a one-pass algorithm to find the left-to-right bottom-up parse of any sentence in the language. There is a variety of equivalent definitions of this class. We give one provided by Lewis and Stearns [8].

G is LR(k) if it is unambiguous and if for all ω_1, ω_2, ω_3, ω_4 in T* and A in N, $S \overset{*}{\Rightarrow} \omega_1 A\omega_3$, $A \overset{*}{\Rightarrow} \omega_2$, $S \overset{*}{\Rightarrow} \omega_1\omega_2\omega_4$, and $FIRST_k(\omega_3) = FIRST_k(\omega_4)$ imply that $S \overset{*}{\Rightarrow} \omega_1 A\omega_4$.

Before giving the LR(k) parsing algorithm, we make a comment about endmarkers. We shall assume that the right-pad symbol \$ is always a member of T and that every input string we shall be considering is followed by $\$^k$, k copies of the right-pad. This ensures that there are always k symbols of lookahead for the parser to inspect.

Our formulation of LR(k) parsing is similar to that of Aho and Ullman [9], with some minor variations.

Let G be an LR(k) grammar, let $A \to \alpha_1\alpha_2$ be a production of G where $\alpha_2 \neq \varepsilon$, and let $\omega \in T^k$. Then $A \to \alpha_1 . \alpha_2 (\omega)$ is a k - item of G. In addition, if $A \to \varepsilon$ is a production of G, then $A \to .\varepsilon(\omega)$ is a k-item of G(The references to k and G can be omitted where there is no ambiguity.) If $\alpha_1 \neq \varepsilon$, then $A \to \alpha_1 . \alpha_2 (\omega)$ is an essential item.

Let $A \to \alpha_1 . B\alpha_2 (\omega)$ and $B \to .\beta(\tau)$ be k-items of G. Then $B \to .\beta(\tau)$ is an immediate descendant of $A \to \alpha_1 . B\alpha_2 (\omega)$ if $\tau \in FIRST_k(\alpha_2\omega)$. The relation "descendant of" is the transitive closure of "immediate descendant of". The closure of an item is the set consisting of the item and all its descendants; this extends to define the closure of a set of items.

If G is an LR(k) grammar, then M, the LR(k) parsing machine for G is a four-tuple (Q,F,q_o,f), where Q is a finite set of non-final states, F is a finite set of final states, $q_o \in Q$ is the initial state, and $f: Q \times V' \times T^k \to Q \cup F$ is the next-state function. There is a one-to-one correspondence between final states of M and rules of G. Each non-final state is a non-empty set of k-items of G.

The initial state q_o is equal to the closure of all items of the form $S \to .\alpha(\$^k)$. The rest of Q and the function f are determined by repeatedly applying the following procedure until no new states are generated.

Let q be an element of Q and $\sigma \in V$. 1) If there is an item of q of the form $A \to \alpha.\sigma(\omega)$, then $f(q,\sigma,\omega)$ equals the final state corresponding to the rule $A \to \alpha\sigma$. 2) If there is an item of q of the form $A \to \alpha.\sigma\beta(\omega)$ where $\beta \neq \varepsilon$, then for each τ in $FIRST_k(\beta\omega)$, $f(q,\sigma,\tau)$ is equal to the state q', which is defined as follows: Let E be the set of all items of q of the form $A \to \alpha.\sigma\beta(\omega)$ where $\beta \neq \varepsilon$; then let E' be the corresponding set of items $A \to \alpha\sigma.\beta(\omega)$. The state q' is defined as the closure of the set E'. 3) If $A \to .\varepsilon(\omega)$ is an item of q, then $f(q,\varepsilon,\omega)$ is the final state corresponding to the $A \to \varepsilon$. 4) Otherwise $f(q,\sigma,\omega)$ is undefined.

It is well-known that if G is an LR(k) grammar, then the function f as described above is indeed well-defined and single-valued. We shall refer to f (q,σ,ω) as the σ/ω-successor of q. Furthermore, it is known that if f (q,ε,ω) and f (q,σ,τ) are both defined, then ω is not equal to the first k symbols of $\sigma\tau$.

The parsing machine M for the LR(k) grammar G processes input strings from T*, and uses a stack in its operation. In order to describe the operation, we assign a unique name to each member of Q and F.

A configuration of M is a triple (q,α,ω), where q is the current state of M, α is the stack contents, and ω is the remaining input string.

Non-final states are read states, while final states indicate that a reduction is to be performed. If $q' = f(q,\varepsilon, FIRST_k (\omega))$ is defined, then q' becomes the new state and it is pushed onto the stack as well. Let a be the next symbol of input and ω' the rest of the input after a; if $q' = f(q,a,FIRST_k (\omega'))$ is defined, then a and q' are pushed onto the stack, q' becomes the new state, and a is removed from the input. If q is the final state for the rule $A \rightarrow \varepsilon$, then the following occurs: the top element of the stack is popped to expose a state name q"; $q' = f(q'',A,FIRST_k (\omega))$ becomes the new state; A and q' are pushed onto the stack; and the input remains unchanged. If q is the final state for $A \rightarrow \beta$, then $2\cdot|\beta|$ elements are popped off the stack to expose state q", and the rest proceeds as in the preceding case.

It follows from our earlier observations that M is deterministic; that is, every configuration of M has at most one successor.

