A NEW GRAMMATICAL TRANSFORMATION INTO DETERMINISTIC TOP-DOWN FORM

Michael M. Hammer

Massachusetts Institute of Technology

Prepared for:

Office of Naval Research

February 1974

BIBLIOGRAPHIC DATA SHEET	1. Report Nos. NSF-OCA-GJ34671 + N00014-70-A-0362-0001, TR-119	2.	3. Recipient's Accession No. AD 775 545

4. Title and Subtitle

A New Grammatical Transformation Into Deterministic Top-Down Form

5. Report Date: Issued February 1974

6.

7. Author(s)

Michael M. Hammer

8. Performing Organization Rept. No. MAC TR-119

9. Performing Organization Name and Address

PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY

545 Technology Square, Cambridge, Massachusetts 02139

10. Project/Task/Work Unit No.

11. Contract/Grant No.s. GJ34671 and N00014-70-A-0362-0001

12. Sponsoring Organization Name and Address

Office of Naval Research
Department of the Navy
Information Systems Program
Arlington, Va 22217

Associate Program Director
Office of Computing Activities
National Science Foundation
Washington, D. C. 20550

13. Type of Report & Period Covered: Interim Scientific Report

14.

15. Supplementary Notes

Ph.D. Thesis, M.I.T., Department of Electrical Engineering, August 1973

16. Abstracts

Although deterministic top-down parsing is an attractive parsing technique, the grammars to which it is applicable (the LL(grammars) are but a small subset of the LR(k) grammars, those that can be parsec tinistically bottom-up. In this thesis, the problem of transforming LR(k) g n'o equivalent LL(k) grammars is studied.

A new transformation procedure is devised which is more powerful than currently available techniques and which preserves the compiling ability of the grammar.

17. Key Words and Document Analysis. 17c. Descriptors

Compilers

Deterministic grammars

Parsing

Syntax Analysis

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited Distribution	UNCLASSIFIED	302
	20. Security Class (This Page) UNCLASSIFIED	22. Price 7.25/1.45

/.

MAC TR-119

A NEW GRAMMATICAL TRANSFORMATION INTO DETERMINISTIC TOP-DOWN FORM

Michael M. Hammer

February 1974

This research was supported in part by the National Science Foundation under research grant GJ-34671, and in part by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 433 which was monitored by ONR Contract No. N00014-70-A-0362-0001.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

\IDGE MASSACHUSETTS 02139

A NEW GRAMMATICAL TRANSFORMATION INTO DETERMINISTIC TOP-DOWN FORM

BY

MICHAEL MARTIN HAMMER

Submitted to the Department of Electrical Engineering on August 24, 1973 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

ABSTRACT

Although deterministic top-down parsing is an attractive parsing technique, the grammars to which it is applicable (the LL(k) grammars) are but a small subset of the LR(k) grammars, those that can be parsed deterministically bottom-up. In this thesis, the problem of transforming LR(k) grammars into equivalent LL(k) grammars is studied.

A new method of parsing, called multiple stack parsing, is introduced. This is a generalization of LR(k) parsing, containing a minimal infusion of top-down predictive techniques into deterministic bottom-up parsing. An automaton, the MSP(k) machine, which parses strings in this way is formally defined, and is shown to be equivalent to the canonical LR(k) parsing machine. A transformation procedure is described which constructs from M, a particular kind of MSP(k) machine for the grammar G (called a cycle-free machine), a new derived grammar $T_M(G)$. The grammar $T_M(G)$ generates the same language as the grammar G does and is (strong) LL(k) as well. No translating ability is lost in effecting this transformation, in the sense that $T_M(G)$ can support the same class of compilation activities, or syntax-directed translations, that G can.

The class of grammars which can be parsed by cycle-free MSP(k) machines and so are amenable to transformation, strictly includes both the LL(k) grammars, and the LC(k) (left corner parsable) grammars. Thus our transformation is more powerful than the previously available one of Rosenkrantz and Lewis. Furthermore, there are good algorithms applicable to many transformable grammars for constructing a cycle-free MSP(k) machine and making the entire transformation process more efficient; and the size of $T_M(G)$ can be systematically reduced without sacrificing its desirable qualities.

ACKNOWLEDGEMENTS

The Talmud (Avot 6) says: "From whomever one learns one topic, one rule, one word, or even one letter, one is obliged to show him honor and respect." Were I to follow this dictum precisely, these acknowledgements would be as lengthy as the body of the thesis. Nevertheless two individuals deserve extraordinary mention.

Professor F. C. Hennie has been my teacher and counselor throughout graduate school, and any education that I have managed to absorb has largely been his doing. Dr. P. M. Lewis II suggested the original topic of this research and stimulated my thinking in profitable directions; he also tried to instill in me some of that unteachable quality called judgment. I also want to thank Dr. D. J. Rosenkrantz and Dr. R. E. Stearns for their many helpful discussions and suggestions, and Professor M. J. Fischer for his patience. Much credit is due Ms. Marsha Baker and Ms. Barbara Kohl for their invaluable assistance in the preparation of the thesis document.

Finally, I should like to dedicate this thesis to my parents and to my wife Phyllis, whose collective support and encouragement have seen me through all of graduate school, and especially through the preparation of this thesis.

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

1.1 Background and Statement of the Problem

For some time, it has been recognized that context-free grammars form a good meta-language for the syntactic description and specification of programming languages. And over the past decade, there has been a developing trend toward eschewal of "hand-coding" the syntactic analysis phase of the compilers for such languages. Rather the tendency has been toward the so-called syntax-directed compiling schemes, wherein a stylized representation of the grammatical specification of the language is directly used as data by some general purpose parsing scheme to process programs in the language. Whatever speed is lost by having the parsing done by a general processor is offset by the ease of implementation of such a compiler and the fact that its clean design makes it easily comprehensible and amenable to change should the occasion arise. Furthermore, the speed problem has become less pronounced as a variety of fast parsing schemes have been developed, each applicable to some particular class of grammars.

This development has been further encouraged by burgeoning interest in compiler-compliers, or automatic translator writing systems. To build a compiler with such a system, a language designer submits to the compiler-compiler a syntactic specification of his language in BNF and a semantic specification of the language, which commonly associates with each rule of the grammar an "action routine," specifying how an instance of that rule is to be processed by a compiler. The system then tailors its general-purpose table-driven parser to the supplied grammar, hooks in the semantic routines, and out comes a

compiler for the language.

The catch in all this is that the grammar devised by a language designer, though reflecting his perception of the language and its features, and also possibly of great descriptive value, might not be amenable to parsing by whatever particular tools and methods a compiler writer has available. A grammatical description which is good for people is not necessarily good for computers, for use in an implemented compiler for the language. On the contrary, the more constrained and idiosyncratic a class of grammars is, the more compact and efficient a compiler designed on its defining principles can be; yet the more specialized such a class is, the less likely it is that an arbitrary human-designed grammar will satisfy the characteristic conditions of that class. For example, the general parsing method of Knuth for the full class of LR(k) (or deterministically parsable in a left-to-right manner) grammars, yields a parser much too large to be of real practical value. A great many subclasses of this general class have been devised, each with its own specific parsing algorithm; most of these are summarised in [1]. However, it is all too easy for a grammar, though meeting the fairly lax and reasonable requirements of the class of LR(k) grammars, to fail to meet all the restrictions of one of these smaller classes. And very often, the failure is not for any systemic reason; that is, there often is another grammar for the same language which does satisfy the appropriate constraints to enable parsing in the desired manner. It is just that the particular grammar, as designed by the language designer, fails to pass all the tests.

This situation is particularly acute in the case of deterministic top-down parsing. This is a mode of parsing that has long been known, has proven

to have a variety of attractive features, and yet has fallen into disuse for quite some time. This is primarily due to the fact that most grammars do not meet the strict requirements under which this method can be applied. For example, if there is even one left-recursive nonterminal in the grammar (as is frequently the case), then the grammar fails to be deterministically top-down parsable.

This is a truly unfortunate state of affairs, for deterministic top-down parsing is indeed a most attractive technique. The grammars amenable to such processing have been termed the LL(k) grammars by Lewis and Stearns [13], and have been studied by many authors [12, 18, 23, 11, 20]. In particular, the latter two are good surveys of the field. In Stearns' words, the two chief advantages of LL(k) parsing (in particular, LL(1) parsing), are:

1) parsing can be carried out by a one state deterministic pushdown machine using very simple logic;

2) the parsing algorithm accommodates very flexible language translation.

LL(k) parsing proceeds in a very direct and efficient manner, which mimics the leftmost derivation of the program in the grammar. When a nonterminal is on top of the parsing stack, examination of k symbols of lookahead resolves the decision as to which grammar rule is to be applied to that nonterminal; the nonterminal is then replaced on the stack by the right-hand side of the rule, and the process continues. (When a terminal symbol is on top of the stack, it is matched against the next input symbol and both are eliminated.) A parser constructed on these principles can be made to run very rapidly; and its design is so simple and clean that it should be easy to implement. In addition,

an LL(k) parser is in a much better position to give meaningful diagnostics and to attempt error recovery when presented with an illegal program, than is an LR(k) parser; because while the LR(k) parser merely becomes stymied when it encounters the error, the LL(k) parser knows the current goal, (i.e., what it is trying to find), and can thus report somewhat more precisely as to the nature of the error.

Furthermore, since the identity of the rule to be applied is determined by an LL(k) parser before all symbols generated by that rule have been read (this is in contrast to bottom-up parsing schemes), it is possible to determine what semantic routine is to be applied before its arguments have already been read and processed. That is, actions can be executed before the end of a rule has been reached, and so it is exceptionally easy to coordinate semantic processing of a general kind with a top-down parser.

Lewis and Rosenkrantz [14] have recently reported on a compiler for ALGOL 60, which uses an LL(1) parser, and which was designed in a structured way based on the principles elucidated above. This compiler achieves both speed in compiling (not just in parsing) and a clean and understandable design. Similar principles have been applied in the rapid construction of an efficient and well-designed FORTRAN compiler. A compiler-compiler, which constructed LL(k) parsers for the grammars supplied it, could reasonably be expected to produce good compilers. Thus it is regrettable that most grammars fail to be LL(k) and are consequently unsuitable for deterministic top-down parsing.

In order to apply LL(k) parsing techniques more often, we would like to be able to apply some transformations to non-LL(k) grammers and convert them into LL(k) form. Even if the transformed grammar were to have a very different structure from, and bear no apparent resemblance to, the original grammar, it would not matter;

for the original grammar could still continue to serve its definitional and pedagogic functions, while the scope of the new grammar would be confined to the compiler implementation. The only point for caution would be that the transformation should preserve translatability (in the sense of syntax-directed translations). That is, any actions performed by semantic routines operating in conjunction with a bottom-up parser for the original grammar, should be implementable by semantic routines working with the LL(k) parser for the transformed grammar. If this condition is met, it is possible to envision a new stage of a compiler-compiler, being a transformation phase. This step would convert a non-LL(k) grammar into LL(k) form, simultaneously transforming the associated semantic specification; it would pass the new grammar to the main body of the translator writing system, which would be designed to construct LL(k) compilers for the grammars provided to it.

In fact there has been a certain amount of research along these lines, [6 , 19, 20], but most of it has concerned variants of the same basic principle, a common transformation to rid the grammar of left-recursive nonterminals. This transformation has been described by a number of authors [8, 17 ,22], and we shall call it the Rosenkrantz transformation. The transformation is rather complex, but basically it changes rules $A \rightarrow Ab$ and $A \rightarrow a$ into rules $A \rightarrow a(A, A)$; $(A, A) \rightarrow b$; and $(A, A) \rightarrow \varepsilon$. Here (A, A) is a new nonterminal that stands for "the rest of an A after an A;" the idea is that if A generated $A\alpha$ in the old grammar, (A, A) will generate α in the new one. Foster's Syntax Improving Device [6] (as well as Wood's improved version of it [24]) basically consists of the application of this transformation, possibly followed by some left factoring. (Left factoring is a process of trans-

forming rules $A \rightarrow BC$ and $A \rightarrow BD$ into $A \rightarrow BX$, $X \rightarrow C$, and $X \rightarrow D$.) These two transforming methods (possibly used in conjunction with substitution) are those cited by Stearns [20] as being the tricks most widely used in attempting to create an equivalent LL(k) grammar, given a grammar which is not LL(k). At the present time however, our understanding of these transformations remains at a rudimentary level; and the method of their application usually consists of heuristics and guesswork. There is little even in the way of good strategies for the use of these transformations. In short, the entire issue of transforming grammars into LL(k) form is not well understood, and the current state of the art is largely trial-and-error.

The purpose of this thesis is to contribute to the state of knowledge and understanding of the process of transforming grammars into LL(k) form. We shall introduce a new transformation which will be applicable to a wide class of grammars; and we shall be able to determine if an arbitrary grammar is in this class. Furthermore, our transformation will proceed in a deterministic manner, and we shall investigate the nature of its process, both on formal and intuitive levels. And so as to be of practical value, the transformation will preserve translatability, in the sense outlined above. In short, we hope to make a contribution towards rationalizing the process of finding equivalent LL(k) forms for non-LL(k) grammars.

We see the possible value of our work coming from two sources. On the one hand, it should serve to further the sum of knowledge in this area and perhaps lead

to a full and complete solution to the problem. At the same time, we can envision the implementation of an actual grammar transformer, embodying the principles of the transformation we shall devise. We shall keep both of these goals in mind throughout. The bulk of our work is of a formal and precise nature, hoping to avoid the vague and often unsatisfying approach frequently taken to work of this kind. Yet we shall also strive to provide intuitive explanations of our development, and we shall not neglect the issues of time and space which assume primary importance in the context of an implementation effort. We shall also include some less formal and semi-heuristic suggestions as to how our transformation may be accomplished most efficiently in many cases.

There has been some research in the spirit just adumbrated. In [19], Rosenkrantz and Lewis have characterized the class of grammars that can be converted into LL(k) form by a single application of the Rosenkrantz transformation. They denote these grammars as the LC(k) grammars; these grammars are distinguished by the fact that they can be parsed in a peculiar hybrid of top-down and bottom-up parsing, known as the left-corner bottom-up parsing method. This technique has been known for some time and was used by a variety of authors in early compiling systems [3, 7, 9]. For a long time this technique was thought of as being pure bottom-up [11], but deeper understanding of it has shown it to be a mixture of the two major classes of parsing schemes. Basically, a left corner parse consists of doing a bottom-up parse in order to find the left corner of a rule (the first symbol on its right hand side), and then switching over to a predictive scheme to determine

the rest of the rule. In their paper, Rosenkrantz and Lewis formally define the LC(k) grammars as those which are amenable to being parsed in such a fashion. They define a formal automaton (the canonical left corner pushdown machine) as a model for the actions of such a parser, and they argue that such a machine, designed to parse a specific LC(k) grammar, does indeed succeed in recognizing precisely the language of that grammar. Then they use this machine model and its relationship to a machine model for LL(k) parsing to show that a grammar is LC(k) if and only if applying the Rosenkrantz transformation to it creates an LL(k) grammar. Finally they show that a large class of translations on the original grammar are also expressible as translations on the transformed grammar.

To our knowledge, this is the only firm and theoretical work on the problem of transforming non-LL(k) grammars into equivalent LL(k) forms. (However, in the only published version of this work, the proofs are omitted, presumably because of their length and difficulty.) We have taken it as our mandate to further the work of Rosenkrantz and Lewis, and to provide additional insight into the transformation process in a precise and rigorous way. We shall use their earlier work as a jumping-off point for our research in several ways. First of all, we shall try to proceed in the same rigorous fashion. Secondly, we shall take it as our mission to find a new method of transformation which will supersede the Rosenkrantz transformation and which will be applicable to a class of grammars which strictly includes the LC(k) grammars. And finally, we shall make use of a similar methodology; namely, we shall devise an even more mixed and general hybrid scheme of parsing, develop an automaton model for it, and show how this machine may be used to derive a new LL(k) form

for the original grammar for which the automaton was constructed.

We shall concentrate on the problem of transforming LR(k) grammars into LL(k) form. (Although not generally stated, most of the other work in the field has also been subject to this same restricted viewpoint.) That is, we shall devise an algorithm for taking a grammar supplied by a language designer, that could as it stands be parsed in a deterministic bottom-up manner, and changing it into an equivalent LL(k) grammar. There are several reasons for limiting our interest to LR(k) grammars as opposed to arbitrary context-free grammars. First, the argument against top-down parsing has always been that many grammars which could be parsed in other deterministic ways could not be handled by an LL(k) parser. But if a grammar is not LR(k), then there is no way to parse it deterministically on a pushdown machine, so the added charge that it is not LL(k) is not very serious. It is just for those grammars that can be parsed in some other way that we want to find top-down forms, and thereby scotch the criticism of detractors of LL(k) parsing. Furthermore, it is too difficult to get a handle on an arbitrary context-free grammar; their general nondeterministic parsing methods do not allow for simple hybridization. And finally, it is not asking too much for a language designer to couch his description in a deterministic way. Arguments have been made that an LR(k) grammar is a natural mode of expression, and that if a grammar fails to meet its criteria, there is some structural defect in the language, some ambiguity of syntax or semantics. Thus we shall restrict our attention to LR(k) grammars throughout the thesis.

1.2 Outline of the Thesis

Chapter 2 consists primarily of a review of the standard terminology

and notation we shall be using. We do introduce some new terms, and we concentrate on the concepts of LR(k) parsing. We define a particular version of the LR(k) parsing machine which we shall find most convenient.

Chapter 3 is devoted to the development and explication of our particular new hybrid of top-down and bottom-up parsing. This technique can be easily summarized. It essentially consists of a standard left-to-right, bottom-up parse, except that the parser has the ability to make predictions as to the identity of nonterminals that it is going to find. That is, at any point during the parse the parser may, based on inspection of some lookahead symbols, predict that some prefix of the remaining input is going to be reduced to, say, an A. This is known as predicting an A. However, once the A is predicted, parsing continues in the normal bottom-up fashion. We can envision the prediction of an A as entailing the invocation of a separate parsing machine, whose goal is to reduce an input string to the an A (in contrast to the main parser, whose goal is to reduce the input to an S). When this A-parser (which operates in a pure bottom-up manner) has completed its task by reducing some prefix of the remaining input to an A, it returns control to the main parser, which picks up where the other one left off. This then is our new model of a parsing machine called an MSP(k) machine (for multiple stack parsing): a collection of bottom-up parsers which can decide to call each other (or themselves) if inspection of the lookahead warrants it. We note that the order of recognition by such a scheme is exactly that of conventional bottom-up parsing. To be precise, an MSP(k) machine for G does the same things as an LR(k) machine for G, except that it takes more steps to do it in (namely, making the predictions); and initially,

this technique seems to be of no particular interest.

Chapter 3 is devoted to laying all the fundamental ground work necessary for the use of these ideas. Sections 3.1 and 3.2 contain a detailed descriptive introduction to the concepts of this mode of parsing. Section 3.3 contains the formal definition of the notion of prediction in bottom-up parsing. Finally in Section 3.4, we introduce the formal model of an MSP(k) parsing machine, a generalization of the LR(k) machine, as an automaton which accommodates the making of predictions during bottom-up parsing. We show in Section 3.5 that every MSP(k) machine for G can be obtained from the LR(k) machine for G, by the repeated application of a relatively simple state-splitting algorithm. In Section 3.6, we prove the expected result that an MSP(k) machine for G does indeed recognize precisely the language generated by G, and that it parses such strings in the conventional bottom-up manner.

In Chapter 4 we restrict our attention to a particular kind of MSP(k) machine, called a cycle-free MSP(k) machine; and we show how to derive an LL(k) grammar from such a cycle-free machine. In Chapter 5 we address the problem of constructing such a cycle-free MSP(k) machine from the LR(k) machine for a grammar G. The combination of these two procedures is our transformation process: the construction of a cycle-free machine and then the derivation of a grammar from it.

A cycle-free MSP(k) machine is one such that each component submachine contains no cycles; hence each component machine needs only a finite stack. Given a cycle-free machine M for G, our transformation, set forth Section 4.2, derives a new grammar $T_M(G)$. The nature of this grammar is heavily dependent on the structure of M. We give a technique for naming the states

of a cycle-free machine; these names serve as the nonterminals of the derived grammar. The transitions among the states are schema for the rules of the grammar.

Once we have defined the transformed grammar $T_M(G)$, we establish its relationship to the original grammar G. First of all, we show that $T_M(G)$ generates the same language that M recognizes; we do this by establishing a correspondence between leftmost derivations in $T_M(G)$ and sequences of configurations of M. This then establishes that $L(T_M(G)) = L(G)$. Since M is defined to be deterministic, this means that the derived grammar is deterministically top-down parsable; i.e., LL(k). A summary of the ideas of this transformation, and some insight into its nature, is provided in Section 4.6.

In the final sections of Chapter 4, we de-emphasize the formal aspects of $T_M(G)$, and concentrate on its utilitarian value. First we establish that $T_M(G)$ is just as useful as G is, in terms of supporting translations; i.e., when it comes to directing compiling activities associated with the appropriate parser. Then we address the problem of bringing the size of $T_M(G)$ (which reflects the size and complexity of M), down to more manageable proportions, without sacrificing either its LL-ness or its compiling ability.

In Section 5.1, we study the class of grammars which can be parsed by cycle-free MSP(k) machines, and which therefore we can transform into equivalent LL(k) grammars. We show that this is a decidable class, so it is possible to determine if a given grammar is transformable. Then we prove that this class of grammars strictly contains both the LL(k) and LC(k) grammars. This means that we have achieved our goal of generalizing the work of Rosenkrantz and Lewis.

However, even though the decision procedure for this class is constructive (i.e., it produces a cycle-free machine for G if one exists), it is terribly unsophisticated and inefficient. In Section 5.2 we discuss a number of semi-heuristic strategies for the efficient construction of cycle-free machines; these techniques do not work for all transformable grammars, but they are applicable to a very wide class of grammars which includes all grammars of reasonable character. These methods would be of great use to a person or a compiler-compiler attempting to apply our transformation. Although the bulk of our work is formal and theoretical, the techniques of this section, and those of 4.8, bring this transformation into the realm of practicality, and make its implementation a feasibility.

Finally, in Chapter 6 we examine some of the peripheral issues and questions raised during the progress of this research. These deal with such topics as various possible generalizations of our work and speculations on some other kinds of transformations.

CHAPTER 2

FUNDAMENTAL CONCEPTS

A (context-free) grammar is a 4-tuple $G = (V_N, V_T, S, P)$, where V_N and V_T are disjoint sets of nonterminal and terminal symbols; S, the sentence symbol, is a member of V_N; and P is a set of productions or rules of the form $A \to \alpha$, where $A \in V_N$ and $\alpha \in (V_N \cup V_T)^*$. An A-rule is any rule $A \to \alpha$.

In general, we shall use the following notation with reference to a grammar G. Lower-case letters like a, b, c denote elements of V_T; symbols x, y, ω, τ, ρ generally represent strings in V_T^*; A, B, X are nonterminals, elements of V_N; α, β, γ, φ, Ψ represent strings in $(V_N \cup V_T)^*$; and σ is any symbol from $V_N \cup V_T$. We use ϵ to stand for the empty string. When we depart from these conventions or use other symbology, we shall explicitly quantify our usage.

If $\omega \in V_T^*$, then $|\omega|$ is the length of ω, the number of symbols in it. The length of ϵ is 0. If $\omega \in V_T^*$, then ω/k is that x such that $|x| = k$ and $\omega = xy$ for some y; if $|\omega| < k$, then $\omega/k = \omega$. V_T^k is the set of $\omega \in V_T^*$ such that $|\omega| = k$.

$\text{FIRST}_k(\alpha) = \{x \mid \alpha \overset{*}{\Rightarrow} xy$ for some y and $|x| = k\}$; $\text{FOLLOW}_k(\alpha) = \{x \mid S \overset{*}{\Rightarrow} \alpha\beta x\gamma$ for some α and γ and $|x| = k\}$.

We define the relation $\underset{G}{\Rightarrow}$ over strings in $(V_N \cup V_T)^*$ as follows: $\alpha \underset{G}{\Rightarrow} \beta$ if and only if $\alpha = \alpha_1 A \alpha_2$, $\beta = \alpha_1 \gamma \alpha_2$, and $A \to \gamma$ is a rule of G. (We shall generally omit explicit mention of G when the reference is clear.) The relation $\overset{*}{\underset{G}{\Rightarrow}}$ is the reflexive transitive closure of $\underset{G}{\Rightarrow}$; if $\alpha \overset{*}{\Rightarrow} \beta$, we say α produces or generates β. We say $\alpha_1 A \alpha_2 \underset{R}{\Rightarrow} \alpha_1 \gamma \alpha_2$ if $\alpha_2 \in V_T^*$; $\alpha_1 A \alpha_2 \underset{L}{\Rightarrow} \alpha_1 \gamma \alpha_2$ if $\alpha_1 \in V_T^*$; $\overset{*}{\underset{R}{\Rightarrow}}$ and $\overset{*}{\underset{L}{\Rightarrow}}$ are the reflexive transitive closures of $\underset{R}{\Rightarrow}$ and $\underset{L}{\Rightarrow}$. If

$\alpha \overset{*}{\Rightarrow} \beta$, then there is a sequence $\alpha_0, \alpha_1, \ldots, \alpha_m$ such that $\alpha = \alpha_0$, $\beta = \alpha_m$, and $\alpha_i \Rightarrow \alpha_{i+1}$; this sequence is called a derivation of β from α. The length of this derivation is m; in such a case, we say $\alpha \overset{m}{\Rightarrow} \beta$. If $\alpha_0 \underset{L}{\Rightarrow} \alpha_1 \underset{L}{\Rightarrow} \cdots \underset{L}{\Rightarrow} \alpha_m$, we call it a leftmost derivation; and similarly for a rightmost derivation.

The language of a grammar G, L(G), is $\{x \in V_T^* \mid S \overset{*}{\Rightarrow} x\}$. It is well known that $S \overset{*}{\Rightarrow} x$ iff $S \overset{*}{\underset{R}{\Rightarrow}} x$ iff $S \overset{*}{\underset{L}{\Rightarrow}} x$. We say α is a sentential form if $S \overset{*}{\Rightarrow} \alpha$; a left sentential form if $S \overset{*}{\underset{L}{\Rightarrow}} \alpha$; and similarly for a right sentential form. Say $S = \alpha_0 \underset{L}{\Rightarrow} \alpha_1 \underset{L}{\Rightarrow} \alpha_2 \underset{L}{\Rightarrow} \cdots \underset{L}{\Rightarrow} \alpha_n = \omega$ is a leftmost derivation of ω; let $\alpha_i = \beta_i A_i \gamma_i$ and $\alpha_{i+1} = \beta_i \varphi_i \gamma_i$, where $A_i \to \varphi_i$ is a rule of G. Then if $A_i \to \varphi_i$ is rule p_i, then the sequence of rules $p_0, p_1, p_2, \ldots, p_{n-1}$ is called the top-down (left-to-right parse) of ω. On the other hand, if $S = \alpha_0 \underset{R}{\Rightarrow} \alpha_1 \underset{R}{\Rightarrow} \alpha_2 \underset{R}{\Rightarrow} \cdots \underset{R}{\Rightarrow} \alpha_n = \omega$, then the sequence of rules $p_{n-1}, p_{n-2}, p_{n-2}, \ldots, p_2, p_1, p_0$ is the bottom-up (left-to-right) parse of ω. The parsing problem is given ω, to find a top-down or bottom-up parse of ω.

The LL(k) grammars, defined by Lewis and Stearns [13], is the largest class of grammars for which a deterministic one-pass algorithm exists to find the left-to-right top-down parse of ω.

Definition Let $k > 0$. A grammar G is LL(k) iff given $\omega \in V_T^k$, $A \in V_N$, and $u \in V_T^*$, there is at most one production p such that for some y and $v \in V_T^*$, $S \overset{*}{\Rightarrow} uAv$, $A \overset{*}{\Rightarrow} y$ beginning with production p, and $(yv)/k = \omega$.

Definition G is strong LL(k) iff given $\omega \in V_T^k$ and $A \in V_N$, there is at most one production p such that for some u, y, and v in V_T^*, $S \overset{*}{\Rightarrow} u A v$, $A \overset{*}{\Rightarrow} y$ beginning with production p, and $(yv)/k = \omega$. (Equivalently, if $A \to \varphi_1$ and $A \to \varphi_2$ are different A-rules, then $FIRST_k(\varphi_1 FOLLOW_k(A)) \cap FIRST_k(\varphi_2 FOLLOW_k(A)) = \emptyset$.)

For any LL(k) grammar, it is possible to find an equivalent strong LL(k) grammar. There is a very simple parsing procedure for any strong LL(k) gram-

mar, which uses a pushdown stack. The input to the algorithm is the string x to be parsed. The procedure begins with the stack containing only S. At any point in the procedure, either a nonterminal or terminal symbol will be on top of the stack. If the top is a terminal symbol, it is compared against the next input symbol: if there is a match, the symbol is popped off the stack and removed from the input; otherwise, there is an error. If a nonterminal A is on top of the stack, then we set ω equal to the next k symbols of the input. By the definition of strong LL(k) grammar, the combination of A and ω specify a single rule p of the form $A \to \alpha$. This rule is the next one in the parse; to continue the procedure, we remove A from the stack and replace it by α.

The LR(k) grammars of Knuth [10] are the largest class of grammars for which there exists a one-pass algorithm to find the left-to-right bottom-up parse of x. There are a variety of equivalent definitions of this class. We give one provided by Lewis and Stearns [13].

Definition G is LR(k) if it is unambiguous and if for all ω_1, ω_2, ω_3, ω_3' in V_T^* and A in V_N, $S \overset{*}{\Rightarrow} \omega_1 A \omega_3$, $A \overset{*}{\Rightarrow} \omega_2$, $S \overset{*}{\Rightarrow} \omega_1\omega_2\omega_3'$, and $\omega_3/k = \omega_3'/k$ imply that $S \overset{*}{\Rightarrow} \omega_1 A \omega_3'$.

This means that the part of a right sentential form last introduced into it (the handle), can be determined by looking ahead k symbols after it.

The construction of a parser for LR(k) grammars is somewhat more complex, and we describe it more carefully. There are several variants of the construction of this parser; our formulation is similar to that of Aho and Ullman [1], but differs in certain respects.

Definition If G is an LR(k) grammar, then $A \rightarrow \alpha.\beta(\omega)$ is an LR(k)-item over G if $A \rightarrow \alpha\beta$ is a rule of G, $\beta \neq \epsilon$, and $\omega \in V_T^k$. Furthermore, if $A \rightarrow \epsilon$ is a rule of G, then $A \rightarrow .\epsilon(\omega)$ is an LR(k)-item over G.

Definition If $I = A \rightarrow \alpha.\sigma\ \beta(\omega)$ is an LR(k)-item over G, then

i) I is an A-item

ii) I is a $.\sigma$-item

iii) if $\sigma \in V_T$, then I is a terminal item; furthermore, $A \rightarrow .\epsilon(\omega)$ is a terminal item

iv) σ is the post-dot component of I

v) $\sigma\ \beta$ is the post-dot section of I

vi) ω is the context of I

vii) if $\alpha \neq \epsilon$, I is an essential item

Informally, the language of an item $A \rightarrow \alpha.\beta(\omega)$ is $\{x \mid \beta\ \omega \stackrel{*}{\Rightarrow} x\}$, and its k-language is $\text{FIRST}_k(\beta\ \omega)$.

Definition The relation "immediate descendant" on LR(k)-items is defined by $B \rightarrow .\gamma(\omega)$ is an immediate descendant of $A \rightarrow \alpha.B\ \beta\ (\tau)$ if $\omega \in \text{FIRST}_k(\beta\ \tau)$. The relation "descendant" is the reflexive transitive closure of "immediate descendant." Item I_1 is an ancestor of I_2 iff I_2 is a descendant of I_1.

Definition The completion of an item I is the set of all items which are descendants of I. The completion of a set of items is the union of the completions of the individual items.

This concept has been called the closure by Aho and Ullman [1].

Definition The completion of the nonterminal A is the completion of the set

of non-essential A-items. The completion of A in the context ω is the completion of all non-essential A-items whose context is ω (i.e., all items of the form $A \rightarrow .\alpha(\omega)$).

Before giving the LR(k) parsing algorithm, we make a comment about endmarkers. We shall assume that every input string we shall be considering is followed by $\dashv^k$, k copies of the right padding symbol. This is to ensure that in looking ahead k symbols into the input, any parser will always have at least k symbols to look at; and that $FIRST_k$ and $FOLLOW_k$ are well-defined.

Our model of an LR(k) parser for G is driven by a finite-state machine-like device called the canonical LR(k) machine for G. This machine consists of a set of non-final states (described below) and a set of final states, one for each rule of G. The entry of the machine to the final state for $A \rightarrow \alpha$ during the course of a parse of x, indicates that α has just been located as the handle, and it is to be reduced to A. In this way, the sequence of final states of the machine entered during a parse provides the sequence of rules of G that form the bottom-up parse of x in G.

The states of an LR(k) machine are connected by a transition function; this function f takes a non-final state q, a symbol σ, and a lookahead of k symbols ω, into a state (final or non-final). At any point during the parse, some state of the machine will be current, the parser will have some symbol in hand, and there will be a lookahead consisting of the next k input symbols that have not yet been read; the transition function will compute from these three arguments, what is to be the next current state of the machine.

But the LR(k) parser does more than just go through states of the LR(k) machine; it also has some side effects on a stack used during the parse. This

stack contains symbols of the grammar alternating with names of states of the machine; at any point, the topmost stack element will be a state name, that of the current state of the machine. If this is a non-final state, then the parser reads the next symbol of input onto the top of the stack; the next state is computed from the current state, the symbol just read, and the next k symbols of input. This next state name is put on top of the stack, and the process continues. If the current state of the machine is a final state, then it is associated with some rule of G, say $A \rightarrow \alpha$. Then it will be the case that the topmost section of the stack consists of the symbols of α alternating with state names. The parser then proceeds to pop off the stack all these symbols, down through the first symbol of α. This exposes some state name q as the stack top. Then A is put on top of q, and q' is put on top of A, where q' is computed from q, A, and the next k symbols of input. This is the next configuration of the parser, and the process continues.

The LR(k) parser begins with the stack containing just q_0, which is the name of the starting state of the machine, and with the remaining input being $x\dashv^k$, where x is the string to be parsed. The parser recognizes x if it eventually reaches a configuration where the stack contains just q_0 S and the remaining input is $\dashv^k$.

We see then that an LR(k) parser consists of a standard driver, which uses the particular LR(k) machine for G to find the bottom-up parse for strings in L(G). The non-final states of the LR(k) machine for G are each comprised of a set of LR(k) items over G. The states of this machine and the transition function connecting them are defined in the following way.

1) The completion of S in the context $\dashv^k$ is the starting state of the machine.

2) Let q be any non-final state of the machine. For any symbol σ, consider the items of q of the form $A \rightarrow \alpha.\sigma\beta(\omega)$ where $\beta \neq \epsilon$. Consider the set E_σ of corresponding items $A \rightarrow \alpha\sigma.\beta(\omega)$. Then there is a non-final state q' which is the completion of the set E_σ. And we set $f(q, \sigma, x) = q'$ if there is some item $A \rightarrow \alpha.\sigma\beta(\omega)$ in q such that $x \in FIRST_k(\beta\omega)$; q' is called a σ-successor of q.

3) If $A \rightarrow \alpha.\sigma(\omega)$ is an item of q, then $f(q, \sigma, \omega)$ is the final state associated with $A \rightarrow \alpha\sigma$.

Repeated application of this definition eventually stops creating new non-final states, and the result is the LR(k) machine for the grammar. Observe that the set E_σ of step 2) forms the set of essential items of the new state; so every non-final state of the machine is precisely equal to the completion of its essential items.

It is well known that it is possible to compute $FIRST_k(\alpha)$ for any LR(k) grammar. Therefore it is possible to compute the completion of any set of items, and hence the definition of LR(k) machine just given can be used to compute the machine.

Furthermore, it is possible to prove the following two results.

Fact If q is any non-final state of the LR(k) machine for an LR(k) grammar G, then for any $a \in V_T^k$, at most one of $f(q, \epsilon, a\omega/k)$ and $f(q, a, \omega)$ is defined.

Fact If q is any non-final state of the LR(k) machine for an LR(k) grammar G, then for any $\sigma \in V_N \cup V_T$ and $\omega \in V_T^k$, $f(q, \sigma, \omega)$ is single-valued.

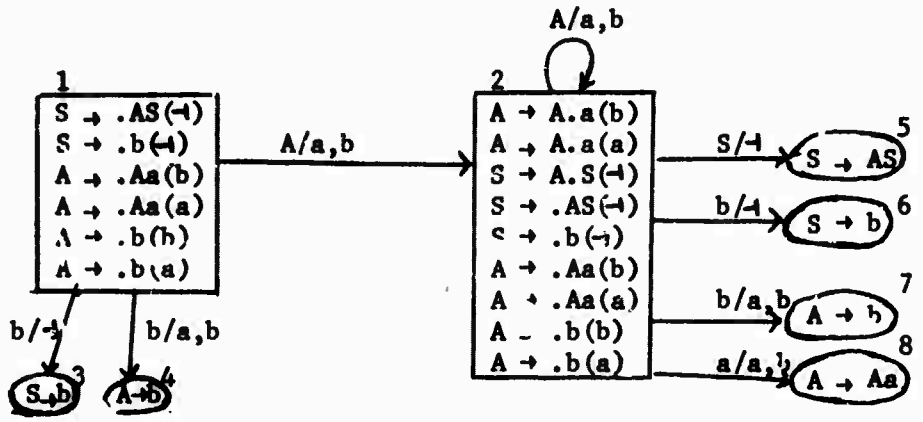
These facts assure us that the parsing algorithm we outlined previously is indeed sensible and can be formalized, and that the parsing algorithm operates deterministically. It can also be shown that this algorithm, using

this machine, does indeed successfully perform the left-to-right bottom-up parse of strings in L(G). That is, it recognizes just the strings in L(G), and the sequence of final states it enters while parsing x defines the bottom-up parse of x in G.

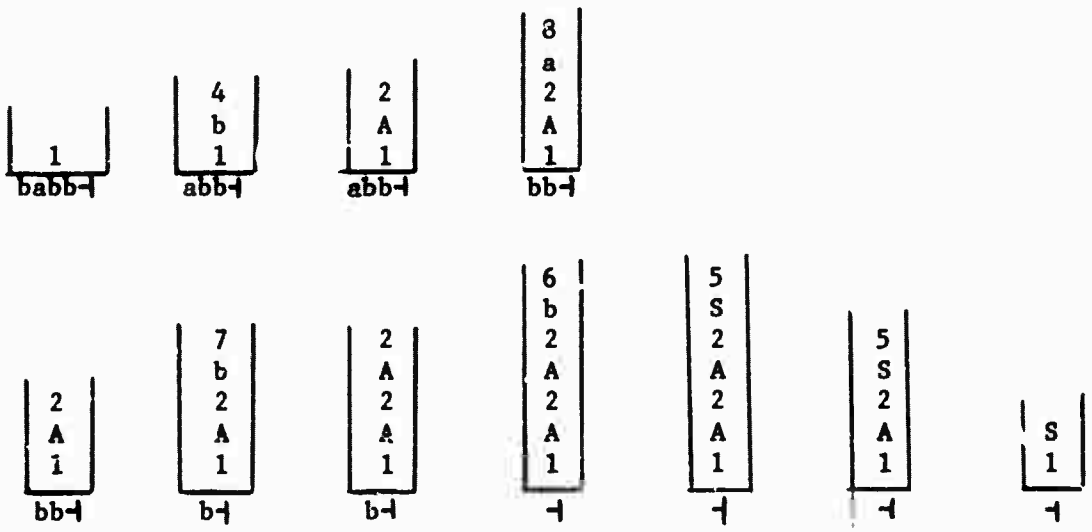
There are several refined models of the LR(k) parsing machine which can often be used to guide the parsing, but the machine construction detailed above produces the "canonical" LR(k) parsing machine for G.

We shall represent LR(k) machines as follows. A non-final state is denoted by a rectangle containing its defining set of items; a final state is an oval containing the name of the rule with which it is associated. If $f(q, \sigma, \omega) = q'$, then an arrow is drawn from q to q' labelled by σ/ω; if $f(q, \sigma, \omega') = q'$ as well, the transition is labelled by $\sigma/\omega,\omega'$.

As an example, the canonical LR(1) machine for the grammar $S \to AS$, $S \to b$, $A \to Aa$, $A \to b$, is given in the figure below. The states are numbered purely for convenience. State 1 is the initial state of the machine and state 2 is the only other non-final state.



The successive stages of the parse of babb using this machine is given below. We show the sequence of stacks, with the remaining input at each stage beneath it. This sequence demonstrates that babb



is indeed in the language generated by the grammar, and the sequence of final states entered does correctly give the bottom-up parse of this string.

CHAPTER 3

PREDICTION IN BOTTOM-UP PARSING: STATE-SPLITTING AND MULTIPLE STACK PARSING

3.1 Basic Ideas of Prediction and State-Splitting

Prediction is a concept usually associated with top-down parsing, rather than bottom-up. The entire process of top-down parsing consists of nothing more than making a sequence of predictions and then seeing them come true. Given a nonterminal and a lookahead string, we predict which rule of the grammar must be applied to the nonterminal in order to effect the eventual generation of the lookahead string. After matching leading terminals on the right-hand side of the rule with an initial segment of the lookahead, we reapply the procedure to the first nonterminal on this right-hand side. Thus we are always predicting the occurrence of a nonterminal in a specific location in the parse tree, as an ancestor of some segment of the sentence being parsed, before we know exactly how that segment is generated from the nonterminal.

On the surface, this concept seems completely alien to the approach embodied in bottom-up parsing. There, nothing whatsoever is predicted, no anticipation of the future ever plays a role in the progress of a parse. It is only when we have reached the end of the handle that we recognize it as such and discover to which nonterminal it is to be reduced. We do not make use of information that we have gained in the past in order to restrict future activities. But in one important sense, there is an air of prediction

to the entire proceedings. Before the bottom-up parse of a sentence begins, we are effectively predicting that we shall reduce it to S, the sentence symbol. The entire parse is predicated on this assumption; and when the prediction is fulfilled, and an S has been located by successive reductions of the input, then the parse is complete. We will generalize this approach and utilize prediction not only at the beginning of a bottom-up parse, but at selected points throughout. That is, at various times we shall predict that the bottom-up parser, proceeding in a normal fashion, will eventually reduce some prefix of the remaining input to a specified nonterminal. Putting it in terms of the LR(k) machine directing a parse, we shall see how to utilize predictive techniques not just at the initial state of the machine (where S is effectively being predicted), but upon entry to other states in the machine.

Before we develop the formal theory of this method of parsing, some informal discussion is in order, to describe the model underlying the development and provide some intuition as to the issues involved and the problems that will arise. The first thing we must understand is how a state of an LR(k) machine is used during the parse of a sentence by that machine. For the time being, we shall be restricting our discussion to so-called "canonical" LR(k) machines, but we shall see later how our work applies to the more efficient parsers of DeRemer, and others.

Suppose then that we have just entered some non-final state q of an LR(k) machine during the parse of a sentence by the machine, and that at this time the remaining input string is $\omega = a_1 a_2 \ldots a_m$. The next step in the parse will be to read a_1 onto the stack and transfer to some state q_1. The symbol a_1 will be the post-dot component of some of the items of q. After further symbols

have been read and possibly some reductions made along the way, the machine will enter the final state associated with one of these $.a_1$ items. Let us assume that this item of q is non-essential, of the form $A_1 \rightarrow .a_1x_1(\tau_1)$. Then we have reduced $a_2 \ldots a_i$ for some i to x_1, and will now reduce $a_1 \ldots a_i$ to A_1 (with the lookahead being τ_1). The machine will pop the stack exposing q, and will then follow an A_1-transition to the state q_2. Again, A_1 will be the post-dot symbol of some items of q, and eventually, we will arrive at the final state associated with one of these items, again assumed to be of the form $A_2 \rightarrow .A_1x_2(\tau_2)$. At this point, we have reduced $a_{i+1} \ldots a_j$ for some j to x_2, and are about to reduce A_1x_2 to A_2, the lookahead being τ_2. We once again return to q and follow an A_2 transition, eventually performing the reduction $A_3 \rightarrow A_2x_3$, with lookahead τ_3. This process continues until we perform a reduction associated with an essential item of q. That is, until we reach the final state for $B \rightarrow \beta A_n x_{n+1}$, where $B \rightarrow \beta.A_n x_{n+1}(\tau_{n+1})$ was an essential item of q. At this point, we will perform the appropriate reduction but will not return to q but rather to a state of which q was a β-successor.

Assume that the remaining input string at the time the reduction to A_n is made is $a_{\ell+1} \ldots a_m$; that is, by that time we have read $a_1 \ldots a_\ell$. Let us see what has been accomplished since our original entry to q. We have constructed the section of the parse tree shown in Figure 3.1. First we reduced some initial segment of ω to A_1; then A_1 and another segment to A_2; and so on, until we had reduced $a_1 \ldots a_\ell$ to A_n, the post-dot component of an essential item of q. After the recognition of each A_i, we returned to q to find the next state to go to; our final return was made after the finding of A_n. In some sense, our mission upon entry to q, was to locate the post-dot component of one of its essential items. Once this had been accomplished,

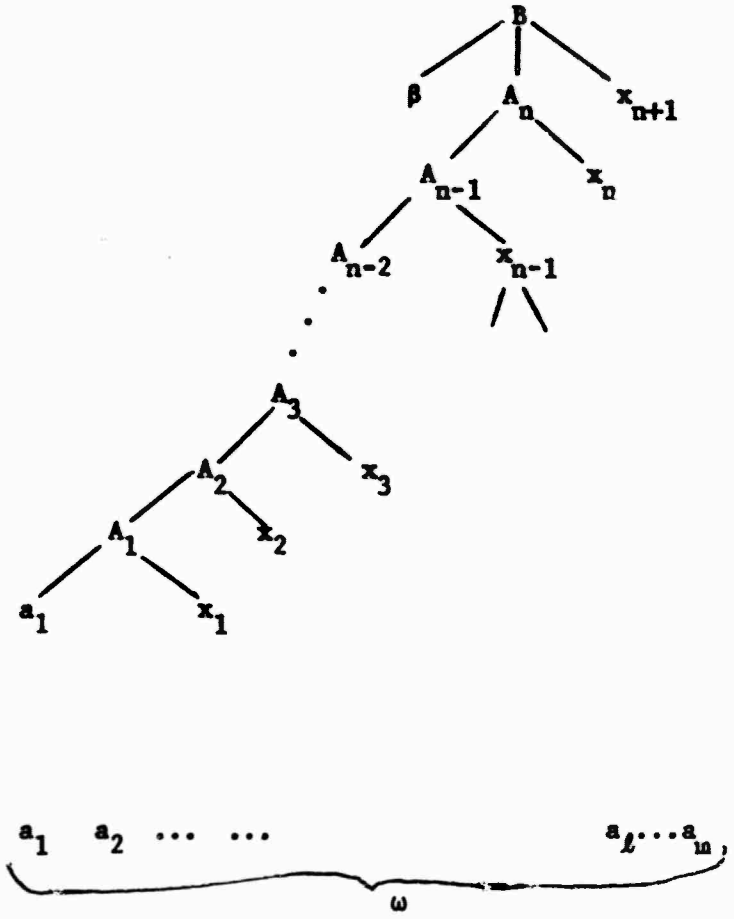


Figure 3.1

we no longer continued to return to q to compute the next state to go to. This task was performed by recognizing a sequence of non-essential items of q, the recognition of one being the first step in the recognition of the next. Each of these recognized nonterminals is an ancestor of a progressively longer initial segment of ω, until all of $a_1 \ldots a_\ell$ has been reduced to A_n.

But there is a further relationship among the rules $A_{i+1} \rightarrow A_i\ x_{i+1}$. Recall that each was underlying the nonessential item $A_{i+1} \rightarrow .A_i\ x_{i+1}(\tau_{i+1})$ of q. By the order in which the reductions were made, it is easy to see that $A_i \rightarrow .A_{i-1}\ x_i(\tau_i)$ is an immediate descendant of $A_{i+1} \rightarrow .A_i x_{i+1}\ (\tau_{i+1})$ in the state q. In summary, then, we know upon entry to the state q, that some initial segment of the remaining input string will be reduced to the post-dot component of one of the essential items of q; and that this reduction will be accomplished by the successive reductions of nonessential items of q, each of which is an immediate descendant of the next one.

How can we use in advance, this information as to what of necessity is going to occur once we enter q? Suppose that all the essential items of q have the same post-dot component, namely the nonterminal A. Then no matter which of these items the parser is really "working on", no matter which of the underlying productions will ultimately be recognized, we know for certain that before we leave q for the last time, we will have reduced some initial segment of the remaining input to an A. This then is the function of state q: to find an A. We can <u>predict</u>, upon entry to q, that an A is going to be found; and once that A has been found, we can leave q behind.

To take a more complex example, consider the LR(0) state q of Figure 3.2. Here there are two essential items, with different post-dot components, so

seemingly neither one could be predicted if we were to enter q during a parse.

$A \rightarrow x.By_1$
$C \rightarrow x.Dy_2$
$B \rightarrow .Ey_3$
$D \rightarrow .Ey_4$
$E \rightarrow .Fy_5$
$E \rightarrow .Gy_6$
$F \rightarrow .y_7$
$G \rightarrow .y_8$

Figure 3.2

But even though neither B nor D can be accurately predicted upon entry to q, it is possible to predict that some initial segment of the remaining input will be reduced to an E. For whether the ultimate reduction is to B or D, a prior reduction must be to an E, since the only items with B and D on the left-hand side have E as their post-dot component. We can not predict how the remaining parse will begin: it might start with $G \rightarrow y_8$ or $F \rightarrow y_7$; nor can we predict the final nonterminal to be found by q: it could be B or D. But somewhere along the way, the parser will find an E, pop the stack to expose q, and then go to an E-successor of q. Thus we see that in certain situations (to be characterized later), it is possible to predict and identify a specific nonterminal which will be recongized at some future time in the parse.

Let it be stressed that this is a prediction about bottom-up parsing, that is, though we predict that an E will be found, the way in which that E will be constructed will be by the further actions of the standard bottom-up LR(0) parser.

Is there any way to utilize this predicted information? Let us use a programming analogy. An LR(k) machine as a whole can be thought of as a main program, with the stack as its working space. Its goal is to find an S, to reduce the input string to the sentence symbol. Thus entry to the initial state of the machine is analogous to calling the main routine. Now suppose it were possible to predict an A in state q of the machine. Then entering state q would be similar to calling a subroutine whose goal would not be the reduction of the entire input to an S, but that of reducing some initial segment of the remaining input to an A. Upon completion of the task assigned it, the subroutine would return control to the main program at the point from which it was called; after that, the main program could carry on as if it itself had found the A in the normal ex post facto fashion, rather than having predicted its occurrence and dispatched the task of finding it to the subroutine. The subroutine itself would operate in much the same way as the main program, performing a bottom-up parse; except that it knows to quit when it has found an A; and let us further say that it has its own work space, its own stack. Note that the subroutine returns no value or information to the calling routine, but merely signals its completion. Thus the order of parse remains unaffected by the added feature of predicting in advance some of the results of the bottom-up parse.

But to complete the analogy, how do we design the subroutine to recognize the A? If the full LR(k) machine is the main program, what is the subprogram?

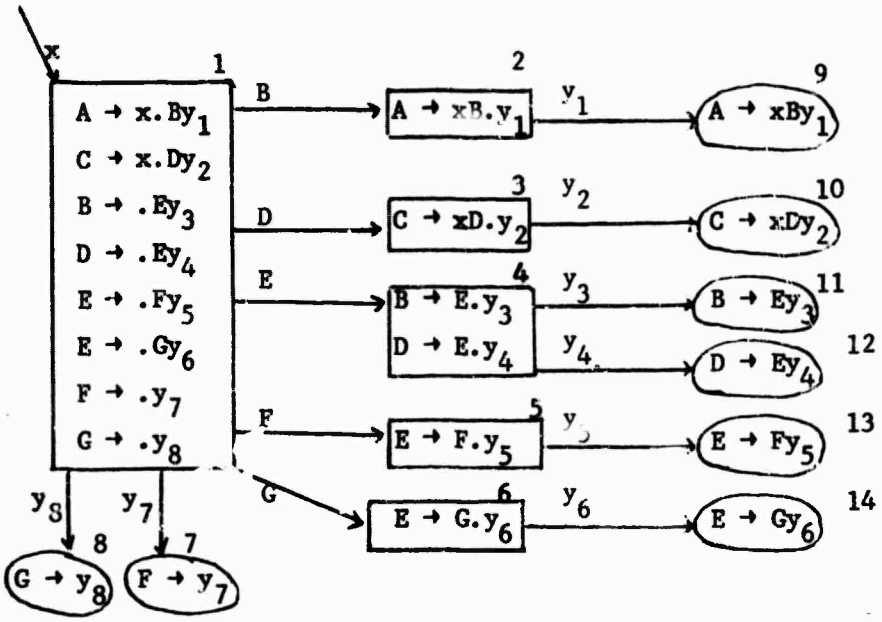
Let us say it will be a smaller machine, in some sense a submachine of the original one. Its construction will be much like that of any LR(k) machine; its states, sets of LR(k)-items. However its initial state, rather than being the completion of all S-items, will be the completion of all A-items. As a first approximation, we can think of the new machine being the LR(k)-parsing machine for the subgrammar of the original grammar which has A as its starting symbol. Successors of states in the submachine will be computed as usual, and the machine will operate like any other on its own stack. The only difference will be that there will be an A-transition from the starting state to a designated state entitled POP; and when POP is entered during a parse, it causes the submachine to erase its stack and return control to the main machine. Thus the submachine knows to suspend operation when its job is completed.

How is the submachine to be called from the main machine, and how does the existence of the new machine affect the design of the original one? We have seen that the function of the new machine is to find an A which can be predicted upon entrance to state q of the main machine. This then is the appropriate place for control to be transferred to the submachine: from state q, a special predictive transition will be made to the initial state of the submachine. This transition basically leaves the original stack unaltered. Control is returned to state q when the submachine is finished, and processing then continues normally by the main machine on its own stack.

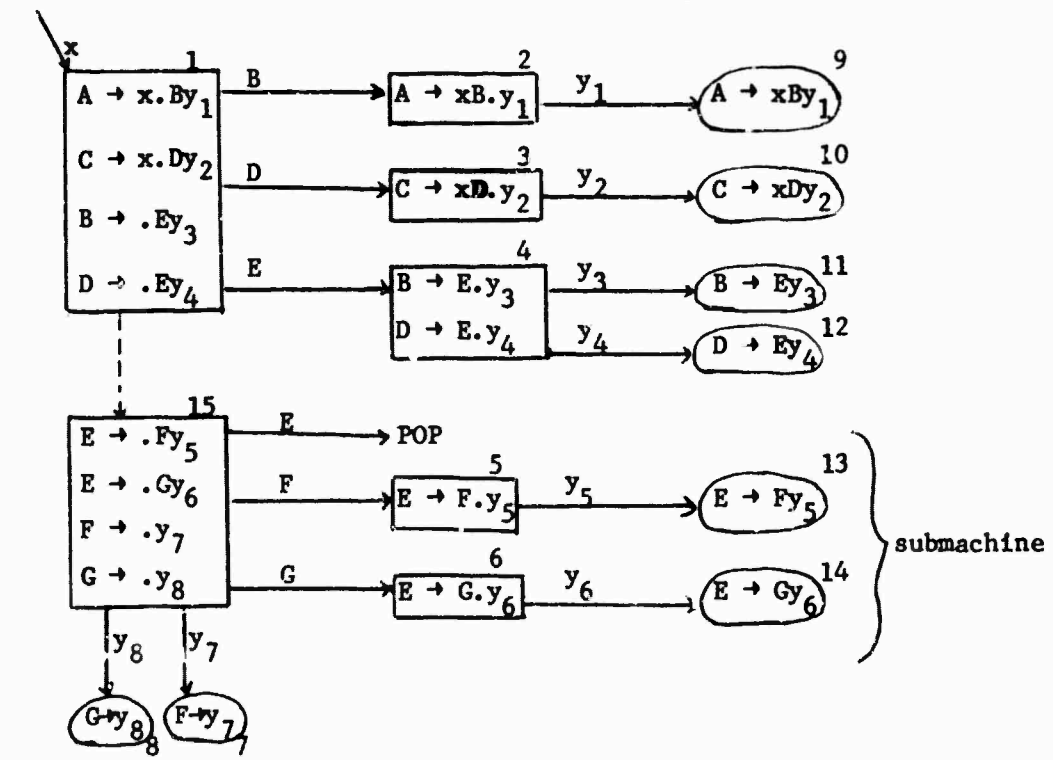
It is certainly reasonable to design the main machine and the submachine so that anything which is performed by the submachine is not also done by the main machine. In particular, since the responsibility of finding an A has been delegated to the submachine, state q no longer has to be concerned

with this. So all A-items and their descendants can be deleted from q, because they are used only to reduce some initial segment of the input to an A. Thus the designer of the machine can utilize the fact that the task for finding an A in q can be delegated to a submachine, by leaving out of q certain items that would otherwise be in it. We can look at this smaller version of q as really being a new state in the main machine. It seems that the items which can be deleted from q are precisely those items which go into the initial state of the submachine. Thus the state q is being split into two states: one which replaces q in the original machine, and one which is the starting state of the new submachine. The items in these two states comprise precisely the set of items in q.

Obviously, delation of some items from q requires the recomputation of the successors of q in the original machine, but this is done in the normal fashion. An example of the entire process is illustrated in Figure 3.3. There we have constructed part of an LR(0) machine containing the states of Figure 3.2 in which, as we argued previously, an E could be predicted. We also show what the revised machine looks like after the state has been "split". (Throughout, we have treated the y_i as though they were single terminal symbols; of course, in reality they could be strings of terminals and nonterminals.) The submachine to find an E has been constructed; transition is made to its initial state (called the predictive state) from a state which occupies the place of q in the main machine (called the base state of the splitting of q). Note that every state in the revised main machine and in the submachine is a subset of some state in the original machine. Further note that the two diagrams are



Before state-splitting



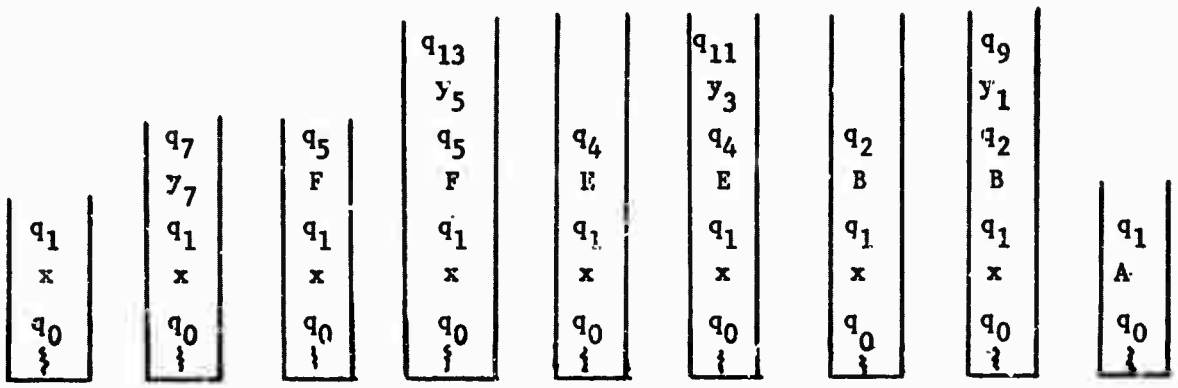
After state-splitting

Figure 3.3

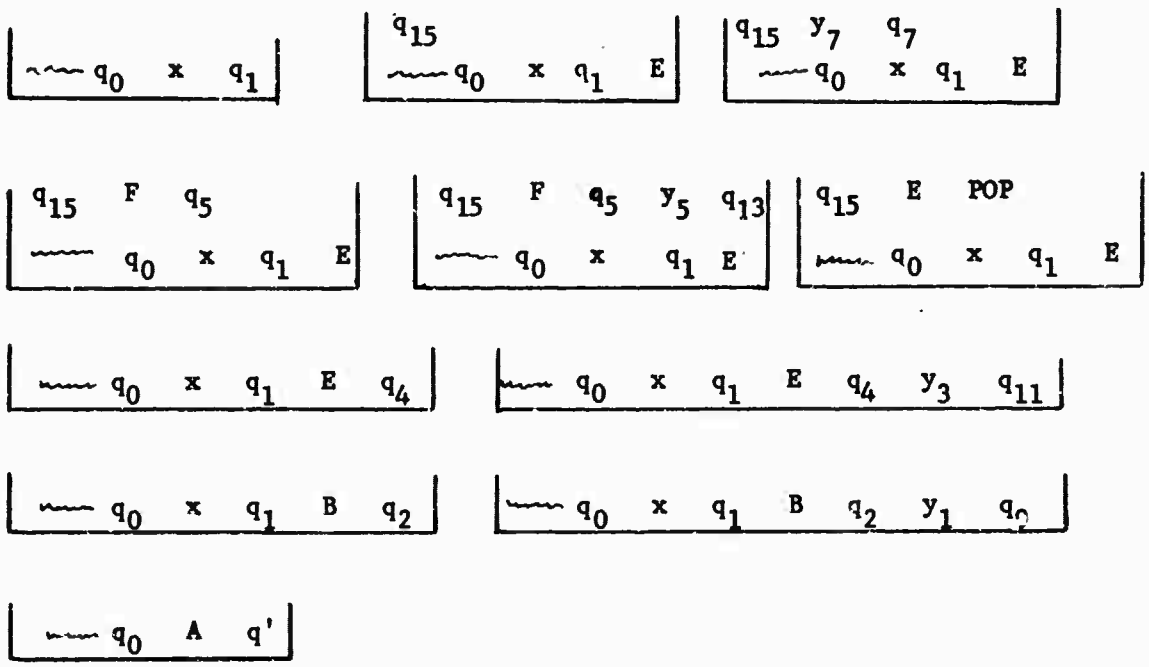
functionally equivalent as far as the rest of the machine is concerned; that is, entrance to q in the original machine and to the base state in the new machine will produce identical results.

In fact, it is easier to think of the new diagram not as being of two LR(0) machines, one of which can call the other, but rather as representing a generalized kind of parsing machine, in which it is possible for one state to transfer control to another part of the machine and get it back when that part has completed its predefined task. Each time a call transition is made, a new stack is created for the benefit of the called submachine. This submachine will use this new stack as its parsing stack, and will destroy it when control is returned to the calling state. Since (as we shall see) it is possible for submachine calls to be nested (i.e., for one submachine to call another, or itself recursively), it is more convenient to think of the machine at large as really having a stack of stacks: each time a call of a submachine is made, a new stack level is created, and all processing utilizes that level (which is itself a stack) until either another call is made or until the current call is completed and a return is effected. When a return is made, the topmost, most recently activated stack level is suspended, and processing continues on the exposed level.

It should be observed that the order of recognition by such a machine is still strictly left-to-right, bottom-up. All the extra work of making predictions and fulfilling them, of creating and suspending stack levels, as of yet makes no appreciable difference. A comparison is made in Figure 3.4 between a conventional LR parse and one utilizing this predictive capability. Actually, it is a portion of a parse, showing how the substring $xy_7y_5y_3y_1$ would be handled by the two machines of Figure 3.3, assuming that the x takes



Conventional LR(0) parse of $xy_7y_5y_3y_1$



Revised parse with prediction

Figure 3.4

us from some state q_0 into state q_1. (We do not know or care what is on the stack underneath the q_0, since it is unaffected by this segment of the parse.)

In the second part of the figure, each stack level is written horizontally; laying the stack levels end to end approximately reproduces the single stack of the first parse. In the revised parse, an E is predicted upon entry to state q_1; this causes an E to be written on the old stack level, and a new stack level to be created, which uses state q_{15} as its starting state. Thereafter parsing continues normally until the POP state is entered; then the top stack level is destroyed, and the machine resumes processing on the lower level, transferring to the E successor of the state from which the call had been made. Note that there are two more steps in the predictive parse, to account for the prediction and fulfillment steps, but otherwise the parses and stack configurations are essentially the same. In particular, the final states are entered in precisely the same order, thus preserving the order of recognition in the parse.

We want to emphasize at this point that state-splitting is not a process employed during the parse of a string by a machine, but during the construction of such a machine by its designer. That is, the designer notices that should state q be entered during a parse, an A is sure to be recognized; he realizes that something can thus be predicted upon entry to q. Therefore, he "splits" q into two states, and divides the machine into a main machine and a submachine. This machine will operate in such a way that when the base state of the splitting of q is entered during a parse, the machine "predicts" the recognition of an A, and transfers to the submachine to accomplish this recognition.

3.2 Prediction with Lookahead

There are numerous ramifications of these notions of state-splitting and prediction that must be considered. First of all, consider the LR(1) state of Figure 3.5. Initially, it would seem impossible, upon entry to this state during the course of a parse, to have any advance information about the nonterminals to be found by this state- that is, to be able to make a prediction. This is because there are two essential items in the state, which have different post-dot components; and these components do not have any common leftmost nonterminal descendant.

$A \rightarrow x.By_1(z_1)$
$C \rightarrow x.Dy_2(z_2)$
$B \rightarrow .a(y_1)$
$D \rightarrow .b(y_2)$

Figure 3.5

However, suppose that upon entry to this state we allowed ourselves to examine the remaining input string in order to assist us in predicting the future course of the parse. In general, we will allow ourselves to look ahead k-symbols into the input. Now in this case, k = 1; and the only two possible 1-lookaheads are a and b. And since $L_1(B) = \{a\}$ and $L_1(D) = \{b\}$, observation of the lookahead does give us enough information to effect a prediction. In particular, upon seeing a, we could predict B; and upon seeing b, we could predict D. This is because the first symbol of the remaining input string must be generated from the post-dot section of one of the essential items of the state. Thus the presence of an a indicates the existence of an A in the parse tree, while b does the same for D.

In examining this state and designing the machine into which it fits, we can easily take advantage of this analysis. We can "split' this state as we did earlier, but now make the predictive transition depend on the lookahead being examined. This is illustrated in Figure 3.6. Here we have

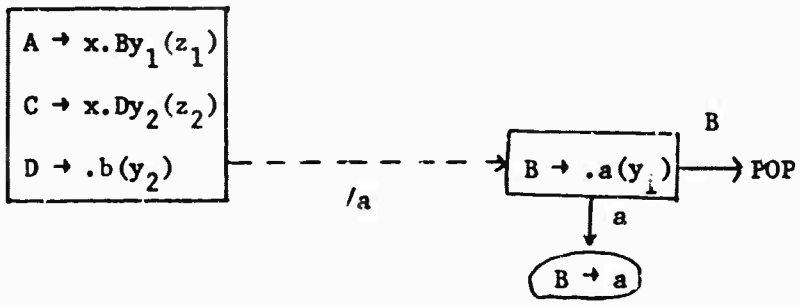


Figure 3.6

shown how this conditional prediction, or prediction on lookahead, could be represented. The notation /a means that if upon entry to the base state of the splitting, the first symbol of the remaining input string should be an a, then the machine would predict the occurrence of B and jump to the predictive state shown. Note that the a would not be read onto the stack by the base state prior to the prediction being made; rather, the a is examined in its place in the input, and left there to be read later by the predictive state. If upon entry to the base state, the lookahead should not be a, then no prediction would be made, and control would remain with the base state, which would proceed to read the next symbol onto its stack.

In the state-splitting of Figure 3.6, we have made no explicit utilization of the fact the appearance of b as the lookahead symbol enables us to predict the occurrence of D. We could have designed an alternate state-splitting, with the predictive transition on /b and the predictive state containing the item $D \rightarrow .b(y_2)$, to account for this case. Or we could split

the original state into three parts - a base and two predictive states - with a predictive transition from the former to each of the latter. This splitting is shown in Figure 3.7.

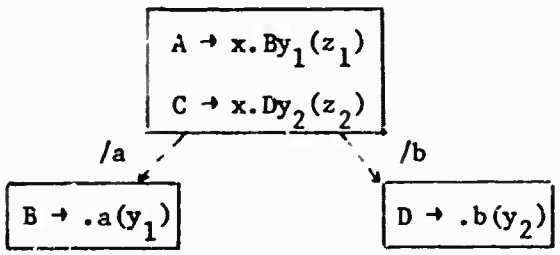


Figure 3.7

The meaning of this splitting is the most intuitive one. Should the parser enter this base state during a parse, the next input symbol would be inspected, and its identity would determine which nonterminal is to be predicted and to which state to transfer.

This then serves as our introduction to the notion of prediction on lookahead. Informally, we can say that a lookahead ω indicates the presence of nonterminal A in state q if upon entry to q, the occurrence of ω as the lookahead assures us that some initial segment of the remaining input will eventually be reduced to an A. That is, if all ways to generating a string beginning with ω, from the post-dot section of <u>any</u> essential item in q, must involve the generation of an A as the ancestor of some prefix of the string. If this is the case, then we know that at some time in the parse the machine will return to q having found an A; and so we can consider instead delegating the discovery of this A to a submachine. Let it be stressed that if a indicates an A in q, it does not tell us that A is the post-dot component of some essential item of q, nor give us any specific information as to the precise

location of this A in the tree; we can only be sure that an A will be there somewhere below the post-dot component of some essential item of q.

Before we decide how and when a nonterminal is to be predicted during a parse, we shall look more closely at the question of deciding when ω indicates A in q. First of all, we shall adopt the following convention: that only lookaheads of length k will be used to indicate nonterminals in LR(k) states. This is purely for convenience and uniformity, and does not impose any significant restrictions on our work. (If we wanted to use a lookahead of length k_1 with an LR(k_2)-state, where $k_1 > k_2$, we could always generate the corresponding LR(k_1)state; while if $k_2 > k_1$, we can pad out the lookahead to make it of length k_2.)

To see if ω indicates A in q, we should determine all ways of generating strings beginning with ω from all the essential items of q, and then ascertain whether or not A appears on the leftmost branch of each generation tree. When we say that we must consider all different ways of generating these strings, we only mean those ways that differ along the leftmost branch of their derivation trees, which is where A should appear. If two methods of generating strings beginning with ω are the same along the leftmost branch, but differ in the interior of the generation trees, then A appears along the leftmost branch of both or of neither; so we need consider only one of these two methods.

That is, suppose $C \to \alpha.B\beta(\tau)$ is any essential item of q and that $B\beta\tau \overset{*}{\Rightarrow} \rho$, $\rho \in V_T^*$ and $\omega = \rho/k$. Then we want to show that for every derivation $L\beta\tau \underset{L}{\Rightarrow} A_1x_1\beta\tau \underset{L}{\Rightarrow} A_2x_2x_1\beta\tau \underset{L}{\Rightarrow} \cdots \underset{L}{\Rightarrow} A_nx_nx_{n-1}\cdots x_2x_1\beta\tau$

$\underset{L}{\Rightarrow} x_{n+1}x_n\ldots x_1\beta\tau \overset{*}{\Rightarrow} \rho$, where x_{n+1} either begins with a terminal symbol or is ϵ, that $A_i = A$ for some i. We know that $C \to \alpha.B\beta(\tau)$ is an item of q, and that $B \to A_1x_1$ is a rule of the grammar. Then for every τ_1 such that $|\tau_1| = k$ and $\beta\tau \overset{*}{\Rightarrow} \tau_1\psi$, $B \to .A_1x_1(\tau_1)$ is also an item of q. In particular, we know that $B\beta\tau \overset{*}{\Rightarrow} \rho$; if we set ρ_1 as the part of ρ which B generates, and ρ_1' the part generated by $\beta\tau$, we can let $\tau_1 = \rho_1'/k$; and then we will have that $\omega \in FIRST_k(A_1x_1\tau_1)$. Similarly, every item $A_1 \to .A_2x_2(\tau_2)$ is an item of q, and a descendant of $B \to .A_1x_1(\tau_1)$, provided $\tau_2 \in FIRST_k(x_1\tau_1)$. In particular, since $A_1x_1\tau_1 \overset{*}{\Rightarrow} \rho_1\tau_1$, we can let τ_2 be the first k symbols of the section of $\rho_1\tau_1$ that are generated by $x_1\tau_1$. We can proceed on in this way, and we can be sure that there will be items of the form $A_{i-1} \to .A_ix_i(\tau_i)$ for each i, in q, where τ_i is the first k symbols of ρ generated by $x_{i-1}\ldots x_1\beta\tau$; and for each of these items, $\omega \in FIRST_k(A_ix_i\tau_i)$. Finally, there will be an item $A_n \to .x_{n+1}(\tau_{n+1})$ in q, with $\omega \in FIRST_k(x_{n+1}\tau_{n+1})$, where x_{n+1} is ϵ or begins with a terminal. Thus we see that for each different derivation $B \Rightarrow A_1x_1 \Rightarrow A_2x_2x_1 \Rightarrow \ldots \Rightarrow A_nx_n\ldots x_1 \Rightarrow x_{n+1}\ldots x_1$, such that $\omega \in FIRST_k(x_{n+1}x_n\ldots x_1\tau)$, there is a different sequence of items of q, $A_{i-1} \to .A_ix_i(\tau_i)$, such that each item in the sequence is a descendant of the previous one, and such $\omega \in FIRST_k(A_ix_i\tau_i)$ for each i.

Thus we see that we have all the information readily available in q, which we need in order to determine if ω indicates A in q. Namely, we will have to consider all sequences of items of q, $I_0, I_1, \ldots, I_n$, such that I_0 is an essential item, I_{i+1} is an immediate descendant of I_i, and I_n is $A_n \to .x_{n+1}(\tau_{n+1})$ where x_{n+1} is either ϵ or begins with a terminal, and such that $\omega \in FIRST_k(x_{n+1}\tau_{n+1})$; and then, with all such sequences in hand, determine whether or not there is an A-item in the interior of each of them. (There may be infinitely many such sequences, but we shall get around that shortly.)

Let us observe at this point that the problem of the presence of an A-item somewhere after the first item in a sequence is equivalent to the presence of a .A-item anywhere at all in the sequence, since in such a sequence, one item can be a .A-item if and only if the next item is an A-item. We are interested in the presence of an A-item somewhere after the essential item in each sequence we are considering; it will be more convenient for us to phrase our condition as a .A-item anywhere in the sequence.

Observe that this approach, in terms of these item sequences, works even in the case where A does not generate all of ω, but only some initial segment of it. If we were to predict an A upon seeing ω upon entry to q, the submachine delegated to finding the A would return before all of ω had been read onto the stack. Nonetheless, there still would be a .A-item in each of these item sequences.

As an example, consider the LR(2) state of Figure 3.8; we wish to know if ab indicates the presence of B in this state. In order to do this, we must identify all those items of the state which might give rise to a lookahead of ab upon entrance to the state; and then consider all sequences of such items, checking whether or not there is a .B-item in each of them.

X → c.B(xy)
X → c.A(xz)
A → .bc(xz)
B → .CE(xy)
B → .DF(xy)
C → .a(bx)
C → .a(dx)
D → .aA(fg)

Figure 3.8

(We assumed, in the construction of this state, that the rules for E in the grammar were E → b and E → d; and for F, F → fg.)

The items beginning with terminal symbols, and having ab among their lookaheads, are C → .a(bx), C → .a(dx), and D → .aA(fg). (In the last case, abc L_2(aA) since A → bc is a rule.) We then consider all appropriate sequences of items, beginning with an essential item and ending in one of these. There are two such sequences, and they are given in Figure 3.9.

X → c.B(xy)	X → c.B(xy)
B → .CE(xy)	B → .DF(xy)
C → .a(bx)	D → .aA(fg)

Figure 3.9

We see by inspection that there is indeed a.B-item in each of these sequences, so ab does indeed indicate B in this state. We also see that ab does not indicate any other nonterminal in this state, since there is not a .Z-item in each sequence for any other Z. If we examined the case for lookahead ad, we would have just the single sequence X → c.B(xy), B → .CE(xy), C → .a(dx). Then we see that ad indicates the presence of B in this state, but it also indicates the presence of C. In general, a single string may indicate the presence of several nonterminals in a given state.

Suppose then that ab indicated the presence of A in state q; how can we use this information to make predictions and split the state q? If we decided to split q by creating a predictive state for A, we would divide the items of q between the new predictive state and the replacement base state for q. Now the goal of this predictive state (and the submachine

which it heads) is to locate an A; so those items which will be used to parse to the level of an A belong in the predictive state. If the lookahead upon entry to q is ω, then we know that an A will eventually be found in q. Any sequence of reductions made in q defines a sequence of items, each of which is a descendant of the following one; in the case of lookahead ω, there will be a .A-item in each item-sequence which q will be going through. So we can "break" each of these sequences just below the .A-item; the items after the .A-item will go into the predictive state, and will be used there to parse the input up to the level of an A. After that, the base state will resume control and use its items to reduce the input up to the appropriate essential item. Transfer between the base and the predictive state will be made on lookahead ω, and the return will be effected once the A has been located.

We now have to ask the crucial question - under what circumstances will we decide to split a state to reflect the prediction of an A, and how will we decide precisely which items are to go into the base and which into the predictive state? The above discussion showed us how, if ω indicates an A in q, to divide up q to handle the processing of string beginning with ω. But when will we avail ourselves of this technique? For the major part of our development, we shall stipulate that q is to be split by creating a predictive state for A if and only if the following condition holds: if ω is the lookahead upon entrance to q when an A will eventually be found in q, then ω must indicate A in q. That is, we will only split a state by means of a predictive state for A if examination of the lookahead upon entry to q will always enable us to detect the presence of an A. To put this condition in other terms: if $B \rightarrow \alpha.A\beta(\tau)$ is an item of q, and $\omega \in L_k(A\beta\tau)$, then ω must indicate A in q for us to try to split q into an A-predictive

state. It is insufficient for A to be indicated by some of the lookaheads it generates, but not by others; the lookahead must always be able to tell us whether or not q is going to find an A. If this is the case, then we will create a submachine, headed by an A-predictive state, to find this A, and devise the state-splitting of q accordingly.

For example, consider again the state of Figure 3.8. The only lookaheads that can exist upon entry to this state, if B is to be eventually found there, are ab and ad; and as we have seen, both of these strings indicate the presence of B. Therefore, we can consider splitting this state in such a way that B gets predicted if the base is entered with the lookahead being ab or ad. But now consider the possibility of predicting a C. The same two strings, ab and ad, are the only possible lookaheads if C is to be found; but while ad does indicate the presence of C, ab does not, as we have seen. Thus C is not a candidate for directing a splitting of that state. In doing conditional predictions, we insist that the lookahead always tell us if the nonterminal in question will be found or not, not just some of the time. Thus we do not consider creating a predictive state for C and jumping to it from the base when the lookahead is ad.

Let us suppose then that this condition is met, that the lookahead always indicates A in q, if it is there. (As is true for B in the state of Figure 3.8.) Precisely how is q to be split? As we discussed above, the predictive state will be transferred to from the base state on any lookahead ω which indicates the presence of an A, and it will parse the string until it finds an A. Then it will return control to the base state, which will finish doing the job that q was supposed to have done. Thus for each such ω, we consider all the

item-sequences defined earlier; in each of these, by hypothesis, there will be a .A-item. The items in the sequence from the essential item at the beginning, down through the .A-item, will go into the base state; while the rest of the sequence, starting with the A-item, go into the predictive state. We create predictive transitions from the base to the predictive state on each ω, and a return transition from the predictive state to POP on A.

But there is more that needs to go into the base state. In general there will be lookaheads that do not occassion an A to be predicted, that will not result in an A being found. Inputs beginning with such lookaheads must be processed in toto by the base state, and so all items that may be used in such processing must be left in the base state. These are all items which may be used in processing any lookaheads other than the ω's that indicate A; that is, items $X \rightarrow \alpha.\ Y\beta(\tau)$, where $\omega' \in L_k(Y\beta\tau)$ and ω' does not indicate A in q.

The splitting of the state of Figure 3.8 is shown in Figure 3.10. The base state is on the left and the predictive state on the right. The items in the base state are those in the "upper" part of the item-sequences used to process strings beginning with ab or ad (namely the item $X \rightarrow c.B(xy)$), and those used to process strings that do not begin with either of these lookaheads ($X \rightarrow c.A(xz)$ and $A \rightarrow .bc(xz)$). The predictive state, meanwhile, contains those items used until the discovery of B.

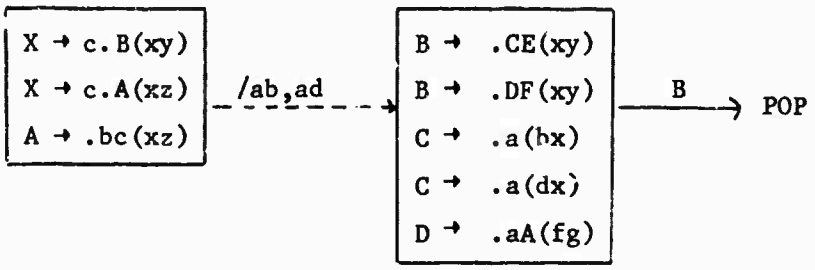


Figure 3.10

It appears that the items in the predictive state for A are just the items that form the completion of A; i.e., all nonessential A-items of q and all their descendants. This view is not entirely accurate, as we shall see shortly. However, one feature of it is correct; namely, that the essential items of q all remain in the base state, and none are included in the predictive state.

In general, there may be several candidates for prediction in a given state q, and a different state-splitting for each choice. For example, consider the LR(2) state of Figure 3.11, which is a substate of the state of Figure 3.8. There are several nonterminals whose prediction is legal in this state and which can induce a state-splitting of it. For example, B is indicated by the lookeaheads ab and ad, as in the earlier state;

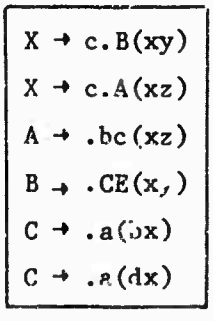


Figure 3.11

but on the other hand, C is also indicated by these same lookahead strings. That i., if either of these strings is sighted on entrance to the state, both a C and a B will be found by this state. And finally, the lookahead bc indicates an A; and since it is the only string that can come from an A, this means we could predict an A. The alternative state-splittings induced by these different predictions are given in Figure 3.12.

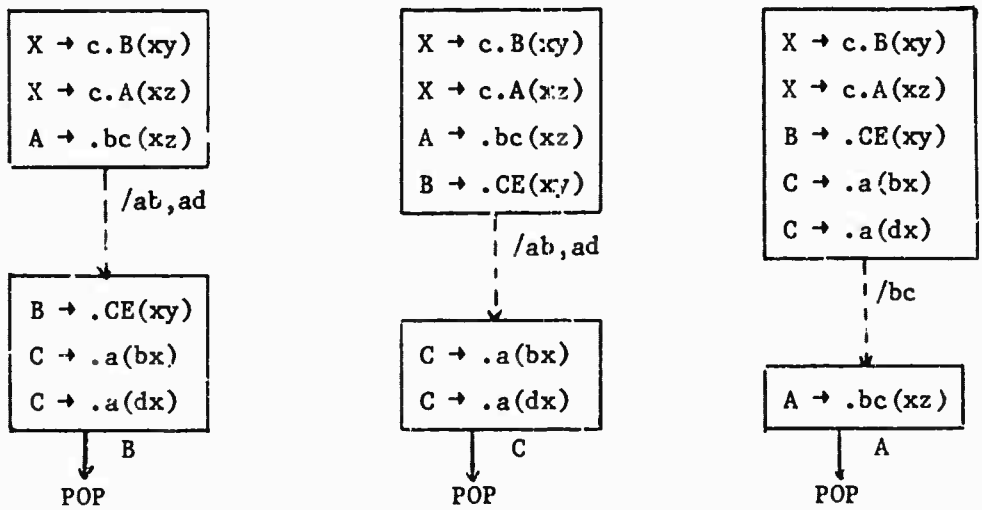


Figure 3.12

Each one of these splittings is valid, and at this time we shall not attempt to assess their relative merits. A machine designer might choose any one of them; later we shall describe some criteria for deciding what a valuable or optimal splitting might be.

Let us examine the implications of Figure 3.12. This indicates that upon entry to q, on lookahead ab or ad, one could predict the occurrence of B, while on lookahead bc, an A could be predicted. Since these two "predictive languages" are disjoint, we should be able to combine these pieces of information, and create a splitting of q with several predictive

states. Here the lookahead tells us not only whether or not to predict, but what to predict. The predictive and base states are constructed by the same principles as before: the item-sequences for the A lookaheads are divided at the .A item, one part going into the base state and the other into the A-predictive state; and similarly for the B lookaheads. And those items with other lookaheads automatically remain in the base state. Figure 3.13 shows a state-splitting involving predictive states for A and for B.

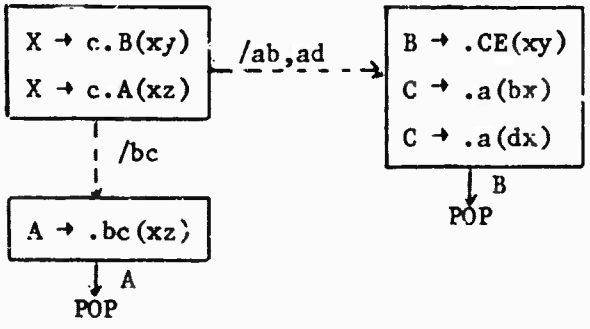


Figure 3.13

It would not be possible to split the state of Figure 3.11 into a base and predictive states for B and C, since the lookaheads that indicate the presence of these two nonterminals and hence cause their prediction, overlap (in fact they are identical). It is possible to have multiple predictive states, associated with different nonterminals, if and only if the lookaheads associated with these various predictions are mutually disjoint.

One other potential problem with our notion of predictive state-splitting needs to be considered. What if A, the nonterminal that is to be predicted, is left recursive? In that case, there may be several A's along the leftmost branch of the tree above the lookahead. Which of these is the one that is being predicted? How will the submachine that is supposed to find an A know

when its task is completed? Should it return as soon as it finds an A, or keep going until it finds a higher level A? Earlier, in describing how q was to be divided into base and predictive states, we said that each relevant item-sequence of q was to be "broken" into two segments, the dividing line being given by the .A-item in the sequence; but if A is left recursive, there may be several .A-items in such a sequence. Where do we choose to break the sequence--how much should we consign to the predictive and how much to the base? We have said that there is to be an A-transition from the predictive state to the POP state; but if there is some .A-item in the predictive state, there will have to be some A-transition to a non-POP state as well. How is the machine to know which transition to follow?

All these questions are aspects of the same problem--namely, if A is left recursive, what does it mean to predict an A? We could give an arbitrary answer, and say that it means to predict the first A that will be found (the lowest in the tree). Then we would know how to break the item-sequences (at the lowest .A-item), and there would never be any .A-items in the predictive state. But this simple solution is much too restrictive for a variety of reasons.

Rather than try to proscribe the kinds of predictions that may be made, we put no limits on it at all, save the restriction that the submachine must be able to determine when it has found the A that it has been dispatched to find. What information can it use to make this determination, and what test shall it perform? We stipulate that the sub-machine can inspect the lookahead <u>after</u> it has found an A, and use the nature of this lookahead to determine whether or not it has completed the job it was sent out to do.

In order for this to be effective, there will have to be two classes of A's: one is the kind of A which is predicted; and the other is the kind of A which is discovered while the predicted A is being constructed. These two classes of A's must have distinct follow sets, so that the membership of an A found by the submachine, with respect to these classes, can be determined by examination of the lookahead after the A has been found. This division of A's into two classes is not a function of the state q itself, but rather of a particular splitting of it. We are no longer giving rules as to how a splitting is to be constructed; we are specifying an additional constraint which any potential candidate for a splitting must satisfy. Given a tentative division of q into a base state and an A-predictive state, we will determine whether or not it conforms to this other requirement. For a given division of q into a base and an A-predictive state, the "predicted A's" are precisely the post-dot A's of the base state, while the "lower-level A's" are those that are in a post-dot position in the predictive state. The latter A's ought to be recognized before a transition is made back to the base state; while the last A to be found by the predictive state, the one that ought to take the machine into the POP state, will turn out, after the return is made, to be one of the post-dot A's in the base state. Thus, to determine if a particular splitting is valid, we should compute f_1, the set of k-length terminal strings that can follow post-dot A's in the base state; and f_2, similarly for the A-predictive state; and then ascertain whether or not $f_1 \cap f_2 = \phi$. If this is satisfied, then A/f_1 will label a transition from the predictive state to POP, while A/f_2 goes to the appropriate A-successor of the predictive state.

For example, consider the very simple LR(1) state of Figure 3.14. Since a is the only lookahead that comes from an A, and a does indicate A in this state, it is possible to split this state into a base and an A-predictive state. However in trying to see how the items should be apportioned between these two, we discover there are several different ways of doing it. For example, let us consider the item-sequence B → x.Ab(y), A → .Ab(b), A → .Ab(b), A → .a(b). If we chose to break this sequence after the first .A item, B → x.Ab(y) will be in the base state, the other items in the predictive state; if instead, we chose to break this sequence after the second .A item, the items B → x.Ab(y) and A → .Ab(b) would be in the base state, with A → .Ab(b) and A → .a(b) in the predictive state (there being nothing a priori wrong with having the same item in both states); while if we chose

B → x.Ab(y)
A → .Ab(b)
A → .a(b)

Figure 3.14

to divide this sequence after the last .A-item in it, this would consign B → x.Ab(y) and A → .Ab(b) to the base state, with A → .a(b) in the predictive state. Now of course, to fully determine the constituents of the components of the splitting, we have to break each item-sequence. But it is not hard to see that we could make our breaking choices for all other sequences consistent with any one of the three we have just described for this sequence; consistent in the sense that the other breakings would not put any items into the base or the predictive, other than those assigned

there by our breaking of this particular sequence. Thus we would tentatively have three different splittings of the state, each with an A-predictive state. These three alternatives are shown in Figure 3.15. However, two of these divisions violate our principles of distinguishing predicted A's from ancillary A's. Specifically, in the first two tentative splittings, b is in the 1-follow set of A's in the base state, as well as of A's in the predictive state. Hence, only the third alternative, the one without any .A items in the predictive state, is a legal splitting of the state.