If τ is the input string to be processed, then the initial configuration of M is $(q_o, q_o, \tau\$^k)$. We say M accepts τ if when starting in the initial configuration for τ, M eventually reaches a configuration where the remaining input is just $\$^k$ and where the stack contains just $q_o S$ (i.e., a reduction to S has just been made wiping out the stack).

If G is an LR(k) grammar, the set of strings accepted by the LR(k) machine M for G is precisely L(G). Furthermore, the sequence of final states M enters during its processing of τ, corresponds to the sequence of productions that define the left-to-right bottom-up parse of τ.

LR(k) machines can be represented in the following way. A non-final state is shown as a rectangle containing the defining set of items; a final state is shown as a circle containing its corresponding production. If f $(q,\sigma,\omega) = q'$, then an arrow is drawn from q to q' labelled by σ/ω.

Prediction and State-Splitting in LR(k) Machines

Before we define the circumstances under which an LR(k) machine can predict, upon inspection of a lookahead, that the nonterminal A is going to be found, we need some preliminary definitions. Throughout the following, G is an LR(k) grammar, M is its parsing machine, and q is an arbitrary non-final state of M.

Definition If I is the item $A \rightarrow \alpha.\sigma\beta(\omega)$, where $\sigma\varepsilon V$, then we say that: i) I is an A - item

ii) I is a .σ - item

iii) If $\sigma\varepsilon T$, then I is a terminal item; in addition, $A \rightarrow .\varepsilon(\omega)$ is a terminal item

Definition If I is the item $A \rightarrow \alpha.\beta(\omega)$, then $FIRST_k (I) = FIRST_k (\beta\omega)$.

Definition If $A \varepsilon N$, $L_k (q,A)$ is the union of $FIRST_k (I)$ over all .A - items in q.

Definition A chain through q is a sequence of items $I_o, I_1, \ldots, I_n$ such that I_o is an essential item, I_n is a terminal item, and I_{j+1} is an immediate descendant of I_j, if $\omega\varepsilon T^k$, then c is a ω-chain if $\omega\varepsilon FIRST_k(I_n)$.

Definition The nonterminal A can be predicted on ω in q, if every ω-chain through q has some .A - item in it.

The significance of the above definition follows from the following observation. The items comprising a non-final state of M determine how that state is "used" by the machine. For example, suppose that $A \rightarrow .\alpha(\omega)$ is an item of q; then the following may occur. At some point in a parse, M will enter the final state for $A \rightarrow \alpha$, with the next k symbols of input being ω; and after popping off $2 \cdot|\alpha|$ elements from the stack, the state that will be exposed will be q. After each new entry to q during a parse, there will be some sequence of such "exposures" of q, each following the "recognition" of some item of q (that is, following the recognition of $A \rightarrow \alpha$ with ω being the lookahead). It is easy to see that there is a relationship between two items of q which are "recognized" in succession; namely, the first one recognized is an immediate descendant of the second one. The sequence of item recognitions must clearly start with a terminal item, since the first exit from q must be on a terminal symbol or on ε. And the recognition sequence ends when the recognition of a rule causes the stack to be popped past q's position on it; this means that some essential item of q has been recognized. Thus the notion of a chain through a state captures the way in which a state is "used" by the parser.

If, upon entry to q, we know that some particular item $A \rightarrow .\alpha (\omega)$ is going to be recognized by this usage of q, then we know that some prefix of the remaining input is going to be eventually reduced to A. On the other hand, if the lookahead upon entry to q is ω, then we know that the sequence of items recognized by this use of q is going to define an ω-chain through q. So if it happens that every ω-chain through q has an A-item in its interior (which is equivalent to its having a .A-item), then we can combine these two observations, and state that a lookahead of ω upon entry to q assures that some prefix of the remaining input at time of entry, will eventually be reduced to A.

If we can predict A in state q on lookahead ω, we should be able to take advantage of this fact in the following way. We could break each ω-chain in two, just below the .A-item which is guaranteed to be in it. The "lower" parts of all these chains could be removed from q and put into a separate state, which would serve as the initial state for the construction of an LR(k)- like machine. Then the operation of the parser would be altered as follows. Upon entry to q, we inspect the next k symbols of input (the lookahead). If they are not equal to ω, then we proceed as usual. If they are equal to ω, we transfer control to the initial state of the constructed sub-machine, the state consisting of the lower parts of the ω-chains through q. This submachine would take as its mandate the recognition of A, which we can guarantee will occur; the processing of this sub-machine will be LR(k)-like, and it will use its own stack. The recognition sequence of items of the initial state will always be the lower part of some ω-chain through q. When this has worked up along the chain to the place where it was broken, the sub-machine will have discovered the A it was looking for, and can suspend operation. It will discard its stack, and transfer control back to q in the main machine. Processing will resume there as if the main machine itself had located the A that has been found. The recognition sequence that q defines will be the upper part of the chain whose lower part was followed by the initial state of the sub-machine.

We call a division of q into two parts, induced by the ability to predict A in q on seeing ω, a splitting of q. The initial state of the sub-machine is called the predictive part of the splitting, while the sub-state of q which holds the place of q in the main machine, is called the base of the splitting. There are obviously very many issues that need to be resolved in the above description. Can the items of the predictive part be so blithely removed from the base state? What if an ω-chain has several .A-items in it; where do we break it? Rather than specifying how to compute a splitting, we shall give below the characteristics which determine whether or not a proposed splitting is indeed valid.

Definition Let c be a chain through q, let I be the terminal item of c, and let X be a subset of T^k. Then c is an X-chain if $FIRST_k(I) \cap X \neq \phi$.