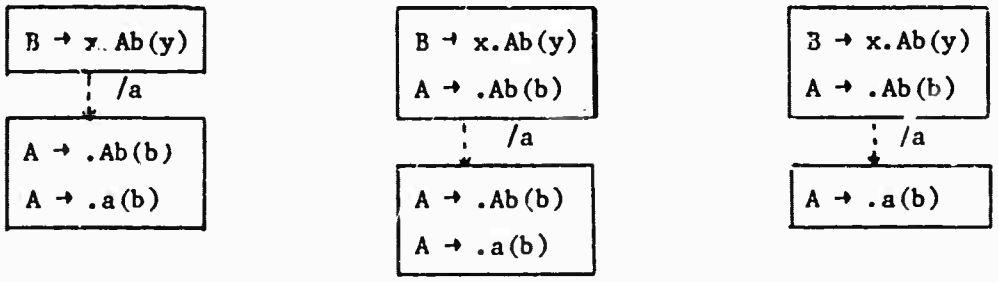


Figure 3.15

This does not mean there can be only one legal splitting of a state for a given nonterminal. The state of Figure 3.16 has two valid splittings,

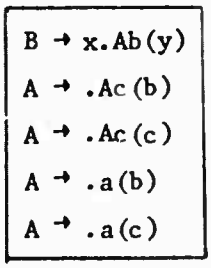


Figure 3.16

shown in Figure 3.17. We also show how the predictive states behave once an A

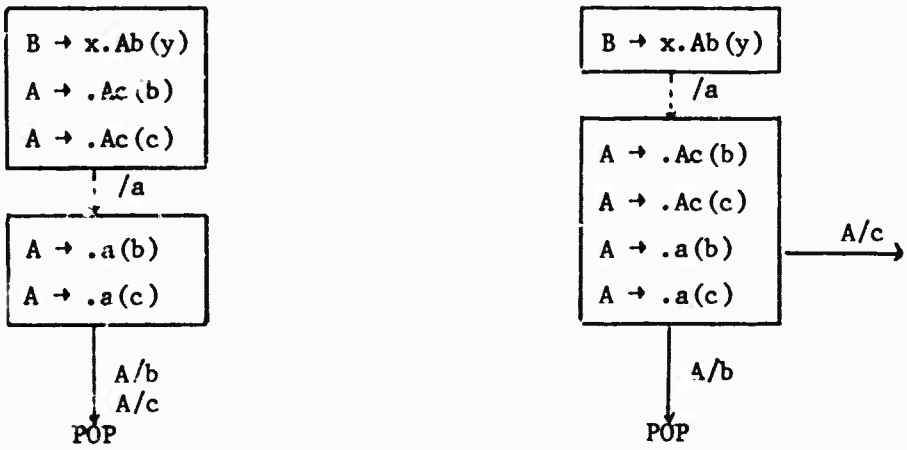


Figure 3.17

has been found. The first one returns to the base state no matter what, while the second does so only if the lookahead is b; otherwise it transfers to some other state in the submachine. The first splitting is designed so that the submachine is to return after finding the first A it encounters; the second, on the other hand, returns only after climbing all the way up the chain of left recursive A's. Both are legal splittings, and our results will apply to both of them.

This, then, is why it is inappropriate to think of an A-predictive state of a splitting of q as containing the completion of all the A-items of q. This is true when A is not left recursive, and there is only one possible A-predictive state. But if A is left recursive, there may be several different possible A-predictive states; and the one consisting of the completion of q's A-items may be invalid for the kinds of reasons we have been discussing.

Because in general, there may be several legal state-splittings of a given state q, the initial line of our formal development will not attempt to dictate

how a state should be split; rather we shall concentrate on the problem of determining whether or not a proposed splitting of a state q is a well-formed splitting. But before we begin this, let us see how a split state is incorporated into the machine at large, how the submachine to locate a predicted nonterminal is constructed.

Let us stress once again that the fact that we have been able to split a state, that we are consequently able to make a prediction during the course of a parse, should not make any material difference to the order of this parse. This is still to be performed in strict left-to-right, bottom-up order. However, we shall be performing this parse on a stack of stacks, a new level being created by the making of a prediction and suspended by its fulfillment. The machine controlling this modified parse is to be our original machine, suitably altered to include these facilities. As we mentioned earlier, the immediate changes to the design of the parsing machine are straightforward: the state q is replaced by the set of states defined by the splitting, the base and the predictive states. All transitions formerly going into q from other states of the machine are made to go into the base state; the predictive states are, of course, linked by predictive transitions to the base. Then the successors of the base and predictive states are computed recursively, until all successor states have been generated. Observe that a number of states of the original machine, which were successors of q, may no longer be accessible in the new machine, and so may be discarded; in addition, there may possibly be new states not in the original machine.

There is one special case to be considered, however. Suppose in the process of generating successors of a base or predictive state, the state q, which was the state originally split, turns up. (This might occur if q were

its own successor in the original machine.) Then we do not proceed normally; the state q is not recreated, but rather any transitions which are scheduled to go to its new incarnation are instead rerouted to go to the base state of the original splitting. We shall provide an example of this, plus its motivation, a little later.

This new machine is a member of a class of machines, which we shall formally define shortly. The splitting procedure may be applied again to states of this machine, resulting in yet another machine. We shall continue this process until we achieve a parsing machine whose structure satisfies certain properties; and then we will be able to utilize our predictability in a new way. Before we go into this, however, we shall present a formal development of the preceding material. First we shall precisely define the notion of a state-splitting, and then develop the theory of multi-stack parsing, proving that a state-splitting does not affect the language recognized by a parsing machine.

3.3 Formal Definition of State-Splitting and Some Properties

First we need a number of preliminary definitions, some of which we have seen before. Throughout the following, q is an LR(k) state, I an LR(k) item, and A a nonterminal.

Definition 3.1 Suppose I is of the form $A \rightarrow \alpha . \beta(\omega)$. Then $FIRST_k(I) = \{x | x \in V_T^k$ and $\beta\omega \overset{*}{\underset{L}{\Rightarrow}} xy$ for some $y\}$.

That is, $FIRST_k$ of an item consists of all terminal prefixes of length k that can be derived from the post-dot portion of the item.

Definition 3.2 $A(q) = \{I \mid I \in q \text{ and } A \text{ is the post-dot component of } I\}$.

Definition 3.3 $L_k(q,A) = \bigcup_{I \in A(q)} FIRST_k(I)$.

$L_k(q,A)$ is the set of k-lookaheads for the items of q which have A as post-dot component.

Definition 3.4 A chain through q from I_0 to I_n is a sequence of items of q, $I_0, I_1, \ldots, I_n$, such that I_{j+1} is an immediate descendant of I_j for each j, $0 \leq j < n$.

Definition 3.5 I is a terminal item if I is of the form $A \rightarrow \alpha.a\beta(\omega)$, where $a \in V_T$, or if I is of the form $A \rightarrow .\epsilon(\omega)$.

Definition 3.6 If $L \subset V_T^k$, then $T(q,L) = \{I \mid I \text{ is a terminal item of } q \text{ and } FIRST_k(I) \cap L \neq \emptyset\}$.

That is, a terminal item of q is in T(q,L) if and only if some element of its k-lookahead set is in the set L.

Definition 3.7 Let $L \subset V_T^k$. A chain through q from I_0 to I_n is an L-chain if and only if I_0 is an essential item of q and $I_n \in T(q,L)$.

Intuitively, the set of L-chains lists all possible ways of achieving a lookahead from the set L in state q. So should a lookahead in L be sighted upon entry to L, the sequence of reductions discovered by q must follow one of these chains.

Definition 3.8 Let R be any set of LR(k) items. Then $FOLLOW_k(A, R) = \{x \mid x \in V_T^k$ and there is an item $B \rightarrow \alpha.A\beta(\omega)$ in R such that $\omega \overset{*}{\underset{L}{\Rightarrow}} xy$ for some y$\}$.

That is, $FOLLOW_k(A,R)$ is the set of k-length terminal strings that might be a lookahead once a post-dot A has been found in R.

We are now in a position to define a legal state-splitting. For pedagogical reasons only, we shall first develop the definition for the case where only a single prediction is being made, and then immediately generalize it.

What information is needed to specify a splitting of state q? In general, we need to know what nonterminal is being predicted and upon what lookaheads, as well as the composition of the base and predictive states. We have decided, for the time being, that the lookaheads which occasion the prediction of an A must be the full set of lookaheads that may be generated from A in q, i.e., $L_k(q,A)$. Thus, only A (the predicted nonterminal), B (the set of items comprising the base state), and P (the predictive state), must be given.

So for a given triple (B, A, P), how do we know if they define a legal splitting of the state q? First of all, we must determine if the nonterminal A can really be predicted in q; and if it can be, we must ascertain that the division of q into B and P is induced by this prediction. Now if A can be predicted in state q, the language causing the prediction to be made is $L_k(q,A)$. In order to be able to safely predict an A in q upon seeing a lookahead from $L_k(q,A)$, it must be the case that every way of generating such a lookahead in q must involve an A on its leftmost branch. Since the $L_k(q,A)$-chains through q list all these leftmost branches, there must be at least one .A-item in each of these chains. So A can be predicted in q iff there is a .A-item in each $L_k(q,A)$ chain.

The acutal splitting of the state, the composition of B and P, is derived directly from the positions of these A-items in the chains. The predictive state P is to contain, for each of these chains, the A-item and all items after it in the chain; while B consists of the items before the A-item, as well as items present in non-$L_k(q,A)$ chains (i.e., items contributing to lookaheads other than those that cause the prediction to be made). These are the criteria which B and P must satisfy for the splitting to be based on the prediction. If A is left recursive, there may be several A-items in any given $L_k(q,A)$ chain. In this case, we can say that the splitting is based on the prediction if every $L_k(q,A)$ chain can be "broken" at some A-item so that B and P satisfy the criteria mentioned above.

Even if a state-splitting reflects the valid prediction of a nonterminal, it may still fail to be a legal splitting, if A happens to be left recursive. For in that case, we require that a "predicted A" be distinguishable from a "lower level A", by insisting that $\text{FOLLOW}_k(A,B) \cap \text{FOLLOW}_k(A,P) = \emptyset$. This then is an additional requirement that must be satisfied.

We summarize all this in the following.

Definition 3.9 Let q be an LR(k) state. Then a bipartite state-splitting of q is a triple (B, A, P) satisfying the following two conditions:

i) for each $L_k(q,A)$ chain $c = I_0,\ldots,I_n$. there is a j, $0 \leq j < n$ such that I_{j+1} is an A-item and such that if we set $H_1(c) = \{I_0, I_1,\ldots,I_j\}$ and $H_2(c) = \{I_{j+1},\ldots,I_n\}$, then $P = \bigcup_c H_2(c)$ and $B = \bigcup_c H_1(c) + \{I \mid I \in q \text{ and } \text{FIRST}_k(I) \not\subset L_k(q,A)\}$.

ii) $\text{FOLLOW}_k(A,B) \cap \text{FOLLOW}_k(A,P) = \emptyset$.

This rather cumbersome definition is the formalization of all the preceding; and while it may seem forbidding, it is very easy to use. There are several alternative formulations of the concepts we are developing here, and we have chosen this particular approach for two main reasons: it mirrors most naturally the underlying conceptual motivation; and it is amenable to the kinds of generalizations we will be making later. In fact, some of the definitions we are making and results we are proving have simpler analogues; but we will need the more general versions later.

As a brief digression, let us examine the meaning of this definition if $k = 0$. Intuitively, if an LR(0) state q can be split with a predictive state for A, then it should be possible to predict, upon entry to q, the eventual discovery of an A, regardless of what the lookahead is at time of entry. And this is precisely what the definition does imply. If $k = 0$, $L_k(q,A)$ is just the set consisting of the empty string ε; therefore, every terminal item of q is in $T(q,L_k(q,A))$. Thus if there is a predictive state for A, there is an A-item in every chain through q from an essential item to a terminal one. Note further if A is left recursive, that there can be no .A items in the predictive state, and so the submachine will have to return after finding the first A that it encounters. This must be so, for if there were .A items in P, the $FOLLOW_0(A,P)$ would be $\{\varepsilon\}$, and so would intersect with $FOLLOW_0(A,B)$; this would violate the second condition of (B, A, P) being a legal splitting.

As promised, we shall generalize the above definition to include the possibility of there being multiple predictive states, with each distinct nonterminal being predicted on its own lookahead set.

Definition 3.10 Let q be an LR(k) state. Then a state-splitting of q is a pair (B,Q), where Q is a finite set of pairs (A_i,P_i), satisfying:

i) $L_k(q,A_i) \cap L_k(q,A_j) = \emptyset$ if $i \neq j$

ii) $FOLLOW_k(A_i,B) \cap FOLLOW_k(A_i,P_i) = \emptyset$

iii) for each $L_k(q,A_i)$ chain $c = I_0,\ldots,I_n$, there is a j, $0 \leq j < n$, such that I_{j+1} is an A_i-item and such that if we set $H_1(c) = \{I_0,\ldots,I_j\}$ and $H_2(c) = \{I_{j+1},\ldots,I_n\}$, then $P_i = \bigcup_c H_2(c)$, where c ranges over all $L_k(q,A_i)$ chains, and $B = \bigcup_i \bigcup_c H_1(c) + \{I \mid I \in q \text{ and } FIRST_k(I) \not\subseteq \bigcup_i L_k(q,A_i)\}$, where for each i, c ranges over all $L_k(q,A_i)$ chains.

This is a straightforward extension of the preceding definition, with the additional requirement that a lookahead causes at most one prediction. Note that the base state is designed to pick up after any one of the predictive states has completed its task, as well as to handle any string which does not occasion a prediction to be made.

We will not devote much effort to studying state-splitting by itself; we are more interested in observing the effects on a parsing machine of splitting one of its states. But one thing we shall do with our definition is show it equivalent to a weaker form.

One problem with Definition 3.10 is that it is expressed in terms of all $L_k(q,A_i)$ chains through q, which in general might be an infinite set. It is not clear that this definition is effective, that there is some finite algorithm to determine whether or not these potentially infinite sets satisfy the required properties. Furthermore, this definition just tells us whether or

not a candidate for a state-splitting is valid or not--it gives us no clue about finding a splitting for a state. We shall solve both these problems by restructuring the definition in terms of a finite, distinguished subset of the set of all $L_k(q,A_i)$ chains.

Definition 3.11 A loop in q is a chain $I_0\ I_1 \ldots I_n$ such that $I_0 = I_n$ and no other pair of items are identical. A chain $K_0K_1\ldots K_m$ contains the loop $I_0I_1\ldots I_n$ if there is an i such that $K_{i+j} = I_j$, for each j, $0 \leq j < n$. The chain contains r occurrences of the loop if there are r different values for such an i.

Definition 3.12 A chain is called simple if it contains no more than two occurrences of any loop. A simple L-chain is an L-chain that is also simple.

We have defined loops in such a way that, for example, no loop can be contained in another. The set of simple chains is sufficient for consideration for purposes of state-splitting, as the following theorem shows.

Theorem 3.13 The set of state-splittings defined by Definition 3.10 is unchanged if the phrase "$L_k(q,A_i)$ chain" is changed to "simple $L_k(q,A_i)$ chain" throughout the definition.

Proof We have to show that (B, Q) satisfies the revised definition if and only if it satisfies the original one. We shall make use of the following result.

Lemma 3.14 Let c be a chain $I_0\ldots I_n$ containing item I_i and I_j, $i < j$. Then there is a simple chain $J_0\ldots J_m$ such that $J_0 = I_0$, $J_m = I_n$, and for some i' and j', with $i' < j'$, $I_i = J_{i'}$ and $I_j = J_{j'}$.

Proof This lemma states that for any chain with distinguished items in a specified order, there is a simple chain with the same endpoints as the original chain, also containing the distinguished items in the same order. We will prove this constructively, by showing how to construct this simple chain. We shall remove all loops from the original chain, except those that contain either of the distinguished items. The result will be the desired simple chain.

So given the chain $c = I_0 \ldots I_n$, with distinguished items I_i and I_j, $i < j$, we apply to c the following procedure.

Algorithm 3.15

1) Set $c' = c$

2) Find the leftmost loop $I_{k_1} I_{k_2} \ldots I_{k_m}$ contained in c' (i.e., the loop with the smallest value of k_1), which satisfies one of the following properties:

   i) $k_m \leq i$
   ii) $k_1 \geq j$
   iii) $k_1 \geq i$ and $k_m \leq j$, but if $k_1 = i$ then $k_m \neq j$

   If there is such a loop, go to step 3). If there isn't, go to step 4).

3) If $i = k_m$ or $j = k_m$, eliminate the subchain $I_{k_1} I_{k_2} \ldots I_{k_{m-1}}$ from c'; otherwise, eliminate the subchain $I_{k_2} I_{k_3} \ldots I_{k_m}$ from c'. Go to step 2).

4) This means we are done. Renumber c' as $J_0 J_1 \ldots J_p$; this will be the desired simple chain.

First we shall show that this algorithm does in fact produce the advertised result. Observe that by the way the loop is chosen and the subchain removal is effected, neither I_i nor I_j is ever removed from the chain c'. Furthermore, since each subchain removal eliminates all but one extremum of a loop, the resulting sequence is still a chain, and one in which I_i is a predecessor of I_j. Finally, the endpoints of c' are the same as the endpoints of c, because the endpoints of the chain could serve at worst as endpoints of eliminated loops, and so are not eliminated from the chain.

It remains to show that the final version of c' is a simple chain. (The algorithm clearly does terminate since the original c was only a finite chain.) We claim that any loop $I_{k_1} \ldots I_{k_m}$ in the final version of c' contains at least one of I_i and I_j in its interior or has them both as the endpoints. That is, either $k_1 < i < k_m$ or $k_1 < j < k_m$ or both $k_1 = i$ and $k_m = j$. This follows because any other relationship among them would satisfy one of the conditions of step 2), and thus cause the loop to have been eliminated by step 3). Now suppose that c' is not simple. Then it contains more than two instances of some loop. By what we have just seen, each instance of this loop must contain I_i or I_j or both. Since there are more than two instances of the loop, by the pigeon-hole principle some two instances must both contain the same item I_i or I_j, say I_i. Thus these two loop instances intersect on the item I_i. But by our definition of loop, a loop has no repetitions in its interior. Thus it is impossible for two instances of the same loop to overlap except on their endpoints -- i.e., the end of one being the beginning of the other. Thus I_i is the end of one loop and the beginning of the other. But if I_i is the end of a loop

(i.e., $i = k_m$ for that loop), the loop would have been eliminated, since it satisfies the first criterion of step 2). Thus we have reached a contradiction, and it follows that c' must be simple.

Intuitively, what we have done in Algorithm 3.15 is divide the chain into three regions; see Figure 3.18. We eliminate entirely any loops contained wholly in Regions I and III. These are loops that do not involve I_i or I_j, except possibly I_i as an endpoint or I_j as a starting point. Similarly, loops wholly in Region II are also eliminated, unless its two extrema are I_i and I_j (this qualification is to prevent either I_i or I_j from disappearing from the chain during a loop elimination). Thus the only loops left when the algorithm terminates are those which overlap two regions;

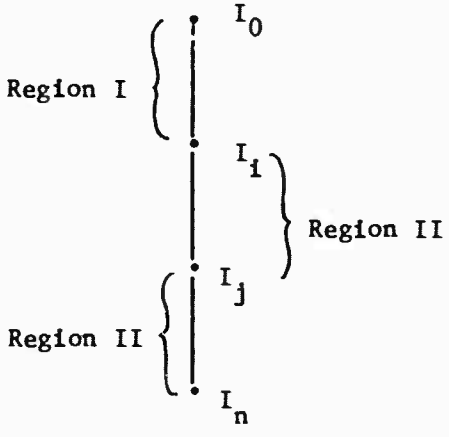


Figure 3.18

and each of these will then contain I_i or I_j. Note that throughout we are concerned with the particular item and location I_i; the same item as I_i may appear elsewhere in the chain c and may be removed from its other positions by loop elimination. But the item will remain in the position held by I_i.

Observe that the items of c' will not be numbered consecutively throughout the process of removing subchains; they retain their original numbering that they had in c, and it is only at the last step that they are renumbered.

This completes the proof and explanation of the lemma. Now we want to use this lemma to prove Theorem 3.13. We will first show that if (B,Q) satisfies the revised definition of a state-splitting, then it also satisfies the original one. Now the first two criteria of the definition are not affected by changing "$L_k(q,A_i)$ chain" to "simple $L_k(q,A_i)$ chain", since they have nothing to do with chains; so if (B,Q) satisfies them in the new definition, it satisfies them in the old definition as well. What we must show is that if each simple $L_k(q,A_i)$ chain can be broken into $H_1(c)$ and $H_2(c)$ so that $P_i = \bigcup_c H_2(c)$, where c ranges over all simple $L_k(q,A_i)$ chains, and the corresponding statement is true for B, then <u>every</u> $L_k(q,A_i)$ chain can be broken into $H_1(c)$ and $H_2(c)$, so that the same equations hold, the unions now being over all $L_k(q,A_i)$ chains.

<u>Definition 3.16</u> If (B,Q) is a revised state-splitting of q, then a $.A_i$ item in B is called an A_i-break candidate.

Then given (B,Q), a revised state-splitting of q, and $c = I_0 \ldots I_n$, a non-simple $L_k(q,A_i)$ chain, we define $H_1(c) = \{I_0, I_1, \ldots, I_j\}$ and $H_2(c) = \{I_{j+1}, \ldots, I_n\}$, where j is the largest value for which I_j is an A_i-break candidate. (Obviously, a revised state-splitting is one that satisfies the definition containing the phrase "simple $L_k(q,A_i)$ chains".) We are trying to use the breakings of simple chains implied by the identity of B and P_i, as models for breaking the non-simple chains as well.

First we must show that this definition of H_1 and H_2 is meaningful, that every non simple $L_k(q,A_i)$ chain contains at least one A_i-break candidate. Starting with a non-simple $L_k(q,A_i)$ chain c, we may repeatedly remove loops until we are left with a loop-free $L_k(q,A_i)$ chain, $c' = J_0 \ldots J_m$. Since (B,Q) is a revised state-splitting, $H_1(c')$ and $H_2(c')$ are defined, because a loop-free chain is also simple. Say $H_1(c') = \{J_0,\ldots,J_k\}$ and $H_2(c') = \{J_{k+1},\ldots,J_m\}$. By the definition, J_{k+1} is an A_i-item which is an immediate descendant of J_k. Therefore J_k is an A_i item. But $J_k \in H_1(c') \subset B$. Thus J_k is an A_i-break candidate. Since c' is contained within c, J_k is an item of c, and so c has at least one A_i-break candidate.

Now we must show that H_1 and H_2, as they are now defined on all $L_k(q,A_i)$ chains, do indeed make (B,Q) a legal state-splitting of q according to the original definition.

We introduce the following notation: Assume q, k, and a set of A_i's are given; then $R = \{I \mid I \in q \text{ and } FIRST_k(I) \notin \bigcup_i L_k(q,A_i)\}$; while for each i, FC_i is the set of all $L_k(q,A_i)$ chains and SC_i is the set of all simple $L_k(q,A_i)$ chains.

What we must show then that for the extended H_1 and H_2, and for (B,Q) as given, $P_i = \bigcup_{c \in FC_i} H_2(c)$ and $B = \bigcup_i \bigcup_{c \in FC_i} H_1(c) + R$. Since (B,Q) is known to be a revised state-splitting, we know that $P_i = \bigcup_{c \in SC_i} H_2(c)$ and similarly for B. What must be shown then is that, for each i, $\bigcup_{c \in FC_i} H_2(c) = \bigcup_{c \in SC_i} H_2(c)$ and $\bigcup_{c \in FC_i} H_1(c) = \bigcup_{c \in SC_i} H_1(c)$. We will

prove the latter of these; precisely the same argument and construction suffice for the former as well.

Since $SC_i \subset FC_i$, we need only show $\bigcup_{FC_i} H_1(c) \subset \bigcup_{SC_i} H_1(c)$. Let $c = I_0 \ldots I_u$ be any element of FC_i, with $H_1(c) = \{I_0,\ldots,I_j\}$ and $H_2(c) = \{I_{j+1},\ldots,I_n\}$. Pick any element of $H_1(c)$, namely I_k, with $0 \leq k \leq j$. We will find a simple chain c', an element of SC_i, such that $I_k \in H_1(c')$. We do this by applying Lemma 3.14 to the chain, using as the distinguished items I_k and I_j. Thus by the Lemma we have a simple chain with the same endpoints as c, also containing I_k as a predecessor of I_j. Since the simple chain ends at a $T(q,L_k(q,A_i))$ item, it is a simple $L_k(q,A_i)$ chain. It only remains to show that $I_k \in H_1(c')$. Now we know that $I_j \in H_1(c)$; in fact, I_j was the last A_i-break candidate in c--that is how $H_1(c)$ and $H_2(c)$ got to be defined as they were. But if I_j is an A_i-break candidate, that means that $I_j \in B$, by definition of A_i-break candidate. So suppose $I_k \notin H_1(c')$. Then $I_k \in H_2(c')$; and since I_j follows I_k in c', that means $I_j \in H_2(c')$ also. But c' is a simple $L_k(q,A_i)$ chain. Therefore, $H_2(c') \subset P_i$, since (B,Q) is a revised state-splitting by hypothesis. Thus $I_j \in B$ and $I_j \in P_i$. But I_j is a $.A_i$ item; that is, A_i is the post-dot component of I_j. Therefore, I_j contributes to both $FOLLOW_k(A_i,B)$ and $FOLLOW_k(A_i,P_i)$; therefore, $FOLLOW_k(A_i,B) \cap FOLLOW_k(A_i,P_i) \neq \emptyset$. But this contradicts the second criterion of (B,Q) being a legal revised state-splitting. Thus we have reached a contradiction. Therefore $I_k \in H_1(c')$, and $\bigcup_{FC_i} H_i(c) \subset \bigcup_{SC_i} H_1(c)$, which is what we wanted to prove. Precisely the same technique can be used to show $\bigcup_{FC_i} H_2(c) \subset \bigcup_{SC_i} H_2(c)$. Thus any state-splitting satisfying the revised definition also satisfies the original one.

To get some feeling for the kind of proof we have just used, and which we shall use again, refer to Figure 3.19. We see how c is divided into $H_1(c)$ and $H_2(c)$, and how c' might be broken if $I_k \notin H_1(c')$. Since I_j is at the end of $H_1(c)$, it must be a break-point, and so must be in B; while the potential division of c' would put I_j in P_i. And this is impossible, as we have seen.

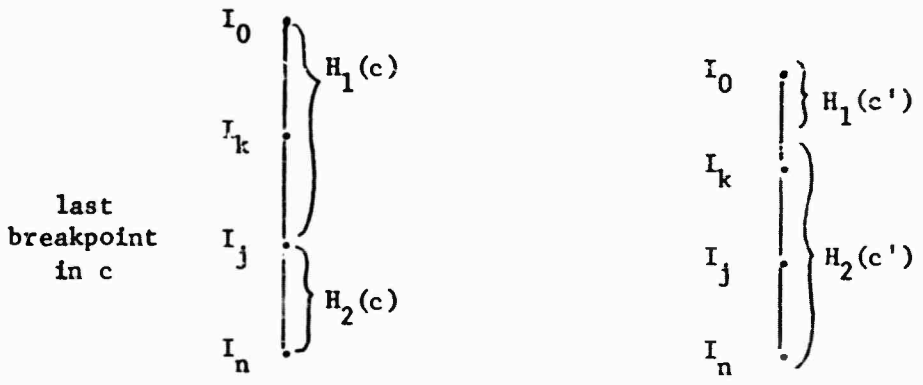


Figure 3.19

To complete the proof of the theorem, we must assume that (B,Q) is a legal state-splitting by the original definition, and then show that it also satisfies the revised definition. We will not have to define a new H_1 and H_2 to use in the revised definition, but can use the restriction of the H_1 and H_2 of the original definiton to the simple $L_k(q,A_i)$ chains. Once again, the proof reduces to showing that $\bigcup_{FC_i} H_1(c) \subset \bigcup_{SC_i} H_1(c)$ and that $\bigcup_{FC_i} H_2(c) \subset \bigcup_{SC_i} H_2(c)$. And to show this, we use exactly the same techniques as we did before. We select any $L_k(q,A_i)$ chain c, let I_j be the last item

in $H_1(c)$, and let I_k be any element of $H_1(c)$. Then we construct the corresponding simple chain c' containing both I_k and I_j, and argue that if $I_k \notin H_1(c')$, then I_j, a $.A_i$ item, would be in both B and P_i, which is impossible. It is almost identical to the first proof. Q.E.D.

This completes the proof of the theorem. In order to make use of it, we should establish the following result which, while intuitively obvious, deserves proof because of our unusual definition of loop.

Proposition 3.17 For any LR(k) state q and any $L \subset V_T^k$, there are only finitely many simple L-chains through q.

Proof If there were infinitely many simple L-chains, then for any number n there would be some simple L-chain with at least n repetitions of the same item. Let m be the number of different loops that can be formed using items in q; since there are only finitely many items in q, and each loop allows for repetition only at the ends, this will be a finite number. Let n = 3m, and choose some simple L-chain c that has at least 3m instances of the same item. Every successive pair of these items defines a subchain of the main chain which starts and ends with the same item. Each of these subchains must either itself be a loop or contain a loop. Thus there are at least 3m-1 loops in the chain c. Since there are only m different loops that can be formed from items of q, there must be some loop which has at least three occurrences in c. But this means c is not simple, which is a contradiction. Therefore, there are only finitely many simple L-chains through q. Q.E.D.

(As a brief aside, we note that the seemingly peculiar choice of our definition for simple chains was motivated by a desire to make Lemma 3.14 be valid. The more intutive choices for the definition (namely a chain with no loops, or a chain with at most one instance of any single loop) would not satisfy this requirement. Some items of a state may not appear in any loop-free chains through the state; and while one particular item may precede another in an unrestricted chain, this may no longer be the case when we consider only chains with at most one instance of any loop.)

Not only is the set of simple L-chains through q a finite set, it is a set that can be effectively computed. Recall that a simple L-chain is a chain that ends with an item in T(q,L); i.e., a terminal item I such that $FIRST_k(I) \cap L \neq \emptyset$. It is well known that $FIRST_k(I)$ can be computed for any item I of an LR(k) grammar; thus, since both q and L are finite, the elements of T(q,L) can be computed. If there are p different items in the state q, and m is the number of different loops in q, then it is clear that the longest simple chain in q must be shorter than 3mp; for if there were a simple chain of length 3mp or greater, there would be at least 3m instances of some particular item in the chain, which we have just seen to be impossible. Thus it is clear that we can compute the full set of simple chains in q, and hence the set of simple L-chains through q, for any given L. For any given nonterminal A_i, $L_k(q,A_i)$ is also computable; therefore we can effectively construct the set of simple $L_k(q,A_i)$ chains.

So suppose we are given an LR(k) state q and (B,Q), a candidate for a splitting of q; we can then effectively determine whether or not (B,Q) is really a splitting of q as follows. Since $L_k(q,A_i)$, $FOLLOW_k(A_i,B)$, and $FOLLOW_k(A_i,P_i)$ are computable for each i, we can check the first two

conditions of the definition. Then it only remains to see whether or not H_1 and H_2 can be defined on the simple $L_k(q,A_i)$ chains in such a way that the defining equations for B and P_i are satisfied. But there are only finitely many A_i's, and for each one, the set of simple $L_k(q,A_i)$ chains is finite and computable; thus there are only finitely many different possible definitions for H_1 and H_2, which we can try in turn. For each definition, the splitting induced by it can be examined and compared with (B,Q). If (B,Q) does match one of them, then it is indeed a legal splitting; otherwise it is not.

Thus we have established the following result.

Corollary 3.18 Given an LR(k) state q and a pair (B,Q), it is decidable whether or not (B,Q) is a splitting of q.

We can go one further step. For a given q, there are only a finite number of pairs (B,Q), where $B \subset q$ and Q is a finite set of pairs (A_i,P_i), where A_i is a nonterminal and $P_i \subset q$. All these pairs (B,Q) can be effectively generated, and each can be tested in turn as to whether or not it is a splitting of q. Therefore we have:

Corollary 3.19 For any LR(k) state q, it is decidable whether or not there is a splitting of q.

Obviously, both these procedures are ridicuously inefficient, but they suffice to show that state-splitting is an effective and computable notion. Later we shall discuss more efficient algorithms for deciding these issues. Actually, we shall rarely be interested in determining whether some arbitrary

splitting is valid or not, or whether there is some splitting of a state q; rather, we shall need to know if there is some splitting of q that satisfies certain additional constraints, and these constraints will help us resolve this question. For the time being, we shall just establish a few general results about state-splittings, to give us some further feeling for them and some intuition as to what might be legal or not.

The first thing we should establish is that the parts of a splitting add up to the state being split. This will entail proving a variety of interesting intermediate results.

Definition 3.20 If (B,Q) is a splitting of q, then B and each P_i is a component of the splitting.

Lemma 3.21 If the item $A \rightarrow \alpha.\beta(\omega)$ is an ancestor of the item $B \rightarrow .\gamma(\tau)$, then $\beta\omega \stackrel{*}{\Rightarrow} \gamma\tau y$, for some $y \in V_T^*$.

Proof If $A \rightarrow \alpha.\beta(\omega)$ is an ancestor of $B \rightarrow .\gamma(\tau)$, then there is a sequence of items $I_0, I_1, \ldots, I_n$, where $I_0 = A \rightarrow \alpha.\beta(\omega)$, $I_n = B \rightarrow .\gamma(\tau)$, and I_{i+1} is an immediate descendant of I_i. We proceed by induction on the length of this sequence. If $n = 0$, the statement is trivial. So say it is true for $n = j$; to show it true for $n = j+1$. Consider the item I_j in the sequence which shows $B \rightarrow .\gamma(\tau)$ is a descendant of $A \rightarrow \alpha.\beta(\omega)$. Since $B \rightarrow .\gamma(\tau)$ is I_{j+1}, the post-dot component of I_j must be B. Let I_j be $C \rightarrow .B\gamma_1(\tau_1)$. By induction, since I_j is a descendant of I_0, $\beta\omega \stackrel{*}{\Rightarrow} B\gamma_1\tau_1 y$, for some y. But since I_{j+1} is an immediate descendant of I_j, we have $\tau \in \text{FIRST}_k(\gamma_1\tau_1)$; this means $\gamma_1\tau_1 \stackrel{*}{\Rightarrow} \tau y_1$, for some y_1. Hence $\beta\omega \stackrel{*}{\Rightarrow} B\gamma_1\tau_1 y \Rightarrow \gamma\gamma_1\tau_1 y \stackrel{*}{\Rightarrow} \gamma\tau y_1 y$. If we let $y' = y_1 y$, then $\beta\omega \stackrel{*}{\Rightarrow} \gamma\tau y'$, and we are done. Q.E.D.

Lemma 3.22 If item I_1 is an ancestor of item I_2, then $FIRST_k(I_2) \subset FIRST_k(I_1)$.

Proof Suppose I_1 is $A \rightarrow \alpha.\beta(\omega)$ and I_2 is $C \rightarrow .\gamma(\tau)$. If $\rho \in FIRST_k(I_2)$, then $\gamma\tau \overset{*}{\underset{L}{\Rightarrow}} \rho x$ for some $x \in V_T^*$. By Lemma 3.21, $\beta\omega \overset{*}{\Rightarrow} \gamma\tau y$, for some $y \in V_T^*$. Thus $\beta\omega \overset{*}{\Rightarrow} \gamma\tau y \overset{*}{\Rightarrow} \rho xy$, for some x and y in V_T^*. Therefore $\beta\omega \overset{*}{\Rightarrow} \rho z$, for some $z \in V_T^*$. Since $\rho \in V_T^k$, $\rho z \in V_T^*$; and it is well known that if $\Psi \overset{*}{\Rightarrow} \rho z \in V_T^*$, then $\Psi \overset{*}{\underset{L}{\Rightarrow}} \rho z$. Therefore $\beta\omega \overset{*}{\underset{L}{\Rightarrow}} \rho z$, so $\rho \in FIRST_k(\beta\omega)$, so $\rho \in FIRST_k(I_1)$. Q.E.D.

Lemma 3.23 If $\omega \in FIRST_k(I)$, then there is a chain of items $I = I_0, I_1, \ldots I_n$, such that I_n is a terminal item and $\omega \in I_n$.

Proof If I is a terminal item, then we are done. So assume that I is $A \rightarrow \alpha.B\beta(\tau)$; then $B\beta\tau \overset{*}{\underset{L}{\Rightarrow}} \omega x$, some $x \in V_T^*$. Then $B \overset{*}{\underset{L}{\Rightarrow}} \omega_1$, some prefix of ωx. In this derivation, there is some first application of a rule whose right-hand side is ϵ or starts with a terminal symbol. Let this be the n^{th} step of the derivation, and let p_i be the rule applied at the i^{th} step, $i \leq n$. For uniformity, say p_i is $B_i \rightarrow B_{i+1}\varphi_i$, while p_n is $B_n \rightarrow \varphi_n$. Each $B_i \overset{*}{\underset{L}{\Rightarrow}} \omega_i$, where ω_i is some prefix of ωx; define ρ_i such that $\omega_i\rho_i = \omega x$, and let $\tau_i = \rho_i/k$. Let the item I_i be $B_i \rightarrow .B_{i+1}\ \varphi_i(\tau_i)$.

The item I_n, $B_n \rightarrow .\varphi_n(\tau_n)$, is a terminal item, by definition of n. Furthermore, $\varphi_n\tau_n \overset{*}{\underset{L}{\Rightarrow}} \omega_n\tau_n$, which is a prefix of ωx of length at least k; hence $\omega \in FIRST_k(I_n)$. All that remains is to show that $\tau_{i+1} \in FIRST_k(\varphi_i\tau_i)$; this will establish that I_{i+1} is an immediate descendant of I_i. We know that $B_i\rho_i \underset{L}{\Rightarrow} B_{i+1}\varphi_i\rho_i \overset{*}{\underset{L}{\Rightarrow}} \omega_{i+1}\varphi_i\rho_i \overset{*}{\underset{L}{\Rightarrow}} \omega x$; but by definition, $\omega_{i+1}\rho_{i+1} = \omega x$. Hence $\varphi_i\rho_i \overset{*}{\underset{L}{\Rightarrow}} \rho_{i+1}$; therefore $\tau_{i+1} = \rho_{i+1}/k \in FIRST_k(\varphi_i\rho_i)$. Q.E.D.

Lemma 3.24 If I is any item, then $FIRST_k(I) = \bigcup FIRST_k(I')$, for all I' which are immediate descendants of I, not equal to I.

Proof By Lemma 3.22, containment in one direction is immediate. If $\omega \in FIRST_k(I)$, then there is a chain $I = I_0, I_1, \ldots, I_n$, where $\omega \in FIRST_k(I_n)$ and I_n is a terminal item. Let I_j be the first item in this chain which is not equal to I. Then by Lemma 3.22, since I_j is an ancestor of I_n, $\omega \in FIRST_k(I_j)$. Furthermore, $I = I_{j-1}$ by definition of j, so I_j is an immediate descendant of I. Q.E.D.

In exactly the same way, we can get the following characterizations.

Lemma 3.25 $FIRST_k(I) = \bigcup FIRST_k(I')$, for all I' which are descendants of I, not equal to I.

Lemma 3.26 $FIRST_k(I) = \bigcup FIRST_k(I')$, for all I' which are terminal items and descendants of I.

These results tells us that it is possible to recursively compute $FIRST_k$, by starting with $FIRST_k$ of terminal items.

Lemma 3.27 Let (B,Q) be a splitting of q, I an item of q. If $FIRST_k(I) \cap L_k(q,A_i) \neq \emptyset$, for some i, then I is an element of some simple $L_k(q,A_i)$ chain.

Proof Since I is an item of q, it is a descendant of some essential item of q. Suppose $\omega \in FIRST_k(I) \cap L_k(q,A_i)$: then by Lemma 3.23, I is the ancestor of some terminal item I', such that $\omega \in FIRST_k(I')$. Join these two chains, the one from an essential item to I, and the one from I to I', and the result is a chain through q to I', that contains I. By Lemma 3.14, we can

construct from this chain a simple chain through q to I', which also contains I. Since I' is an $L_k(q,A_i)$ terminal item by definition, this chain is a simple $L_k(q,A_i)$ chain. Q.E.D.

Lemma 3.28 Let (B,Q) be a splitting of q; then every item of q appears in at least one component of the splitting.

Proof Let I be any item of q. If $FIRST_k(I) \not\subset \bigcup L_k(q,A_i)$ then $I \in B$ by the defining equation for B. If $FIRST_k(I) \cap L_k(q,A_i) \neq \emptyset$ for some i, then by Lemma 3.27, I is contained in some simple $L_k(q,A_i)$ chain c; then by definition, H_1 and H_2 are both defined on c, so $I \in H_1(c)$ or $I \in H_2(c)$. But $H_1(c) \subset B$ and $H_2(c) \subset P_i$; hence $I \in B$ or $I \in P_i$. Q.E.D.

Corollary 3.29 If (B,Q) is a splitting of q, then q equals the union of all the components of the splitting.

Proof By Lemma 3.28 and the fact that each component consists of items of q. Q.E.D.

Proposition 3.30 If (B,Q) is a splitting of q, and if $I \in T(q,L_k(q,A_i))$ for some i, then $FIRST_k(I) \subset L_k(q,A_i)$.

Proof Intuitively, what this says is that if some k-lookahead of the item I of state q triggers the prediction of an A_i, then every lookahead of that item does cause a prediction of an A_i. The proof is as follows. Since $I \in q$, there is a chain to I from some essential item of q, and hence some simple chain. Since $I \in T(q,L_k(q,A_i))$ (that is, since $FIRST_k(I) \cap L_k(q,A_i) \neq \emptyset$), by definition any simple chain ending in I is an $L_k(q,A_i)$ chain. Since (B,Q) is a splitting of q, there is an $.A_i$ item in each such chain. Hence I is

the descendant of some $.A_i$ item, I'. Therefore $FIRST_k(I) \subset FIRST_k(I')$, by Lemma 3.22. Since by definition, $L_k(q,A_i)$ is the union of $FIRST_k$ for all $.A_i$ items, we have $FIRST_k(I) \subset FIRST_k(I') \subset L_k(q,A_i)$. Q.E.D.

Theorem 3.31 Let q be an LR(k) state, (B,Q) a splitting of q, and I an item of q of the form $A \rightarrow \alpha.\sigma\beta(\tau)$, where $\sigma \in V_N \cup V_T$ and σ is not a left-recursive nonterminal. Then I appears in only one component of the splitting (B,Q).

Proof Suppose $I \in P_i$ and $I \in P_j$. Then by definition, I is a descendant of some $.A_i$ item and a descendant of some $.A_j$ item. Therefore $FIRST_k(I) \subset L_k(q,A_i)$ and $FIRST_k(I) \subset L_k(q,A_j)$; therefore $L_k(q,A_i) \cap L_k(q,A_j) \neq \phi$. But this contradicts the definition of a state-splitting, since all the predictive languages must be disjoint.

Now suppose $I \in B$ and $I \in P_j$. Since $I \in P_j$, I is a descendant of a $.A_j$ item, and so $FIRST_k(I) \subset L_k(q,A_j)$. By definition, B is the union of the "upper half" of all simple $L_k(q,A_i)$-chains, and those items I' such that $FIRST_k(I')$ is not contained in the union of all the $L_k(q,A_i)$. Since $I \in B$, I must fall into one of these sets. Since $FIRST_k(I) \subset L_k(q,A_j)$ by Proposition 3.30, I does not fall into the latter category. Then I must be above a $.A_i$ item in some simple $L_k(q,A_i)$ chain, for some i. If this i is not equal to j, then we get that $FIRST_k(I) \cap L_k(q,A_i) \neq \phi$; but since $FIRST_k(I) \subset L_k(q, A_j)$, this means that $L_k(q, A_i) \cap L_k(q, A_j) \neq \phi$, which is a contradiction. Thus I must be above a $.A_j$ item in some simple $L_k(q,A_j)$ chain, in order for it to be in B. Say the post-dot element of I is C. Since I is above some $.A_j$ item, we have $C \overset{*}{\Rightarrow} A_j x$ for some x; since I is below some $.A_j$ item, we have $A_j \overset{*}{\Rightarrow} Cy$ for some y. Thus C, the post-dot component of I, is left recursive. This contradicts our hypothesis, and so we are done. Q.E.D.

Theorem 3.32 Let q be an LR(k) state, (B,Q) a splitting of q, and I any item of q. Then I is in at most one predictive state of the splitting.

Proof The first half of the preceding proof works for any item at all.

Q.E.D.

Corollary 3.33 Any terminal item of q appears in exactly one component of the splitting (B,Q).

This result follows both from Theorem 3.31 or from Proposition 3.30.

Theorem 3.34 Let (B,Q) be a splitting of q. If nonterminal A_i is not left recursive, then P_i is the completion of all the A_i-items in q.

Proof First of all, if $I \in P_i$, then I is in $H_2(c)$ for some simple $L_k(q,\ldots)$ chain; hence I is a descendant of some A_i item (the one in the chain), and thus in the completion of the A_i items. On the other hand, if I is an item in this completion, then I is a descendant of some A_i-item.

Therefore, by Lemma 3.14, I is a follower of this A_i-item in some simple chain through q. By definition, this chain will be a simple $L_k(\ldots,A_i)$ chain. Therefore, this chain must be breakable at some A_i-item into H_1 and H_2. Since A_i is not left-recursive, there is only one A_i-item in this chain, since I follows it in the chain, I will be in H_2 of the chain, and therefore in P_i by definition. Q.E.D.

In summary then, a state splitting distributes all the items of a state among the components of the splitting. Any particular item occurs in only one component unless its post-dot symbol is a left-recursive

nonterminal; in that case, the item may appear in the base state and also in the predictive state for A_i, where A_i and the post-dot nonterminal of the item in question are mutually left-recursive. Similarly, the contents of the predictive state for A_i are determined to be the completion of the A_i-items in the state being split, unless A_i is left recursive; in this case, the predictive state is a subset of this completion.

3.4 MSP(k) Machines

As we have stated several times previously, the main line of our interest is to investigate the effects of replacing a state in a parsing machine by its split equivalent; that is, by a base state and some number of attached predictive states. We have briefly described how this replacement would be effected and in what manner the resulting machine would operate. We now would like to study this replacement more carefully. The first step is to develop a formal model for the kind of parsing machine that results after a state of an LR(k) machine is replaced by its split equivalent. Since we will be interested in possibly splitting more than one state of a machine, we will allow for multiple split states in this new kind of machine. The operation of this machine will be as described previously. It will operate on a stack of stacks, reading and reducing on the topmost stack level just like an LR(k) machine, until it encounters a predict or suspend transition, at which point it will either create or delete the top stack level. We will call this machine a multiple stack parsing (MSP(k)) machine; the k indicates the length of the lookahead the machine may use during its operation.