Definition Let R be a set of items, and $A \in N$. Then $FOLLOW_k(R,A) = \{\tau | \tau \in T^k$ and there is an item $B \to \alpha .A\beta (\omega)$ in R with $\tau \in FIRST_k(\beta\omega)\}$.

Definition A bipartite splitting of q is a triple (B,A,P), where B and P are subsets of the items of q and $A \in N$, satisfying:

1) if $c = I_o, I_1, \ldots I_n$ is an $L_k(P,A)$ chain through q, then there is a j, $o \leq j < n$, such that I_{j+1} is an A-item, and such that if we set $H_1(c)=\{I_o, \ldots I_j\}$ and $H_1(c)=\{I_{j+1}, \ldots, I_n\}$, the following two equations hold, where in each case the union is over all L_k (P,A)-chains through q:

$$P = \bigcup H_2(c)$$

$$B = \bigcup H_1(c) + \{I \in q | FIRST_k(I) \not\subset L_k(P,A)\}$$

2) $FOLLOW_k(B,A) \cap FOLLOW_k(P,A) = \phi$.

In a splitting (B,A,P), B represents the base of the splitting, P the predictive state, and A is the nonterminal which is to be predicted. Rather than predicting A on a single lookahead, we now concern ourselves with predicting A whenever the lookahead is a member of a set of strings. This set is called the predictive language; it need not be specified explicitly, for it is implicitly specified by the other components of the splitting: it is $L_k(P,A)$. That is, A must be predictable in q for every ω in $L_k(P,A)$; we want to be able to predict an A whenever it may be there. Given the fact that we want to predict A does not fully determine how the state is to be split into the base and predictive states. We allow any of the various possibilities, so long as it satisfies the basic constraints. The predictive state must consist exactly of the lower parts of all the $L_k(P,A)$ chains, while the base state must equal the upper parts of these chains plus all the items of q which appear in other (non-$L_k(P,A)$) chains. There may be several ways of breaking some chains; definition requires only that each chain be breakable in a way consistent with the splitting, and that the splitting be that precisely induced by the breaking of the chains.

The second condition is only relevant in the case where A is left-recursive; in that case, there may be .A-items in B and P both. If this situation obtains, there is a potential ambiguity concerning the operation of the sub-machine which is delegated to find an A. If A is left-recursive, how will the sub-machine know if it has discovered the A it was supposed to find, or some "lower-level" A? Put another way, the sub-machine has to know when it has worked its way up the chain to the break; if A is left-recursive, finding an A-item will not be proof that the sub-machine is done. We shall allow the sub-machine to resolve its dilemma by inspecting the lookahead after it has made a reduction to an A. In order for this lookahead to convey sufficient information, we impose the second constraint of the definition.

$FOLLOW_k(B,A)$ is the set of lookaheads that indicate the sub-machine has worked its way up to a break in a chain and so should return, while $FOLLOW_k(P,A)$ is the set of lookaheads that follow "lower-level" A's.

As a simple example of a state-splitting, consider the LR(1) grammar $S \rightarrow Ax$, $A \rightarrow aD$, $A \rightarrow ac$, $A \rightarrow ad$, $D \rightarrow Ax$, $D \rightarrow Axy$, $D \rightarrow b$. One state of the LR(1) machine for this grammar would be comprised of the following items: $A \rightarrow a.D(x)$, $A \rightarrow a.c(x)$, $A \rightarrow a.d(x)$, $D \rightarrow .Ax(x)$, $D \rightarrow .Axy(x)$, $D \rightarrow .b(x)$, $A \rightarrow .aD(x)$, $A \rightarrow .ac(x)$, $A \rightarrow .ad(x)$. A splitting of this state would be the triple (B,D,P), where B consists of the items $A \rightarrow a.D(x)$, $A \rightarrow a.c(x)$, and $A \rightarrow a.d(x)$, while P would be comprised of the other items of the state. In this case the predictive language, $L_1(P,D)$ would equal {a,b}.

We denote a splitting by drawing two boxes, one for the base and one for the predictive state, each containing the defining items. We shall draw a dotted arrow from the base to the predictive state, and label it with a slash followed by the elements of the predictive language. The name of the predicted nonterminal A will be denoted in the predictive state.

The foregoing definition is readily generalizable to the case where the lookahead not only tells us whether or not to predict a nonterminal, but which of several possibilities to predict. The only additional requirement is that the predictive languages be disjoint, so that a particular lookahead causes at most one prediction to be made.

<u>Definition</u> A <u>state-splitting</u> of q is a pair (B,R), where R is a finite set of pairs (A_i,P_i), satisfying:

1) $L_k(P_i,A_i) \cap L_k(P_j,A_j) = \emptyset$ if $i \neq j$
2) $FOLLOW_k(B,A_i) \cap FOLLOW_k(P_i,A_i) = \emptyset$
3) for each $L_k(P_i,A_i)$ chain $c = I_o, I_1, \ldots\ldots, I_n$, there is a j, $o \leq j < n$, such that I_{j+1} is an A_i-item and such that if we set $H_1(c) = \{I_o, \ldots\ldots, I_j\}$ and $H_2(c) = \{I_{j+1}, \ldots\ldots, I_n\}$, then the following equations hold:
   $P_i = \bigcup H_2(c)$, where the union is over all $L_k(P_i,A_i)$ chains
   $B = \bigcup_i \bigcup H_1(c) + \{I \epsilon q \mid FIRST_k(I) \not\subset \bigcup_i L_k(P_i,A_i)\}$, where the union in the first term is over all $L_k(P_i,A_i)$ chains.

Even though the definition of a state-splitting makes reference to the set of all $L_k(P_i, A_i)$ chains, it can be demonstrated that this is an effective definition, that it is possible to determine whether or not a proposed splitting of a state is legal or not. It can also be shown that it is possible to compute all splittings of a state; but this is not of great interest, for we shall only be interested in splittings that satisfy certain criteria described below. These criteria will guide us in the construction of these splittings.

<u>MS(k) Parsing Machines</u>

We use the ideas of state-splitting to define our formal multiple-stack parsing machine. As we stated earlier, this machine is to consist of a collection of LR(k)-like parsers which can call each other (or themselves). Rather than just pasting together an arbitrary collection of such parsers to get a member of this machine class, we shall have our multiple-stack parsers evolve from LR(k) machines, in which some states have been replaced by splittings. The various sub-machines will be those constructed from predictive states of splittings and designed to find the predicted nonterminals.

<u>Definition</u> Let G be an LR(k) grammar (T,N,P,S). <u>An MS(k)</u> machine for G is a six-tuple (Q,F,I,q_o, f, g), where Q is a finite set of states, F is a finite set of final states, I is a finite set of initial states, $q_o \epsilon I$ is the starting state, $f:(I \cup Q) \times V' \times T^k \rightarrow Q \cup F$ is the next-state function, and $g: Q \times T^k \rightarrow I \times N$ is the predictive function, subject to the following restrictions:

1) The LR(k) machine for G is an MS(k) for G, with $I = \{q_o\}$, g undefined everywhere, and $f(q_o, S, \$^k)$ = RETURN (a special state in F)
2) If M is an MS(k) machine for G, $q \epsilon Q$, and (B,R) a splitting of q, then the result of replacing q by (B,R) (as defined below) is also an MS(k) machine for G.
3) Only machines given by 1) and 2) are MS(k) machines for G.

<u>Algorithm</u> If M is an MS(k) machine for G, $q \epsilon Q$, <u>and (B,R)</u> a splitting of q, then the following procedure replaces q by (B,R):

1) Remove q from Q, add B to I, add each P_i to I.
2) All images under f equal to q are set equal to B.
3) If $\omega \epsilon L_k(P_i,A_i)$, set $g(B,\omega) = (P_i,A_i)$.
4) If $\omega \epsilon$ $FOLLOW_k(B,A_i)$, set $f(P_i,A_i,\omega)$ equal to the special state RETURN
5) Recursively apply the following procedure to B and each of the P_i and their successors, until no new states are computed.

   Let q' be an element of $I \cup Q$.
   If there is an item in q' of the form $D \rightarrow \alpha.\sigma(\tau)$ where $\sigma \epsilon V$, then set $f(q',\sigma,\tau)$ equal to the final state for the rule $D \rightarrow \alpha\sigma$.

If there is an item of the form $D \rightarrow .\varepsilon(\tau)$ in q', set $f(q',\varepsilon,\tau)$ equal to the final state for the rule $D \rightarrow \varepsilon$.

If there is an item of q' of the form $D \rightarrow \alpha.\sigma\beta(\omega)$ where $\sigma \varepsilon V$ and $\beta \neq \varepsilon$, let E be the set of all such items. Let E' be the corresponding set of items $D \rightarrow \alpha\sigma.\beta(\omega)$. If there already is some state in Q whose essential items precisely equal E', set q" equal to that state; else compute the closure of E", add it to Q, and set q" equal to it. Then for any τ such that $\tau \varepsilon \mathrm{FIRST}_k(\beta\omega)$ for some item $A \rightarrow \alpha.\sigma\beta(\omega)$ in the class E, set $f(q',\sigma,\tau) = q''$.

6) When step 5) is completed, delete from $T \cup Q$ any states which are not accessible from the starting state.

An MS(k) machine is similar to an LR(k) machine but has several additional features. The set of initial states I is the set of those states into which there are no next-state transitions; this includes the starting state of the machine as well as the predictive states of splittings introduced into the machine. The predictive function g takes the base state of a splitting and a lookahead string, and produces the name of the nonterminal which should be predicted from that base on that lookahead, and the predictive state which heads the submachine dedicated to locating that nonterminal. The final state set of an MS(k) machine includes one state for each rule of G as well as a distinguished RETURN state, to which submachines transfer when they have found their assigned nonterminal. Note that the main parser is dedicated to finding S.

The LR(k) machine for G is trivially an MS(k) machine; other MS(k) machines are obtained by replacing a state in an MS(k) machine by a splitting of it. The replacement algorithm substitutes the base state for the state being split, and hooks things up appropriately. Since the items of q are scattered among B and the P_i, it is necessary to recompute the successors of q after the replacement has been made; some new states may be introduced and some old ones rendered inaccessible and hence deleted. The algorithm for computing successors of the new states is essentially the same used in computing an LR(k) machine, with an additional proviso to prevent reintroduction of a state which has previously been replaced by a splitting.