An MSP(k) machine M is always associated with some grammar G, which, for the time being, we will require to be LR(k). There are four kinds of states in an MSP(k) machine. First of all, there are the final states: there is exactly one final state for each rule of the grammar G, plus one additional state POP. Then there are the initial states: these include the starting state of the machine, as well as all predictive states of state-splittings. Then there are the base states, which are the states from which predictions are made. And finally, there are intermediate states, which are regular garden-variety LR(k) states. Each state in these last three classes is comprised of a set of LR(k)-items over G, but these states are not always the completions of their essential items, as was the case with LR(k) states. There are precise rules governing the composition of these states.

There are several functions which tie the states together. The regular transition function f_M takes an initial, base, or intermediate state into a base, intermediate, or final state, in the same way that the LR(k) transition function operates. Namely, for a state q_1, a symbol (terminal or nonterminal or ϵ) σ, and a string x of k lookahead symbols, $f_M(q_1, \sigma, x)$ is the successor of q_1 on σ with lookahead x. This successor function is defined as follows. If there is no item of q_1 of the form $A \rightarrow \alpha.\sigma\beta(\omega)$ with $x \in FIRST_k(\beta\omega)$, then $f_M(q_1, \sigma, \omega)$ is undefined. If there is an item $A \rightarrow \alpha.\sigma(x)$ in q_1, then $f_M(q_1, \sigma, \omega)$ is the final state for the rule $A \rightarrow \alpha\,\sigma$. Otherwise, consider all items of q_1 of the form $A \rightarrow \alpha.\sigma\beta(\omega)$, where $\beta \neq \epsilon$; let q_2 be the state (stipulated to be unique) whose essential items are precisely $\{A \rightarrow \alpha\sigma.\beta(\omega)\}$. Then $f_M(q_1, \sigma, x) = q_2$. We shall require that the states be so constructed that the function f_M is always single-valued.

(In cases where there is no ambiguity, we shall leave off the subscript M from f_M and the other functions and sets of M.)

In addition to this almost standard transition function, there is a predictive transition function g_1. If q_1 is a base state and x a string of k terminal symbols, then $g_1(q_1,x)$ is the predictive successor of q_1 on lookahead x. If $g_1(q_1, x)$ is undefined, it means that if the lookahead is x on entry to q_1, no prediction is to be made. If it is defined, $g_1(q_1, x)$ is the initial state to which we should jump. There is another function associated with this prediction. If q_2 is an initial state, $g_2(q_2)$ is the name of the predicted nonterminal associated with q_2. Thus upon entry to state q_1 with the lookahead being x, we would jump to $g_1(q_1,x)$ and predict $g_2(g_1(q_1,x))$. Of course, we will require that the state q_1 and all its associated predictive successors define a legal state-splitting of some LR(k) state; and that the set of x such that $g_1(q_1,x)$ equals q_2 is precisely $L_k(g_2(q_2))$ with respect to the split state.

All of this is summarized as follows.

Definition 3.35 Let G be an LR(k) grammar (V_N, V_T, S, P). Then an MSP(k) machine associated with G is a tuple $(Q_1, Q_2, Q_3, Q_4, q_0, f, g_1, g_2)$, where:

1) each of Q_1, Q_2, and Q_3 are finite state sets, each of whose elements is a set of LR(k)-items over G
2) $Q_4 = P \cup \{POP\}$, the set of final states
3) $q_0 \in Q_1$, the starting state
4) $f: (Q_1 \cup Q_2 \cup Q_3) \times (V_N \cup V_T \cup \{\epsilon\}) \times V_T^k \rightarrow Q_2 \cup Q_3 \cup Q_4$
5) $g_1: Q_3 \times V_T^k \rightarrow Q_1$, the predictive transition
6) $g_2: Q_1 \rightarrow V_N$

and which satisfy the following properties:

7) no two states in $Q_2 \cup Q_3$ have precisely the same essential items

8) every element of Q_2 is the completion of its essential items

9) if $\sigma \in V_T$, then if $A \rightarrow .\epsilon(\omega)$ and $B \rightarrow \beta_1.\sigma\beta_2(\tau)$ are both items of the same state, then $\omega \notin FIRST_k(\sigma\beta_2\tau)$; if $\sigma \in V_N \cup V_T$, then if $A \rightarrow \alpha.\sigma(\omega)$ and $B \rightarrow \beta_1.\sigma\beta_2(\tau)$ are both items of the same state, then $\omega \notin FIRST_k(\beta_2\tau)$

10) if $A \rightarrow \alpha.\sigma(\omega)$ is an item of q, then $f(q, \sigma, \omega)$ is the member of Q_4 associated with $A \rightarrow \alpha\sigma$; if $A \rightarrow .\epsilon(\omega)$ is an item of q, then $f(q, \epsilon, \omega)$ is the member of Q_4 associated with $A \rightarrow \epsilon$

11) consider the set of items of q of the form $B \rightarrow \beta_1.\sigma\beta_2(\tau)$, where $\beta_2 \neq \epsilon$; then there is a state q' whose essential items are the items $B \rightarrow \beta_1\sigma.\beta_2(\tau)$; and $f(q, \sigma, \omega) = q'$ for each $\omega \in FIRST_k(\beta_2\tau)$, for some item $B \rightarrow \beta_1.\sigma\beta_2(\tau)$ in q

12) $g_1(q,\omega_1) = g_1(q',\omega_2) \Rightarrow q = q'$; that is, each element of Q_1 is the predictive image of at most one base state

13) for any $q \in Q_3$, let $q_1, q_2, \ldots, q_n$ be the different images of q under g_1; and let $q' = q \cup q_1 \cup \ldots \cup q_n$; then q' is an LR(k) state, the completion of the essential items of q, and $(q,((g_2(q_1),q_1),(g_2(q_2),q_2),\ldots,(g_2(q_n),q_n)))$ is a splitting of q'; furthermore, for any i, $1 \leq i \leq n$, $\{x \mid g_1(q,x) = q_i\} = L_k(q', g_2(q_i))$, called the predictive language of $g_2(q_i)$

14) $g_2(q_0) = S$ and q_0 is the completion of S in the context of $\dashv^k$

15) $f(q, \sigma, \omega) = POP$ if $q \in Q_i$, $\sigma = g_2(q)$, and $\omega \in FOLLOW_k(q', \sigma)$, where q' is the state such that $g_1(q', x) = q$ for some x; in addition, $f(q_0, S, \dashv^k) = POP$

16) $f(q, \sigma, x)$ or $g_1(q,x)$ is undefined unless its value is given by one of the above clauses

Admittedly this is a very complex definition, but we have made no claims about the simplicity of this machine model. All we are really after is the formalization of what the result is of performing several state-splittings on an LR(k) machine. Thus we have designed MSP(k) machines so that they hang together in just the right way. In an MSP(k) machine, Q_1 is the set of initial states, Q_2 the intermediates, Q_3 the base states, and Q_4 the final states. Intuitively, an LR(k) machine has one initial state, the set of final states, and all the rest intermediates. Each time a splitting is performed, a base and several initials replace an intermediate. Note however that the essentials of the new base are precisely those of the replaced intermediate, and that in some sense the vanished intermediate could be reconstructed by combining together the base and all the predictives. Intutively, we have formulated the definition of an MSP(k) machine so that it will be satisfied by the result of performing arbitrarily many such replacements of an intermediate by a splitting, after each replacement recomputing successors of the affected states.

Looking at it this way, the starting state is associated with the nonterminal S, and is precisely the starting state of an LR(k) machine for G. By conditions 9, 10, and 11, the successors of q_0 are precisely the same successors that it would have in an LR(k) machine. This is true until we reach a successor that has been "exploded", that has been replaced by a splitting. Condition 13 requires that if some state is not a full fledged intermediate state, but is just a base state with the same essentials as the intermediate that "should" be there, then the base and its associated

initials do indeed define a legal splitting of the missing intermediate. The missing intermediate is, of course, the completion of its essential items, and this is what the base and initials must add up to. The process of computing successors of the base state and its predictive states continues on in the same way.

The other properties specify various constraints to keep the model accurate. Every initial state is associated with some nonterminal, and is attached to at most one base state. The set of strings which cause transfer to be made from a base to one of its attached initials and which cause prediction of the associated nonterminal to be made, must be precisely the lookaheads generated by that nonterminal in the state which was replaced by the splitting. Furthermore, transition is made to POP only from an initial state and only on the nonterminal associated with the initial state and only when the lookahead is in the follow set of the nonterminal with respect to the associated base state (this latter implying that the prediction has been completed).

Finally, the reason that no two states may have the same essential items is that if two LR(k) states have the same essential items, then they are the same state. Thus there can be only one intermediate state with a given set of essential items. Now state-splitting leaves all the essential items in the base state; thus splitting a state can never create a base state with the same essential items as some other state. So if two states do have the same set of essentials, they both must be derived from the same intermediate. But either an intermediate state is replaced by a state-splitting, or it is not; and if it is, it is replaced by only one base state with attached initial

states, not by several bases. Thus merely by performing several successive state-splittings on an LR(k) machine, it is impossible to construct two states with the same essential items; therefore, we prohibit such an eventuality in our model of MSP(k) machines.

For an example of an MSP(1) machine, let G be the LR(1) grammar S → Ax, A → aB, B → BA, B → b. The LR(1) machine for G is shown in Figure 3.20. Recall our conventions for representing LR(k) machines: Final states are represented by circles containing the name of the associated rule, while other states have the component items inside a box. All input strings are assumed to be padded with ⊣ 's, so ⊣ is the context of all S-items. Permissible lookaheads for a transition are written after a slash.

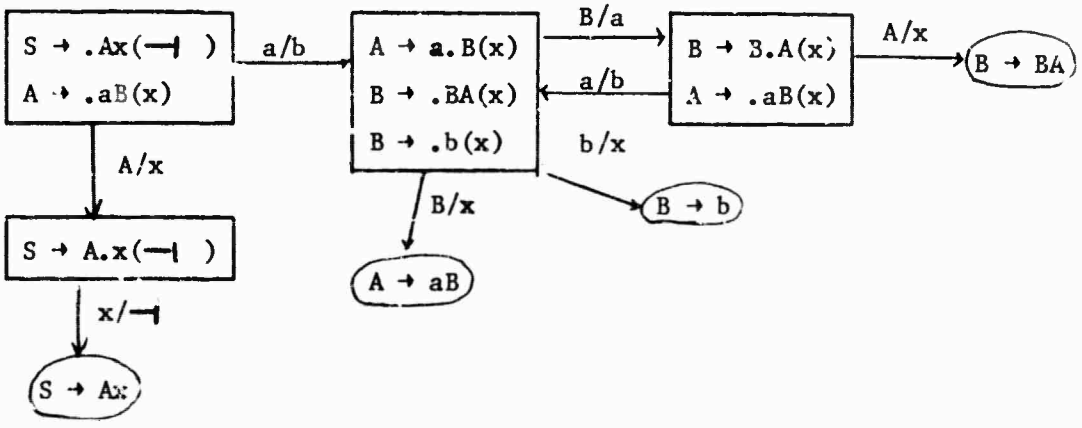
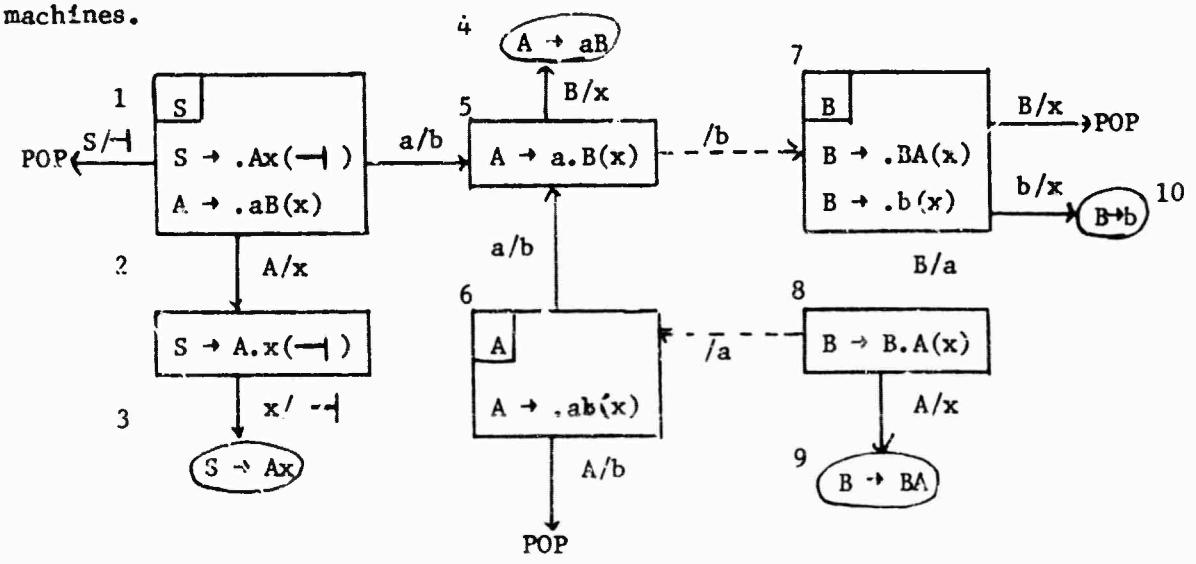


Figure 3.20

Figure 3.21 illustrates one possible MSP(1) machine for the grammar G. There are three initial states, each with its associated nonterminal written in its upper left-hand corner. Two of the initial states are attached to base states by predictive transitions, a predictive transition is represented by a dotted line, and the lookaheads which cause it to be followed are written next to it after a slash. Even though POP appears twice in the

machine, there is only one POP state, which is written three times only to facilitate representation of the machine. We shall continue to use the conventions employed here in our further representations of MSP(k) machines.



MSP(1) Machine for G

Figure 3.21

For the sake of completeness and precision, we will cast this example in the formal terms of the definition. For convenience only, we have numbered the states as indicated in the figure. The initial state set, Q_1, is {1,6.7}; Q_2, the set of intermediate states, equals {2}; Q_3, the base states, is {5,8}; and Q_4, the set of final states, is {3,4,9,10,POP}. The starting state is state 1. The values for the function f are given by the following table:

f(1,a,b) = 5	f(6,a,b) = 5
f(1,A,x) = 2	f(6,A,x) = POP
f(2,x,⊣) = 3	f(7,B,a) = 8
	f(7,B,x) = POP
f(5,B,x) = 4	f(7,b,x) = 10
	f(8,A,x) = 9

For all other q, σ, x triplets, f(q, σ, x) is undefined.

Similarly, g_1 is given by $g_1(5,b) = 7$, $g_1(8,a) = 6$. We also have $g_2(1) = S$, $g_2(6) = A$, and $g_2(7) = B$.

It is not hard to see that this machine does indeed satisfy all the requirements of the definition. By inspection we see that no two states have the same essential items. State 2 is the only member of Q_2, and it has precisely one essential item which is its own completion. Properties 9, 10, and 11 are seen to be true by examining the contents of the states and the nature of the f-values given above. It is also clear that each initial state is the image under g_1 of at most one base state: state 7 is the g_1-image of state 5, 8 of 6, and 1 of no state. Finally, it is easy enough to establish that states 5 and 7 do indeed define a legal state-splitting; that those σ for which $g_1(5,\sigma) = 7$ equals precisely $L_1(g_2(7)) = L_1(B)$; and that $f(7,B,\sigma) = POP$ for the appropriate σ (namely, $FOLLOW_1(5,B)$). Analogous statements can be verified for states 8 and 6.

Intuitively, it is possible to think of this MSP(1) machine as being the result of splitting two of the states of the LR(1) machine of Figure 3.20.

Before we proceed to describe the operation of an MSP(k) machine, we refine the model a little further. We shall only be interested in dealing with MSP(k) machines that have no useless states, that can not be reached from the starting state.

Definition 3.36 Let M be an MSP(k) machine, q and q' states of M. Then q' is immediately accessible from q if $q' = f(q,\sigma,\tau)$ for $\sigma \in V_N \cup V_T \cup \{\epsilon\}$ and $\tau \in V_T^k$, or if $q' = g_1(q,\tau)$ for some $\tau \in V_T^k$. The state q' is accessible from q if there is a sequence of states of M, $q = q_1, q_2, \ldots, q_n = q'$ such that q_{i+1} is immediately accessible from q_i. A state q of M is accessible if it is accessible from q_0, the initial state of M.

Definition 3.37 An MSP(k) machine is reduced if and only if every one of its states is accessible.

We restrict our attention from here on to reduced MSP(k) machines; "MSP(k) machine" will mean "reduced MSP(k) machine".

We note that in a reduced MSP(k) machine, every initial state other than the starting state is guaranteed to be attached to exactly one base state. By definition, it is attached to at most one; while if it is attached to none, it would not be accessible from the starting state, and hence would not be in a reduced machine. The starting state may or may not be attached to a base state.

In order to describe the operation of an MSP(k) machine, we establish the following result.

Lemma 3.38 Let M be an MSP(k) machine for G, q a state of M. Then:

i) $g_1(q,x)$ is single-valued, for any $x \in V_T^k$

ii) $f(q,\sigma,x)$ is single-valued, for any $\sigma \in V_N \cup V_T \cup \{\epsilon\}$ and $x \in V_T^k$

iii) at most one of $f(q, \epsilon, \sigma x/k)$, $f(q, \sigma, x)$, and $g_1(q, \sigma x/k)$ is defined, for any $\sigma \in V_T$ and $x \in V_T^k$

Proof

i) If $g_1(q,x)$ is defined, then q is a base state and $g_1(q,x)$ is one of its associated initial states. By definition, q and its associated initial states form a valid splitting of some state. Suppose that $g_1(q,x) = q_1$ and $g_1(q,x) = q_2$; then $x \in L_k(q',A_1) \cap L_k(q',A_2)$, where $A_i = g_2(q_i)$ and q' is the state of whose splitting q is the base. But by definition of state-splitting, this is an illegal state of affairs.

ii) Suppose $f(q, \sigma, x)$ had two values, for some σ and x. Call these two values q_1 and q_2. If both q_1 and q_2 are final states corresponding to rules of G, these two rules must be $A_1 \rightarrow \alpha_1\sigma$ and $A_2 \rightarrow \alpha_2\sigma$. Then there must be the following items in q: $A_1 \rightarrow \alpha_1.\sigma(x)$ and $A_2 \rightarrow \alpha_2.\sigma(x)$. But this is impossible by condition 9 of the definition.

By condition 11, there is at most one non-final state which is a σ-successor of q, so both q_1 and q_2 can not be non-final. If q_1 is final, associated with $A \rightarrow \alpha\sigma$, and q_2 is non-final, then there must be items $A \rightarrow \alpha.\sigma(\gamma)$ and $B \rightarrow \beta_1.\sigma\beta_2(\omega)$, with $x \in FIRST_k(\beta_2\omega)$, in q, which is again impossible by condition 9.

So the only other possibility is that q_1 is POP and that q_2 is some other state. We know that $f(q_0,S, \dashv^k) = POP$; if $f(q_0,S, \dashv^k)$ equals some other state as we l, there must be an item $A \rightarrow .S(\dashv^k)$ in q_0, where $\$ \overset{*}{\Rightarrow} A$; thus we would have $S \overset{*}{\Rightarrow} S$, which is impossible in an unambiguous

grammar (which every LR(k) grammar is). The only other possibility is that $f(q,\sigma,x) = POP$ iff $\sigma = g_2(q)$ and x is in the k-follow set of σ in the base state which q is associated with. But if $f(q, \sigma, x)$ also equals some other state, there is some item in q, $B \rightarrow \beta_1.\sigma\beta_2(\omega)$, with $x \in FIRST_k(\beta_2\omega)$. Thus $x \in FOLLOW_k(\sigma, q)$ and $x \in FOLLOW_k(\sigma, q')$, where q' is the associated base. But since q and q', together with any other associated initial states, form a state-splitting, this is impossible, by the definition of state-splitting.

iii) The fact that both $f(q, \sigma, x)$ and $f(q, \epsilon, \sigma x/k)$ can not both be defined follows directly from condition 9 of the definition of MSP(k) machines. Suppose then that $f(q, \sigma, x)$ and $g_1(q, \sigma x/k)$ were both defined for some σ and x. Then q must be an element of Q_3 for g_1 to be defined. Then $g_1(q, \sigma x/k)$ is a state in Q_1, say q'; let $A = g_2(g_1(q, \sigma x/k))$, the nonterminal associated with q'. Since $g_1(q, \sigma x/k)$ is defined, by condition 13, $\sigma x/k \in L_k(q'', A)$, where q'' is the state created by combining q and all its associated initial states (all initial states q_1 such that $q_1 = g_1(q, \tau)$ for some τ). Since q and these associated initial states define a splitting of q'', any item in $T(q'', L_k(q'', A))$ will be in q' and not in q (since q' is the initial state associated with A); this is by Corollary 3.33. But if $f(q, \sigma, x)$ is defined, there must be an item $B \rightarrow \beta_1.\sigma\beta_2(\omega)$, $x \in FIRST_k(\beta_2\omega)$, in the state q. But since $\sigma x/k \in L_k(q'', A)$, the item $B \rightarrow \beta_1.\sigma\beta_2(\omega)$ is in $T(q'', L_k(q'', A))$, and so $B \rightarrow \beta_1.\sigma\beta_2(\omega)$ can not be in q, the base state of the splitting of q''. We thus have a contradiction, and our contention is proved.

The same technique shows that $f(q, \epsilon, \alpha x/k)$ and $g_1(q, \alpha x/k)$ can not both be defined, and we are done. Q.E.D.

We are now in a position to describe the operation of an MSP(k) machine.

Definition 3.39 Let M be an MSP(k) machine for G. A stack level is an element of $V_N \cdot Q_1 \cdot (V \cdot (Q_2 \cup Q_3))^* \cdot V_N$; that is, a string of the form $A\ q_0\ x_1\ q_1\ x_2\ q_2 \ldots x_n\ q_n\ B$, where q_0 is an initial state such that $A = g_2(q_0)$, each x_i is in $V_N \cup V_T$, and $n \geq 0$. A topmost stack level is an element of $V_N \cdot Q_1 \cdot (V \cdot (Q_2 \cup Q_3))^* \cdot (\lambda \cup V \cdot (Q_2 \cup Q_3 \cup Q_4))$, that is, a string of the form $A\ q_1\ x_1\ q_1\ x_2\ q_2 \ldots x_n\ q_n$, where $n \geq 0$, each x_i is in $V_n \cup V_T$, and q_n may possibly be a final state. A stack is a sequence of the form $\ell_1 \Delta \ell_2 \Delta \ldots \Delta \ell_m \Delta t$, where $m \geq 0$, each ℓ_i is a stack level, Δ is a new marking symbol, and t is a topmost stack level.

Definition 3.40 A configuration of the MSP(k) machine M is a triple (q, α, ω), where q is a state of M, α is a stack of M, and $\omega \in V_T^* \dashv^k$.

These definitions reflect the operation of the machine as we have discussed it previously. In a configuration (q, α, ω), q represents the state that the machine M is in, α denotes the contents of M's stack, and ω is the remainder of the input string that has not been read yet. (Observe that the input to an MSP(k) machine is always padded with k end-markers.) In practice, the state q will also be the last element on the topmost stack level of α. The organization of the stack accurately mirrors the meanings of the stack levels. Processing proceeds on the topmost level, which resembles any LR(k) stack, until it is time to create a new level. At that time, the name of

the predicted nonterminal is written both on the old level (the last symbol on a stack level) and on the new one (the first symbol on a level, topmost or not). Also the name of the initial state jumped to is written on the new level, this being the element of Q_1 that begins any level. Note that only the last state of the topmost level can be final.

A move by M is represented by the relation $\vdash$ on configurations.

Definition 3.41 Let M be an MSP(k) machine for G. The relation $\vdash$ on configurations of M is defined by:

1) $(q, \alpha, \sigma\omega) \vdash (q_1, \alpha\sigma q_1, \omega)$ if $f(q, \sigma, \omega/k)$ is defined, where $\sigma \in V_T \cup \{\epsilon\}$ and $q_1 = f(q, \sigma, \omega/k)$

2) $(q, \alpha, \omega) \vdash (q_1, \alpha\ A\ \Delta\ A\ q_1, \omega)$ if $g_1(q, \omega/k)$ is defined, where $q_1 = g_1(q, \omega/k)$ and $A = g_2(q_1)$

3) $(q, \beta q_1\tau_1 q_2\tau_2 q_3 \cdots \tau_m q, \omega) \vdash (q', \beta q_1 A q', \omega)$ if q is the final state for the rule $A \rightarrow \tau_1\tau_2\cdots\tau_m$, where $q' = f(q_1, A, \omega/k)$

4) $(q, \beta q_1\ A\ \Delta\ A\ q_2\ A\ q, \omega) \vdash (q, \beta q_1 A q_3, \omega)$ if q is the POP state, where $A = g_2(q_2)$ and $q_3 = f(q_1, A, \omega/k)$

This is just a formalization of the machine operation that we have been describing all along. The first case describes the effect of reading the symbol σ onto the topmost level of the stack. Both the symbol σ and the new state q_1 are written on the stack, and σ is removed from the input. In thi case, the new configuration is called a read-successor of the old configuration.

The second case describes how the configuration reflects the making of a prediction. If the machine was in a base state with the lookahead indicating

that a prediction should be made (that is, if $g_1(q, \omega/k)$ is defined), then the name of the predicted nonterminal is written on the former topmost stack level and a new topmost level is created; on it too is written the name of the predicted nonterminal as well as the appropriate initial state with which that nonterminal is associated. In this case, the new configuration is a prediction successor of the old one.

The third case shows how a reduction is made on the topmost level. If the state of the old configuration is the final state for $A \rightarrow \tau_1\tau_2\ldots\tau_m$, then $\tau_1\ldots\tau_m$ is wiped off the topmost level as well as the interleaved state names, while A and the new state name are written on the stack. The new state is computed using A, the old state uncovered by popping off $\tau_1\ldots\tau_m$, and the lookahead ω/k. Here the new configuration is a reduction successor of the old one.

Finally, the last case shows how a prediction is fulfilled and the topmost stack level eliminated. If the state of the configuration is the POP state, then the topmost stack level is just dispensed with; the new state is computed by dropping back to the next-to-top level, and observing the state q_1 from which the prediction of the A was made. The lookahead used is of course ω/k, the lookahead at the time the A was found and POP entered. We call the new configuration here a suspension successor of the old one.

Let us note that in each of these cases, the new configuration is indeed a well-defined configuration. In particular, the stack component is a properly constructed stack, with the topmost stack level ending with a state as it should.

Theorem 3.42 If M is an MSP(k) machine for the grammar G, then M is deterministic. That is, for any configuration (q, α, ω) there is at most one configuration (q', α', ω') such that $(q, \alpha, \omega) \vdash (q', \alpha', \omega')$.

Proof The state q is either a final state for a production, the POP state, or something else. If it is a final state, only the third clause of the definition of $\vdash$ can hold; since f is single-valued, this means (q, α, ω) will have at most one successor. If q is POP, only the fourth clause can be relevant, and again there can be at most one successor. So assume q is non-final. Then if $(q, \alpha, \omega) \vdash (q', \alpha', \omega')$, the latter must be either a read or prediction successor of the former. Let $\omega = a\rho$, where $a \in V_T$. By Lemma 3.38, $g_1(q, \omega/k)$ is not defined if either $f(q, a, \rho/k)$ or $f(q, \epsilon, \omega/k)$ is. Hence, (q, α, ω) can not have both read and prediction successors. Now if (q, α, ω) does have a prediction successor, it must be unique by the single-valuedness of g_1. On the other hand, both $f(q, a, \rho/k)$ and $f(q, \epsilon, \omega/k)$ are not defined, again by Lemma 3.38; therefore by the single-valuedness of f, (q, α, ω) has at most one read successor, and we are done. Q.E.D.

We will let $\vdash^*$ denote the reflexive, transitive closure of $\vdash$.

Definition 3.43 An initial configuration of M is one of the form $(q_0, Sq_0, \omega \dashv^k)$; that is, one where the state is the starting state of M and the stack has one level consisting of S and q_0. A string ω in V_T^* is accepted by M if $(q_0, Sq_0, \omega \dashv^k) \vdash^* (q, Sq_0Sq, \dashv^k)$, where q is the POP state. That is, if starting M off with ω as the input, M eventually reads all ω and ends up with a one-level stack which shows that S has

been found. The language of M, L(M), is $\{\omega \mid \omega$ is accepted by M$\}$. If $\omega \in L(M)$, then the sequence of configurations $c_0, c_1, \ldots, c_n$, where $c_0 = (q_0, Sq_0, \omega \dashv^k)$, $c_n = (POP, Sq_0 \$ POP, \dashv^k)$ and $c_i \vdash c_{i+1}$, is called the accepting sequence of M for ω. Observe by Theorem 3.42, that this sequence is unique. In general, we let $c_i = (q_i, \alpha_i, \omega_i \dashv^k)$.

3.5 Replacing a State with a Splitting

Now that we have some feeling for the nature and design of MSP(k) machines, let us show that it indeed is a good model for the result of splitting states in an LR(k) machine. That is, we shall show that if we start with an LR(k) machine and split one of its states; take the result and split one of its states; and continue this process, then at every stage we will be dealing with a well-constructed MSP(k) machine. We begin by showing that this process starts off correctly.

Proposition 3.44 If M is the LR(k) machine for G, then M is an MSP(k) machine associated with G.

Proof This is trivial to check. We let Q_1 be just the starting state of M, Q_2 be the nonfinal states, Q_4 the final states, and Q_3 will be empty. Then conditions 8, 9, 10, and 11 are true because G is LR(k) and by the method of construction of the canonical LR(k) machine for a grammar, and the rest of the conditions are either trivial or vacuously true. Q.E.D.

We further note that the conventional way for describing the operation of an LR(k) machine coincides, for all practical purposes, with our specification of how it operates as an MSP(k) machine.

Before we proceed further, we can use some additional terminology.

Definition 3.45 Let q and q' be states of an MSP(k) machine M. Then q dominates q' if the following condition holds: for every sequence of states $q_0, q_1, \ldots, q_n = q'$, starting with the starting state and ending with q' such that q_{i+1} is immediately accessible from q_i, it must be that $q = q_j$ for some j, $0 \leq j < n$.

We note that dominance is transitive: if q dominates q' and q' dominates q", then q dominates q".

Definition 3.46 Let $q_1, q_2, \ldots, q_n$ be a sequence of states of M such that q_{i+1} is immediately accessible from q_i. If $q_{i+1} = f(q_i, \sigma, \tau)$ for σ and τ, let $\sigma_i = \sigma$; if $q_{i+1} = g_1(q_1, \tau)$ for some τ, let $\sigma_i = e$. Then q_n is accessible from q_1 by α, where α is the string $\sigma_1\sigma_2\ldots\sigma_{n-1}$.

This is just an extension of the notion of α-successor of a state, that is common in machine theory. Since a base and its predictives are in some sense a single state, the next state in a sequence to determine an α-successor can either be a successor of a base or a successor of one of its initials. Let us recall the conventional definition of successor.

Definition 3.47 If $q' = f(q, \sigma, \tau)$, then q' is a σ-successor of q; in addition, q is an e-successor of itself. If there is a sequence $q = q_1, q_2, \ldots, q_n = q'$, such that q_{i+1} is a σ_i-successor of q_i, then q' is a <u>α-successor</u> of q, where $\alpha = \sigma_1\sigma_2\ldots\sigma_{n-1}$.

Lemma 3.48 Let M_0 be the LR(k) machine for G, M an MSP(k) machine for G; q_0 is the starting state of M_0, q_0' the starting state of M. Then the

non-final state of M_0 which is an α-successor of q_0 is the union of all non-final states of M which are accessible by α from q_0'.

Proof Let us first observe that in M_0, there is at most one non-final state which is an α-successor of q_0, for any α. This is easy to show by induction on the length of α, since there is at most one non-final state which is a σ-successor of q, for any state q and symbol σ.

Further note that the statement of the theorem assumes that there is a non-final α-successor of q_0 in M_0 if and only if some non-final states are accessible by α from q_0' in M. This statement will come for free in the proof of the lemma.

The proof is by induction on the length of α. If $|\alpha| = 0$, then $\alpha = \epsilon$. In M_0, there are no ϵ-transitions from q_0 to a non-final state, since ϵ labels only transitions to final states for rules like $A \to \epsilon$; hence the single non-final state of M_0 in this case is q_0. Similarly in M, there are no ϵ-transitions from q_0' to non-final states; and since q_0' is an initial state, it has no attached initial states. Hence the only non-final state accessible from q_0' by ϵ in M, is q_0' itself. But $q_0 = q_0'$ by construction of M and M_0, so this case holds.

Suppose then that the statement is true for $|\alpha| = n$; we shall show it true if $|\alpha| = n+1$. Let $\alpha = \beta\sigma$, where $|\beta| = n$ and $|\sigma| = 1$. Then a state of M_0 is an α-successor of q_0 if and only if it is a σ-successor of a β-successor of q_0. By our remarks on ϵ-transitions, q is a non-final α-successor of q_0 if and only if $q = f(q_1,\sigma,\tau)$, where q_1 is a non-final β-successor of q_0. By induction, there are non-final states of M, $q_1', q_2', \ldots, q_m'$, each of which is accessible by β from q_0', such that

$q_1 = \bigcup q_i'$. That means that the items of q_1 are somehow spread out among the q_i'.

Now consider the state q. It is the completion of its essential items, which in turn are formed by moving the dot one place to the right on all items of q_1 of the form $A \rightarrow x.\sigma y(\omega)$, where $y \neq \epsilon$. These items occur somewhere in the q_i'; there will be a state accessible by σ from some particular q_i' if and only if there are one or more such $.\sigma$ items in it. The non-final states accessible by σ from such a q_i' include the non-final σ-successor of q_i', and any attached initial states, if that successor is a base state. Thus the union of the non-final states accessible by σ from q_i' form the completion of the essential items of the σ-successor of q_i'. Since $q_i' \subset q_1$, this union is contained in the σ-successor of q_1; hence the union of all these accessible states, for each q_i', is contained in q as well. But any item of q is a descendant of one of the essential items of q. Any essential item of q is contained in the σ-successor of some q_i'; any descendant of this essential item is either in this successor or in one of its predictives. In either event, any item of q is contained in a non-final state accessible from q_0' by $\beta\sigma = \alpha$; and so q equals the union of these states and we are done. Q.E.D.

Corollary 3.49 Every non-final state of an MSP(k) machine for G is contained in some state of the LR(k) machine for G.

We now describe the process by which a state of a machine may be replaced by a split version of itself.

Algorithm 3.50 Let M be an MSP(k) machine for G, $q \in Q_2$ an intermediate state of M, and (B,Q) a splitting of q. Then we replace q by (B,Q) by performing the following steps:

I) add B to Q_3 and each P_i in Q to Q_1; define $g_2(P_i) = A_i$ and $g_1(B,x) = P_i$ for each $x \in L_k(q,A_i)$; define $f(P_i,A_i,\omega) =$ POP if $\omega \in$ FOLLOW$_k(A_i, B)$.

II) remove the state q from the set Q_3; for any q_1 in M, if $f(q_1,\sigma,\omega) = q$, set $f(q_1,\sigma,\omega) = B$

III) designate B and each P_i as a "new state"; recursively apply step IV to B and each P_i and to any other "new states" created by step IV

IV) let q_1 be a new state;

a) for each symbol σ, let E_σ be the items in q of the form $A \to \alpha.\sigma\beta(\omega)$, where $\beta \neq \epsilon$, and let E'_σ be the corresponding set of items $A \to \alpha\sigma.\beta(\omega)$;

b) define q' as follows: if E'_σ equals the set of essential items of any state q" currently in Q_2 or Q_3, then q' = q"; otherwise, q' is the completion of the set of essential items E'_σ, it is added to the set Q_2 and it is also defined as a "new state";

c) then for any τ such that $\tau \in$ FIRST$_k(\beta\omega)$ for some item $A \to \alpha.\sigma\beta(\omega)$ in E_σ, let $f(q_1,\sigma,\tau) = q'$;

d) and for any ω such that $A \to \alpha.\sigma(\omega)$ is an item of q, let $f(q_1,\sigma,\omega)$ equal the final state associated with the grammar rule $A \to \alpha\sigma$;

e) after the definition of $f(q_1,\sigma,\omega)$ for each σ and ω, remove q_1 from the list of new states.

V) eliminate all states in Q_1, Q_2, or Q_3 which are no longer accessible from q_0; the modified values of Q_1, Q_2, Q_3, f, g_1, and g_2, together with Q_4 and q_0, define the <u>result</u> of this algorithm.

This replacement algorithm is quite straightforward. The old state is discarded, replaced by the base state which is appropriately attached to its associated initials. All transitions into the old state are reconnected to the base of the splitting. Then successors of the base and initials are recursively computed in the conventional way, with one exception: should the conventional successor of a state be an intermediate state which has already been replaced by a state-splitting, the actual successor is instead chosen to be the base of that state-splitting. This is to prevent re-introduction of a state that has already been eliminated, and to avoid having two states with identical essential items. We note in passing, that it is both possible and legal to have different initial states which consist of precisely the same set of items; because initial states do not have any essential items, this would not violate the definition. However, if two different initial states do have the same set of items, their successors will be the same states. We shall see later when it is possible to merge different copies of the same initial state.

After all successors of the newly introduced states have been computed, all leftover states from the original machine which can no longer be reached in the new version are eliminated. Thus it is possible for one state-splitting to "undo" the work of another. For example, suppose q_2 is the σ-successor of q_1, and that both q_1 and q_2 can be split. If we first split q_2, then the

base of the splitting will be the new σ-successor of q_1. But if we then split q_1, it may happen that the .σ items of q_1 are so scattered among the base and initials of the splitting of q_1 that the E_σ of q_1 items are not all together in some state. Then the σ-successors of the base and initials of the q_1-splitting will each contain just some of the essentials of q_2, and these successors will be computed as new states. And unless there are other transitions to it, the base state of the splitting of q_2, together with its associated initial states, will vanish. In general, if q dominates q', then the splitting of q may make q' no longer accessible, leading to its elimination from the machine.

We must show two things about Algorithm 3.50 in order for it to be meaningful. We have to prove that it always terminates and that the result of performing the replacement as specified by the algorithm will be a well-constructed MSP(k) machine. The fact that it terminates is immediate, since the algorithm iterates on Step IV only so long as new states are being produced; and there can be only finitely many new states, since there are only finitely many collections of LR(k) items altogether, and once a new state has been treated it is never generated again as a new state.

In order to show that the procedure is well-defined and that the result is a valid MSP(k) machine, we really need establish one result: that each new state generated during the algorithm satisfies Clause 9 of the definition of MSP(k) machines; namely that if $A \to .\sigma(\omega)$ and $B \to \beta_1.\sigma\beta_2(\tau)$ are items of the state, with $\sigma \in V_T$, then $\omega \notin \mathrm{FIRST}_k(\sigma\beta_2\tau)$; and that if $A \to \alpha.\sigma(\omega)$ and $B \to \beta_1.\sigma\beta_2(\tau)$ are items, then $\omega \notin \mathrm{FIRST}_k(\beta_2\tau)$. This suffices because replacing q by a splitting does not affect those states not dominated by q,

and adds only new intermediate states to the machine. But this required result is very easy to establish. Since q is a state of M, it is contained in q', some state of the LR(k) machine for G; hence the same is true of each component of the splitting. Each new state is the α-successor of some component of the splitting for some α, and is readily seen to be contained in an α-successor of q'. Since clause 9 holds for any LR(k) state, it holds for any subset of one, and hence for each new state created by the algorithm.

Now we consider the other side of the coin.

Theorem 3.51 Let M be any MSP(k) machine with n base states, $n > 0$. Then there exists an MSP(k) machine M' with n-1 base states, and a state q of M' and a splitting (B,Q) of q, such that M is the result of replacing q in M' by the splitting (B,Q).

Proof We will describe how to construct the machine M' from M. Let the base states of M be $q_1, q_2, \ldots, q_n$. Let q_j be any one of these base states which does not dominate any other one. (If every base state dominates some other one, then some state must domainate itself since there are only finitely many, and that is ridiculous.) Since q_j is a base, it is the base state of some splitting (B,Q). To get M' from M we will "undo" this splitting, replacing q_j by the state of which (B,Q) is a splitting.

We let $q = q_j + \bigcup P_i$; that is, the union of all the items of the components of the splitting (B,Q). We observe that the essentials of q are the same as those of q_j. In order to get M' from M, we do the following: eliminate states q_j and P_i, for each P_i in the splitting, set all transitions formerly going into q_j to go into q'; compute the successors of q in the conventional way; and finally, eliminate any other states which become

inaccessible. When we say "compute the successors in the conventional way", we mean that the essential items of any successor state are compared with the essential items of existing states: if there is a match, that state becomes the successor; otherwise, a new intermediate state is generated.

It is straightforward to show that M' is indeed a well-defined MSP(k) machine. This new machine M' differs from M only with respect to the state q_j and its successors. That is, all states of M that can be reached from q_0 without going through q_j (i.e., all states not dominated by q_j) are in M' as well, and connected in the same way. In particular, all the base states of M other than q_j are in M' too. Since no new base states are generated in computing successors of the new state q, it is true that M' has only n-1 base states.

Now let us consider the result of replacing q in M' by the splitting (B,Q). By construction of q, (B,Q) is indeed a splitting of q. Those states not dominated by q_j in M are not dominated by q in M', so those states are unaffected by replacing q_j by q, and subsequently replacing q by the splitting (B,Q). The new states introduced into M' as successors of q, are no longer accessible once q is split into (B,Q); and the states eliminated when q_j and the P_i are eliminated, are reintroduced when (B,Q) replaces q in M'. All connections among the states of M that are severed when q_j and the P_i are removed are reconnected when they are reintroduced. Thus M indeed results when q is replaced by (B,Q) in M'. Q.E.D.

Theorem 3.52 Let M be any MSP(k) machine for G with n base states. Then there is a sequence of MSP(k) machines for G, $M_0, M_1, \ldots, M_n$, such that M_0 is

the LR(k) machine for G, $M_n = M$, and M_{i+1} is obtained by replacing some state of M_i by a splitting of it.

Proof Immediate from the preceding theorem. Q.E.D.

This last theorem justifies our definition of MSP(k) machines. We have already seen that starting with an LR(k) machine, and performing a series of state-splittings, always results in an MSP(k) machine. Now we see that every MSP(k) machine can be so obtained. Thus the notion of MSP(k) machine precisely captures the notion we were striving for, namely the result of applying a number of state-splittings to an LR(k) machine. This is a very good state of affairs, for while our model of MSP(k) machines may be a complex one and hence MSP(k) machines may be difficult to design from scratch, replacing a state by a splitting is an elementary procedure and one that is easy to perform repeatedly, starting with the LR(k) machine. Thus it is enough for us to know how to split states in order to build MSP(k) machines.

As an example of how this state-splitting process works, consider the LR(1) machine of Figure 3.22. Each of the three intermediate states can be split, some in several ways. The result of sequentially replacing two of these states by splittings is shown in Figure 3.23; we do not bother to show how this is done in two steps since these two states occur in separate areas of the machine.