The operation of an MS(k) machine is easily described. It utilizes a two-dimensional doubly infinite stack; effectively a stack of stacks (called stack levels). The machine processes just like an LR(k) machine, using the first stack level, until it enters a state which is the base of a splitting. At that point, the lookahead is inspected. If it indicates that no prediction is to be made, processing continues on the first level; if a prediction is to be made, a new stack level is started, and on it is written the name of the nonterminal the sub-machine has been delegated to find, and the predictive state of the splitting that is the initial state of that sub-machine. Processing continues on the new level just as on the original level, including the possibility of starting further new levels, until the RETURN state is entered. This means that the currently operating submachine has completed its task; so the topmost stack level is wiped off and processing resumes on the next highest level.

In formally describing the operation of the machine, we use the symbol Δ as a break between stack levels.

<u>Definition</u> A configuration of an MS(k) machine is a triple (q,α,ω), where q is the current state, α is the stack contents, and ω is the remaining input string. The relation $\vdash$ on configurations is defined as follows:

1) $(q,\alpha,\sigma\omega) \vdash (q',\alpha\sigma q',\omega)$ where $\sigma \varepsilon T \cup \{\varepsilon\}$ and $q' = f(q,\sigma,\mathrm{FIRST}_k(\omega))$

2) $(q,\alpha,\omega) \vdash (q',\alpha A \Delta A q',\omega)$ if $g(q,\mathrm{FIRST}_k(\omega)) = (q',A)$

3) $(q,\alpha q_1\beta_1 q_2\beta_2 \ldots q_m\beta_m\, q,\omega) \vdash (q',\alpha q_1 A q',\omega)$ if q is the final state for the rule $A \rightarrow \beta_1 \ldots \beta_m$ and $q' = f(q_1,A,\mathrm{FIRST}_k(\omega))$

4) $(q,\alpha q_1 A \Delta q_2 A\, q,\omega) \vdash (q,\alpha q_1 A q_3,\omega)$ if q is the RETURN state and $q_3 = f(q_1,A,\mathrm{FIRST}_k(\omega))$.

<u>Definition</u> Let M be an MS(k) machine for G. M_k accepts $\omega \varepsilon T^*$ if $(q_o,Sq_o,\omega\,\$^k) \overset{*}{\vdash} (q,Sq_oSq,\,\$^k)$, where q is the RETURN state.

The basic results concerning the operation of MS(k) machines are easily stated.

<u>Theorem</u> Let M be an MS(k) machine for the LR(k) grammar G. Then:

1) M is deterministic (any configuration of M has at most one successor configuration)

2) M accepts precisely L(G)

3) The sequence of final states M enters in accepting $\omega \varepsilon L(G)$ gives the left-to-right bottom-up parse of ω.

This result means that nothing is lost or gained by making predictions in bottom-up parsing, by anticipating what is going to happen.

The proof of this result proceeds by showing that the operation of an MS(k) machine for G effectively simulates the operation of the LR(k) machine for G. The arguments used to demonstrate this rely heavily on showing how constituent items of LR(k) states are "used" during the course of a parse, and where these items are located in the MS(k) machine.

In processing $\omega \varepsilon L(G)$, an MS(k) machine performs the exact same reads and reductions that the LR(k) machine does, but performs some extra steps as well, corresponding to the making and fulfilling of predictions. The MS(k) machine takes just 2n extra steps more than the LR(k) machine, where n is the number of predictions the former makes. Thus an MS(k) machine does exactly what the LR(k) machine does, only somewhat slower.

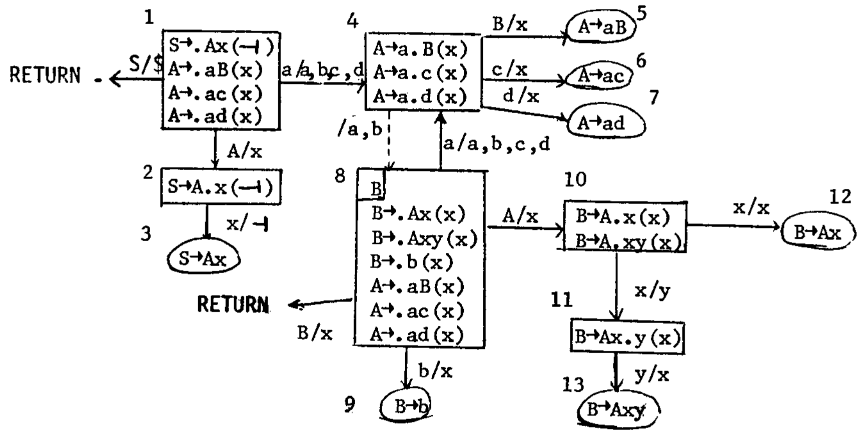


Figure 1.

Figure 1 shows an MS(1) machine for the LR(1) grammar $S \rightarrow Ax$, $A \rightarrow aB$, $A \rightarrow ac$, $A \rightarrow ad$, $B \rightarrow Ax$, $B \rightarrow Axy$, $B \rightarrow b$. The states are numbered purely for convenience.

The Basic Transformation

We now wish to consider a restricted kind of MS(k) machine. As we have said, an MS(k) machine uses a two-dimensional doubly infinite stack (of stacks) in its processing. Now consider an MS(k) machine such that each of its constituent sub-machines is cycle-free; that is, such that there is no sequence of next-state transitions (as specified by the function f)from a state back to itself.Such a machine uses its stack in a very constrained and restricted manner. Since each constituent sub-machine is cycle-free,each sub-machine can never use more than a bounded portion of its stack level; so the stack levels need not be stacks at all, and each sub-machine is basically a finite-state machine. The MS(k) machine as a whole is thus essentially a collection of finite-state machines that can call each other.

Since an MS(k) machine can recognize a non-regular language, it must make effective use of its stack. An unrestricted MS(k) machine uses the stack in two ways; while a cycle-free machine uses only the potentially recursive call sequences of the sub-machines, rather than the loop structure of an individual sub-machine.

In the case of a cycle-free MS(k) machine, no stack level will ever have more than a finite number of symbols on it. Looking just at the grammatical symbols (and not at the state names), we can interpret the contents of an individual stack level as being not a sequence of symbols, but as the representation of a single symbol from some new finite alphabet. Under this interpretation, the two-dimensional stack becomes one-dimensional again. And if we were to observe the sequence of stack configurations such a machine goes through while performing an MS(k) parse, it would look remarkably like the sequence of stack configurations that a top-down (LL(k)) parser goes through, for a different underlying grammar. This new grammar would have rules of the form $X_1 \rightarrow aX_2$, $X_1 \rightarrow X_2X_3$, $X_1 \rightarrow X_2$, and $X_1 \rightarrow \varepsilon$, corresponding to the MS(k) machine's performing a read, a prediction, a reduction, or a termination of a prediction. We can derive this new grammar directly from the structure of the machine; the first step of this derivation involves assigning "names" to all the states.

Definition Let M be an MS(k) machine for the LR(k) grammar G, q and q' states of M. We say q' is an α-successor of q if there is a sequence of transitions going from q to q' which spell out α.

Definition M is a cycle-free MS(k) machine if no state of M is its own α-successor for any $\alpha \in V^*$.

Definition If M is a cycle-free MS(k) machine for G, a state name is a pair (X_i, α) where $X \in N$ and $\alpha \in V^*$.

Algorithm (State-naming) Let M be a cycle-free MS(k) machine for G. The states of M are assigned names as follows:

1) Each initial state is given a unique name of the form (X_i, ε), where X is the nonterminal with which that initial state is associated (i.e., which it is dedicated to find)
2) If q is an α-successor of an initial state named (X_i, ε), then q is named (X_i, α).

The reason for the subscripts is to assign different names to different initial states which happen to be associated with the same nonterminal. We shall drop them for the remainder of our discussion. Note that several states may have the same name and that a given state may have more than one name; however, initial states have only one name each. This naming algorithm terminates because M is cycle-free. We use these state names as the nonterminals of the derived grammar.

Definition If M is a cycle-free MS(k) machine for G, the grammar T (M,G) is defined as follows:

1) The terminals of T(M,G) are the terminals of G
2) The nonterminals of T(M,G) are the names assigned the states of M by the foregoing algorithm.
3) The sentence symbol is the name of the starting state of M (usually written (S,ε))
4) The productions of $T_M(G)$ are defined as follows:

   i) If q is a state of M, (X,α) is a name of $q, a \varepsilon T$, and $f(q,a,\omega)$ is defined for some ω, then:

   $(X,\alpha) \rightarrow a\ (X,\alpha a)$

   is a production.

   ii) If q is a state of M, (X,α) is a name of q, $g(q,\omega) = q'$ for some ω, and (Y,ε) is the name of q', then:

   $(X,\alpha) \rightarrow (Y,\varepsilon)\ (X,\alpha Y)$

   is a production

   iii) If q is a final state of M corresponding to the rule of $G, A \rightarrow \beta$, and $(X,\alpha\beta)$ is a name of q, then:

   $(X,\alpha\beta) \rightarrow (X,\alpha A)$

   is a production

   iv) For each name (X,ε),

   $(X,X) \rightarrow \varepsilon$

   is a production.

There are four kinds of productions in T(M,G), which are induced by different constructs in M.

Rules of the form $(X,\alpha) \rightarrow a(X,\alpha a)$ are caused by transitions on terminal symbols. For each prediction in the machine, there will be one or more rules of the form $(X,\alpha) \rightarrow (Y,\varepsilon)\ (X,\alpha Y)$. Final states cause the introduction of rules like $(X,\alpha\beta) \rightarrow (X,\alpha A)$. And erasing rules $(X,X) \rightarrow \varepsilon$ are induced by the connections from initial states to the RETURN state.

In Figure 2 we show the grammar derived from the MS(1) machine of Figure 1, which is clearly cycle-free. Note that states 4,5,6, and 7 of the machine are each assigned two names by the naming procedure; while states 11 and 12 are given the same name.

We can establish a close connection between leftmost derivations in the grammar T(M,G) and configurations of the machine M. The precise expression of this relationship is cumbersome, but it can be approximately stated by the following two lemmas.

<u>Lemma</u> Suppose that after reading the string ω, the cycle-free machine M has a stack configuration of n levels, with α_i being the contents of the ith level. Then there is a leftmost derivation in T(M,G)

$$(S,\varepsilon) \overset{*}{\underset{L}{\Rightarrow}} \omega\ A_n\ A_{n-1} \ \cdots\cdots\ A_1$$

where A_i is a nonterminal of T(M,G) representing α_i.

<u>Lemma</u> Suppose there is a leftmost derivation in T(M,G), $(S,\varepsilon)\$^k \overset{*}{\underset{L}{\Rightarrow}} \omega_1\ A_n \cdots\cdots A_1\ \$^k \overset{*}{\underset{L}{\Rightarrow}} \omega_1\ \omega_2\ \$^k$.