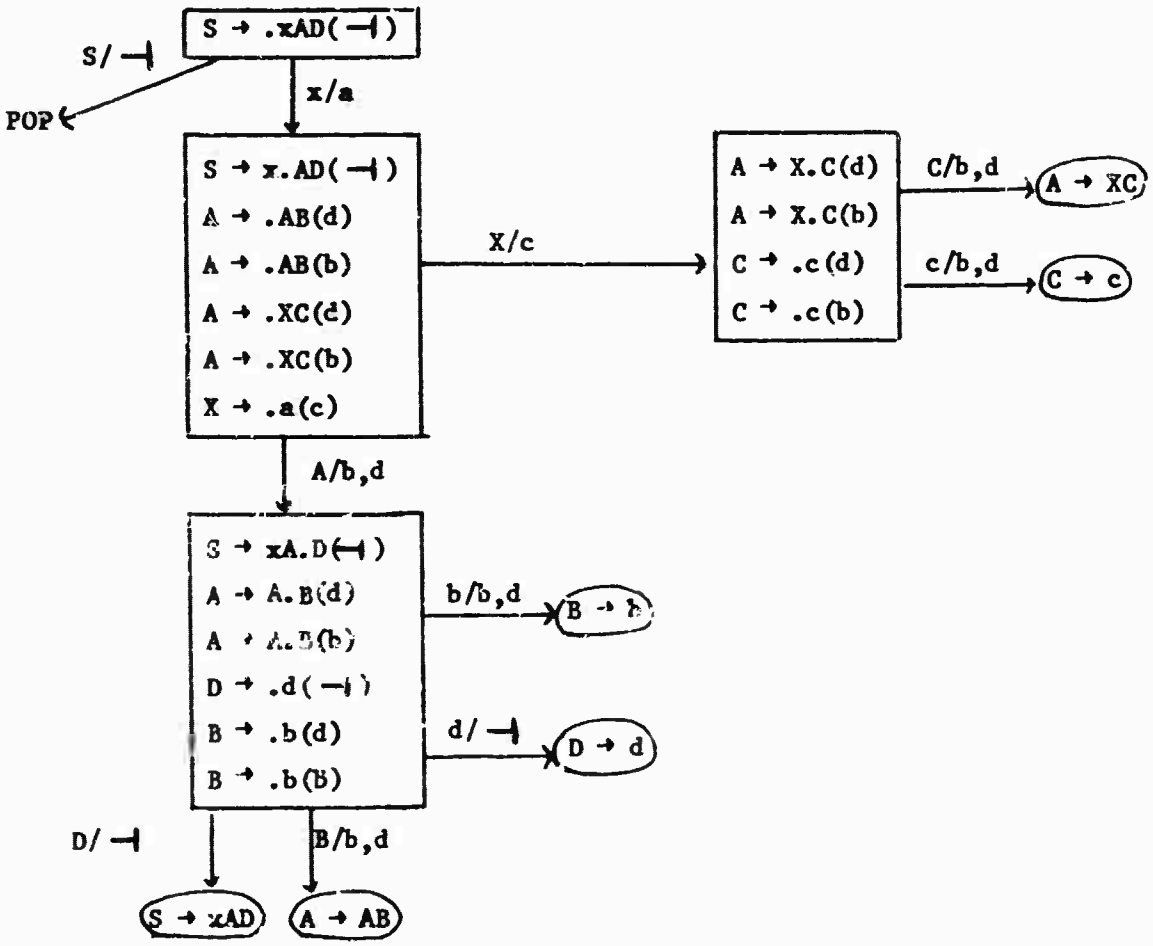

S → .xAD(⊣)
S/⊣
POP
x/a
S → x.AD(⊣)
A → .AB(d)
A → .AB(b)
A → .XC(d)
A → .XC(b)
X → .a(c)
X/c
A → X.C(d)
A → X.C(b)
C → .c(d)
C → .c(b)
C/b,d
A → XC
c/b,d
C → c
A/b,d
S → xA.D(⊣)
A → A.B(d)
A → A.B(b)
D → .d(⊣)
B → .b(d)
B → .b(b)
b/b,d
B → b
d/⊣
D → d
D/⊣
B/b,d
S → xAD
A → AB

Figure 3.22

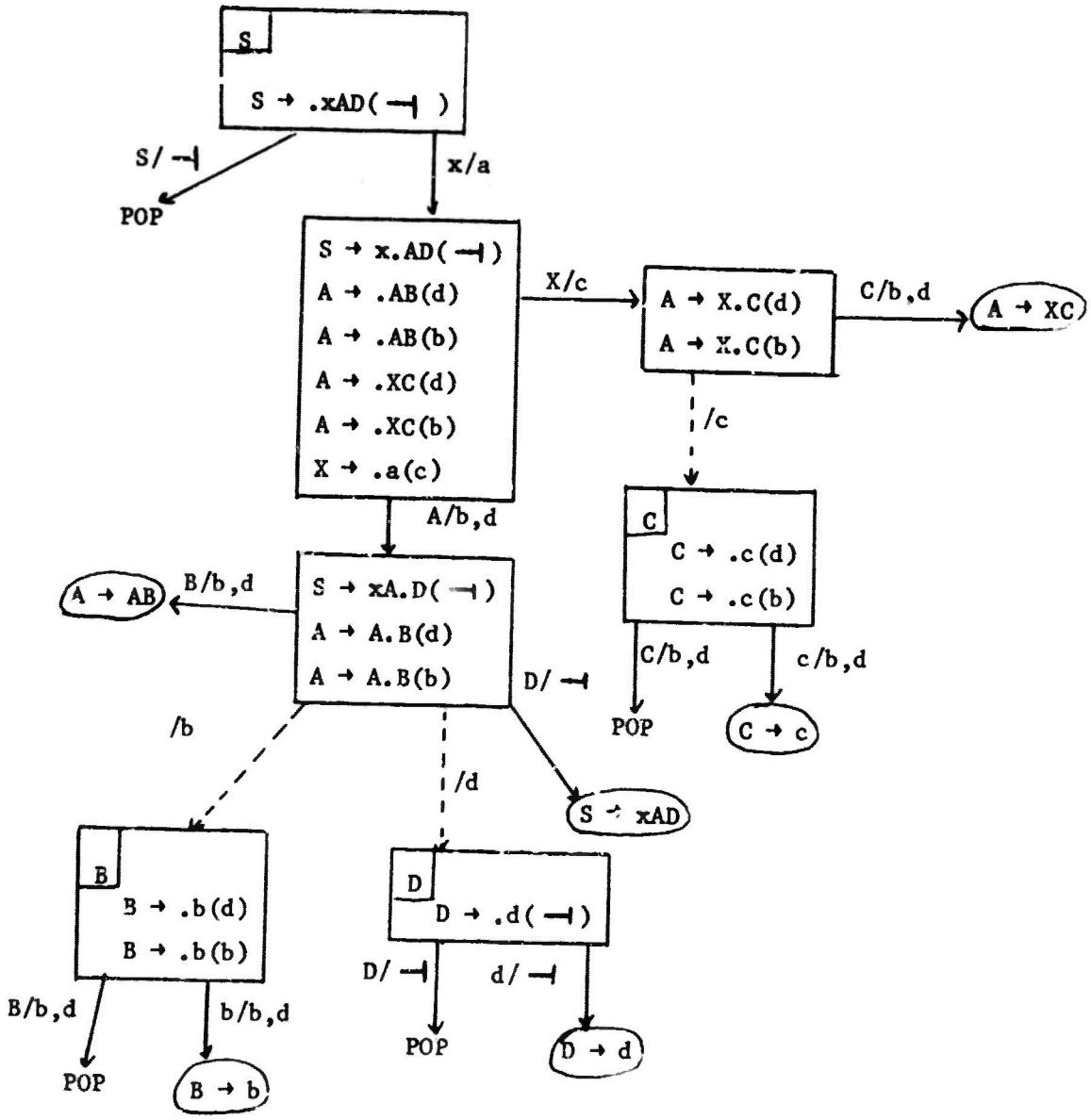

S
S → .xAD(⊣)
S/ ⊣
POP
x/a
S → x.AD(⊣)
A → .AB(d)
A → .AB(b)
A → .XC(d)
A → .XC(b)
X → .a(c)
X/c
A → X.C(d)
A → X.C(b)
C/b,d
A → XC
/c
A/b,d
C
C → .c(d)
C → .c(b)
C/b,d
c/b,d
POP
C → c
A → AB
B/b,d
S → xA.D(⊣)
A → A.B(d)
A → A.B(b)
D/ ⊣
/b
/d
S → xAD
B
B → .b(d)
B → .b(b)
D
D → .d(⊣)
D/ ⊣
d/ ⊣
B/b,d
b/b,d
POP
B → b
POP
D → d

Figure 3.23

Now suppose that we want to split the remaining intermediate state of this MSP(1) machine, and that we choose a splitting such that the only item in the base state of the splitting is the essential item S → x.AD(⊣). (It is straightforward that this is indeed a legal splitting.) Then the result of this splitting is shown in Figure 3.24.

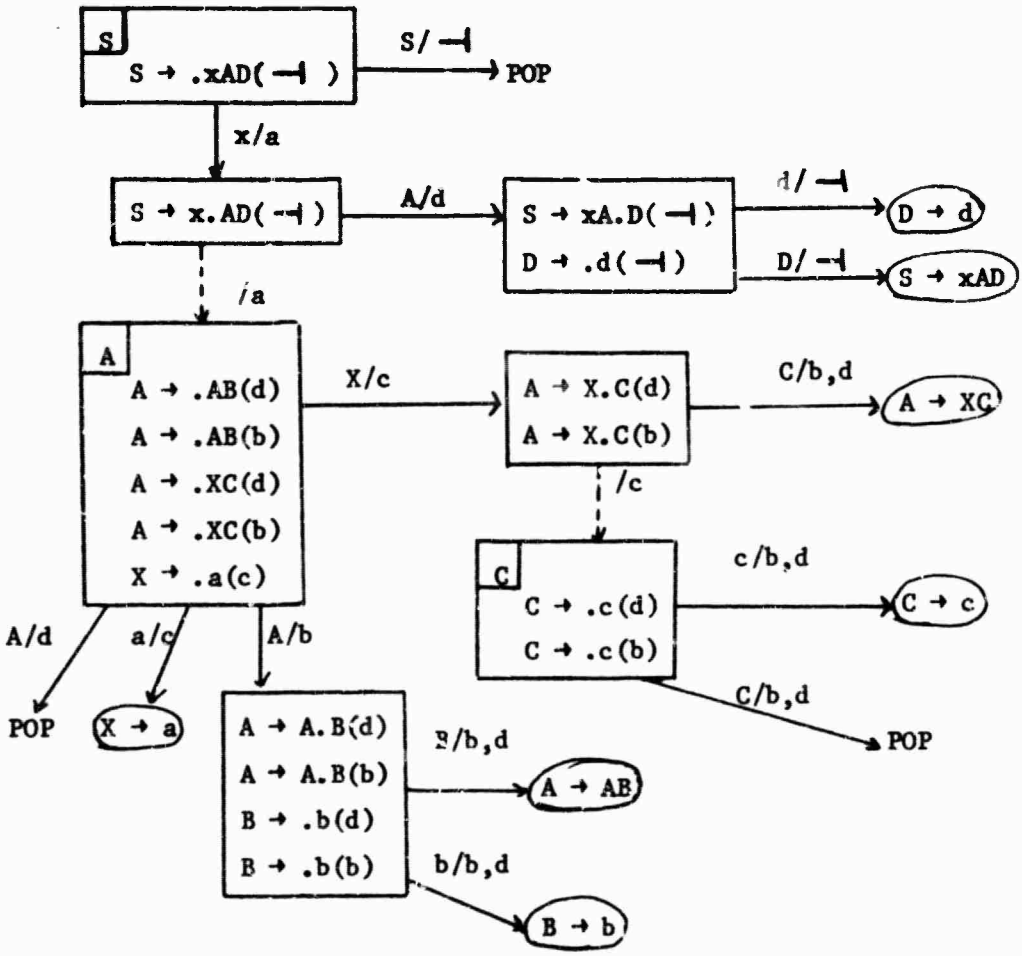


Figure 3.24

Splitting this state has effected changes in the machine. The A-successors of the base and predictive states of this splitting are new states, which add up to the A-successor of the state being split. Moreover, the A-successor of the state being split here (which itself was the base state of a splitting) is no longer accessible once the splitting has been done; so it and its attached predictive states are eliminated from the machine. Thus one splitting can "undo" the work of a previous one. Our latter results, especially Theorem 3.52, assure us that this resulting MSP(1) machine could have been achieved in a less roundabout fashion, by making just two splittings on the LR(1) machine; rather than three, which interfere with each other.

3.6 Parsing with MSP(k) Machines

It is now our purpose to show that the language accepted by an MSP(k) machine for the grammar G is precisely L(G), the language generated by G. This will confirm our design of MSP(k) machines as alternate parsing machines for G. We shall establish this result by showing that $L(M) = L(M_0)$, where M_0 is the canonical LR(k) machine for G. Since it is well known that $L(M_0) = L(G)$, this will give us our desired theorem. The proof of this assertion is fairly tedious, and uses standard "twin-machine" proof techniques: we show how to construct from an accepting sequence in M_0, a corresponding sequence in M, and vice versa. First, we need a number of preliminary definitions and results about the way an LR(k) machine operates.

Definition 3.52 Let M_0 be an LR(k) machine, q a non-final state of M_0, and $c_0, c_1, \ldots, c_n$ an accepting sequence of configurations of M_0 for the string ω. Then an entry to q in the sequence is any stack α_i of the form βXq, $X \in V_N \cup V_T$.

We note that there may be many entries to q in the course of a parse; an entry to q is simply the occasion on which it is transferred to, either as the result of a read or after a reduction.

Definition 3.53 Let βXq be an entry to q in an accepting sequence; the corresponding exit from q is the first stack α_j after the entry, of which βXq is not a proper prefix.

Since we are dealing with an accepting sequence of configurations, the last stack in the sequence will not have βXq as a prefix, and so there is an exit for each entry. The exit corresponding to an entry is the first stack in which that entry no longer figures; that is, it is the point where a reduction corresponding to one of the essential items of q is made.

Definition 3.54 Let $\alpha_i = \beta Xq$ be an entry to q, and α_j the corresponding exit. We say that the nonessential item $A \to .\Psi(\tau)$ is recognized by this entry to q if there is a stack α_m, $i < m < j$, such that $\alpha_m = \beta Xq\Psi' q'$ where q' is the final state for $A \to \Psi$ and Ψ' consists precisely of the symbols of Ψ alternating with state names and where $\omega_m \dashv\!\vdash^k /k = \tau$.

We observe that if these conditions hold for α_m then the next step performed by M_0 will be to reduce Ψ to A; so $\alpha_{m+1} = \beta X q A q''$, for some q". We also have the following:

Lemma 3.55 If an entry to q recognizes the item $A \rightarrow .\Psi(\tau)$, then $A \rightarrow .\Psi(\tau)$ is an item of q.

Proof This follows readily from the way an !R(k) machine operates and the way its states are composed. Suppose stack α_m is $\beta X q \Psi' q'$, where $\Psi' = \Psi_1 q_1 \Psi_2 q_2 \cdots q_{r-1} \Psi_r$. Then $q' = f(q_{r-1}, \Psi_r, \omega_m \dashv^k / k)$; therefore $q' = f(q_{r-1}, \Psi_r, \tau)$. Therefore there must be an item $A \rightarrow \Psi_1 \Psi_2 \cdots \bullet \Psi_r(\tau)$ in q_{r-1}. Working backwards through Ψ, we see that $A \rightarrow \Psi_1 \cdots \bullet \Psi_i \cdots \Psi_r(\tau)$ is an item of q_{i-1}. Thus $A \rightarrow .\Psi(\tau)$ is an item of q. Q.E.D.

Lemma 3.56 Let $\alpha_i = \beta X q$ be an entry to q; suppose $A \rightarrow .\Psi(\tau)$ is some non-essential item recognized by q, and that $B \rightarrow .\varphi(\rho)$ is the next non-essential item recognized by q. Then $A \rightarrow .\Psi(\tau)$ is an immediate descendant of $B \rightarrow .\varphi(\rho)$.

Proof At the time $A \rightarrow .\Psi(\tau)$ recognized, the stack is $\beta x q \Psi' q'$, which then becomes $\beta X q A q''$. The only way for this A to be removed from the stack is by the performance of a reduction that exposes either q or some state lower down in the stack; but this can not happen before the next non-essential item is recognized by q. So when $B \rightarrow .\varphi(\rho)$ is found, the A is still on the stack. But since recognition of $B \rightarrow .\varphi(\rho)$ entails popping off φ and exposing q, we have that A is the first symbol of φ. Thus the two items are really $A \rightarrow .\Psi(\tau)$ and $B \rightarrow .A\varphi(\rho)$. To show the former is an immediate descendant of the latter, we need only show $\tau \in \text{FIRST}_k(\varphi\rho)$.

To show this, let us let ω_1 be the remaining input at the time $A \to .\Psi(\tau)$ is recognized and ω_2 the remaining input when $B \to .A\varphi(\rho)$ is found. Observe that ω_2 is a suffix of ω_1; some prefix of ω_1, call it ω_3, has been read between those two recognitions. That is, $\omega_1 = \omega_3\omega_2$ and what has occurred is that ω_3 has been reduced to φ by the machine. By a well-known result of LR(k) machines, this means that $\varphi \overset{*}{\Rightarrow} \omega_3$. We know that $\tau = \omega_1/k$ and that $\rho = \omega_2/k$. If $|\omega_3| \geq k$, then τ is a prefix of ω_3, and so $\tau \in FIRST_k(\varphi)$. If $|\omega_3| < k$, we can set $\tau = \omega_3\omega_4$, where ω_4 is a prefix of ω_2 of length less than or equal to k. Since ρ is the prefix of ω_2 of length k, this means ω_4 is a prefix of ρ; hence $\tau \in FIRST_k(\varphi\rho)$. In either case, we are done. Q.E.D.

We want to extend these ideas to the concept of an entry to a state recognizing an essential item.

Definition 3.57 Let $\alpha_1 = \beta Xq$ be an entry to state q, and α_j the corresponding exit. We say that the essential item $A \to \varphi.\Psi(\tau)$ is recognized by this entry if there is a stack α_m, $i < m < j$, such that $\alpha_m = \beta Xq\Psi'q'$, where Ψ' consists of the symbols of Ψ alternating with state names and q' is the final state for $A \to \varphi\Psi$, and $\omega_2 \overset{k}{\to} /k = \tau$.

Note that if these conditions are met, α_{m+1} is obtained by removing all of Ψ' and some of βXq from the stack, thus exposing some state lower down in the stack, and applying A to that state. In other words, α_{m+1} will be the exit from q, and so it is α_j. Thus m = j-1. We note from this that if α_j is the exit corresponding to entry α_i, then α_{j-1} satisfies the above definition as recognizing some essential item. Thus we have:

Lemma 3.58 If $\alpha_i = \beta X q$ is an entry to state q, then it recognizes exactly one essential item $A \rightarrow \varphi.\Psi(\tau)$.

Furthermore, there is the following analogy to Lemma 3.55.

Lemma 3.59 If an entry to q recognizes the essential item $A \rightarrow \varphi.\Psi(\tau)$, then $A \rightarrow \varphi.\Psi(\tau)$ is an item of q.

Proof Similar to the proof of Lemma 3.55.

Lemma 3.60 If an entry to q recognizes the essential item $A \rightarrow \varphi.\Psi(\tau)$, and if the last non-essential item it recognizes is $B \rightarrow .\eta(\rho)$, then $B \rightarrow .\eta(\rho)$ is an immediate descendant of $A \rightarrow \varphi.\Psi(\tau)$

Proof Similar to the proof of Lemma 3.56.

Lemma 3.61 The first item recognized by an entry to q is a terminal item.

Proof If $\alpha_i = \beta X q$, α_{i+1} must be $\beta X q \sigma q'$, where σ is ϵ or an element of V_T, since q is non-final; thus by definition, σ will be the first symbol of the first item found by this entry. Q.E.D.

Lemma 3.62 If $\alpha_i = \beta x q$ is an entry to state q, and $A \rightarrow .\alpha\rho(\tau)$ is the first item recognized by the entry, then $\omega_i \dashv^k / k \in \text{FIRST}_k(\alpha\varphi\tau)$.

Proof Similar to the second half of the proof of Lemma 3.56.

Theorem 3.63 Suppose α_i is an entry to state q, and that $\omega_i \dashv^k / k \in L$. Then the items recognized by α_i form an L-chain through the state q.

Proof By the preceding lemmas and the definition of an L-chain. Q.E.D.

This is the first intermediate result toward which we have been heading. We have succeeded in formalizing the notions we discussed earlier, of how an LR(k) state is used during the course of a parse, and have shown the utility of our notion of a chain. We all proceed to apply this result in order to show that $L(M_0) \subset L(M)$.

Definition 3.64 Let M_0 be the LR(k) machine for G, M some MSP(k) machine for G. Suppose $\omega \in L(M_0)$, and $c_1, c_2, \ldots, c_n$ is the accepting sequence of configurations of M_0. Then we define a sequence of configurations of M, $h(c_1), \ldots, h(c_n)$, as follows: $h(c_1)$ is the initial configuration of M for ω; $h(c_{i+1})$ is the first succesor configuration of $h(c_i)$ whose state is neither POP nor an initial state.

We want the sequence $h(c_i)\ldots h(c_n)$ to keep track of where M_0 is in the sequence $c_1 \ldots c_n$. Predictions and suspensions are extra steps done by M when compared with M_0, so the results of these operations are skipped over. As of yet, we have no assurance that $h(c_i)$ is well defined; but if it is, its state is either a base, an intermediate, or a final state. We shall show that this state of $h(c_i)$ can do exactly what the state of c_i is called upon to do in this accepting sequence.

Definition 3.65 If ℓ is a stack level, then ℓ is a string of the form $A\, q_0 x_1 q_1 x_2 q_2 \cdots x_n q_n B$, where $n \geq 0$; then CON(ℓ), the contents of ℓ, is the string $x_1 x_2 \cdots x_n$. If t is a topmost stack level, then t is of the form $A\, q_0 x_1 q_1 \cdots x_n q_n$; then CON(t) is the string $x_1 \ldots x_n$. If α is a stack,

then $\alpha = \ell_1 \Delta \ell_2 \Delta \ldots \Delta \ell_m \Delta t$; LIN($\alpha$), the linearization of α, is the string $CON(\ell_1) \cdot CON(\ell_2) \cdots CON(\ell_m) \cdot CON(t)$, the concatenation of the contents of its stack levels.

The contents function ignores the state names, as well as the names of the nonterminals predicted on any level.

Proposition 3.66 If $\omega \in L(M_0)$, M is an MSP(k) machine for G, and $h(c_1),\ldots,$ $h(c_n)$ is defined as above, then for each i, $1 \leq i \leq n$:

i) $h(c_i)$ is a well-defined configuration of M, $(q'_i,\alpha'_i,\omega_i' \dashv^k)$
ii) $\omega_i = \omega_i'$
iii) $LIN(\alpha_i) = LIN(\alpha_i')$
iv) if q_i is a final state, q_i' is the same final state
v) if q_i is non-final then $q_i' \subset q_i$; furthermore, $I \in q_i'$, where I is the item of q_i recognized by α_i, which is an entry to q_i

Proof We shall prove this by induction on i. The last two clauses of the propostion are the ones that "carry" the proof, making the others immediate.

Basis i = 1; This is straightforward by the definition of $h(c_1)$.

Induction Suppose the statement is true for $i \leq m$; we shall show it true for i = m+1. Since M_0 is an LR(k) machine, c_{m+1} can be either a read or a reduce successor of c_m.

Let us first consider the case where c_{m+1} is a read successor of c_m. In this case, q_m must be an intermediate state, and q_m' is a base or intermediate such that $q_m' \subset q_m$. In going from c_m to c_{m+1}, M_0 reads the first symbol (call it a) from the remaining input ω_m. As we saw in Lemma 3.61, the first item recognized by c_m, an entry to q_m, will be an item

$A \rightarrow .a\Psi(\tau)$; and by Theorem 3.63, this item will be a descendant of I, the essential item recognized by c_m.

If q_m' is an intermediate state, since $I \in q_m'$, $A \rightarrow .a\Psi(\tau)$ will be a member of q_m' as well, since it is in the completion of I. There may be other .a items in q_m' as well, but they will all be in q_m too. So since $f_{M_0}(q_m,a,\rho)$ is defined, where $a\rho = \omega_m \dashv^k/k+1$, $f_M(q_m',a,\rho)$ will also be a well-defined state. Thus machine M, when in the configuration with q_m' as the state and remaining input $\omega_m' = \omega_m$, will have a well-defined successor, gotten by reading the symbol a from the input. Thus $h(c_{m+1})$ is well-defined. We had $\omega_m = \omega_m'$, both headed by an a; ω_{m+1} and ω'_{m+1} are gotten from their predecessors by removing the first symbol, so $\omega_{m+1} = \omega'_{m+1}$ as well. Furthermore, α_{m+1} is the same as α_m, except for the addition of the symbol a and one more state name; since an analogous statement is true for α'_{m+1}, and since $LIN(\alpha_m) = LIN(\alpha'_m)$, we have $LIN(\alpha_{m+1}) = LIN(\alpha'_{m+1})$.

If q_{m+1} is not a final state, then the essential item it recognizes is is $A \rightarrow a.\Psi(\tau)$, which by construction will be in q'_{m+1} too. Furthermore, since $q_m' \subset q_m$, it is immediate that the a/ρ successor of q_m' is contained in the a/ρ successor of q_m. If q_{m+1} is the final state for $A \rightarrow a$, then q'_{m+1} will be the same state, and so this sub-case is done.

Now we must consider the possibility that q_m' was a base state, rather than an intermediate; this is still under the assumption that c_{m+1} is a read-successor of c_m. Let (B,Q) be the splitting of which q_m' is the base state, where $Q = \{(A_i,P_i)\}$. There are two possibilities: either $\omega_m \dashv^k/k$ is an element of the predictive language of A_i, for some i; or it is in none of the predictive languages. If it is in the predictive language for the nonterminal A_i, then the first item recognized by q_m, which we shall call

$A \to .a\Psi(\tau)$, is an $L_k(A_i)$ terminal item and also a descendant of I, the essential item recognized by the α_m entry to q_m. Since I is in q_m', the base state of the splitting (B,Q), and since $A \to .a\Psi(\tau)$ is a descendant of I, this item will have to be in one of the components of the splitting. Since it is an $L_k(A_i)$ - terminal item, it will be in P_i, the predictive state for A_i. Now consider the actions of M after entering configuration $h(c_m)$. The lookahead is $\omega_m' \dashv^{\kappa}/k$; by hypothesis, this equals $\omega_m \dashv^{k}/k$, which is in the A_i predictive language. So since q_m' is the base state of the splitting (B,Q), M will then create a new level and predict A_i, jumping to state P_i. The state P_i is a predictive state, so even though $h(c_m)$ has a successor c', we can not let it be $h(c_{m+1})$. But now observe the next action of M. It is in state P_i, which is a predictive state associated with base q_m'. Since $q_m' \subset q_m$, we have $P_i \subset q_m$ as well. The item $A \to .a\Psi(\tau)$ is in q_m as well as in P_i. Since it is the first item recognized by q_m, $f_{M_0}(q_m,a,\rho)$ is well-defined, where $a\rho = \omega_m \dashv^{k}/k+1$; so $f_M(P_i,a,\rho)$ is also well-defined. But $a\rho$ are exactly the k+1 symbols of lookahead M will see when it jumps to P_i. Thus this configuration c' has a successor as well, which it gets to by reading the symbol a onto the stack and following the a/ρ-transition out of P_i. This configuration will be by definition $h(c_{m+1})$, and it is easy to see that it satisfies all the required properties, as follows.

Either q_{m+1} is the final state for $A \to a$ or the essential item it recognizes is $A \to a.\Psi(\tau)$; in the first case, q'_{m+1} is the same state, while in the other q'_{m+1} contains $A \to a.\Psi(\tau)$ as well. In the latter case, since q'_{m+1} is the a/ρ-successor of P_i, which is contained in q_m, we have $q_{m+1}' \subset q_{m+1}$, which is the a/ρ-successor of q_m. The remaining inputs, ω_{m+1} and

ω_{m+1}', are the same, since they are gotten from ω_m and ω_m' by eliminating the first symbol. Finally, α_{m+1} differs from α_m in having the symbol a and a new state name; α_{m+1}' differs from α_m' in having a new stack level, with the predicted nonterminal written there and on the previous level, and the symbol a and two state names on the top level. But the only part of this that effects LIN is the symbol a; that is, $LIN(\alpha_{m+1}) = LIN(\alpha_m) \cdot a$ and $LIN(\alpha_{m+1}') = LIN(\alpha_m') \cdot a$. Since $LIN(\alpha_m) = LIN(\alpha_m')$, we are done with this case.

The other possibility, still assuming that c_{m+1} is a read successor of c_m and that q_m' is a base state, is that $\omega_m \dashv {}^k/k$ is not in any predictive language of the splitting (B,Q), of which q_m' is the base. In this case, the first item recognized by c_m is a descendant of I which will also be in the base state, q_m'; and so $f_M(q_m', a, \rho)$ will be well-defined. The successor of $h(c_m)$ will be well-defined, and it will be $h(c_{m+1})$; the necessary conditions are verified just as we have done above.

This completes the first half of our proof, for the case where c_{m+1} is a read-successor of c_m. We note that the above proof is also valid where the symbol being read is ε, i.e., where the first item recognized by c_m is $A \cdot .\varepsilon(\tau)$; or where the first item recognized by c_m is an essential item.

Now we must consider the case where c_{m+1} is a reduce successor of c_m. In this case, c_m must be the final state for some rule $A \rightarrow \Psi$, so $h(c_m)$ must be the same state, by induction. The configuration c_{m+1} will be obtained by popping Ψ and the interleaved state names off the stack, exposing some state q, and applying A to q. The stack α_{m+1} will be of the form $\beta X q A q_{m+1}$. Now there must have been some earlier entry to this state q; let α_1 be the last

previous stack equal to $\beta X q$. In other words, c_i is an entry to state q, and c_m is the recognition of one in the sequence of items that c_i will recognize before its exit. By Theorem 3.63 this sequence forms a chain of items; the item $A \rightarrow .\Psi(\omega_m \dashv^k/k)$ is one in this chain. We can construct the part of this chain below this item by looking back at all configurations c_j, where $i < j < m$, such that $\alpha_j = \beta X q \varphi' q_j$, where q_j is the final state for $B \rightarrow \varphi$; in that case, $B \rightarrow .\varphi(\omega_j \dashv^k/k)$ is some item in the chain which follows $A \rightarrow .\Psi(\omega_m \dashv^k/k)$. All of these items are descendants of I, the essential item recognized by the entry to q_i at c_i.

We have seen that at configuration $h(c_m)$, machine M is in the final state for $A \rightarrow \Psi$. By the way M is designed to operate, it will try to pop Ψ off the top level of its stack, expose some state name, and apply A to that state. We have to show that M can successfully complete this task.

By the way an MSP(k) machine operates, it can enter the final state for $A \rightarrow \Psi$ only if all the symbols of Ψ are on the topmost stack level. So M will be able to pop Ψ off the stack; but what state will this expose? Let us examine what has happened to the stack of M, in between configurations $h(c_i)$ and $h(c_m)$. By the inductive hypothesis and what we have already shown, we see that M performs all of the reads and reductions that M_0 does; the only difference between the operations of the two machines before c_m is that M may have made and suspended some predictions. We are not interested in any predictions that M may have made after $h(c_i)$ that have been fulfilled before $h(c_m)$; but let us suppose that M made a prediction at $h(c_j)$, $i < j < m$, that has not yet been fulfilled at $h(c_m)$. But if this is to be the case, all of Ψ must be on the topmost level created at $h(c_j)$; but the first symbol of Ψ has to be on the same level as q_i', if popping off Ψ is to reveal q_i. So either no

unfulfilled predictions have been made between $h(c_i)$ and $h(c_m)$, or exactly one had been made at $h(c_i)$. In the former case, the state exposed when M pops off Ψ at $h(c_m)$, will be none other than q_i', the state at $h(c_i)$; in the latter case, q_i' is a base state, and the state exposed by popping off Ψ is some predictive state associated with it.

Let us consider, then, what happened at (c_i). There are several possibilities:

i) That q_i' was an intermediate state. We know that $A \rightarrow .\Psi(\omega_m \dashv^k/k)$, the item recognized at c_m, is a descendant of I, the essential item recognized by c_i. Let $B \rightarrow .A\varphi(\tau)$ be the next item recognized by c_i after c_m. Then $B \rightarrow .A\varphi(\tau)$ is also a descendant of I, and $A \rightarrow .\Psi(\omega_m \dashv^k/k)$ is its immediate descendant. By hypothesis, $I \in q_i'$; therefore, since q_i' is the completion of its essential items, $A \rightarrow .\Psi(\omega_m \dashv^k/k)$ and $B \rightarrow .A\varphi(\tau)$ are both in q_i' as well. When Ψ is popped off the stack at $h(c_m)$, q_i' is exposed, and the lookahead is $\omega_m \dashv^k/k$; since $f_M(q_i', A, \omega_m \dashv^k/k)$ is well-defined (since $q_i' \subset q_i$, and since $B \rightarrow .A\varphi(\tau)$ is in q_i', with $\omega_m \dashv^k/k \in \text{FIRST}_k(\varphi\tau)$), $h(c_m)$ will thus have a well-defined succesor configuration. Its state will be $f(q_i', A, \omega_m \dashv^k/k)$. If $\varphi \neq \epsilon$, then since $B \rightarrow .A\varphi(\tau)$ is the next item to be recognized by c_i, $B \rightarrow A.\varphi(\tau)$ will be the essential item recognized by c_{m+1}; but this item will be in q_{m+1}' as well; and since $q_i' \subset q_i$, q_{m+1}' will be contained in q_{m+1}. If $\varphi = \epsilon$, then both q_{m+1} and q_{m+1}' will be final states for $B \rightarrow A$. The other properties of $h(c_{m+1})$ are easy to verify.

ii) The other possibility is that q_i' was a base state, say of the splitting (B,Q). There are two possibilities to consider here: that the lookahead at the time of entry to q_i' was in some predictive language, or that it was in none.

a) Suppose $\omega_i \dashv {}^k/k$ was not in any predictive language. Since I, the essential item recognized by c_i, in in q_i', every item recognized by c_i will be in some component of the splitting (B,Q). Any item $A \to .\Psi(\tau)$ recognized by c_i will have $\omega_i \dashv {}^k/k \in \text{FIRST}_k(\Psi\tau)$. Since $\omega_i \dashv {}^k/k$ was not in any predictive language, this means that any item $A \to .\Psi(\tau)$ recognized by c_i, will be in the base of the splitting, q_i', by the definition of the base of a splitting.

At configuration $h(c_i)$, since $\omega_i' \dashv {}^k/k = \omega_i \dashv {}^k/k$ was not in any predictive language, M will not have made a prediction. So when it enters the final state, at $h(c_m)$, for $A \to \Psi$, it will pop off Ψ and expose the state q_i'. Using the same arguments as we did before, the item $B \to .A\varphi(\tau)$, corresponding to the next item after $A \to .\Psi(\omega_m \dashv {}^k/k)$ found by c_i, will also be in q_i'. Thus, since $q_i' \subset q_i$, $f_M(q_i', A, \omega_m \dashv {}^k/k)$ is well-defined, and satisfies all the required properties.

b) Now suppose $\omega_i \dashv {}^k/k$ was an element of some predictive language of the splitting (B,Q), say for A_j. Then at $h(c_i)$, M would have predicted an A_j and started a new level to find an A_j. Intutitively, there are three possibilities: by the time $A \to .\Psi(\omega_m \dashv {}^k/k)$ is recognized, an A_j has already been found and the prediction fulfilled; an A_j has not yet been found; or $A = A_j$, and the reduction of Ψ to A fulfills the prediction. We shall see what causes each of these three possibilities.

Let us consider the sequence of items recognized by c_i between c_i and c_m. As we have seen, these items, when taken together with the rest of the items found by c_i after c_m, form an $L_k(A_j)$-terminal chain, starting with I, the essential item found by c_i. Since $q_i' \subset q_i$, this chain is also a chain through the state of which (B,Q) is a splitting; and so there must be an A_j-item in this chain. Let us see whether or not there has been an A_j-item among those recognized by c_i before c_m.

If c_i has not yet recognized an A_j-item, then M has not found an A_j yet either after predicting one at $h(c_i)$; so the prediction that has been made at $h(c_i)$ has not yet been fulfilled; thus the state that will be exposed when Ψ is popped off at $h(c_m)$ will be the initial state P_j. Furthermore, since no A_j-item has yet been found, all items found up till now, including $A \rightarrow .\Psi(\omega_m \dashv {}^k/k)$, will be "below" the A_j-item in the chain, and so will be in P_j. Let us consider what happens when Ψ is popped off, exposing P_j, and we try to apply A to it. If $A \neq A_j$, then the next item to be found by c_i, $B \rightarrow .A\varphi(\tau)$, will also be in P_j; so since $P_j \subset q_i$, $f_M(P_j,A,\omega_m \dashv {}^k/k)$ will be well-defined, as will be $h(c_{m+1})$, with all the required properties.

If, however, $A = A_j$, it may be that we have just fulfilled the prediction made at $h(c_i)$; however, it is also possible that the A_j found is a lower-level one, and does not fulfill the prediction. By definition, each $L_k(A_j)$-terminal chain can be broken into H_1 and H_2. So consider the chain corresponding to the items recognized by c_i. Either the item $A_j \rightarrow .\Psi(\omega_m \dashv {}^k/k)$ is the first item of H_2 or it is not. If it is not, then the next item in the chain, $B \rightarrow .A_j\varphi(\tau)$, is also in P_j; and so $f_M(P_j,A_j,\omega_m \dashv {}^k/k)$ is a well-defined state, and we are done. However, if it is the first item of H_2, then the next item in the chain, $B \rightarrow .A_j\varphi(\tau)$, is in the base state q_i'; and by definition, $f_M(P_j,A_j,\omega_m \dashv {}^k/k)$ will be equal to POP. So after exposing P_j, $h(c_m)$ will transfer to POP, wiping off the top stack level, and exposing q_i' as the state to which it tries to apply A_j. Since $B \rightarrow .A_j\varphi(\tau)$ is the next item to be found by c_i, and since it is also in q_i', $h(c_{m+1})$ will be well-defined and have the desired properties.

Finally, we must consider the case where c_i found an A_j-item before c_m. We again consider whether or not that A_j-item was the first element of H_2, of the chain of items found by c_i. If not, then as we have already seen, the prediction made at $h(c_i)$ would not yet have been fulfilled; and so the state exposed by popping off Ψ

would still be P_j, and the analysis would be as above. If however, the A_j-item already found was the first item of H_2, the prediction would have been fulfilled. Thus when Ψ is popped off at $h(c_m)$, the base state q_1' will be exposed. The item $A \rightarrow .\Psi(\omega_m \dashv^k/k)$ as well as the next item in the chain, $B \rightarrow .A\varphi(\tau)$, will both be in H_1 of the chain, and hence in q_1'. Thus after exposing q_1', M will be able to apply A to it, since $f(q_1', A, \omega_m \dashv^k/k)$ is well-defined. Thus $h(c_{m+1})$ is well-defined, and it is easy to see that it satisfies the required properties.

This completes the final subcase of the analysis, and establishes the stated result. Q.E.D.

Theorem 3.67 $L(M_0) \subset L(M)$.

Proof Suppose $\omega \in L(M_0)$, and let $c_1,\ldots,c_n$ be its accepting sequence. Then by the foregoing result, there is a sequence of configurations of M: $h(c_1),\ldots,h(c_n)$. By definition c_n is the configuration $(q, S\ q_0\ S\ q, \dashv^k)$. According to the preceding theorem, $LIN(\alpha_n) = LIN(\alpha_n')$. Since an initial state can not be a base state, it follows that every level of an MSP(k) stack α must add at least one symbol to $LIN(\alpha)$. Therefore since $LIN(\alpha_n) = S$, it must be that α_n' has only one level. This level must be of the form $S\ q_0'\ S\ q'$, where $q' = f_M(q_0', S, \dashv^k)$. Thus $q' = POP$, and $h(c_n)$ is the configuration $(POP, S\ q_0'\ S\ POP, \dashv^k)$, since $\omega_n = \omega_n'$. In each case, we either had $h(c_i) \vdash h(c_{i+1})$ or $h(c_i) \vdash c' \vdash h(c_{i+1})$; in any event, we have $h(c_1) \vdash^* h(c_n)$; since $h(c_1)$ is the initial configuration of M_0 for ω, this means that $\omega \in L(M_0)$. Q.E.D.

We shall now show the converse of the preceding result, and prove that $L(M) \subset L(M_0)$, that any string recognized by an MSP(k) machine for G is also recognized by the LR(k) machine for G. We again use a twin machine approach, but the basic lemma is much easier to obtain in this case. We do not need to make use of all the details of the definition of a state-splitting, but largely only one: that each component of a splitting is a subset of the state being split.

Definition 3.68 Let M be an MSP(k) machine for G. Suppose $\omega \in L(M)$, and $d_1, d_2, \ldots, d_n$ is the accepting sequence of configurations of M for ω. Then we define a sequence of configurations of M_0, $h(d_1), h(d_2), \ldots, h(d_n)$, as follows: $h(d_1)$ is the initial configuration of M_0 for ω; if d_{i+1} is a read or reduce successor of d_i, then $h(d_{i+1})$ is the successor of $h(d_i)$; otherwise $h(d_{i+1}) = h(d_i)$.

We want the sequence of configurations of M_0 to keep track of what M is doing. The performance of a prediction or a suspension by M has no counterpart in activities by M_0, so in that case M_0 does not have to change its configuration in order to keep up with M.

Lemma 3.69 If $\omega \in L(M)$, with accepting sequence $d_1, d_2, \ldots, d_n$, and with $h(d_1), \ldots, h(d_n)$ defined as above, then for each i, $1 \leq i \leq n$:

i) $h(d_i)$ is a well-defined configuration of M_0, $(q_i', \alpha_i', \omega_i' \dashv^k)$
ii) $\omega_i = \omega_i'$
iii) $LIN(\alpha_i) = LIN(\alpha_i')$
iv) if q_i is the final state for $A \to \Psi$, q_i' is the same final state
v) if q_i is not a final state, then either is q_i', and $q_i \subset q_i'$.

Proof

Basis $i = 1$, immediate.

Induction Suppose the statement is true for i, $1 \leq i \leq m$; we shall show it is true for $m+1$.

Since M is an MSP(k) machine for G, there are four possible relationships between d_m and d_{m+1}: d_{m+1} can be a read, a reduce, a prediction, or a suspension successor of d_m. We consider each in turn.

i) Suppose d_{m+1} is a read successor of d_m. That is, $\omega_m = a\rho$, $q_{m+1} = f_M(q_m, a, \rho \dashv^k/k)$, and $\omega_{m+1} = \rho$. This can be the case only if there is some item $A \rightarrow \varphi.a\Psi(\tau)$ in state q_m, with $\rho \dashv^k/k \in \text{FIRST}_k(\Psi\tau)$. But if this item is in q_m, it is also in q_m', since $q_m' \subset q_m$ by hypothesis. Since q_m' is a state of a well-formed LR(k) machine, $f_{M_0}(q_m', a, \rho \dashv^k/k)$ is thus a single-valued, well-defined state. But at configuration $h(d_m)$, the next symbol of input is a and the following lookahead is $\rho \dashv^k/k$. So $h(d_{m+1})$ is a well-defined read successor of $h(d_m)$. Since $\omega_m = a\omega_{m+1}$ and $\omega_m' = a\omega_{m+1}'$, we have $\omega_{m+1} = \omega_{m+1}'$. Also $\text{LIN}(\alpha_{m+1})$ and $\text{LIN}(\alpha_{m+1}')$ are obtained from their identical predecessors by appending an a, so they are identical. Finally, if $\Psi = \epsilon$, then both q_{m+1} and q_{m+1}' are the final state for $A \rightarrow \varphi a$; if $\Psi \neq \epsilon$, then the essential items of q_{m+1} will be contained in the essential items of q_{m+1}', since all items of q_m are in q_m'. Then since q_{m+1}' is the completion of its essential items while q_{m+1} (which is either an intermediate or a base) is contained in the completion of its essential items, we have $q_{m+1} \subset q_{m+1}'$.

ii) Suppose d_{m+1} is a prediction successor of d_m. Then q_m is a base state and q_{m+1} is an associated initial state; $\omega_{m+1} = \omega_m$, and α_{m+1} is the same as α_m except for the addition of a new stack level. By definition, $h(d_{m+1}) = h(d_m)$. So $\omega_{m+1}' = \omega_m' = \omega_m = \omega_{m+1}$; $\text{LIN}(\alpha_{m+1}') = \text{LIN}(\alpha_m') = \text{LIN}(\alpha_m) = \text{LIN}(\alpha_{m+1})$, the last equality holding

because adding a new level to a stack does not affect its linearization. Finally, since q_{m+1} is a predictive state associated with the base state q_m, we know that q_{m+1} is contained in the completion of the essential items of q_m; since $q_m \subset q_m'$, and q_m' is an intermediate state and hence the completion of its essential items, we have $q_{m+1} \subset q_m'$. By definition $q_{m+1}' = q_m'$, and hence $q_{m+1} \subset q_{m+1}'$, and we are done.

iii) Suppose d_{m+1} is a reduction sucessor of d_m. In this case q_m is a final state, say for $A \rightarrow \Psi$. In going from d_m to d_{m+1}, the machine M will pop Ψ off its topmost level, exposing some state, and will then apply A to this state. Suppose that $\alpha_{m+1} = \beta X q A q_{m+1}$, where q is the state exposed by popping off the Ψ. Let α_i be the last stack before α_{m+1} which was equal to $\beta X q$; by the operation of M, it is easy to see that there must have been such a stack. That is, $q = q_i$.

Now let us consider what happens to machine M_0 at configuration $h(d_m)$. It has entered the final state for $A \rightarrow \Psi$, and would like to pop Ψ off the stack. Since $LIN(\alpha_m) = LIN(\alpha_m')$, and α_m had Ψ on its topmost level to pop off, it must be that Ψ can be popped off α_m' as well. After Ψ is removed, some state q' is exposed, and M_0 tries to apply A to this state. We claim that q' is precisely the state q_i', the analogue of the state M exposes when it pops off Ψ. This is shown as follows: since simply by popping off Ψ at d_m, M exposes the state q_i, M can not have made any unfulfilled predictions between d_i and d_m. By the inductive hypothesis, M_0 between $h(d_i)$ and $h(d_m)$ precisely mimics the read and reduce activities of M between d_i and d_m. Thus the topmost level of α_m records the reads and reductions that have been done to q_i, while the upper regions of α_m' do the same for q_i'. Thus if the net effect of going from d_i to d_m is to put Ψ on top of q_i, the effect of going from $h(d_i)$ to $h(d_m)$ is to place Ψ on top of q_i'.

When q_i' is exposed by M_0 at $h(d_m)$, M_0 tries to apply A to it. When q_i is exposed to M at d_m, M successfully applies A to q_i, with lookahead $\omega_m \dashv^k/k$. There are three possibilities for the nature of this success. If there is some item $B \rightarrow \gamma.A\varphi(\tau)$ in q_i, with $\omega_m \dashv^k/\kappa \in FIRST_k(\varphi\tau)$ and $\varphi \neq \epsilon$, so that $f_M(q_i, A, \omega_m \dashv^k/k)$ is a non-final state, then since $q_i \subset q_i'$, this .A item and all others in q_i will be in q_i' as well. Since M_C is an LR(k) machine $f_{M_0}(q_i', A, \omega_m' \dashv^k/k)$ will be a single well-defined non-final state of M_0, which will contain all the essential items of q_{m+1} and therefore all of q_{m+1}. Since $\omega_m = \omega_m'$, $h(d_{m+1})$ is thus well-defined. Similarly, if there is an item $B \rightarrow \gamma.A(\tau)$ in q_i, it will be in q_i' as well, with q_{m+1} and q'_{m+1} both being the final state for $B \rightarrow \gamma A$. In both these cases, $\omega'_{m+1} = \omega_m' = \omega_m = \omega_{m+1}$, since the input is not affected by the performance of a reduction; and also in both cases, the stacks are identically affected by the removal of Ψ and its replacement by A.