Then if M is presented with input $\omega_1\ \omega_2\ \$^k$, it eventually reaches a configuration where the remaining input is $\omega_2\ \$^k$ and where the stack has n levels, the ith level α_i representing A_i.

The key to these lemmas lies in the nature of the relationship between α_i and A_i. The correspondence is a simple one. Let $X_1\ X_2 \ldots\ X_m$ be the grammatical symbols of G on α_i; then $A_i = (X_1, X_2 \ \cdots\cdots\ X_m)$. The other direction is similar. (Recall that the first symbol written on a new stack level is always the name of the

$(S,\epsilon) \rightarrow a(S,a)$	$(B,\epsilon) \rightarrow a(B,a)$	$(B,aB) \rightarrow (B,A)$
$(S,a) \rightarrow (B,\epsilon)\ (S,aB)$	$(B,\epsilon) \rightarrow b(B,b)$	$(B,A) \rightarrow x(B,Ax)$
$(S,a) \rightarrow d(S,ad)$	$(B,a) \rightarrow c(B,ac)$	$(B,Ax) \rightarrow (B,B)$
$(S,a) \rightarrow c(S,ac)$	$(B,a) \rightarrow d(B,ad)$	$(B,Ax) \rightarrow y(B,Axy)$
$(S,aB) \rightarrow (S,A)$	$(B,a) \rightarrow (B,\epsilon)\ (B,aB)$	$(B,Axy) \rightarrow (B,B)$
$(S,ad) \rightarrow (S,A)$	$(B,b) \rightarrow (B,B)$	$(B,B) \rightarrow \epsilon$
$(S,ac) \rightarrow (S,A)$	$(B,ac) \rightarrow (B,A)$	
$(S,A) \rightarrow x(S,Ax)$	$(B,ad) \rightarrow (B,A)$	
$(S,Ax) \rightarrow (S,S)$		
$(S,S) \rightarrow \epsilon$		

Figure 2

predicted nonterminal.)

These results yield the following theorem.

Theorem $L(T(M,G)) = L(M)$.

But since M is an MS(k) machine,

Corollary $L(T(M,G)) = L(G)$.

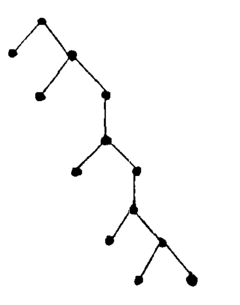
Thus our transformation preserves the language of the grammar. Since M is known to be deterministic, we can further exploit the relationship between leftmost derivations in T(M,G) and sequences of configurations of M.

Theorem Let M be a cycle-free MS(k) machine for the LR(k) grammar G. If $k > o$, then T(M,G) is strong LL(k); if $k = 1$, T(M,G) is strong LL(1).

Thus if we are provided with a cycle-free MS(k) machine M for the LR(k) grammar G, we can derive an equivalent strong LL(k) grammar T(M,G). The T(M,G) trees for strings in L(G) are quite different from the corresponding G-trees. In fact, we have the following result.

Theorem Let $\omega\epsilon T^*$. If $(A,\alpha) \overset{*}{\underset{L}{\Rightarrow}} \omega$ in T(M,G), then $A \overset{*}{\underset{R}{\Rightarrow}} \alpha\omega$ in G.

Because of this, we can think of (A,α) as standing for "the rest of A after α". In these terms, the rules T(M,G) can be interpreted in an intuitive manner. But the proof of this result indicates that the T - transformation radically alters the structure of trees. Now obviously, since T(M,G) is LL(k) even if G is not, we know T must replace left recursion by right recursion; but the effect is more than that. For a given stack level in M, i.e., an (A,ε) node in the T(M,G) tree, the tree's structure mimics the activities of M on that level. Thus the tree will locally look like:



The bifurcating branches reflect reads, while the single branches correspond to reductions that M makes. The tree as a whole is obtained by pasting together a number of these local trees, at the places where M makes a prediction.

However, even though the tree structure has been altered, the pertinent information is still available.

Theorem The LR(k) parse for ω in G can be reconstructed from the LL(k) parse of ω in T(M,G).

The reconstruction procedure is straightforward. We cull out from the list of rules in the LL(k) parse just those of the form $(X,\alpha\beta)\to(X,\alpha A)$, and form the list of corresponding rules G, $A\to\beta$. This list gives the order of reductions performed by the MS(k) parser M, and hence is the LR(k) parse.

Furthermore, the grammar T(M,G) can be used to effect the same compilation activities as those which G could drive. One model for these activities is the syntax directed translation of Lewis and Stearns [8].

Theorem Every simple Polish translation scheme based on G can be expressed as a simple translation on T(M,G).

Since simple translations can be performed in conjunction with LL(k) parsing, and simple Polish translations are one characterization of the kinds of translating activities that can be done by an LR(k) parser, this is one way of saying that T(M,G) is as useful as G. A stronger result, which uses a different model of compiling activities, can be found in [10].