There is another way for M to be able to apply A to q_i. That is if q_i is an initial state for A, and $f_M(q_i, A, \omega_m \dashv^k/k) = POP$. But if q_i is an initial state, q_{i-1} is its associated base state. By the construction of MSP(k) machines, if $f_M(q_i, A, \omega_m \dashv^k/k) = POP$, there is some item $B \rightarrow \gamma.A\varphi(\tau)$ in q_{i-1}, with $\omega_m \dashv^k/k \in FIRST_k(\varphi\tau)$. Since q_i was a prediction successor of q_{i-1}, it follows that $q_i' = q'_{i-1}$; and since by induction $q_{i-1} \subset q'_{i-1}$, we have $q_{i-1} \subset q_i'$. Thus the item $B \rightarrow \gamma.A\varphi(\tau)$ is in q_i', and so $f_{M_0}(q_i', A, \omega_m' \dashv^k/k)$ is defined, since $\omega_m = \omega_m'$. Thus $h(d_m)$ has a successor, with $q'_{m+1} = f_{M_0}(q_i', A, \omega_m' \dashv^k/k)$. Since q_{m+1} is the POP state, we do not have to establish any relationship between q_{m+1} and q'_{m+1}. Since $\omega_{m+1} = \omega_m$, and $\omega'_{m+1} = \omega_m'$, we have $\omega_{m+1} = \omega'_{m+1}$. Finally, in both machines, the new stack is obtained from the old one by replacing Ψ by A, so $LIN(\alpha_{m+1}) = LIN(\alpha'_{m+1})$.

iv) Finally, let us suppose that d_{m+1} is a suspension successor of d_m; i.e., $d_m = POP$. The only way for q_m to be the POP state is if q_{m-1} is the final state for some rule $A \rightarrow \Psi$, where A was the non-terminal predicted at the last unfulfilled prediction, say at

configuration d_i. Then as we have seen, the discovery of the A at d_{m-1} involved popping off Ψ, exposing state q_i, and then applying A to q_i to reach the POP state, q_m. In going from d_m to d_{m+1}, then, M wipes off the top level of α_m, exposing state q_{i-1}, the base state of the prediction that has just been fulfilled; then $q_{m+1} = f_M(q_{i-1}, A, \omega_m \dashv\ ^k/k)$.

By definition, if q_m is POP, then $h(d_{m+1}) = h(d_m)$; thus as we have seen, $q'_{m+1} = q_m' = f_{M_0}(q_i', A, \omega_m' \dashv\ ^k/k)$. But since d_i was a prediction successor of d_{i-1}, we know that $h(d_i) = h(d_{i-1})$; hence $q_{i-1} \subset q_i'$. Therefore, since $\omega_m = \omega_m'$ by induction, if $f_M(q_{i-1}, A, \omega_m \dashv\ ^k/k)$ is defined, so is $f_{M_0}(q_i', A, \omega_m' \dashv\ ^k/k)$, and the former will be a subset of the latter, unless they are both the same final state. Thus q_{m+1} and q'_{m+1} bear the required relationship. Since wiping off the top level of the stack does not affect either the input or the linearization of the stack, we have $LIN(\alpha_{m+1}) = LIN(\alpha_m)$ and $\omega_{m+1} = \omega_m$; by definition, $\omega'_{m+1} = \omega_m'$ and $\alpha'_{m+1} = \alpha_m'$, while $\omega_m' = \omega_m$ and $LIN(\alpha_m') = LIN(\alpha_m)$; hence $\omega'_{m+1} = \omega_{m+1}$ and $LIN(\alpha'_{m+1}) = LIN(\alpha_{m+1})$, and we are done with this last case.

This completes the induction and the proof. Q.E.D.

Theorem 3.70 $L(M) \subset L(M_0)$

Proof If $\omega \in L(M)$, let $d_1,\ldots,d_n$ be its accepting sequence. Then $h(d_1),\ldots, h(d_n)$ is well-defined. In each case, either $h(d_i) \vdash h(d_{i+1})$ or $h(d_i) = h(d_{i+1})$; hence $h(d_1) \vdash^* h(d_n)$. Since d_n is the last of the accepting sequence, $\alpha_n = S\, q_0\, S\, q$; since $LIN(\alpha_n) = LIN(\alpha_n')$, it follows that $\alpha_n' = S\, q_0'\, S\, q'$, where $q' = f_{M_0}(q_0', S, \dashv\ ^k)$. So $h(d_n)$ is the end of an accepting sequence of M_0 for ω, and we are done. Q.E.D.

Theorem 3.71 Let M be any MSP(k) machine for the LR(k) grammar G, and M_0 the LR(k) machine for G. Then $L(M) = L(M_0) = L(G)$.

Proof By Theorem 3.67 and Theorem 3.70. Q.E.D.

Theorem 3.72 Let G be an LR(k) grammar, $\omega \in L(G)$, and M any MSP(k) machine for G. Then M, when presented with ω, enters the same final states and in the same order as does M_0 when presented with ω.

Proof Immediate from the proofs of Proposition 3.66 and Lemma 3.69.

Q.E.D.

Corollary 3.73 If M is an MSP(k) machine for G and $\omega \in L(G)$, then M, when presented with ω, produces the left-to-right, bottom-up parse of ω.

Corollary 3.74 The number of steps taken by M in parsing ω is equal to the number of steps taken by M_0 in parsing ω plus twice the number of predictions made during the parse by M.

This last result also comes from the proofs of 3.66 and 3.69. We saw there that M precisely mimics M_0, except for the extra steps of making and suspending predictions. For each such prediction, both these actions must occur, which leads to this last result.

We have thus seen that our model of multiple-stack parsing machines does indeed capture the idea we started out with; we shall now see how this model may be usefully employed.

CHAPTER 4

THE BASIC TRANSFORMATION

4.1 Cycle-free MSP(k) Machines

In the preceding chapter, we defined a new kind of parsing machine, the multiple stack parsing machine, and showed that it did precisely the same things as the canonical LR(k) machine for the same grammar, except that the MSP(k) machine took a few more steps to do it. So far, this new model seems completely useless. But now we shall restrict our attention to a special kind of MSP(k) machine, and we shall see that our painstaking preparations will begin to pay off.

The most obvious way in which the parse performed by an MSP(k) machine differs from that done by an LR(k) machine, is in the condition of the stack. The LR(k) machine uses a conventional, one-dimensional stack, while the MSP(k) machine must have available a stack-of-stacks; a two-dimensional stack, singly infinite in both dimensions. Why exactly must each stack level of the MSP(k) stack itself be a potentially infinite stack, rather than of bounded finite width? Because in general, upon creation of a new stack level, we can not be sure how many symbols will have to appear on that level before it is destroyed. But how do an unbounded number of symbols get to appear on a level? Only by the existence of cycles in the state-graph of the machine. Each time a transition is made in the machine, a new symbol is put onto the stack; a sequence of transitions puts on a sequence of symbols. And should some sequence of transitions return the machine to the state whence the sequence began, having placed

on the stack some sequence of symbols, the whole process might begin anew, putting those symbols on again, and so on. In a cycle-free machine, however, only a bounded number of symbols could ever appear on a single stack level. The two-dimensional stack could then be considered as a one-dimensional stack again, with interesting implications.

Definition 4.1 If M is an MSP(k) machine, then a path in M is a sequence of states of M, $q_1 q_2 \ldots q_m$, such that for each i, there exists a symbol σ_i and a string x_i such that $f(q_i, \sigma_i, x_i) = q_{i+1}$. The length of this path is m, and it is labelled by $\sigma_1 \sigma_2 \ldots \sigma_{m-1}$. A path $q_1 \ldots q_m$ is a straight path if $q_i = q_j$ iff $i = j$. A path $q_1 \ldots q_m$ is a cycle in M if $q_1 = q_m$. M is cycle-free if there are no cycles in M.

Proposition 4.2 If M is a cycle-free MSP(k) machine, then during the parse of any string by M, there are a bounded number of symbols on any stack level of the stack. The bound is 2n, where n is the length of the longest straight path in M.

Proof Suppose that at some time there were some stack level with more than 2n symbols on it; we can assume that it is the topmost level at the time. Now any topmost level is always of the form $A_1\ q_1\ A_2\ q_2 \ldots A_m\ q_m$, where each q_i is a state and each A_i is an element of $V_n \cup V_T$. By the way M operates, for each i there is an x_i such that $q_{i+1} = f(q_i, A_{i+1}, x_i)$. Since M is cycle-free, $q_1 \ldots q_m$ must be a straight path. If there are more than 2n symbols on this level, it must be that $m > n$. But n is the length of the longest straight path in M. This is a contradiction, and we are done. Q.E.D.

In the case where M is a cycle-free LR(k) machine, it is well-known that M accepts a finite-state language. Since the stack never contains more than a bounded number of symbols, it is not really needed at all, and its use can be simulated by having the state keep track of what would be on the stack. For a general MSP(k) machine, however, this is not the case. Even if M is cycle-free, that applies only to each individual stack level; the stack as a whole is of unbounded size, because indefinitely many new stack levels may be created by repeated predictions. M can accept non-regular languages; the only constraint is that the stack can only be effectively used by means of predictions.

For a cycle-free machine, the maximum number of vocabulary symbols that can appear on any stack level is also equal to n, the length of the longest straight path in M. Since there are only finitely many vocabulary symbols, it follows that there are only finitely many different sequences of vocabulary symbols that can appear on a stack level. Let us ignore the state symbols that occur on a stack level, and just concentrate on the vocabulary symbols. We can think of each sequence of such symbols as being a single symbol from some other, finite set of symbols. We shall consider the sequence $A_1A_2A_3...A_m$ as representing the single symbol $(A_1,A_2A_3...A_m)$. Note that A_1 is always the name of the predicted nonterminal for that level.

It is thus possible for us to think of the two-dimensional MSP(k) stack as being, in the case of a cycle-free machine, a single-dimensional stack, on which are written symbols from some new vocabulary, symbols of the form (X, Φ). For each of these symbols, X is a nonterminal of the original grammar G, while Φ is a sequence of terminals and nonterminals from that grammar. Intuitively, an LR(k) stack is read vertically; in going to an MSP(k) stack, we cut up the LR(k)

stack into segments, and lay them out horizontally on top of one another. As we have seen, reading the MSP(k) stack in this way makes it look essentially the same as the original LR(k) stack. But now we are changing our perspective, reading the MSP(k) stack vertically, regarding each stack level as a single symbol.

Using this new perspective, let us observe what happens to the stack during the course of an MSP(k) parse. (From now on, our discussion is always referring to cycle-free MSP(k) machines.) If a terminal symbol is read from the input onto the stack, the topmost stack symbol (stack level) changes from (X, φ) to $(X, \varphi a)$. If a prediction is made, where Y is the name of the predicted nonterminal, the only change to the stack is that the symbol (Y, ϵ) is placed on top of the stack (i.e., a new level is created). If a reduction is made, some suffix of the topmost level is replaced by a single nonterminal; or, as we see things now, $(X, \varphi\alpha)$ becomes $(X, \varphi A)$. And finally, a suspension is made only if the topmost symbol is (X, X); the effect of performing the suspension is to eliminate this symbol (discard the topmost stack level).

These four alternatives are depicted in Figure 4.1; in each case, the remaining input string is indicated below the stack. (Note that only the first alters the input string.) Now suppose we were watching a "stack movie" of such a parse, observing the changes made to the contents of the stack and the occasional symbol disappearing from the input stream, without knowing what kind of parse was being performed, or the identity of the underlying grammar. Then by a stretch of the imagination, we could believe that we were watching not an MSP parse, but a deterministic top-down (LL) parse. The identity of the grammar driving the parse could be induced from observation of steps in the parse. Thus,

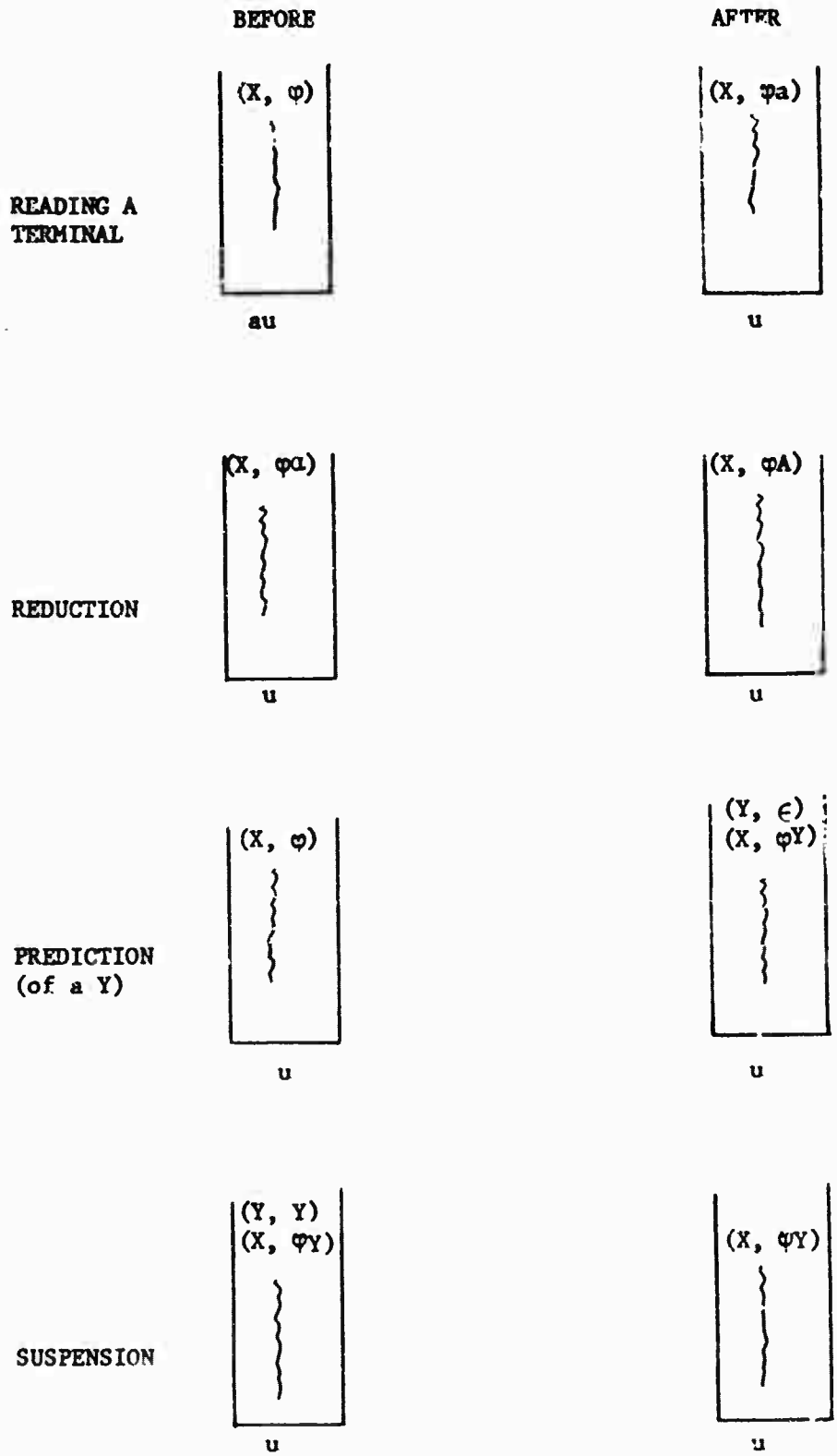

BEFORE
AFTER
READING A TERMINAL
(X, φ)
au
(X, φa)
u
REDUCTION
(X, φα)
u
(X, φA)
u
PREDICTION (of a Y)
(X, φ)
u
(Y, ϵ)
(X, φY)
u
SUSPENSION
(Y, Y)
(X, φY)
u
(X, φY)
u

Figure 4.1

reading the symbol a from the input and replacing (X, φ) as the topmost stack symbol by $(X, \varphi a)$, looks exactly like the effect of applying the rule $(X, \varphi) \rightarrow a(X, \varphi a)$ during a top-down parse. Replacing $(X, \varphi\alpha)$ by $(X, \varphi A)$ could be caused by applying the rule $(X, \varphi\alpha) \rightarrow (X, \varphi A)$. Putting (Y, ϵ) on top of (X, φ) might be effected by using the production $(X, \varphi) \rightarrow (Y, \epsilon)\ (X, \varphi Y)$. And discarding (Y, Y) from the top of the stack can be viewed as nothing more than applying the rule $(Y, Y) \rightarrow \epsilon$. Thus, an MSP(k) parse, when viewed in the proper light, looks remarkably like an LL(k) parse, based on another grammar.

We don't have to sit around watching a lot of parses in order to determine what this other grammar is--we can read it directly off the machine, as seen below. This new grammar we shall call $T_M(G)$, becuase it depends on the MSP(k) machine chosen.

4.2 The Transformation $T_M(G)$

Algorithm 4.3 For a grammar G and a cycle-free MSP(k) machine M associated with G, the grammar $T_M(G)$ is defined as follows:

1) The terminals of $T_M(G)$ are the terminals of G.

2) To determine the nonterminals, first name every initial state of of M with a distinct name (X_i, ϵ) where X is the nonterminal of G associated with the state. Then assign names to all other states of M as follows: if a state q can be reached from the initial state named (X_i, ϵ) by a path labelled by φ, then q is given the name (X_i, φ). The set of all such state names is the set of nonterminals of $T_M(G)$. The sentence symbol of $T_M(G)$ is is (S_1, ϵ), the name given the starting state of M.

3) The rules of $T_M(G)$ are derived as follows:

   i) If there is a transition labelled with the terminal symbol a coming from a state named (X_i, φ), add the rule $(X_i, \varphi) \rightarrow a(X_i, \varphi a)$

   ii) If the initial state named (Y_i, ϵ) is attached to the base state named (X_j, φ), add the rule $(X_j, \varphi) \rightarrow (Y_i, \epsilon)\ (X_j, \varphi Y)$

   iii) If $(X_i, \varphi\alpha)$ is the name of the final state of M associated with the rule of G, $A \rightarrow \alpha$, add the rule $(X_i, \varphi\alpha) \rightarrow (X_i, \varphi A)$.

   iv) For each name (X_i, ϵ), add the rule $(X_i, X) \rightarrow \epsilon$.

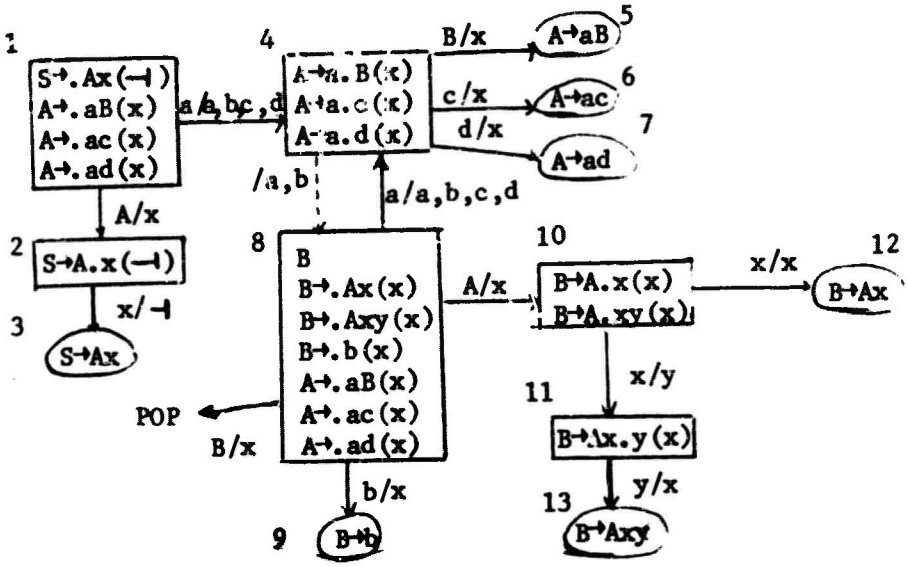
Before we give an example of this algorithm, a few comments are in order. First of all, the naming procedure is not unique, neither for states nor for names; two states may have the same name, and one state may have several names. Only initial states are uniquely named. Secondly, since M is cycle-free, no state can have more than a finite number of names; thus the set of state names (and the set of nonterminals of $T_M(G)$) is finite. Finally, it is easy to see that all paths into the final state associated with $A \rightarrow \alpha$ must end with α; so all names of such a state will be of the form $(X, \varphi\alpha)$, each giving rise to a rule $(X, \varphi\alpha) \rightarrow (X, \varphi A)$.

We shall establish certain conventions for subsequent use. First of all, in our examples, if there is but one initial state associated with the nonterminal X, we shall name it just (X, ϵ), omitting the subscript. Similarly, we shall often refer to a typical nonterminal of a grammar as (X, φ), dropping the subscript and identifying (X, ϵ) as the name of an initial state associated with nonterminal X. However, we shall have occasion to refer to (X_i, φ_i) as one nonterminal in a sequence of them; in these cases, we shall specifically

give a name to the nonterminal of G associated with the initial state (X_i, ϵ) (usually A_i); for in this context, we shall not mean that (X_i, φ_i) and (X_j, φ_j) are successors of the same initial state associated with nonterminal X.

For a simple example, consider the MSP(1) machine shown in Figure 4.2. It is associated with the grammar $S \rightarrow Ax$, $A \rightarrow aB$, $A \rightarrow ac$, $A \rightarrow ad$, $B \rightarrow Ax$, $B \rightarrow Axy$, $B \rightarrow b$; by inspection, it is cycle-free. (State numbers are given for reference purposes only.) To derive the grammar $T_M(G)$, we first apply our state-naming procedure. First, we assign name (S, ϵ) to state 1 and (B, ϵ) to state 8; this takes care of the initial states. Then state 2 is named (S, A), since there is a path labelled by A from state 1 to state 2. Similarly, state 3 is given the name (S, Ax). State 4 has two names (S, a) and (B, a), because there are paths labelled by a from both state 1 and state 8 to state 4. Furthermore, both state 11 and state 12 have the name (S, Ax), because there is a path labelled Ax from state 8 to state 11 as well as one from state 8 to state 12. The different lookaheads associated with these two paths do not affect the naming of the states. Completing the naming process, we get the information summarized in Figure 4.3. The set of names in the second column is the nonterminal set of $T_M(G)$, and (S, ϵ) is the sentence symbol.

To apply the first step of rule generation, we must locate those states which have some transition on a terminal symbol coming out of them. In this case, this includes states 1, 2, 4, 8, 10, and 11. For each of these states, we consider each of its names (X, φ) and each terminal symbol σ coming out of it, and generate the rule $(X, \varphi) \rightarrow \sigma\,(X, \varphi\,\sigma)$. Thus state 8 is named (B, ϵ) and has transitions on a and b leaving it; so we create rules $(B, \epsilon) \rightarrow a(B, a)$ and $(B, \epsilon) \rightarrow b(B, b)$. Similarly, state 4 has names (S, a) and (B, a), with



Cycle-free MSP(1) Machine

Figure 4.2

State Number	Name(s)
1	(S, ϵ)
2	(S, A)
3	(S, Ax)
4	(S, a) (B, a)
5	(S, aB) (B, aB)
6	(S, ac) (B, ac)
7	(S, ad) (B, ad)
8	(B, ϵ)
9	(B, b)
10	(B, A)
11	(B, Ax)
12	(B, Ax)
13	(B, Axy)

Figure 4.3

outgoing transitions on c and d; this gives rise to the rules (S, a) → c(S, ac), (S, a) → d(S, ad), (B, a) → c(B, ac), and (B, a) → d(B, ad). The set of rules generated by this step of the algorithm is given in Figure 4.4.

(S, ϵ) → a(S, a)	(B, ϵ) → a(B, a)
(S, A) → x(S, Ax)	(B, ϵ) → b(B, b)
(S, a) → c(S, ac)	(B, A) → x(B, Ax)
(S, a) → d(S, ad)	(B, Ax) → y(B, Axy)
(B, a) → c(B, ac)	
(B, a) → d(B, ad)	

Figure 4.4

Next we locate each base state and examine each initial state attached to it. If (X, φ) is a name of the base state, and (Y, ϵ) the name of the attached initial state, then we create the production (X, φ) → (Y, ϵ) (X, φY). In this example, there is only one base state, with one attached initial state; since the base state has two names, we get the following two rules: (S, a) → (B, ϵ) (S, aB) and (B, a) → (B, ϵ) (B, aB).

Then we consider each final state of M. We have seen that if a final state is associated with production A → α, then each name of that state will be of the form (X, $\varphi\alpha$). Then for each such name, we create the rule (X, $\varphi\alpha$) → (X, φA). In our example, the rules generated are:

(S, Ax) → (S, S)

(S, aB) → (S, A) (B, aB) → (B, A)

(S, ac) → (S, A) (B, ac) → (B, A)

(S, ad) → (S, A) (B, ad) → (B, A)

(B, Ax) → (B, B)

(B, Axy) → (B, B)

Note that since (B, Ax) is the name of both a final state and of a non-final state with a terminal transition, there are two different kinds of rules generated with (B, Ax) on the left-hand side: (B, Ax) → y(B, Axy) and (B, Ax) → (B, B).

Finally, we add one rule for each initial state. In this case, the rules are (S, S) → ϵ and (B, B) → ϵ. The entire grammar is given in Figure 4.5

(S, ϵ) → a(S, a)

(S, a) → (B, ϵ) (S, aB)

(S, a) → d(S, ad)

(S, a) → c(S, ac)

(S, aB) → (S, A)

(S, ad) → (S, A)

(S, ac) → (S, A)

(S, A) → x(S, Ax)

(S, Ax) → (S, S)

(S, S) → ϵ

(B, ϵ) → a(B, a)

(B, ϵ) → b(B, b)

(B, a) → c(B, ac)

(B, a) → d(B, ad)

(B, a) → (B, ϵ) (B, aB)

(B, b) → (B, B)

(B, ac) → (B, A)

(B, ad) → (B, A)

(B, aB) → (B, A)

(B, A) → x(B, Ax)

(B, Ax) → (B, B)

(B, Ax) → y(B, Axy)

(B, Axy) → (B, B)

(B, B) → ϵ

Figure 4.5

By the observations made earlier, it is clear that for any grammar G and associated cycle-free machine M, that the set of nonterminals given by Algorithm 4.3 is finite, and that the productions generated are indeed well-formed; thus $T_M(G)$ is indeed a well-formed grammar.

4.3 The Relationship Between G and $T_M(G)$

We shall discuss at length various properties of the grammar $T_M(G)$, but first we want to establish the most important, basic result. The grammar $T_M(G)$, read off the cycle-free machine M, generates precisely the set of strings which M accepts; and since M is an MSP(k) machine associated with G, this in turn equals the language generated by G. We shall prove this result by showing that a leftmost derivation of a string in $T_M(G)$ essentially simulates the progress of the parse of that string by the machine M.

Lemma 4.4 Let M be a cycle-free MSP(k) machine associated with grammar G, and $T_M(G)$ the grammar derived from M. Say $\omega \in L(M)$, ω is accepted by M, and consider any configuration of M in accepting ω. Suppose there are n stack levels to this configuration, and that the remaining input is ω_2, where ω_1 is the portion of ω already read. Let A_j be the first (leftmost) symbol on the j^{th} stack level of this configuration, and let φ_j represent the string of remaining vocabulary symbols on that level. Then there is a leftmost derivation in $T_M(G)$

$$(S_1, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \omega_1(X_n, \varphi_n)\ (X_{n-1}, \varphi_{n-1})\ldots(X_1, \varphi_1),$$

where (X_i, ϵ) is the name of the initial state of the i^{th} level.

Proof We prove this by induction on the number of the configuration in the accepting sequence for ω.

Basis The first configuration has a stack with only one level, on which the only inscribed vocabulary symbol is the sentence symbol S. Thus $A_1 = S$ and $\varphi_1 = \epsilon$. Furthermore, none of the input has yet been read, so $\omega_1 = \epsilon$. Now (S_1, ϵ) is the name of the starting state, which is the initial state of the first level. Therefore, since $(S_1, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} (S_1, \epsilon)$, we are done.

Inductive Step Suppose the lemma is true for the first m configurations in the accepting sequence for ω; we want to show it true for the (m+1)st. Let us examine more closely the mth configuration, as shown below. We have indicated only the vocabulary symbols on each level,

A_n	φ_n q
A_{n-1}	φ_{n-1}
⋮	
A_1	φ_1

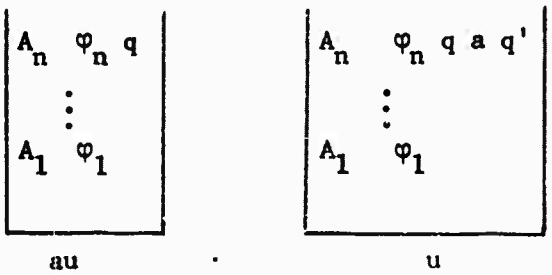
remaining input = ω_2; $\omega = \omega_1\omega_2$

except that we have indicated q, the state which M is in at this configuration. Observe that there is a path in M labelled by φ_n from the initial state of the n^{th} level to the state q; therefore, if (X_n, ϵ) is the name of the initial state of the n^{th} level, (X_n, φ_n) is a name for the state q.

How can the (m+1)st configuration follow from the m^{th}? There are only four ways: by M performing a reduction, reading a symbol, making a prediction, or suspending a prediction. We shall consider each of these cases.

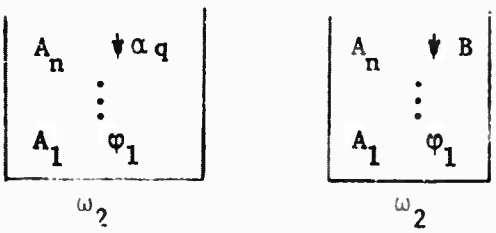
(From now on, we indicate the remaining input under the stack.)

Suppose M reads the terminal symbol a from the input in going from the m^{th} configuration to the $(m+1)^{st}$. Then the first symbol of the remaining input at the m^{th} stage must have been a; we can compare the two configurations as seen below. For the change to be effected, there must be a transition on the terminal symbol a from state q to state q'.



But since a name for q is (X_n, φ_n), this means there is a rule in $T_m(G)$: $(X_n, \varphi_n) \rightarrow a(X_n, \varphi_n a)$. By hypothesis, $(S_1, \epsilon) \underset{L}{\overset{*}{\Rightarrow}} \omega_1(X_n, \varphi_n)\ldots(X_1, \varphi_1)$. Therefore, $(S_1, \epsilon) \underset{L}{\overset{*}{\Rightarrow}} \omega_1 a(X_n, \varphi_n a)\ldots(X_1, \varphi_1)$. But this is precisely what we have to show, since at this new stage, the input already read is $\omega_1 a$, the topmost stack level is $A_n\ \varphi_n a$ with the same initial state as before, and all lower levels are unchanged.

Now suppose M performs a reduction according to the rule of G, $B \rightarrow \alpha$, in going from the m^{th} configuration to the next one. Then the two configurations are as pictured below. The right-hand side of the rule, α, must be a suffix of φ_n; so $\varphi_n = \Psi\alpha$. Thus the state q has a name $(X_n, \Psi\alpha)$; since q is a final state corresponding to the rule $B \rightarrow \alpha$, this gives rise to

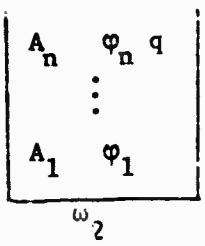


a rule in $T_M(G)$, $(X_n, \Psi\alpha) \rightarrow (X_n, \Psi B)$. Therefore,

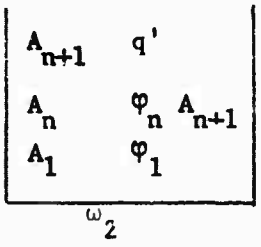
$$(S_1, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \omega_1(X_n, \varphi_n)\ldots(X_1, \varphi_1) \underset{L}{\Rightarrow} \omega_1(X_n, \Psi B)\ (X_{n-1}, \varphi_{n-1})\ldots(X_1, \varphi_1),$$

which is just what we have to show.

If the change of configuration is caused by a prediction, say of A_{n+1}, we get the following two stacks:

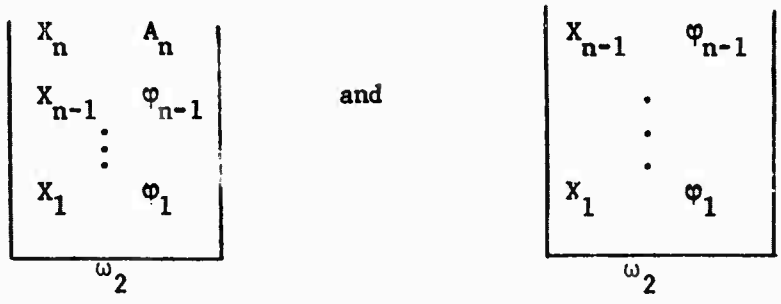


and



In both cases, we show only the last state on the topmost level. Note that in the (m+1)st configuration, $\varphi_{n+1} = \epsilon$, since only the predicted nonterminal is written on the new level when a prediction is made. The initial state for this level is q', which is associated with nonterminal A_{n+1}; by hypothesis, we shall call the name of this state (X_{n+1}, ϵ). Then since (X_n, φ_n) is a name for q, and since q' is an initial state associated with the base state q, the production $(X_n, \varphi_n) \rightarrow (X_{n+1}, \epsilon)\ (X_n, \varphi_n A_{n+1})$ is a rule of $T_M(G)$. Therefore, $(S_1, \epsilon) \overset{*}{\underset{L}{\Rightarrow}}$ $\omega_1(X_{n+1}, \epsilon)\ (X_n, \varphi_n A_{n+1})\ (X_{n-1}, \varphi_{n-1})\ldots(X_1, \varphi_1)$, which is what needs to be shown to prove the induction for this case.

Finally, we must consider the case where the change of configuration is caused by the suspension of a prediction. Then the "before" and "after" stacks are:



By definition, (X_n, ϵ) is the name of the initial state of the n^{th} level of the m^{th} configuration, and A_n is the nonterminal associated with this state. Therefore, $(X_n, A_n) \rightarrow \in$ is a rule of $T_M(G)$. Therefore, $(S_1, \in) \overset{*}{\underset{L}{\Rightarrow}} \omega_1(X_n, \varphi_n)\ldots (X_1, \varphi_1) \overset{}{\underset{L}{\Rightarrow}} \omega_1(X_{n-1}, \varphi_{n-1})\ldots(X_1, \varphi_1)$, which is the statement of the lemma for the $(m+1)^{st}$ configuration.

Thus the induction is complete, and the lemma is proved. Q.E.D.

Theorem 4.5 $L(M) \subset L(T_M(G))$

Proof Suppose $\omega \in L(M)$. Then the final configuration of the accepting sequence of M for ω, has a one-level stack, containing just $S\, q_0\, S\, q$, and an empty remaining input string - i.e., the input read so far is ω, the entire input string. Then by the preceding lemma, $(S_1, \in) \overset{*}{\underset{L}{\Rightarrow}} \omega(S_1, S)$, since (S_1, ϵ) is the name of the starting state q_0. But $(S_1, S) \rightarrow \in$ is a rule of $T_M(G)$. Therefore $(S_1, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \omega$, and since $(S_1, \in)$ is the sentence symbol of $T_M(G)$, this means $\omega \in L(T_M(G))$.

Q.E.D.

Since M is an MSP(k) machine associated with the grammar G, we know that $L(M) = L(G)$. Thus the theorem tells us that $L(G) \subset L(T_M(G))$. Now we want to show the other direction, that $L(T_M(G)) \subset L(G)$. We shall resort to a rather different proof technique for this theorem. First we establish a lemma that

defines the relationship between nonterminals of G and those of $T_M(G)$.

Lemma 4.6 Let M be a cycle-free MSP(k) machine for G, and $T_M(G)$ the derived grammar. Let (A, φ) be any nonterminal of $T_M(G)$. Then for any string of terminals ω, if $(A, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega$ via a derivation in $T_M(G)$, then $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$ in G.

Proof The proof is by induction on the length of ω.

Basis Assume $|\omega| = 0$; i.e., $\omega = \epsilon$. We have to show that if $(A, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \epsilon$, then $A \overset{*}{\underset{R}{\Rightarrow}} \varphi$. We prove this in turn by induction on the length of the derivation of ϵ from (A, φ) in $T_M(G)$.

Basis Assume that the length of the derivation of ϵ from (A, φ) is one; that is, $(A, \varphi) \rightarrow \epsilon$. But inspection of the possible rules of $T_M(C)$ shows that there is a rule $(A, \varphi) \rightarrow \epsilon$ if and only if $\varphi = A$. Since it is true that $A \overset{*}{\underset{R}{\Rightarrow}} A$, the basis is complete.

Inductive Step Assume for all nonterminals (A, φ) of $T_M(G)$, that if $(A, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \epsilon$ in less than k steps, then $A \overset{*}{\underset{R}{\Rightarrow}} \varphi$. We will show that if $(A, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \epsilon$ in exactly k steps, then it is also the case that $A \overset{*}{\underset{R}{\Rightarrow}} \varphi$.

So assume that $(A, \varphi) \overset{k}{\underset{L}{\Rightarrow}} \epsilon$. The first step in this derivation is an application of a rule either of the form $(A, \varphi) \rightarrow (B, \epsilon)\ (A, \varphi B)$ or one of the form $(A, \varphi) \rightarrow (A, \varphi')$. (This must be since no rule $(A, \varphi) \rightarrow a(A, \varphi a)$ can be used in the generation of the empty string, while $(A, A) \rightarrow \epsilon$ is obviously the last rule applied; and there are only these four kinds of rules in $T_M(G)$.) If the first rule applied is $(A, \varphi) \rightarrow (B, \epsilon)(A, \varphi B)$, then since we are dealing with context-free derivations, it must be that $(B, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \epsilon$ in fewer than k steps and also $(A, \varphi B) \overset{*}{\underset{L}{\Rightarrow}} \epsilon$ in fewer than k steps. So by the inductive assumption, $A \overset{*}{\underset{R}{\Rightarrow}} \varphi B$ and

$B \overset{*}{\underset{R}{\Rightarrow}} \epsilon$. Thus $A \overset{*}{\underset{R}{\Rightarrow}} \varphi B \overset{*}{\underset{R}{\Rightarrow}} \varphi$, and the induction is complete.

If $(A, \varphi) \rightarrow (A, \varphi')$ is the first rule in $(A, \varphi) \overset{k}{\underset{L}{\Rightarrow}} \epsilon$, it must be that $\varphi = \Psi\alpha$, and $\varphi' = \Psi D$, for some Ψ, α, and D. Then $(A, \Psi D) \overset{k-1}{\underset{L}{\Rightarrow}} \epsilon$, and so $A \overset{*}{\underset{R}{\Rightarrow}} \Psi D$ by inductive assumption. Since $(A, \Psi\alpha) \rightarrow (A, \Psi D)$ is a rule of $T_M(G)$, it must be that $D \rightarrow \alpha$ is a rule of G. Then we have $A \overset{*}{\underset{R}{\Rightarrow}} \Psi D \underset{R}{\Rightarrow} \Psi\alpha$, and since $\Psi\alpha = \varphi$, we are done. Q.E.D.

Thus we have shown that if $(A, \varphi) \overset{*}{\underset{T_M(G)}{\Rightarrow}} \epsilon$, then $A \overset{*}{\underset{G}{\Rightarrow}} \varphi$; thus the basis of our first induction is complete.

Inductive Step Assume for all nonterminals (A, φ) of $T_M(G)$, that if $(A, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega$, where $|\omega| < k$, then $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$ in G. Now we will show that if $(A, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega$ where $|\omega| = k$, then $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$. This proof will also be by induction on the length of the derivation of ω from (A, φ) in $T_M(G)$.

Basis Just to be safe and make sure that the proof is well-constructed, we shall not take for our basis the case where the length of the derivation is one, because if $|\omega| > 0$, it can't happen that (A, φ) generates ω in one step. Let us consider then, the shortest possible derivation of ω from (A, φ). Suppose $\omega = a_1 a_2 \ldots a_k$. By construction of the rules of $T_M(G)$, not more than one terminal symbol can be introduced into a sentential form by the application of a single rule. Therefore there must be at least k steps in the derivation, to introduce all the terminal symbols. Furthermore, the rightmost symbol in any sentential form of the derivation is of the form (A, Ψ); to get rid of this, last nonterminal, the rule $(A, A) \rightarrow \epsilon$ will have to be eventually applied. Furthermore, at least one rule will have to be applied to make the last nonterminal be (A, A). (Even if $(A, \varphi) = (A, A)$, the first rule applied to (A, φ)

will change its second component to something other than A, requiring another rule later to change it back.) Putting all this together, we see that the very shortest possible leftmost derivation of ω from (A, φ) is the following: $(A, \varphi) \Rightarrow {}_1(A, \varphi a_1) \Rightarrow a_1a_2(A, \varphi a_1a_2) \Rightarrow \ldots \Rightarrow a_1a_2\ldots a_k(A, \varphi a_1a_2\ldots a_k) \Rightarrow a_1a_2\ldots a_k(A, A) \Rightarrow a_1\ldots a_k = \omega$. Any other derivation will have to be longer. But if this derivation is valid, there must be a final state in M for a rule of G, $A \rightarrow \varphi a_1a_2\ldots a_k$. Since $a_1\ldots a_k = \omega$, this means that $A \rightarrow \varphi\omega$ is a rule of G, and so $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$. This completes the basis of our induction.

Inductive Step Assume that for all nonterminals (A, φ) and for all strings ω of length k, if there is a leftmost derivation of length less than m of ω from (X, φ), then $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$. We want to show that if $(X\ \varphi) \overset{m}{\underset{L}{\Rightarrow}} \omega$, then too $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\ \omega$.

Observe that we are doing a double induction, and so have two inductive hypotheses at our disposal. First that if $(A, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega$, where $|\omega| < k$, then $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$; and secondly, if $(A, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega$, where $|\omega| = k$ and the length of the derivation is less than m, then $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\ \omega$.

So suppose $(A, \varphi) \overset{m}{\underset{L}{\Rightarrow}} \omega$, $|\omega| = k$. Consider the first step of the derivation. If it is an application of the rule $(A, \varphi) \rightarrow a(A, \varphi a)$, then $(A, \varphi a) \overset{m-1}{\underset{L}{\Rightarrow}} \omega'$, where $\omega = a\omega'$. Thus by hypothesis, $A \overset{*}{\underset{R}{\Rightarrow}} \varphi a\omega' = \varphi\omega$. Another possibility is that the first rule applied is $(A, \varphi) \rightarrow (B, \epsilon)\ (A, \varphi B)$. Then $(B, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \omega_1$ and $(A, \varphi B) \overset{*}{\underset{L}{\Rightarrow}} \omega_2$, where $\omega = \omega_1\omega_2$, both derivations being shorter than m steps. There are three cases: $\omega_1 = \epsilon$, $\omega_2 = \epsilon$, or neither ω_1 nor ω_2 equals ϵ.

1) If $\omega_1 = \epsilon$, then $(B, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \epsilon$ and we have already established this implies $B \overset{*}{\underset{R}{\Rightarrow}} \epsilon$; furthermore, $(A, \varphi B) \overset{*}{\underset{L}{\Rightarrow}} \omega$ in fewer than m steps, so by hypothesis $A \overset{*}{\underset{R}{\Rightarrow}} \varphi B\omega$.

But since $B \overset{*}{\underset{R}{\Rightarrow}} \epsilon$ and ω is a string of terminals, this means $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$.

2) If $\omega_2 = \epsilon$, then $(B, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \omega$ in fewer than m steps, so $B \overset{*}{\underset{R}{\Rightarrow}} \omega$. And since $(A, \varphi B) \overset{*}{\underset{L}{\Rightarrow}} \epsilon$, we know that $A \overset{*}{\underset{R}{\Rightarrow}} \varphi B$. Thus $A \overset{*}{\underset{R}{\Rightarrow}} \varphi B \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$. 3) If $\omega_1 \neq \epsilon$ and $\omega_2 \neq \epsilon$, then (B, ϵ) generates a string shorter than k. Then by the first inductive hypothesis, since $(A, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \omega_1$, and $|\omega_1| < k$, we have $B \overset{*}{\underset{R}{\Rightarrow}} \omega_1$. Similarly, since $(A, \varphi B) \overset{*}{\underset{L}{\Rightarrow}} \omega_2$, and $|\omega_2| < k$, we have $A \overset{*}{\underset{R}{\Rightarrow}} \varphi B\omega_2$. Therefore $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega_1\omega_2 = \varphi\omega$.

Thus all cases are satisfied, and so if $(A, \varphi) \overset{m}{\underset{L}{\Rightarrow}} \omega$, $|\omega| = k$, we have that $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$. This completes our second induction, which in turn completes our first one: namely, we have established that if $(A, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega$, $|\omega| = k$, then $A \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$. Thus all inductive steps are complete, as is the proof. Q.E.D.

Theorem 4.7 $L(T_M(G)) \subset L(G)$.

Proof. If $\omega \in L(T_M(G))$, then $(S_1, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \omega$. Then by the lemma, $S \overset{*}{\underset{R}{\Rightarrow}} \omega$, since S is associated with the starting state, which is named (S_1, ϵ). Q.E.D.

Theorem 4.8 $L(T_M(G)) = L(G)$.

Proof. By Theorems 4.5 and 4.7 Q.E.D.

4.4 Some Technical Lemmas

Before we proceed to discuss the full implications of the preceding theorems and the precise nature of our transformations, we shall establish a few more results that will be very useful.

A brief comment is appropriate here on the effect of ϵ - transitions in M on the generation of $T_M(G)$. No rule will be generated of the form $(X, \varphi) \rightarrow (X, \varphi\epsilon)$, if there is an ϵ-transition from an (X, φ) state, since ϵ is not a terminal symbol. (Such a rule would be a poor thing to have anyway, since it really is $(X, \varphi) \rightarrow (X, \varphi)$.) However, an ϵ-transition always goes to a final state for a rule of the form $A \rightarrow \epsilon$. Thus we will get a rule $(X, \varphi) \rightarrow (X, \varphi A)$ in $T_M(G)$.

Lemma 4.9 If M is a loop-free MSP(k) machine, then the state-naming procedure of Algorithm 4.3 will assign any given name (X, φ) to at most one non-final state.

Proof We have noted that it is possible for two or more different states of M all to be given the name (X, φ); this lemma maintains that at most one of these states will be non-final. By the naming procedure, a state q is given the name (X, φ) if there is path to it from the initial state named (X, ϵ) so that the path spells out φ. If $\varphi = \varphi_1\varphi_2 \ldots \varphi_n$, the transitions along the path must be on the φ_i, except possibly including some transitions on ϵ (since inclusions of ϵ does not affect the spelling of the path). But the only way ϵ-transitions can occur in the machine M is if some state has an item of the form $A \rightarrow .\epsilon(\omega)$; then there will be an ϵ-transition from that state to the final state of the rule $A \rightarrow \epsilon$. Thus ϵ-transitions go only to final states; so if there is an ϵ-transition in a path spelling out φ, from initial state (X, ϵ) to a state named (X, φ), it can only be at the very end, in which case the (X, φ) state will be a final state.

Thus if non-final states q and q' are both named (X, φ), there must be paths of length exactly n from initial state (X, ϵ) to each of the states. Since q and q' are different states, the paths are not identical. Let the sequence of states along the path to q be $q_0, q_1, \ldots, q_n$, where q_0 is the initial state (X, ϵ), q_n is the state q, and $q_i = f(q_{i-1}, \varphi_i, \tau_i)$ for some τ_i; similarly the path to q' goes through $q_0', q_1', \ldots, q_n'$. Now $q_0 = q_0'$, and $q_n \neq q_n'$; thus there is some smallest i such that $q_i \neq q_i'$, $i > 0$. Then for some τ_i and τ_i', $q_i = f(q_{i-1}, \varphi_i, \tau_i)$ and $q_i' = f(q_{i-1}, \varphi_i, \tau_i')$, since $q_{i-1} = q_{i-1}'$. But by construction and definition of MSP(k) machines, a state has two successors under the same symbol only if at least one of them is final. Thus either q_i or q_i' is final. So in order for the paths to q and q' to be well-defined, it must be that $i = n$, and either q or q' is final, which is what we wanted to prove. Q.E.D.

The preceding result assures us that we can refer to the non-final state named (X, φ). The next lemma is cumbersome to state, and rather technical. But it establishes a connection between the structure of the machine M and the derivations of strings generated by the grammar derived from M.

Lemma 4.10 Let M be a loop-free MSP(k) machine for the grammar G, and $T_M(G)$ the grammar derived from M by Algorithm 4.3. Suppose that $(S_1, \epsilon) \vdash^k \overset{*}{\underset{L}{\Rightarrow}} \omega_1(X, \varphi)\Psi \vdash^k \overset{*}{\underset{L}{\Rightarrow}} \omega_1\omega_2 \vdash^k$, is a derivation in $T_M(G)$, where $\omega_1, \omega_2 \in V_T^*$. Then:

1) If $\varphi = \epsilon$, then for some state q and lookahead ρ, $g_1(q, \rho) = q'$, where q' is the initial state named (X, ϵ) and $\rho = \omega_2 \vdash^k / k$.

2) If $\varphi \neq \epsilon$, let $\varphi = \tau\sigma$, where $\sigma \in V_N \cup V_T$, and $\tau \in (V_N \cup V_T)^*$ and let q be the single non-final state named (X, τ). Consider the first rule applied in the leftmost derivation $(X,\varphi) \vdash^k \underset{L}{\overset{*}{\Rightarrow}} \omega_2 \dashv^k$.

   a) If this rule is $(X, \sigma) \rightarrow a(X, \tau\sigma a)$ or $(X, \tau\sigma) \rightarrow (Y, \epsilon)$ $(X, \tau\sigma Y)$, then $f(q, \sigma, \rho) = q'$, where q' is the single non-final state named $(X, \tau\sigma)$ and $\rho = \omega_2 \dashv^k/k$.

   b) If this rule is of the form $(X, \tau\sigma) \rightarrow (X, \eta)$, where $\tau\sigma = \alpha\beta$ and $\eta = \alpha A$, then $f(q, \sigma, \rho) = q'$, where q' is the final state for the rule $A \rightarrow \beta$ and $\rho = \omega_2 \dashv^k/k$.

   c) If this rule is $(X, X) \rightarrow \epsilon$ (i.e., $\tau = \epsilon$ and $\sigma = X$), then $f(q, X, \rho) = \text{POP}$ where $\rho = \omega_2 \dashv^k/k$.

Proof This lemma tries to establish a relationship between the structure of the machine M and the nature of derivations in the grammar $T_M(G)$ derived from it. We will prove it by induction, in the following way. By inspection, we can see that the only part of each case that requires much in the way of proof is the fact that $\rho = \omega_2 \dashv^k/k$. That is, for each such case, it is apparent that the transition in question is defined as required for some lookahead ρ. For example, if $(X, \tau\sigma) \rightarrow a(X, \tau\sigma a)$ is a rule of $T_M(G)$, it could only have arisen from an a-transition out of a non-final state named $(X, \tau\sigma)$; the fact that there is a non-final state named $(X, \tau\sigma)$ means there is a transition on σ from a non-final state named (X, τ) to some other non-final state (which will thus be named $(X, \tau\sigma)$). There will be a number of lookaheads associated with this σ-transition -- the lemma requires that one of these be equal to $\omega_2 \dashv^k/k$. Similarly, for case b), if $(X, \tau\sigma) \rightarrow (X, \eta)$ is a rule, where $\tau\sigma = \alpha\beta$ and $\eta = \alpha A$, then the final state for rule $A \rightarrow \beta$ of G is named

$(X, \tau\ \sigma)$, and so there is a σ-transition to it from the non-final state named (X, τ). Similar analyses hold for the other cases. What remains for us to establish is that for each of the transitions in question, one of its associated lookaheads is equal to $\omega_2 \dashv^k/k$. We will do this by showing for each $i \leq k$, there is a lookahead ρ associated with the transition such that $\rho/i = \omega_2 \dashv^k/i$. For the case $i = k$, $\rho/i = \rho$, and our result will be established. Thus we do induction on i.

Basis $i = 0$. This case is trivially true, since for any ρ at all, $\rho/0 = \epsilon = \omega_2 \dashv^k/0$.

Induction We assume that the lemma holds for lookahead ρ in each case with $\rho/i = \omega_2 \dashv^k/i$. We want to show $\rho/i+1 = \omega_2 \dashv^k/i+1$. (Of course, we're assuming $i < k$.) We will examine one case at a time. We will start with case 2a first.

Case I Assume the first rule being applied in $(X, \varphi)\Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}} \omega_2 \dashv^k$ is $(X, \tau\ \sigma) \rightarrow a(X, \tau\ \sigma\ a)$. Then the picture in Figure 4.6 obtains,

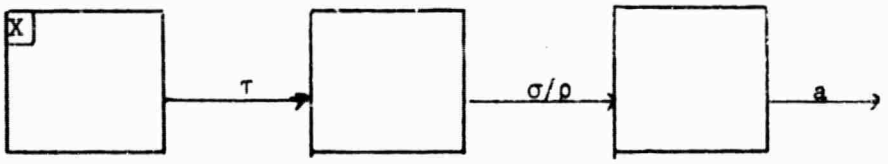


Figure 4.6

and may help in seeing what's going on. We know there is an a-transition from the non-final $(X, \tau\ \sigma)$ state, and that there is a σ-transition from the non-final state named (X, τ) to that non-final $(X, \tau\ \sigma)$ state, with lookahead ρ such that $\rho/i = \omega_2 \dashv^k/i$.

By assumption, $(X, \tau\ \sigma)\ \Psi \dashv^k \underset{L}{\Rightarrow} a(X, \tau\ \sigma\ a)\ \Psi \dashv^k \overset{*}{\underset{L}{\Rightarrow}} \omega_2 \dashv^k$. Thus a is the first symbol of ω_2; let $\omega_2 = a\omega_3$. Then $(X, \tau\ \sigma\ a)\ \Psi \dashv^k \overset{*}{\underset{L}{\Rightarrow}} \omega_3 \dashv^k$. Now the first rule applied in this derivation will either be of the form $(X, \tau\ \sigma\ a) \rightarrow b(X, \tau\ \sigma\ ab)$, $(X, \tau\ \sigma\ a) \rightarrow (Y, \epsilon)\ (X, \tau\ \sigma\ aY)$, or $(X, \tau\ \sigma\ a) \rightarrow (X, \varphi')$; that is, it will <u>not</u> be $(X, X) \rightarrow \epsilon$. Then by induction, there will be an a-transition from the non-final $(X, \tau\ \sigma)$ state to a $(X, \tau\ \sigma\ a)$ state (which is final or not depending on which kind of rule is being applied to $(X, \tau\ \sigma\ a)$), with associated lookahead ρ' such that $\rho'/i = \omega_2 \dashv^k/i$.

This means there is some item I in the non-final $(X, \tau\sigma)$ state of the form $A \rightarrow \gamma_1 . a\gamma_2(u)$, such that $\omega_3 \dashv^k/i \in \text{FIRST}_i(\gamma_2 u)$. Therefore $a\omega_3 \dashv^k/i+1 \in \text{FIRST}_{i+1}(a\gamma_2\mu)$. Now since σ is a non-null symbol, it is easy to see that every essential item of the non-final $(X, \tau\ \sigma)$ state has a σ in the pre-dot position. Either the item I is essential or it is not; if it is not, it is the leftmost descendant of some essential item. Thus in either event there is some essential item $B \rightarrow \alpha\ \sigma . \beta(\pi)$ such that $a(\omega_3 \dashv^k/i) \in \text{FIRST}_{i+1}(\beta\pi)$. But $a(\omega_3 \dashv^k/i) = a\omega_3 \dashv^k/i+1 = \omega_2 \dashv^k/i+1$. And if $B \rightarrow \alpha\sigma . \beta(\pi)$ is an item of the $(X, \tau\ \sigma)$ state, $B \rightarrow \alpha . \sigma\beta(\pi)$ is an item of the non-final (X, τ) state. Thus for some lookahead ρ'', $f(q, \sigma, \rho'') = q'$, with $\rho''/i+1 = \omega_2 \dashv^k/i+1$.

Case II Assume that the first rule in $(X, \varphi)\ \Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_2 \dashv^k$ is $(X, \tau\ \sigma) \rightarrow (Y, \epsilon)\ (X, \tau\ \sigma\ Y)$. Then we have the picture shown in Figure 4.7. We want to show that for some lookahead ρ as indicated, $\rho/i+1 = \omega_2 \dashv^k/i+1$.

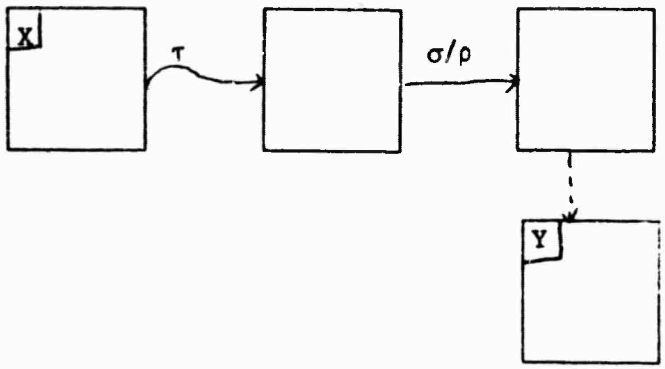


Figure 4.7

We know that $(Y, \epsilon)\ (X, \tau\ \sigma\ Y)\ \Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_2 \dashv^k$. If $\omega_2 = \epsilon$, then we have by induction that for some lookahead ρ in the desired position, $\rho/i = \omega_2 \dashv^k/i = \dashv^k/i = \dashv^i$. By the construction of the machine M and the nature of $\dashv$ as a right-padding symbol, if the first i symbols of a lookahead ρ are all $\dashv$'s, then the entire string is $\dashv$, and so $\rho/i+1 = \dashv^{i+1}$. Thus we may assume that $\omega_2 \neq \epsilon$.

The first rule in the derivation $(Y, \epsilon)\ (X, \tau\ \sigma\ Y)\ \Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_2 \dashv^k$ will be applied to (Y, ϵ); let us assume that this rule is $(Y, \epsilon) \rightarrow a(Y, a)$. Then $(Y, a)\ (X, \tau\ \sigma\ Y)\ \Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_3 \dashv^k$, where $\omega_2 = a\omega_3$. Then by induction, there is an a-transition from the non-final (Y, ϵ) state (which is the initial state for Y), to a (Y, a) state, with lookahead ρ such that $\rho/i = \omega_3 \dashv^k/i$. Therefore there is an item in the (Y, ϵ) state of the form $B \rightarrow .a\alpha(\pi)$, such that $\omega_3 \dashv^k/i \in \text{FIRST}_i(\alpha\pi)$. Therefore $\omega_2 \dashv^k/i+1 \in \text{FIRST}_{i+1}(a\ \alpha\ \pi)$.

Let us recall that an initial state, when taken together with the base state with which it is associated, forms the completion of the essential items of that base state. Thus every item of the initial state is a descendant of some essential item in the base state. Thus there is some essential item I in the associated base state such that $\omega_2 \dashv^k/i+1 \in \text{FIRST}_{i+1}(I)$. But this associated base state is the $(X, \tau\ \sigma)$ state. Thus as we argued before in Case I, there will be a σ-transition from the (X, τ) state with some associated lookahead ρ'' with $\rho''/i+1 = \omega_2 \dashv^k/i+1$.

The preceeding argument rested on the assumption that the rule applied to (Y, ϵ) was $(Y, \epsilon) \rightarrow a(Y, a)$. The only other possibility would be an application of a rule $(Y, \epsilon) \rightarrow (Y, A)$. This rule could be in the grammar $T_M(G)$ if there were in M an ϵ-transition from the initial state for Y to the final state for the rule $A \rightarrow \epsilon$. But even in this case, there must be an eventual application of a rule $(Z, \alpha) \rightarrow a\ (Z, \alpha\ a)$, since $\omega_2 \neq \epsilon$. Let us assume for the moment that this rule is $(Y, \alpha) \rightarrow a(Y, \alpha\ a)$, and that the derivation $(Y, \epsilon) \overset{*}{\Rightarrow} (Y, \alpha)$ consisted solely of rules $(Y, \beta) \rightarrow (Y, \beta\ A)$, each of which arises from an ϵ-transition from the state named (Y, β) to the final state for $A \rightarrow \epsilon$. It is thus easy to see that $\alpha \overset{*}{\Rightarrow} \epsilon$. By induction, there is an a/ρ transition from the (Y, α) state to a $(Y, \alpha\ a)$ state such that $\rho/i = \omega_3 \dashv^k/i$. Thus there is a lookahead ρ' on entrance to the (Y, α) state such that $\rho'/i+1 = \omega_2 \dashv^k/i+1$. Since $\alpha \overset{*}{\Rightarrow} \epsilon$, any lookahead on entry to the (Y, α) state is also a lookahead on an ϵ-transition out of the (Y, ϵ) initial state. Thus for some item I in the base state associated with the (Y, ϵ) state (namely, the non-final $(X, \tau\ \sigma)$ state), $\omega_2 \dashv^k/i+1 \in \text{FIRST}_{i+1}(I)$. Therefore there is some lookahead ρ'' on entrance to the $(X, \tau\ \sigma)$ state such that $\rho''/i+1 = \omega_2 \dashv^k/i+1$, which is the

desired result. Figure 4.8 depicts the local structure of the machine M and the various lookaheads.

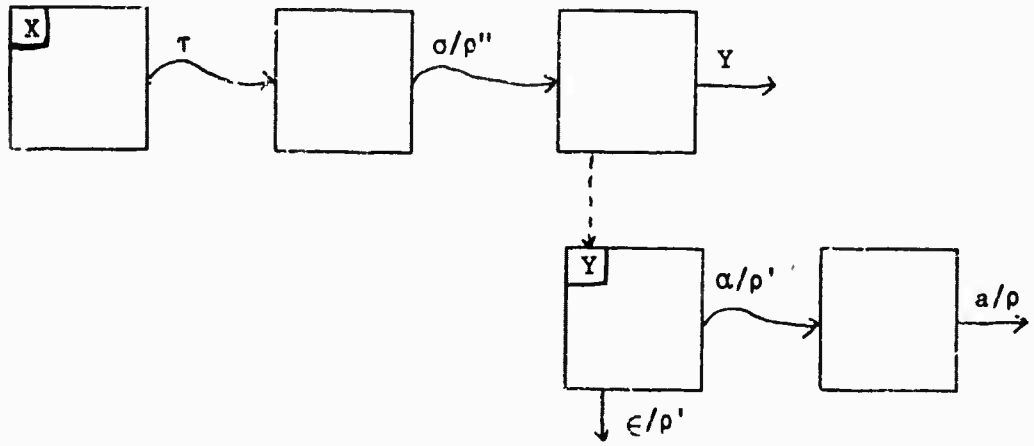


Figure 4.8

What other possibility exists for the first application of a rule that introduces a terminal symbol in the derivation $(Y, \epsilon)\ (X, \tau\ \sigma\ Y)\Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}}$ $\omega_2 \dashv^k$? There is always the chance that $(Y, \epsilon) \overset{*}{\Rightarrow} \epsilon$ (if for example, $Y \rightarrow \epsilon$ is a rule of G), and that the introduction in question occurs later in the derivation; or that for some α, $(Y, \alpha) \rightarrow (Z, \xi)\ (Y, \alpha\ Z)$, and that the terminal introduction is via $(Z, \beta) \rightarrow a(Z, \beta\ a)$. In any of these remaining cases, a tedious extension of the preceeding argument suffices to establish the desired result. The motif of the proof is that corresponding to the generation of the terminal symbol there will be an entry to some state q of M with lookahead ρ such that $\rho/i = \omega_3 \dashv^k/i$, with entry to this state's predecessor consequently having lookahead ρ', $\rho'/i+1 = \omega_2 \dashv^k/i+1$; and since none of the intervening steps in the derivation introduce any terminal symbols, the entrance to q's predecessor will have the same lookaheads as the entrance to the $(X, \tau\ \sigma)$ state.

Case III Assume that the first rule in $(X, \varphi)\ \Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_2 \dashv^k$ is of the form $(X, \varphi) \rightarrow (X, \varphi')$. That is $(X, \tau\ \sigma) = (X, \varphi) = (X, \alpha\ \beta)$ and $(X, \varphi') = (X, \alpha\ A)$. Then we have the picture in Figure 4.9, where σ is the last symbol of β.

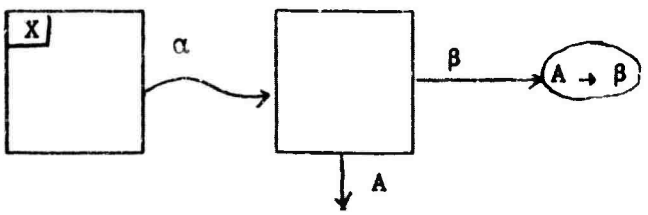


Figure 4.9

The analysis in this case resembles that of Case II. First of all, if $\omega_2 = \epsilon$, the induction is trivially true, just as in the previous cases; so we may assume $\omega_2 \neq \epsilon$. Therefore in the derivation $(X, \alpha\ A)\ \Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_2 \dashv^k$, there is the application of at least one rule that introduces a terminal symbol into the sentential form. Let us assume for the moment that the first such rule is applied to $(X, \alpha\ A)$; i.e., $(X, \alpha\ A)\ \Psi \dashv^k \underset{L}{\Rightarrow}\ a\ (X, \alpha\ A\ a)$ $\Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_2 \dashv^k$. Then $(X, \alpha\ A\ a)\ \Psi \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_3 \dashv^k$, so by induction the a-transition from $(X, \alpha\ A)$ to $(X, \alpha\ A\ a)$ is associated with lookahead ρ, $\rho/i = \omega_3 \dashv^k/i$. Therefore the A-entry to $(X, \alpha\ A)$ from (X, α)has associated lookahead ρ', $\rho'/i+1 = \omega_2 \dashv^k/i+1$. But since $A \rightarrow \beta$ is a rule of G, the lookaheads on entrance to the $(X, \alpha\ \beta)$ state must be the same as those on entry to the $(X, \alpha\ A)$ state. Therefore, ρ' is also a lookahead on entry to the $(X, \alpha\ \beta)$ state, which is the (X, φ) state; and so we are done.

If the first rule which introduces a terminal symbol is not applied to $(X, \alpha\ A)$, but later in the derivation, an extension of the above argument suffices. Once again, the key point is to show that the lookahead on entry to the $(X, \alpha\ \beta)$ state is the same as the lookahead on entry to the state named by the nonterminal to which the terminal-introducing rule is applied. The argument is straightforward, but tedious, and follows the pattern of the previous case.

<u>Case IV</u> Assume the first rule in $(X, \varphi)\ \Psi \dashv^k \overset{*}{\underset{L}{\Rightarrow}}\ \omega_2 \dashv^k$ is $(X, X) \rightarrow \epsilon$. That is, $(X, \varphi) = (X, X)$, so $\tau = \epsilon$ and $\sigma = X$. Here we have the picture of Figure 4.10.

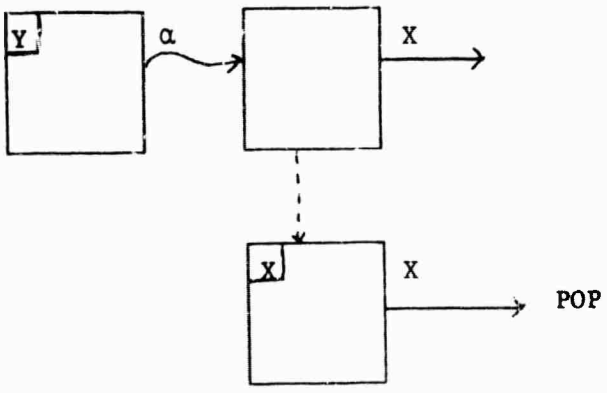


Figure 4.10

As before, the case is trivial if $\omega_2 = \epsilon$, so we assume $\omega_2 \neq \epsilon$. Let us say that the first nonterminal symbol of Ψ is $(Y, \alpha\ Y)$; it must be of this form, since (X, ϵ) must have been introduced into the sentential form by $(Y, \alpha) \rightarrow (X, \epsilon)\ (Y, \alpha\ X)$. Then the base state associated with intital (X, ϵ) state is named (Y, α). Once again, let us consider the first rule in the derivation

that introduces a terminal symbol. Assume this rule is $(Y, \alpha X) \rightarrow a(Y, \alpha X a)$ Then by induction, the a-transition from $(Y, \alpha X)$ to $(Y, \alpha X a)$ has an associated lookahead ρ such that $\rho/i = \omega_3 \dashv^k/i$. Therefore, the X-transition from (Y, α) to $(Y,\alpha X)$ has associated lookahead ρ', $\rho'/i+1 = \omega_2 \dashv^k/i+1$. Now recall from the definition of MSP(k) machines, that if q is an intitial state named (X, ϵ) and q' is the base state associated with q, that $f(q, X, \rho'') = POP$ iff $\rho'' \in FOLLOW_k(q', X)$. In our case, since ρ' is a lookahead on the X-transition from (Y, α) to $(Y, \alpha X)$, it must be that ρ' is in the k-follow set of X in the (Y, α) state. Therefore, since the (Y, α) state is the base associated with the initial (X, ϵ) state, ρ' will also be a lookahead for the X-transition from that initial state to POP. Thus there is a lookahead ρ' on entry to the POP state such that $\rho'/i+1 = \omega_2 \dashv^k/i+1$, which was the desired result.

Now what if the first rule that introduces a terminal is not applied to $(Y, \alpha X)$? If it is applied to some nonterminal (Y, β) such that $(Y, \alpha X) \stackrel{*}{\Rightarrow} (Y, \beta)$, it is apparent that the lookahead on entry to the (Y, β) state is the same as it is upon entry to $(Y, \alpha X)$, and the above argument suffices. If it is applied to some other nonterminal (Z, β) further along in Ψ, or introduced into the derivation by $(Y, \alpha') \rightarrow (Z, \epsilon)\ (Y, \alpha' Z)$, the same kind of argument outlined previously applies, the key idea being that lookaheads on entry to (Z, β) will be the same as those on entry to $(Y, \alpha X)$.

Case V This is the case where $\varphi = \epsilon$. We want to show that if $(X, \epsilon)\ \Psi \dashv^k \stackrel{*}{\underset{L}{\Rightarrow}} \omega_2 \dashv^k$, then there is a lookahead ρ on entry to the (X, ϵ) initial state such that $\rho/i+1 = \omega_2 \dashv^k/i+1$. The analysis used in case II is directly applicable here. There we established that there was an item I in the initial state (and hence in the associated base) such that $\omega_2 \dashv^k/i+1 \in FIRST_{i+1}(I)$.

But by definition and construction of an MSP(k) machine, any member of $FIRST_k(I)$, where I is in the initial state, is a lookahead for entry to the initial state. Thus, since $i+1 \leq k$, there is some lookahead ρ, on entry to the initial state, with $\rho/i+1 = \omega_2 \dashv^k/i+1$.

This is the final case, and we have established the induction step of the proof. Thus the proof of this lemma is complete. Q.E.D.

4.5 The LL(k)-ness of $T_M(G)$

The preceeding lemma is not so interesting in itself, but it enables us to establish the following, which is the converse of Lemma 4.4.

Lemma 4.1' Let M be a cycle-free MSP(k) machine for G, $T_M(G)$ the derived grammar. Suppose $\omega \in L(T_M(G))$, and that $(S_1, \epsilon) \dashv^k \overset{*}{\underset{L}{\Rightarrow}} \omega_1(X_n, \varphi_n)(\lambda_{n-1}, \varphi_{n-1})\ldots(X_1, \varphi_1) \dashv^k \overset{*}{\underset{L}{\Rightarrow}} \omega_1\omega_2 \dashv^k = \omega \dashv^k$. Then upon applying M to $\omega \dashv^k$ M is eventually in the following configuration:

$$\left|\begin{array}{ll} A_n & \varphi_n \\ A_{n-1} & \varphi_{n-1} \\ \vdots & \\ A_1 & \varphi_1 \end{array}\right|$$

ω_2

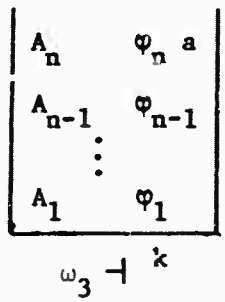
where A_i is the nonterminal associated with the initial state named (X_i, ϵ).

Proof We prove this by induction on the length of the leftmost derivation of $\omega_1(X_n, \varphi_n)\ldots(X_1, \varphi_1)$ from (S_1, ϵ).

Basis If the length of the derivation is zero, then we have yet to read any input, so $\omega_1 = \epsilon$, $\omega_2 = \omega$, and $\omega_1(X_n, \varphi_n)\ldots(X_1, \varphi_1) = (S_1, \epsilon)$. Thus the statement is trivially true in this case, since the condition of the stack before any input is read is precisely as required: one level, with S the only vocabulary symbol on it.

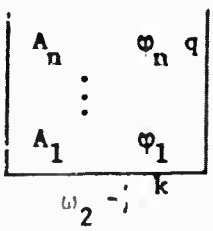
Inductive Step Suppose the lemma is true for derivations of length m; we shall show it true for derivations of length m+1. We shall proceed by considering the different possibilities for the last rule applied in the derivation.

Case I First assume that the last rule applied in the derivation is of the form $(X, \varphi) \rightarrow a(X, \varphi\ a)$. Then we have $(S_1, \epsilon) \overset{m}{\underset{L}{\Rightarrow}} \omega_1(X_n, \varphi_n)\ldots(X_1, \varphi_1) \overset{*}{\underset{L}{\Rightarrow}} \omega_1\ a(X_n, \varphi_n\ a)\ldots(X_1, \varphi_1)$. We have to show that after M has read $\omega_1\ a$, it is in the following configuration:



, where $\omega_1\omega_2 = \omega_1\ a\ \omega_3 = \omega$.

By inductive assumption, we know that after reading ω_1, M is in:



, where q is the state of M.

Assume first that $\varphi_n \neq \epsilon$; so $\varphi_n = \tau\ \sigma$, $\sigma \in V_n \cup V_T$, $\tau \in (V_N \cup V_T)^*$. Since $(X_n, \varphi_n) \rightarrow a(X_n, \varphi_n a)$ is a rule of $T_M(G)$, we know there is a non-final state of M named (X_n, φ_n) and that there is an a-transition from it. Furthermore, q, the state of M after reading ω_1, is named (X_n, φ_n); however we do not know ab initio that it is the non-final state so named. However, we can deduce this by using the preceeding lemma.

Let us examine the topmost stack level of M after reading ω_1; since $\varphi_n = \tau\ \sigma$, we may consider it as: $A_n\ \tau\ q'\ \sigma\ q$. Here q and q' are states of M. The last step in the operation of M entailed an entrance to state q (whether after a read, reduce, or suspension) from state q', on symbol σ and lookahead $\omega_2 \dashv^k/k$. By Lemma 4.10, since $(X_n, \tau\ \sigma)\ldots(X_1, \varphi_n) \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_2 \dashv^k$ starting with $(X_n, \tau\ \sigma) \rightarrow a(X_n, \tau\ \sigma\ a)$, there is a transition from the non-final state named (X_n, τ) (which in our case is q') to the non-final state named $(X_n, \tau\ \sigma)$, on symbol σ with lookahead $\omega_2 \dashv^k/k$. Since the machine M is deterministic, this means that our state q is <u>the</u> non-final $(X, \tau\ \sigma)$ state.

If $\omega_2 = a\omega_3$, we have that $(X_n, \tau\ \sigma\ a)\ (X_{n-1}, \varphi_{n-1})\ldots(X_1, \varphi_1) \dashv^k \underset{L}{\overset{*}{\Rightarrow}}\ \omega_3 \dashv^k$. Then by the lemma again, there is a transition from the non-final $(X_n, \tau\ \sigma)$ state to <u>some</u> $(X_n, \tau\ \sigma\ a)$ state, on symbol a with lookahead $\omega_3 \dashv^k/k$ After reading ω_1, we have ascertained that M is in the non-final state named $(X_n, \tau\ \sigma)$, with the remaining input being $\omega_2 \dashv^k$, or a $\omega_3 \dashv^k$. Then, since M is deterministic, and there is an a-transition from that state in M with lookahead $\omega_3 \dashv^k/k$, the action that M will take at this juncture will indeed be to follow that transition, thus reading the symbol a onto the stack, and leaving M in the desired configuration. (We had to go through this latter analysis to make sure at this point M would not decide, based on the lookahead, to make a prediction.)

If $\varphi_n = \epsilon$, we have $(X_n, \epsilon)\ldots(X_1, \varphi_1) \vdash^k \underset{L}{\Rightarrow} a(X_n, a)\ldots(X_1, \varphi_1) \vdash^k \overset{*}{\underset{L}{\Rightarrow}} a\,\omega_3 \vdash^k$. Then $(X_n, a)\ldots(X_1, \varphi_1) \vdash^k \overset{*}{\underset{L}{\Rightarrow}} \omega_3 \vdash^k$, and so by the lemma, we know that there is an a-transition from the non-final (X_n, ϵ) state to some (X_n, a) state, on symbol a with lookahead $\omega_3 \vdash^k/k$. We know that after reading ω_1, M is in a state named (X_n, ϵ); but can we be sure that this is the initial (X_n, ϵ) state? Perhaps M read an ϵ and went to some final state also named (X_n, ϵ). We can discount this possibility as follows: Even if this were the case, prior to this ϵ-transition, M must have been in the initial (X_n, ϵ) state, with the remaining input being a $\omega_3 \vdash^k$. Then if an ϵ-transition were to be effected, it would be on lookahead a $\omega_3 \vdash^k/k$. But we already know that there is an a-transition from the initial (X_n, ϵ) state with lookahead $\omega_3 \vdash^k/k$; so since M is deterministic, there can be no such ϵ-transition.

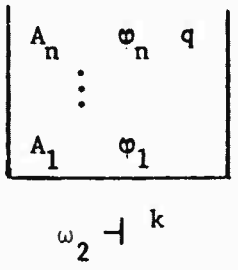
Thus we know that after reading ω_1, M will be in the initial (X_n, ϵ) state, with the remaining input a $\omega_3 \vdash^k$. Then the a-transition from that state will be followed, reading a onto the stack, and achieving the desired configuration.

Case II Assume that the last rule applied in the derivation is of the form $(X, \varphi) \rightarrow (Y, \epsilon)\ (X, \varphi\ Y)$. Then we have $(S_1, \epsilon) \underset{L}{\overset{m}{\Rightarrow}} \omega_1\ (X_n, \varphi_n)\ldots(X_1, \varphi_1) \underset{L}{\Rightarrow} \omega_1\ (Y, \epsilon)\ (X_n, \varphi_n\ Y)\ldots(X_1, \varphi_1) \overset{*}{\underset{L}{\Rightarrow}} \omega_1\ \omega_2$. We want to show that M enters the following configuration:

A_{n+1}	ϵ
A_n	φ_n
A_{n-1}	φ_{n-1}
$\vdots$	$\vdots$
A_1	φ_1

$\omega_2 \vdash^k$

By inductive assumption, we know that one configuration achieved by M after reading ω_1 is the following:



, where q is the state of M.

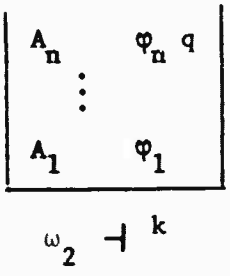
$\omega_2 \dashv^k$

First let us observe that $\varphi_n \neq \epsilon$. Since $(X_n, \varphi_n) \rightarrow (Y, \epsilon)(X_n, \varphi_n Y)$ is a rule of $T_M(G)$, some (X_n, φ_n) state is a base with the (Y, ϵ) initial state an associated initial. But a (X_n, ϵ) state is either itself an initial state, or a final state--neither of which can be a base.

The first thing we have to show is that q is the non-final state named (X_n, φ_n). This proof is similar to that of Case I. Let $\varphi_n = \tau\ \sigma$, and let the known earlier top stack level be $A_n\ \tau\ q'\ \sigma\ q$. The last previous action of M caused it to follow the σ transition from q' to q', where the lookahead was $\omega_2 \dashv^k/k$. By Lemma 4.10, since $(X_n, \tau\ \sigma)\ldots(X_1, \varphi_1) \overset{*}{\underset{L}{\Rightarrow}} \omega_2 \dashv^k$ beginning with the rule $(X_n, \tau\ \sigma) \rightarrow (Y, \epsilon)\ (X_n, \tau\ \sigma\ Y)$, there is a transition from q' on σ, with lookahead $\omega_2 \dashv^k/k$, to the non-final state named $(X_n, \tau\ \sigma)$. Thus q is this non-final $(X_n, \tau\ \sigma)$ state.

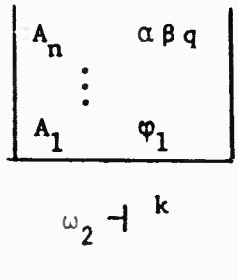
We have that $(Y, \epsilon)\ (X_n, \tau\ \sigma\ Y)\ldots(X_1, \varphi_1) \dashv^k \overset{*}{\underset{L}{\Rightarrow}} \omega_2 \dashv^k$. Then by the first clause of Lemma 4.10, there is a predictive transition from some base state to the (Y, ϵ) initial state on lookahead $\omega_2 \dashv^k/k$. Since $(X_n, \varphi_n) \rightarrow (Y, \epsilon)\ (X_n, \varphi_n\ Y)$ is a rule, the non-final (X_n, φ_n) state must be the base state for the (Y, ϵ) initial state.

Let us recapitulate. We know that after reading ω_1, M will at some time be in state q, the non-final (X_n, φ_n) state, with the remaining input being $\omega_2 \dashv^k$. Furthermore, we know that there is a predictive transition from that (X_n, φ_n) state to the (Y, ϵ) initial state, on lookahead $\omega_2 \dashv^k/k$. Therefore we can be sure that in the configuration



M's action will be to jump to the predictive state, causing a new stack level to be created, and leaving M in the required configuration.

Case III Here we assume that the last rule in the derivation is of the form $(X, \varphi) \rightarrow (X, \varphi')$. That is, for some α, β, and A, we have that $\varphi_n = \alpha\ \beta$ and that $(S_1, \epsilon) \overset{m}{\underset{L}{\Rightarrow}} \omega_1\ (X_n, \alpha\ \beta)\ldots(X_1, \varphi_1) \overset{=}{\underset{L}{\Rightarrow}} \omega_1\ (X_n, \alpha\ A)\ldots(X_1, \varphi_1) \overset{*}{\underset{L}{\Rightarrow}} \omega_1\ \omega_2$. We know by hypothesis that M will reach the configuration



It remains to show that q is the final state corresponding to the rule $A \rightarrow \beta$ of G.

If σ represents the last symbol of β, we can set $\alpha\ \beta = \tau\ \sigma$. Then the top level of M's stack is $A_n\ \tau\ q'\ \sigma\ q$. Now we have $(X_n, \tau\ \sigma)\ldots(X_1, \varphi_1) \dashv^k \overset{*}{\underset{L}{\Rightarrow}}$

$\omega_2 \dashv^k$, where the first rule applied is $(X_n, \alpha\ \beta) \rightarrow (X_n, \alpha\ A)$. Then by Lemma 4.10, there is a transition on σ from q' to the final state for $A \rightarrow \beta$, with lookahead $\omega_2 \dashv^k/k$. Now the last step performed by M involved following the σ transition from q' with lookahead $\omega_2 \dashv^k/k$; therefore q is the final state for $A \rightarrow \beta$, and M will perform the reduction, leaving the required configuration. (Note that this works as well for the case that β, and hence σ, is ϵ.)

Case IV We assume that the last rule applied in the derivation is $(X, X) \rightarrow \epsilon$. That is, $(S_1, \epsilon) \overset{m}{\underset{L}{\Rightarrow}} \omega_1 (X_n, X_n)(X_{n-1}, \varphi_{n-1}) \ldots (X_1, \varphi_1) \underset{L}{\Rightarrow} \omega_1 (X_{n-1}, \varphi_{n-1}) \ldots (X_1, \varphi_1) \overset{*}{\underset{L}{\Rightarrow}} \omega_1 \omega_2$. The known configuration of M is

$$\left|\begin{array}{ll} A_n & X_n\ q \\ \vdots & \\ A_1 & \varphi_1 \end{array}\right|$$
$$\omega_2 \dashv^k$$

we must show that q is the POP state.

This case is easy. We have that $(X_n, X_n)(X_{n-1}, \varphi_{n-1}) \ldots (X_1, \varphi_1) \dashv^k \underset{L}{\Rightarrow} (X_{n-1}, \varphi_{n-1}) \ldots (X_1, \varphi_1) \dashv^k \overset{*}{\underset{L}{\Rightarrow}} \omega_2 \dashv^k$. Then by Lemma 4.10, there is an X_n-transition from the (X_n, ϵ) initial state to the POP state, on lookahead $\omega_2 \dashv^k/k$. Now the top level of the stack is $A_n\ q'\ X_n\ q$, where q' is the (X_n, ϵ) initial state. The entrance to q was made by following an X_n-transition with lookahead $\omega_2 \dashv^k/k$ from the (X_n, ϵ) initial state. Therefore q is the POP state, and M will suspend the top level, leaving the desired configuration.

This is the final case for our induction argument, and so the proof is done. Q.E.D.

Lemma 4.11 has a number of interesting consequences, but the most important is the following.

Theorem 4.12 Let M be a cycle-free MSP(k) machine for G, $T_M(G)$ the derived grammar. Then $T_M(G)$ is strong LL(k), if $k > 0$; if $k = 0$, $T_M(G)$ is LL(1).

Proof We use the definition of strong LL(k) given by Rosenkrantz and Stearns [13]: G is strong LL(k) if and only if given a word $\omega \in V_T^k$ and a nonterminal A, then there is at most one production p such that for some ω_1, ω_2, and ω_3 in V_T^*:

1) $S \overset{*}{\Rightarrow} \omega_1 A \omega_3$

2) $A \overset{*}{\Rightarrow} \omega_2$ beginning with an application of production p

3) $\omega_2 \omega_3 : k = \omega$.

We want to recast this definition in terms of leftmost derivations. It is clear that $S \overset{*}{\Rightarrow} \omega_1 A \omega_3$ if and only if $S \overset{*}{\underset{L}{\Rightarrow}} \omega_1 A \Psi$, for some Ψ such that $\Psi \overset{*}{\underset{L}{\Rightarrow}} \omega_3$. Furthermore $A \overset{*}{\Rightarrow} \omega_2$ beginning with rule p if and only if $A \overset{*}{\underset{L}{\Rightarrow}} \omega_2$ beginning with p. Thus we may give the following characterization: G is strong LL(k) if and only if given $\omega \in V_T^k$ and a nonterminal A, then there is at most one production p such that for some ω_1 and $\omega_2 \in V_T^*$ and $\Psi \in (V_N \cup V_T)^*$:

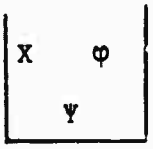
1) $S \overset{*}{\underset{L}{\Rightarrow}} \omega_1 A \Psi$

2) $A \overset{*}{\underset{L}{\Rightarrow}} \omega_2$, beginning with the application of production p

3) $\omega \in \text{FIRST}_k(\omega_2 \Psi)$.

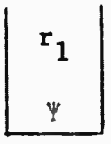
Let us assume then that $T_M(G)$ is not strong LL(k). Then for some ω_1 and ω_2, and some nonterminal (X, φ), we have $(S_1, \epsilon) \vdash^k \overset{*}{\underset{L}{\Rightarrow}} \omega_1 (X, \varphi) \Psi \vdash^k$ and $(X, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega_2$ beginning with rule p_1 and $(X, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega_2$ beginning with rule p_2.

Let ω_3 be any string such that $\Psi \overset{*}{\underset{L}{\Rightarrow}} \omega_3$, and let us consider the result of applying the string $\omega_1\omega_2\omega_3$ to the machine M. By Lemma 4.11, after reading ω_1, the configuration of M will be:



$\omega_2\omega_3$

Since $(S_1, \epsilon) \overset{*}{\Rightarrow} \omega_1\omega_2\omega_3$, M will accept the string $\omega_1\omega_2\omega_3$. Therefore there must be some next configuration of M; furthermore, since M is deterministic, there is only one such possible next configuration. But what will it be? Suppose production p_1 is $(X, \varphi) \rightarrow r_1$; then by Lemma 4.11, since $(X, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega_2$ beginning with production p_1, the next configuration of M should be:



$\omega_2\omega_3$

or very similar to it (if r_1 is $a(X, \varphi a)$ then the top stack level will be $(X, \varphi a)$ while the first symbol will be removed from $\omega_2\omega_3$). Similarly, if p_2 is $(X, \varphi) \rightarrow r_2$, since $(X, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega_2$ beginning with p_2, the next configuration should be:

But since p_1 and p_2 are different productions with the same left-hand side, it must be that $r_1 \neq r_2$. Thus there must be two different next configurations for M, which is impossible, since M is deterministic. Thus $T_M(G)$ is strong LL(k).

We observe that this analysis works both for the case where $k > 0$, in which case it show that $T_M(G)$ is strong LL(k); and for $k = 0$, where it can be used to show that $T_M(G)$ is strong LL(1). The difference for the case where $k = 0$ is that the machine, even an MSP(0) one, always needs to examine the first symbol of the lookahead when it reads it, in order to decide which state to go to. Hence the derived grammar has a similar property and so is LL(1). Q.E.D.

4.6 Discussion and Explanation

We are now in a position to look back on what we have done in this chapter and to try to get some perspective on it. We began with the notion of a cycle-free MSP(k) machine, an MSP(k) machine that did not have indefinitely long paths through it and so would never need more than a finitely wide stack level at any point during a parse. We then proceeded to name the states of such a machine; intuitively, we gave a state the name (X, φ) if it could be reached by a path spelling φ from an initial state corresponding to the prediction of an X. (In this way, several states could have the same name while any one state might have several names.) Since M was cycle-free, we could be assured that there would be only finitely many such names assigned in total. Then using these names as a

set of nonterminals, we derived a grammar from the machine M, and denoted it $T_M(G)$. The rules of this grammar were of four different types:

$(X, \varphi) \rightarrow a(X, \varphi a)$; $(X, \varphi) \rightarrow (Y, \epsilon)(X, \varphi Y)$; $(X, \varphi\alpha) \rightarrow (X, \varphi A)$; $(X, X) \rightarrow \epsilon$.

The first type of rule would be created if there was an a-transition in M from a state named (X, φ); the second if some state named (X, φ) were the base for the prediction of Y; the third if some $(X, \varphi\alpha)$ state was a final state for the rule of G, $A \rightarrow \alpha$; and the fourth for every predicted nonterminal.

We note in passing here that it is only for convenience that we have described the derivation of $T_M(G)$ as a two-step process, first the naming of the states and then the construction of the rules. It should be apparent that these two steps could be combined into one, that the states could be named "on the fly" as the rules were being generated.

We have established some relationships between derivations in the grammar $T_M(G)$ and the processing of strings by the machine M. In Lemma 4.4 we showed that if ω was a string accepted by the machine M, then there was a leftmost derivation of ω in $T_M(G)$, such that the sequence of sentential forms of the derivation represented the sequence of stack configurations M was passing through while processing ω. In other words, the leftmost derivation of ω in $T_M(G)$ simulated the MSP(k) parse of ω by M. The converse of this was proved in Lemma 4.11. There we showed that if ω was a string in $L(T_M(G))$, then if we gave ω to M, the sequence of stack configurations M would go through would precisely mimic the sequence of sentential forms in the leftmost derivation of ω in $T_M(G)$. These two lemmas taken together show that there is a one-one relationship between leftmost derivations in $T_M(G)$ and MSP(k) parses by M.

This suffices to show that $L(T_M(G)) = L(M)$. Since M is an MSP(k) machine for G, we already know that $L(M) = L(G)$; thus we have that the transformed grammar $T_M(G)$ generates precisely the same set of strings as the original grammar G. Furthermore, since M operates deterministically, we were able to deduce in Theorem 4.12 that the process of leftmost derivations in $T_M(G)$ was also deterministic--in other words, that $T_M(G)$ was strong LL(k).