The grammar T(M,G), as we have seen, is strong LL(k), and so can be used to drive a top-down parser for the language L(G). However the number of steps in an LL(k) parse using T(M,G) is equal to the number of steps in an MS(k) parse performed by M based on G, which, as we have seen, will in general be greater than the number of steps in an LR(k) parse based on G. Thus we may sacrifice some parsing speed in converting into LL(k) form. Furthermore, the grammar T(M,G) may be very uneconomical; the number of nonterminals and rules in T(M,G) can be very large, depending on the number of predictive states in M and the interconnections between states of different submachines. In [10] we provide reduction schemes for eliminating by substitution a maximal number of nonterminals of T(M,G), thereby reducing the size of the grammar. Since T(M,G) is usually highly redundant, these techniques almost always result in very significant improvements in the size of T(M,G). Such reductions can also have a very salubrious effect on the parsing speed of T(M,G) by eliminating chains of nonterminals and combining several rules together. The substitution algorithm is so designed so as not to disturb the LL-ness of the derived grammar; but the price we may pay for decreasing the size of the grammar is a potential increase in the value of k for which the grammar is strong LL(k), i.e., the amount of lookahead needed by a top-down parser. Such an increase can, of course, dramatically increase the size of the parser's driving table. There are, however, well-known techniques by which this potential increase can be somewhat ameliorated; and the substitution algorithms allow for explicit consideration of trade-offs between decrease in the size of the grammar and increase in the lookahead value.

Transformable Grammars

The starting point for the transformation described above is a cycle-free MS(k) machine for G. Several problems still remain: determining whether or not a given LR(k) grammar has a cycle-free MS(k) machine; finding that machine if it does exist; and choosing, if there are several such machines, the one that will result in the "best" derived grammar.

Definition A grammar G is k-transformable if there exists a cycle-free MS(k) machine for G.

Theorem It is decidable if an LR(k) grammar G is k-transformable.

The simplest decision procedure finds the cycle-free machine if it exists, but it is inefficient; nor does it distinguish "good" machines from "bad" ones. In [10] we provide some heuristic guides for improving on this algorithm. The approach taken is to start with the LR(k) machine for G and try to eliminate all cycles from it, one by one, by splitting states in the cycles in certain ways. We derive methods for finding splittings of a state that effect the removal of the cycles of which that state is a member. These techniques work in most practical cases, for non-pathological grammars, and are much more directed and efficient than the simple decision procedure. Furthermore, it is possible to direct these methods so that the resulting machine has various desirable properties which tend to minimize the size of its derived grammar.

These desiderata include such properties as minimizing the number of predictive states in the machine or the number of items in the base state of a splitting. In analyzing these procedures, we determined that in most realistic cases, bipartite splittings suffice to break all the cycles in the LR(k) machine of a k-transformable grammar.

While we have no precise characterization of the class of k-transformable grammars, we do have some idea of its extent. We can show that it strictly includes the LC(k) grammars of [4].

Definition The core of the item $A \to \alpha.\beta(\omega)$ is $A \to \alpha.\beta$.

Lemma Let q be any non-final state of the LR(k) machine for the LC(k) grammar G, and let I_1 and I_2 be any two essential items of q with different cores. Then $FIRST_k\ (I_1) \cap FIRST_k\ (I_2) = \phi$.

Lemma Let q be as in the preceding lemma. Then there is a splitting (B,R) of q such that B consists of precisely the essential items of q.

Because of the first lemma, it is possible to predict in q every nonterminal that follows the dot of an essential item.

Theorem If G is LC(k), then G is k-transformable.

The preceding lemma enables us to establish a simple procedure to create a cycle-free MS(k) machine for an LC(k) grammar, given the LR(k) machine for it. The procedure is simply to select some state of the machine and replace it by the specified kind of splitting; pick some state of the resultant machine, and do the same thing; and iterate this process until a cycle-free machine is constructed.

Corollary If G is LL(k), then G is k-transformable.

Corallary The languages accepted by the class of cycle-free MS(k) machines are precisely the LL(k) languages.

Theorem The class of k-transformable grammars strictly includes the class of LC(k) grammars.

The grammar $S \to bAc$, $A \to ABx$, $A \to ABy$, $A \to a$, $B \to Bd$, $B \to d$ is k-transformable but not LC(k), for every value of $k \geq 0$.

Acknowledgements

The author wishes to thank P.M. Lewis, D. J. Rosenkrantz, and R.E. Stearns for many helpful discussions and suggestions.

References

1. Stearns, R. E., "Deterministic Top-Down Parsing," Proceedings of Fifth Annual Princeton Conference on Information Sciences and Systems, March 1971, 182-189.

2. Foster, J. M., "A Syntax Improving Program," Computer Journal 11, 31-34 (1968)

3. Wood, D., "The Normal Form Theorem-Another Proof," Computer Journal 12, 139-147 (1969).

4. Rosenkrantz, D. J., and Lewis, P. M., "Deterministic Left Corner Parsing," Conference Record IEEE 11th Annual Symposium on Switching and Automata Theory, October 1970, 139 - 152.

5. Tixier, V., "Recursive Functions of Regular Expressions in Language Analysis," Ph.D. Thesis, Stanford University, Stanford, California, 1967.

6. Lomet, D. B., "A Formalization of Transition Diagram Systems," JACM 20, 235-257 (1973)

7. Knuth, D. E., "On the Translation of Languages from Left to Right," Information and Control 8, 607-639 (1965)

8. Lewis, P.M., II, and Stearns, R. E., "Syntax-Directed Transduction," JACM 15, 464-488 (1968)

9. Aho, A. V., and Ullman, J. D., The Theory of Parsing, Translation and Compiling, Prentice-Hall, Englewood Cliffs, N.J., 1972

10. Hammer, M., "A New Grammatical Transformation into Deterministic Top-Down Form," MIT Project MAC Technical Report TR-119, February 1974