Actually, we showed that $L(T_M(G)) = L(M)$ in a slightly different manner, by relating the meaning of nonterminals in $T_M(G)$ to those in G. We showed in Lemma 4.6 that if $(X, \varphi) \overset{*}{\underset{L}{\Rightarrow}} \omega$ in $T_M(G)$, where ω is a string of terminals, then $X \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$ in G. This result is actually interesting in its own right, because it begins to give us some insight into the nature of $T_M(G)$. Because of this lemma, we can begin to think of the nonterminal (X, φ) in $T_M(G)$ as generating "the rest of an X (in G) after φ". Thinking in these terms, the rules of $T_M(G)$ make a lot of sense. For example, $(X, \varphi) \rightarrow a(X, \varphi a)$ means that one possibility for the rest of an X after φ is an a followed by the rest of an X after φa; which indeed makes sense, if we think of X generating (in G) φ a τ for some τ. Then the rest of the X after the φ is a τ, while the rest of the X after φa is just τ.

Similarly, $(X, \varphi\ \tau) \rightarrow (X, \varphi\ A)$ just means that the rest of an X after $\varphi\ \tau$ may be the same as the rest of an X after φA. And this is indeed reasonable, if $A \rightarrow \tau$ is a rule of G. Or that $(X, X) \rightarrow \epsilon$ just means that the rest of an X after an X is just ϵ.

We do not want to stretch this analogy too far, though it is useful for an intuitive feeling for $T_M(G)$. First of all, Lemma 4.6 was not an "if and only if"; it is not the case that (X, φ) necessarily generates all of the strings which might be called the rest of an X after φ, only that any string which (X, φ) does generate may be so described. In particular, (X, ϵ) does not necessarily generate all strings which X generates in G; specifically, (X, ϵ) generates precisely those strings whose generations from X in G begin with one of the X-rules underlying an item in the (X, ϵ) initial state. This follows directly from the proof of Lemma 4.6. In particular, if X is left recursive in G, some of the rules for X may be "left back" in the base state of the splitting, and so (X, ϵ) would not generate any strings generated in G using these rules. By analogy, when an MSP(k) machine enters a predictive state and predicts that an X will be found, it is not any X that will be found and that will satisfy the prediction; rather, an X whose generation begins or whose parse ends with one of the items included in the predictive state. This notion of (X, φ) as generating the rest of an X after φ extends to the machine M also; if in a given configuration of M, the topmost stack level is $X\varphi$, then the portion of the remaining input that must be read in order to cause this level to be suspended is also a string ω such that $X \overset{*}{\underset{R}{\Rightarrow}} \varphi\omega$ in G. That is, a string which is the rest of an X after φ will complete the parse at that level.

There is one other important piece of information that can be gleaned from Lemma 4.6 and from the other main lemmas. We know that if $(S, \epsilon) \overset{*}{\underset{L}{\Rightarrow}} \omega$ in $T_M(G)$, then $S \overset{*}{\underset{R}{\Rightarrow}} \omega$ in G. But we know even more than this; we know how to reconstruct the tree for ω in G from the tree for ω in $T_M(G)$. Let ω be a string in L(G),

and consequently in L(M) and $L(T_M(G))$. We know that applying M to ω produces the LR(k) parse of ω according to G; that is, the order in which reductions are performed by M are the same in which they would be performed by the LR(k) machine for G doing a conventional bottom-up parse of ω. Thus these reductions can be pieced together appropriately to form the tree for ω in G.

We have established that there is a relationship between the stack configurations M goes through while parsing ω and the sequence of sentential forms in the leftmost derivation of ω in $T_M(G)$. In particular, performance of a reduction by the machine M causes the topmost level of the stack to change from X $\alpha\beta$ to X αA, without an input symbol being read. The only kind of rule in $T_M(G)$ that effects a similar transformation on the sentential form is the rule $(X, \alpha\beta) \rightarrow (X, \alpha A)$. Thus in order to reconstruct the LR(k) parse of ω in G from the leftmost derivation of ω in $T_M(G)$, we may perform the following procedure. The leftmost derivation of ω in $T_M(G)$ is a series of rules of $T_M(G)$, $r_1 r_2 \ldots r_n$. Consider the subsequence of this list which consists just of those rules of the form $(X, \alpha\beta) \rightarrow (X, \alpha A)$. From this subsequence, form the corresponding list of rules of G, $A \rightarrow \beta$. This list will give the order of reductions performed by an LR(k) parse of ω - i.e., the list is the bottom-up parse, and can be used to construct the tree for ω in G. Furthermore, the sequence of rules of $T_M(G)$ used in the leftmost derivation of ω is precisely the information provided by the deterministic top-down parse of ω, i.e., the LL(k) parse. We may summarize all this information in the following statement.

Theorem 4.13 The LR(k) parse for ω in G can be reconstructed from the LL(k) parse of ω in $T_M(G)$.

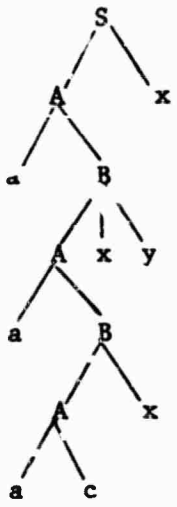


Figure 4.11

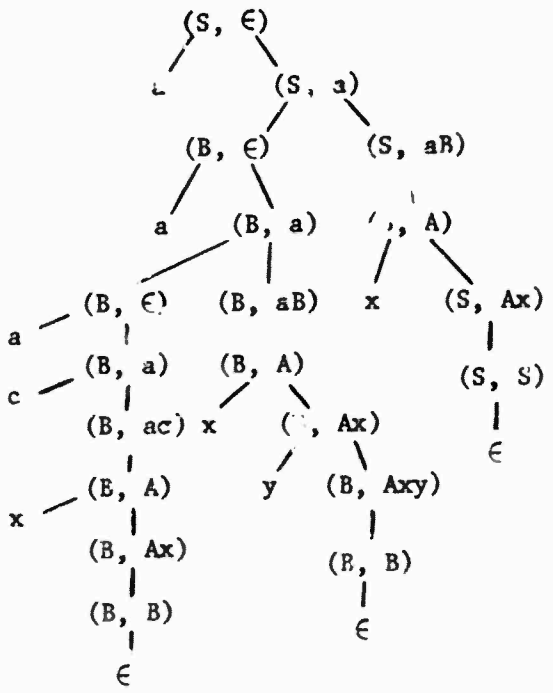


Figure 4.12

Let us give an example of how this reconstruction is performed. Recall the grammar G and its associated cycle-free MSP(1) machine shown in Figure 4.2, with the derived grammar $T_M(G)$ shown in Figure 4.5. Consider the string aaacxxyx; the syntax tree for this string in G is shown in Figure 4.7 , while the one for $T_M(G)$ is given in Figure 4.12.

The following, then, is the sequence of rules of $T_M(G)$ which is the left-most derivation of ω: (S, ϵ) → a(S, a); (S, a) → (B, ϵ) (S, aB); (S, ϵ) → a(B, a); (B, a) → (B, ϵ) (B, aB); (B, ϵ) → a(B, a); (B, a) → c(B, ac); (B, ac) → (B, A), (B, A) → x(B, Ax); (B, Ax) → (B, B); (B, B) → ϵ; (B, aB) → (B, A); (B, A) → x(B, Ax); (B, Ax) → y(B, Axy); (B, Axy) → (B, B); (B, B) → ϵ; (S, aB) → (S, A); (S, A) → x(S, Ax); (S, Ax) → (S, S); (S, S) → ϵ. The subsequence of rules of the form (X, αβ) → (X, αA) is: (B, ac) → (B, A); (B, Ax) → (B, B); (B, aB) → (B, A); (B, Axy) → (B, B); (S, aB) → (S, A); (S, Ax) → (S, S). The sequence of rules of G that can be derived from this sequence is: A → ac, B → Ax, A → aB, B → Axy, A → aB, S → Ax; and this is precisely the left-to-right, bottom-up sequence of rules used in the tree for ω in G.

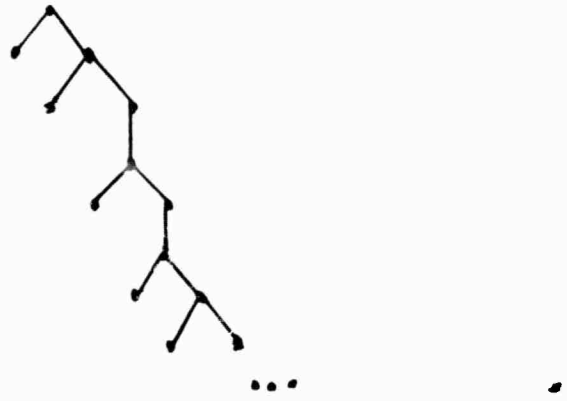
We thus see that there is no essential loss of information in going from G to $T_M(G)$, that the G tree can be recovered from the $T_M(G)$ tree. But a casual inspection of the two preceding figures indicates that there can be great structural differences between these two trees. In order to try to understand the transformational effect of T_M on trees, let us attempt another view of the grammatical transformation.

Suppose M were a cycle-free LR(k) machine for G, a machine without cycles and without initial predictive states (other than the starting state). Then it is well known that the language M accepts is regular, that M has no more power than a finite-state machine. Since M is cycle-free, M's stack can only grow to a finite bounded depth, so the information that would be contained on M's stack can be coded into the state structure of the equivalent finite-state machine. Alternatively, there will be a right-linear grammar whose nonterminals stand for the states of the FSM and whose derivations simulate the actions of this machine and hence the successive conditions of M's stack. And this is exactly what our grammar $T_M(G)$ would be in this case. It would basically have two kinds of rules, either of the form $(S, \varphi) \rightarrow a(S, \varphi a)$ or $(S, \alpha\beta) \rightarrow (S, \alpha A)$, plus in addition the single rule $(S, S) \rightarrow \epsilon$. (This would be all, since there are no predictions in M.) This grammar would indeed be right-linear; and the second part of a nonterminal name could quite clearly be seen as representing the progress of M's processing. The rule $(S, \varphi) \rightarrow a(S, \varphi a)$, when used in a derivation, means that if M had φ on the stack with a as the next input symbol, then M would read a onto the stack. And $(X, \alpha\beta) \rightarrow (X, \alpha A)$ means that M, with $\alpha\beta$ on the stack, would decide to reduce β to A. In other words, the details and nature of the grammar G get somewhat garbled in transforming to $T_M(G)$; it is primarily the structure of the machine M derived from G that influences the structure of $T_M(G)$.

When we consider M as a general cycle-free MSP(k) machine for G, much of this approach is still valid. It is no longer true that $T_M(G)$ will be regular; in general, there will be ϵ-linear rules like $(X, \varphi) \rightarrow (Y, \epsilon)$ $(X, \varphi Y)$. But we can think of $T_M(G)$ as being "almost" regular, of being composed of a number

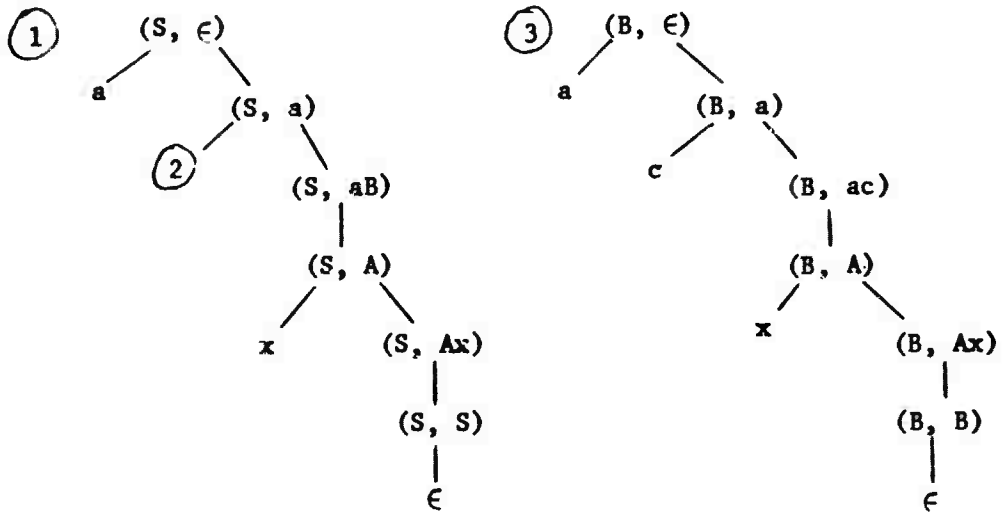
of smaller almost regular grammars, each with a different sentence symbol (X, ϵ). All the nonterminals (X, φ) for a given X may be thought of as keeping track of the status of one stack (level), of bounded depth, in much the same way that φ) kept track of the whole stack in the cycle-free case. It is only the predictive rules that disturb the regularity of this picture; these are the rules that correspond to starting new stack levels in the more complicated MSP(k) machine structure, and that serve to glue together the various sub-grammars, each of which suffices for a single stack level. And the way in which we defined state-splitting was designed in such a way that the top-down parser would always know when to switch to a new subgrammar, because the machine would deterministically know when to start a new level. The two conditions - cycle-free and MSP(k) machine - ensure two different properties of the machine that make the derived grammar desirable. The first assures us that a finite-state grammar will suffice to describe the actions of any one stack level, while the second ensures that the decision on which sub-grammar to use is determinable in advance. The LL(k)-ness of $T_M(G)$ derives from the fact that $T_M(G)$ is basically a number of finite-state grammars cleverly pasted together, so that one grammar knows when to turn control over to another.

With all this in mind, we can begin to think about the effect of T_M on trees. Since $T_M(G)$ is LL(k) even if G is not, we can be sure that T_M will change the orientation of the trees, replacing left recursion in G by right recursion in $T_M(G)$. But the effect is far more than that. For a given stack level in M, i.e., an (X, ϵ) node in the $T_M(G)$ tree, the tree's structure mimics the activities of M. Thus the tree will look locally like:



The bifurcating branches are for rules like $(X, \varphi) \rightarrow a(X, \varphi a)$, while the single branches are for $(X, \alpha\beta) \rightarrow (X, \alpha A)$. This represents the activities of M: read, read, until it's time for a reduction, then reduce, then read again.

The full tree is obtained by pasting a number of these local trees together. That is, occasionally the branching is for a rule $(X, \varphi) \rightarrow (Y, \epsilon)$ $(X, \varphi Y)$; in that case, the left node of the branching is just the root node of another such local tree. It is in this way that the tree at large is constructed from smaller sub-trees. The tree of Figure 4.12 is made up of three subtrees, with roots (S, ϵ), (B, ϵ), and (B, ϵ), as shown in Figure 4.13. If we label these trees 1, 2, and 3, the figure indicates how they fit together, how the root of one is effectively a leaf of another.


1
(S, ∈)
a
(S, a)
2
(S, aB)
(S, A)
x
(S, Ax)
(S, S)
∈
3
(B, ∈)
a
(B, a)
c
(B, ac)
(B, A)
x
(B, Ax)
(B, B)
∈

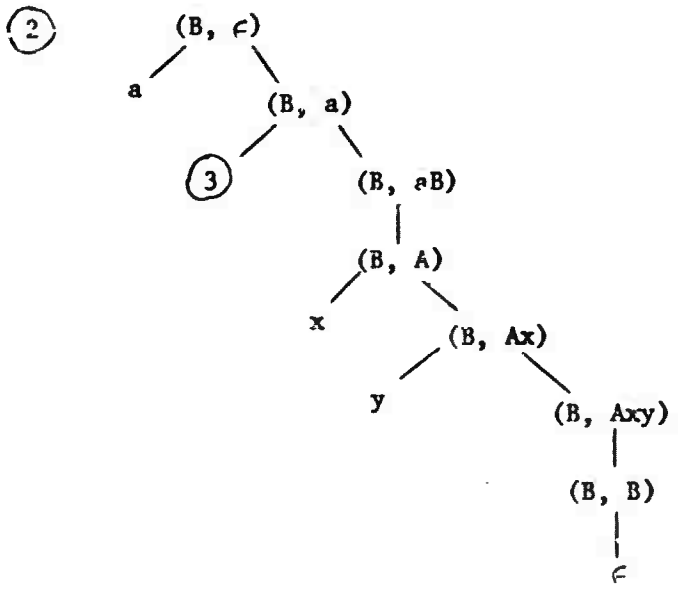

2
(B, ∈)
a
(B, a)
3
(B, aB)
(B, A)
x
(B, Ax)
y
(B, Axy)
(B, B)
∈

Figure 4.13

4.7 Compiling Power of $T_M(G)$

While it is certainly gratifying to know that $T_M(G)$ generates the same language as G and is LL(k) to boot, this information by itself is insufficient to prove the value and worth of the transformation. For we are not interested only in the problem of pure parsing, but in the rather more complex issue of compiling; and the fact that $T_M(G)$ can be deterministically parsed in a top-down fashion is only one aspect of this problem. We must convince ourselves that $T_M(G)$ is at least as useful as G, in terms of directing the compilation of sentences of $L(G)$. That is, we wish to show that all transformations on the input performed by a compiler which is driven by an LR(k) parser for G can also be effected by some compiler driven by an LL(k) parser for $T_M(G)$. This issue is particularly sensitive since we have just seen how T_M can distort the phrase structure of a sentence in the language; we must be reassured that this distortion is not so violent as to make the new grammar unusable in compilation. We are of course referring to a single-pass compilation process here, where the parser makes occasional calls on "semantic routines" which produce as output some representation of the meaning of the program being parsed; since it is possible to reconstruct the G-tree for a string from its $T_M(G)$-tree, any multipass compilation scheme based on G can be simulated by a similar scheme based on $T_M(G)$.

The most popular formal model for the compilation process is that of syntax-directed translations, as developed by Lewis and Stearns [13]. We briefly review their terminology.

Definition 4.14 A translation grammar based on the grammar G is a triple (G, V_T', g), where V_T' is a set of output terminal symbols disjoint from

V_T, and g is a mapping which takes the right-hand side of the rule $A \to \alpha$ of G into a string $g(A, \alpha)$ over $(V_N \cup V_T')^*$ such that the nonterminals of $g(A, \alpha)$ are some permutation of the nonterminals of α. We refer to $g(A, \alpha)$ as the translation element for the rule $A \to \alpha$.

We shall usually represent a translation grammar by writing the translation element for a rule in braces next to the rule itself. For example, $A \to aBC\{xCyB\}$ might be one rule of a translation grammar.

Definition 4.15 If for each rule $A \to \alpha$ of G, the nonterminals of α appear in the same order in $g(A, \alpha)$ as they do in α, then (G, V_T', g) is called a simple translation grammar. If the nonterminals of $g(A, \alpha)$ appear to the left of the terminals of $g(A, \alpha)$, then (G, V_T', g) is a simple Polish translation grammar.

Thus $A \to aBC\{xByCz\}$ is not a simple Polish rule (though it is simple) while $A \to aBC\{BCz\}$ is simple Polish.

The purpose of a translation grammar is to specify for each string $\omega \in L(G)$, a string ω' over $(V_T')^*$, which will be called the translation of ω; this represents the alternative form of ω into which we wish to compile it. The translation elements specify how ω' is defined in terms of ω, as follows.

Definition 4.16 If (G, V_T', g) is a translation grammar, then the associated grammar G' has nonterminals V_N, terminals V_T', and its rules are given by: if $A \to \alpha$ is a rule of G, then $A \to g(A, \alpha)$ is a rule of G'.

Definition 4.17 Every derivation in G has an associated derivation in G', which is obtained by substituting associated rules for corresponding nonterminals. If ω is in L(G), then the derivation in G' associated with the

the derivation of ω in G generates a terminal string over $(V_T')^*$, called the translation of ω.

This last definition can be made more precise and formal (see [1], p. 219), but for our purposes this specification will suffice.

Definition 4.18 Let (G, V_T', g) be a translation grammar. Then $\{(x,y) \mid x \in L(G)$ and y is the translation of x$\}$ is the translation defined by the translation grammar.

We note that we will be restricting our attention to LR(k) grammars, which are unambiguous; hence every string will have a unique translation.

The translation of a string may be interpreted either as the representation of the meaning of that string in terms of some intermediate language, or as a sequence of calls to semantic "action routines" which will perform the appropriate compilation activities for that string, or even as some mixture of the two concepts.

This model has proven to be an understandable and useful model of compiling. Writing a translation element for each rule can frequently be a convenient and simple method of specifying in a local way the transformation which is to be performed by a compiler for the language; for each rule, the translation element describes what actions are to be taken during and after the recognition of that rule. The question then naturally arises as to whether and how the translation specified by a translation grammar can be actually implemented by some formal automata-theoretic model. The kind of model that has customarily been used for these purposes is the (deterministic) pushdown transducer [1,13], which is basically a pushdown machine with one added feature: namely, when making any move the transducer can emit as out-

put some string of symbols from some new output vocabulary. This model is more fully defined in [1, 2, 13]. We shall briefly restate some of the principal results pertaining to this model, with reference to translations defined on LL(k) and LR(k) grammars.

Definition 4.19 A pushdown transducer performs (or implements) a translation grammar based on G if the machine accepts precisely L(G) and if in accepting $\omega \in L(G)$ it produces as output the translation of ω.

Theorem 4.20 Any simple translation based on an LL(k) grammar G can be performed by a deterministic pushdown transducer.

Sketch of proof: Since G is an LL(k) grammar, it can be recognized by some deterministic pushdown acceptor. We modify this acceptor in the following way. Consider any rule of G, $A \to a_1 B_1 a_2 B_2 \cdots a_n B_n a_{n+1}$, where each a_i is either ϵ or a string of terminal symbols; the general simple translation element that can be associated with this rule is $\{x_1 B_1 x_2 B_2 \cdots x_n B_n x_{n+1}\}$, where each x_i is some string of output symbols. Now for some configurations of the machine, with A on top of the stack and some lookahead, the machine will specify that A is to be replaced by $a_1 B_1 \cdots B_n a_{n+1}$ on the stack. We change these entries in the acceptor's state table so that A will be replaced on the stack in these cases by $a_1 \bar{x}_1 B_1 a_2 \bar{x}_2 B_2 \cdots a_n \bar{x}_n B_n a_{n+1} \bar{x}_{n+1}$, where $\bar{x}_i$ is a new symbol unique for x_i. We further modify the machine so that when $\bar{x}_i$ is on the top of the stack, it is popped off, and x_i is emitted as output.

A related statement pertains to the capability of translating LR(k) grammars.

Theorem 4.21 Any simple Polish translation based on an LR(k) grammar G can be performed by a deterministic pushdown transducer.

Sketch of proof: Since G is an LR(k) grammar, it can be recognized by some deterministic pushdown acceptor. We shall modify this acceptor as follows. If $A \rightarrow a_1 B_1 \ldots B_n a_{n+1}$ is a rule of G, the most general simple Polish translation element for this rule is $\{B_1 \ldots B_n x\}$. We then alter the acceptor so that when it reduces $a_1 B_1 \ldots B_n a_{n+1}$ to A, by popping the former off the stack and replacing it by A, it also emits x as output.

There is a converse to this latter result, namely that any translation performed by a deterministic pushdown transducer can be specified as a simple Polish translation on some LR(1) grammar.

These results can be interpreted on an intuitive basis. If G is LL(k), then it can be parsed deterministically top-down, and so the right-hand side of a rule can be identified before it has been read; thus the identity of the output symbols that have to be emitted for that rule can be determined in advance, and some output symbols which are peculiar to that rule can be emitted before the entire right-hand side has been read. For an LR(k) grammar, however, a rule is only identified once its end has been reached; while output symbols that call for emission at the end of the rule can be safely determined at that time, it is too late by then to emit symbols that should have been put out earlier.

These results can also be misinterpreted and be made to seem much more significant than they actually are. It has sometimes been said that these results imply that the most general kind of translation that can be performed by a translator based on an LL(k) (LR(k)) parser for G are those that

can be described by simple (simple Polish) translation grammars based on G. This is not the case, and only appears to be the case because of the paucity of the translator model usually employed. Sometimes these results are used as an argument for preferring LL(k) parsing of a particular grammar G to LR(k) parsing for G, the argument being that more translations expressed on G can be implemented by a translator based on an LL(k) parser than by one based on an LR(k) parser for G. This statement is simply not the case. Aho and Ullman have shown in a recent paper [2], that any simple translation on an LL(k) grammar G can be performed by a translator based on the LR(k) parser for G. In other words, while any simple Polish translation on an LR(k) grammar can be performed by a deterministic pushdown translator, it may be the case that some non-simple Polish translations can also be performed by a translator based on the LR(k) parser for G.

In order to focus clearly on the appropriate issues of translations, let us reestablish the kind of context in which our transformation might be applied. The scenario is that a language designer has specified an LR(k) grammar for his language, which reflects his conception of the language's primitives and features; and that he also has defined the semantics of the language in compilation-oriented terms, that is, by means of a translation grammar or its equivalent. Now he would like to obtain an equivalent grammar, but one more suitable for use in a compiler; an LL(k) grammar, for which a compiler can be cleanly and efficiently designed and implemented. So he supplies his grammar to a transforming program which finds a cycle-free MSP(k) machine for the grammar (as described in the next chapter), and which then derives an LL(k) grammar from that machine. However, we want to be sure that a compiler which uses this new LL(k) grammar can implement the kind of

compilation activities which the designer specified for his original grammar. In particular, even if the original translation grammar were not simple Polish, but could nevertheless have been performed by a compiler which used an LR(k) parser for the original grammar, then it behooves us to demonstrate that such a translation can also be effected by some compiler using the new grammar. Therefore we want to look more closely at the kinds of translations describable on LR(k) grammars that can be implemented by compilers using LR(k) parsers; these are the kinds of translations we want to be able to implement on a compiler based on an LL(k) parser for the transformed grammar. Thus while it would be straightforward to show that every simple Polish translation of G has an equivalent simple translation on $T_M(G)$, this is only part of what we want to establish.

In order to minimize (hopefully) confusing terminology, from now on we shall use the term "compiler" rather than "translator," to refer to a machine model that parses and emits output. A compiler will be based on a particular parser if it emits output in conjunction with the actions of that parser in recognizing a string. We shall have occasion to refer to compilers based on LR(k) parsers as well as to compilers based on MSP(k) parsers; rather than make separate definitions for these two cases, we make just one, for the MSP(k) case: and the LR(k) compiler is just a special case of that.

Definition 4.22 Let M be an MSP(k) machine for G. Then P, a compiler based on M, is a triple (M, V_T', δ), where V_T' is a finite output vocabulary and δ is the output function, which takes $(Q_1 \cup Q_2 \cup Q_3) \times (V_N \cup V_T \cup \{\epsilon\}) \times V_T^k$ into $(V_T')^*$. A configuration of such a translator is a quadruple $(q, \alpha, \omega, \gamma)$, where the first three components are the configuration of M and $\gamma \in (V_T')^*$ is the output emitted so far. As for successor configurations, we define (q, α, ω, y)

$\vdash_P$ (q', α', ω', y') as follows: q', α', and ω' are as given by (q, α, ω) $\vdash_M$ (q',α', ω'); if (q', α', ω') is a read-successor of (q, α, ω), then q' = f_M(q, a, τ) for some a and τ, so we let y' = y $\cdot$ δ_P(q, a, τ); if (q', α', ω') is a reduce-successor of (q, α, ω), then q' = $f_M(q_1, A, \tau)$, for some q_1, A, and τ, so we let y' = y $\cdot$ $\delta_P(q_1, A, \tau)$; otherwise y' = y.

All this says is that we associate an output string with every non-predictive transition between states of M; that output is emitted whenever that transition is followed, whether that be after a read or a reduce.

Definition 4.23 If P is a compiler based on the MSP(k) machine M, then the translation defined by P is {(x,y) such that $(q_0, S\ q_0, x \dashv^k, \epsilon) \vdash_P^* (POP, S\ q_0\ S\ POP, \dashv^k, y)$}. If (x,y) is in the translation defined by P, then $x \in L(M)$ and y is called the translation of x.

Definition 4.24 If (G, V_T', g) is a translation grammar, and P is a compiler for G, then P implements (G, V_T', g) if for each $\omega \in L(G)$, the translations of ω defined by P and by (G, V_T', g) are the same.

This general machine-oriented notion of a translation is only partly helpful. We are not really interested in the full range of translations that can be computed by a compiler based on an LR(k) parser, because people do not, in general, conceive of the meaning of a program purely and directly in terms of how a compiler should operate on it and produce output; the translation grammar is a much more effective model for the description of the semantics of a grammar. We want to characterize those translation grammars that can be implemented by a compiler. We already know that simple Polish translations on an LR(k) grammar can be implemented by a compiler based on the LR(k) parser for the gram . But that is not necessarily the only kind of translation that can be so implemented. For example, suppose

that A → aBb was the only rule of G which contained the terminal symbol a, and suppose its translation element is {x By}. Then even though this is not a simple Polish translation element, an LR(k)-based compiler could nonetheless handle it correctly; namely, on making an a-transition from an LR(k) state containing A → . aBb(τ), it would emit output x. The problems of course arise if there are rules like A → aBb{xBy} and A → aBc{uBv}; then during a bottom-up parse, it really is impossible to know whether to emit x or u on the a until the end of the rule has been reached. And of course, by then it is too late to make the decision and emit the appropriate output, because by then the output for the B will already have been generated.

We try to capture the idea of a compiler which is constructed from the specification of a translation grammar in the following definition.

Definition 4.25 Let (G, V_T', g) be a translation grammar based on the LR(k) grammar G. A compiler P based on the LR(k) parser for G is designed to implement (G, V_T', g) if the following two conditions hold:

i) P implements (G, V_T', g)

ii) let $A \to \alpha\sigma\beta$ be any rule of G; then there is a string $y \in (V_T')^*$ such that if $A \to \alpha.\sigma\beta(\tau)$ is an item of state q and $\omega \in FIRST_k(\beta\tau)$, then $\delta_p(q, \sigma, \omega) = y$.

The meaning of this definition is that each symbol of each rule is to have an output associated with it, and that when the parser locates a symbol of a rule, the compiler emits that symbol's associated output.

The motivation for this definition is that it is too easy for a randomly designed compiler based on the LR(k) parser for G to "accidentally" implement a translation grammar, without having been explicitly designed to

do so. And since such compilers may be capriciously constructed and may often bear no visible relationship to the translation they implement, it is very difficult to discuss, or make any general statements about, the full class of compilers that implement a particular translation grammar. Furthermore, we are really interested only in compilers that are designed by specification to implement a given translation grammar, rather than those that just turn out to do so.

For example, consider a rule of a translation grammar $A \rightarrow BaD\{Bx_1x_2D\}$. It would be possible for a compiler that implemented this translation to have items $A \rightarrow .BaD(\tau_1)$ and $A \rightarrow .BaD(\tau_2)$ in two different states, and yet have different outputs associated with the B transition out of these states. This situation is illustrated in Figure 4.14; we have not written the lookaheads on the transitions, but rather have we indicated the outputs on the transitions.

It is situations like this that we wish to exclude for the reasons described above. Note that we

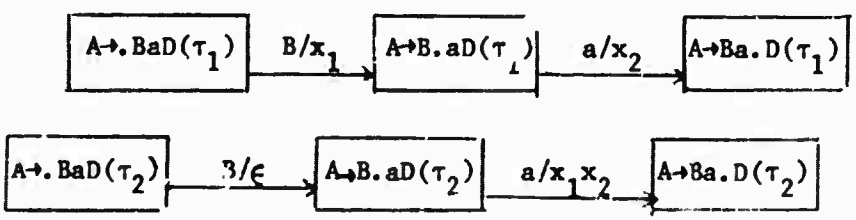


Figure 4.14

have not restricted the compiler model so that if $A \rightarrow BaD\{x_1Bx_2x_3D\}$ is a rule of the translation grammar being implemented, then the output associated with the B-transition out of any state with item $A \rightarrow .BaD(\tau)$ in it, must be x_2x_3. Such a restriction would effectively say that outputs in the com-

piler are only associated with nonterminals, which is not necessarily a good model of how translations are designed. In particular, no such LR(1) compiler could implement a translation grammar with rules $A \to BabD(Bx_1D)$ and $A \to BacD(Bx_2D)$, since items $A \to .BabD(\tau)$ and $A \to .BacD(\tau)$ would be in the same state; and then what should the output be on a B/a transition from that state - x_1 or x_2?

We feel that our restricted class of compilers is large enough to effectively capture the utility of the compiler model of implementing translations, yet small enough to be manageable. Furthermore, a little reflection confirms one's intuition that if some compiler based on the LR(k) parser for G implements (G, V_T', g), then there is some compiler designed to implement the translation, at the expense of a possible increase in the value of k used by the designed compiler to determine what outputs to emit.

Since this result is not critical to our work, we shall not pursue the laborious constructions needed to verify it; we only mention it in passing, as another justification for the reasonableness of our model.

Thus the class of translation grammars that are of effective use in building compilers are those that have compilers designed to implement them. We now wish to show that our transformation T_M preserves this class and so maintains the compilation power of the underlying language, even using a different grammar.

Theorem 4.26 Let G be an LR(k) grammar, (G, V_T', g) a translation grammar based on G, and let M be any MSP(k) machine for G. Then if there is some compiler P_0, based on the LR(k) parser for G, that is designed to implement

(G, V_T', g), then there also is a compiler P based on M designed to implement (G, V_T', g).

Proof We must define the output function δ_P. This is done as follows. Let q be any state of M; then by Corollary 3.49, q is a substate of q', some state of M_0, the LR(k) machine for G. Then if $A \to \alpha.\sigma\beta(\tau)$ is an item of q, and $\omega \in FIRST_k(\beta\tau)$, we define $\delta_P(q, \sigma, \omega)$ as equal to $\delta_{P_0}(q', \sigma, \omega)$.

The first thing we must show is that δ_P is well-defined. First of all, if q is a substate of q', and if $A \to \alpha.\sigma\beta(\tau)$ is an item of q, then it is an item of q' as well; and so $\delta_{P_0}(q', \sigma, \omega)$ is defined. Next, suppose q is a substate of both q' and q''. Then $A \to \alpha.\sigma\beta(\tau)$ will be an item of both q' and q'', and since the compiler based on M_0 is designed to implement (G, V_T', g), we have $\delta_{P_0}(q', \sigma, \omega) = \delta_{P_0}(q'', \sigma, \omega)$, if $\omega \in FIRST_k(\beta\tau)$. Thus $\delta_P(q, \sigma, \omega)$ is defined and single-valued.

It is clear that this new compiler satisfies the second condition for being designed to implement (G, V_T', g). Suppose $A \to \alpha.\sigma\beta(\tau_1)$ is an item of q_1 in M and $A \to \alpha.\sigma\beta(\tau_2)$ is an item of q_2 in M, where $\omega \in FIRST_k(\beta\tau_1) \cap FIRST_k(\beta\tau_2)$. Then we must show that $\delta_P(q_1, \sigma, \omega) = \delta_P(q_2, \sigma, \omega)$. But $q_1 \subset q_1'$ and $q_2 \subset q_2'$, which are states of M_0. Then $\delta_P(q_1, \sigma, \omega) = \delta_{P_0}(q_1', \sigma, \omega) = \delta_{P_0}(q_2', \sigma, \omega) = \delta_P(q_2, \sigma, \omega)$, the middle equality holds because the compiler P_0 is by hypothesis designed to implement (G, V_T', g).

Finally, we must show that the M-based compiler implements (G, V_T', g). We do this by showing that it is equivalent to the M_0-based compiler. We know that $L(M) = L(M_0)$, so we need only demonstrate that if $x \in L(M)$, then the M-based compiler produces the same output for it as does the M_0based one. We know by 3.66 and 3.69 that M and M_0 process any string in essentially the same

way; that every time M makes a σ/ω transition from a state q, M_0 also makes a σ/ω transition from a state q' that contains q. (A crucial point to be noted here is that if M makes an A/ω transition to the POP state, it will make another A/ω transition after disposing of the topmost stack level; however, M_0 makes only one corresponding A/ω transition, so it might be feared that M emits an extra output. But this is not the case, for if $f_M(q, A, \omega) =$ POP, then there is no item $B \rightarrow \alpha.A\beta(\tau)$ with $\omega \in FIRST_k(\beta\tau)$ in q, so $\delta_P(q, A, \omega)$ will not be defined.) Thus M emits the same outputs as M_0 in processing x, and in the same order; furthermore, M does not emit any additional output, since outputs are not emitted upon making a prediction or transferring to POP. Thus for any x, M and M_0 produce the same outputs, and so P is indeed a compiler designed to implement (G, V_T', g). Q.E.D.

Now that we have shown that no real translating power is lost by parsing with M instead of with M_0, we will show that nothing is lost by parsing with $T_M(G)$ rather than by using M for G.

Theorem 4.27 Let M be a cycle-free MSP(k) machine for G, and P a compiler based on M. Then there is a compiler based on the LL(k) parser for $T_M(G)$, which is equivalent to P.

Proof The new compiler is specified in the following way. Suppose q is some non-final state of M named (X, φ), and suppose $f_M(q, \sigma, \tau)$ is defined, with $\delta_P(q, \sigma, \tau) = y$. Then if $(X, \varphi\sigma)$ is the top symbol of the stack during an LL(k) parse using $T_M(G)$, the new compiler is to emit y as output before replacing $(X, \varphi\sigma)$ by the appropriate right-hand side.

Since there is only one non-final state in M named (X, φ), it is immediate

that this compiler is well-defined. Now we know that $L(T_M(G)) = L(M)$. Furthermore, we know by Lemma 4.11 that the LL(k) parse of a string based on $T_M(G)$ effectively simulates the processing of that string by M. In particular, if a σ/τ-transition is ever followed out of q to a state named $(X, \varphi\sigma)$, then at a corresponding time in the LL(k) parse, $(X, \varphi\sigma)$ will be on top of the stack with τ as the lookahead. The new compiler is designed to emit y at that time, so it has the same effect as the original one.

This sketch can readily be expanded into a full proof. Q.E.D.

We make a note about the timing of these two translations. In P, the output y is emitted as the transition is followed on σ from q; however, in the compiler using $T_M(G)$, the output is not emitted upon recognition of the rule $(X, \varphi) \rightarrow \sigma(X, \varphi\sigma)$ or $(X, \varphi\alpha) \rightarrow (X, \varphi\sigma)$, whichever one puts $(X, \varphi\sigma)$ onto the stack; but rather a little later, when $(X, \varphi\sigma)$ already resides on top of the stack, and prior to its replacement by some right-hand side. In a sense, it is like P emitting the output upon leaving state $(X, \varphi\sigma)$, rather than on entering it. The reason for this is that there is a difference in the way MSP(k) compilers and LL(k) compilers can use k symbols of lookahead to determine what output to emit. An MSP(k) compiler can inspect k symbols after σ, while the LL(k) compiler can just look an absolute k symbols ahead into the remaining input. In the case where σ is a terminal symbol, this is a real difference, and so the $T_M(G)$ compiler has to wait until after the σ has been read in order to see as much as the MSP(k) compiler could when it was reading σ, in order to decide what output to produce.

We can summarize all of the foregoing as follows.

Theorem 4.28 Let (G, V_T', ρ) be a translation grammar such that there is

some compiler based on the LR(k) parser for G which is designed to implement (G, V_T', g). Then if M is a cycle-free MSP(k) machine for G, there is a compiler based on the LL(k) parser for $T_M(G)$ which implements (G, V_T', g).

This theorem follows immediately from the preceding results. It says that any simple translation of G that can be implemented by a naturally defined LR(k) based compiler, can also be implemented by a compiler based on the LL(k) parser for $T_M(G)$. Or in other words, any useful translation of G can be done by some $T_M(G)$ compiler.

This is the most useful form in which to state this result. It would profit us little to find some other formal characterization of this set of translations on G, for our goal is not to study in abstract the kind of G-based translations that can be implemented on a $T_M(G)$ compiler, but just to demonstrate that the $T_M(G)$ compilers can handle anything that a reasonably constructed compiler for G could.

However, if another characterization is desired, the rudiments of one already exist. In a difficult section of [13], Lewis and Stearns introduce the notion of a "distinction index." Briefly, the distinction index of two instances of the same symbol in right-hand sides of two rules of a grammar, is the amount of lookahead necessary to distinguish between occurrences of these instances in a sentential form. In a roundabout way, Lewis and Stearns discuss a general class of simple translations on an LR(k) grammar which can be implemented by a pushdown translator using the LR(k) grammar. The class consists of those translation grammars that satisfy the following property: two instances of a symbol on right-hand sides of rules may have different outputs "associated" with them in the rules' respective translation elements, only if the distinction index of the two instances is less than or equal to k.

Of course we have paraphrased their results in an informal way, but an inspection of [13] will verify that our class of transformable translations is a generalization of their class of implementable translations.

We note that the translation performed by the compiler based on the $T_M(G)$ parser of Theorem 4.27, may not be expressible as a translation grammar on $T_M(G)$. But that is of little real significance, for we are really only interested in the compiling power of $T_M(G)$ parsers. The LL(k) grammar $T_M(G)$ and its associated parser and compiler need never see the light of day in the kind of automatic compiler writing system that this work might be applied to. To use such a system, as we have envisioned it, a programming language designer would construct a grammar for his language and provide it, together with translation elements specifying the semantics of each rule, to a grammatical processor. This processor would convert the designer's grammar into an equivalent LL(k) grammar by applying our transformation, and it would also design a compiler, based on the LL(k) parser for the transformed grammar, to implement the originally specified translation. This compiler would be used to actually compile programs in the language; its effective activities would be described by the original translation grammar; and the fact that there is no compact way of describing this compiler's precise actions by means of a translation grammar on the transformed grammar would not be so important.

However, for the sake of completeness, we will characterize a class of translations on G that can be described by simple translations on $T_M(G)$.

Theorem 4.29 Let P_0 be a compiler based on the LR(k) parser for G, designed to implement (G, V_T', g), and also satisfying the property: if $f_{M_0}(q, \sigma, \tau_1) =$

$f_{M_0}(q, \sigma, \tau_2)$, then $\delta_{P_0}(q, \sigma, \tau_1) = \delta_{P_0}(q, \sigma, \tau_2)$. Then if M is a cycle-free MSP(k) machine for G, there is a simple translation grammar based on $T_M(G)$ which is equivalent to (G, V_T', g).

Proof First we define the compiler P based on M that is equivalent to P_0, the one based on M_0; this can be done by Theorem 4.26. Then we define the translation elements for $T_M(G)$ as follows. For the rule $(X, \varphi\sigma) \to a(X, \varphi\sigma a)$ or $(X, \varphi\sigma) \to (Y, \epsilon)\ (X, \varphi\sigma Y)$, consider the unique non-final state named $(X, \varphi\sigma)$; by hypothesis, all entries to that state from q, the non-final (X, φ) state, have an identical output y associated with each one. Then the translation elements for these rules are to be $(X, {}_{\varphi}\sigma) \to a(X, \varphi\sigma a)\ \{y(X, \varphi\sigma a)\}$ and $(X, {}_{\varphi}\sigma) \to (Y, \epsilon)(X, \varphi\sigma Y)\{y(Y, \epsilon)(X, \varphi\sigma Y)\}$. For the rule $(X, \varphi\sigma) \to (X, \varphi')$, there is exactly one final state whose rule expresses the relationship between $\varphi\sigma$ and φ'; let z be the output associated with entry to that state from q. Then we have $(X, {}_{\varphi}\sigma) \to (X, \varphi')\ \{z(X, \varphi')\}$. If a rule is not assigned a translation element by these stipulations, its translation element is to be just the nonterminals of its right-hand side. It is immediate that this translation grammar is well-defined.

Let us design a $T_M(G)$-based compiler P' to implement this translation grammar as follows. If $(X, \varphi) \to \Psi\ \{y\ \Psi'\}$ is a rule of the translation grammar, then whenever (X, φ) is on top of the stack, with τ as the lookahead, where $\tau \in FIRST_k(\Psi)$, then the compiler is to emit y as it replaces (X, φ) by Ψ on the stack. This compiler is well-defined; since $T_M(G)$ is strong LL(k), for a given (X, φ), any τ will be in $FIRST_k$ of at most one (X, φ)-rule. And it is clear that it does implement this translation grammar.

But as we have seen in Lemma 4.10, τ is in $FIRST_k$ of a rule $(X, \varphi\sigma) \to \Psi$ only if τ is the lookahead on some transition into an (X, φ_σ) state, which is

non-final if Ψ is $a(X, \omega\sigma\alpha)$ or (Y, ϵ) $(X, \omega\sigma Y)$, and final otherwise. By the way the translation grammar has been defined, the y output associated with all σ/τ entries into the same state is the output specified for the appropriate $(X, \omega\sigma) \rightarrow \Psi$ rule(s). Thus this compiler P' not only implements the translation grammar, but is also precisely the compiler described in the proof of Theorem 4.27, which is equivalent to the M-based compiler P. Thus by a chain of equalities, the $T_M(G)$ based translation grammar is equivalent to (G, V_T', g). Q.E.D.

Corollary 4.30 If M is a cycle-free MSP(k) machine for G, then every simple Polish translation on G can be expressed as a simple translation on $T_M(G)$.

Proof We define an LR(k)-based compiler for the simple Polish translation as follows. If $A \rightarrow \Psi$ $\{\Psi' y\}$ is a rule of the translation grammar, then the compiler is to emit an y as output upon every entry to the final state for $A \rightarrow \Psi$; no other outputs are to be emitted.

This compiler satisfies the hypotheses of the previous theorem, so there is a simple translation on $T_M(G)$ which is equivalent to the original simple Polish translation. The proof of the preceding result tells us that this translation will be the following:

$$(X, \varphi) \rightarrow a(X, \varphi a) \; \{(X, \varphi a)\}$$

$$(X, \varphi) \rightarrow (Y, \epsilon)(X, \varphi Y) \; [(Y, \epsilon)(X, \varphi Y)]$$

$$(X, \varphi\Psi) \rightarrow (X, \varphi A) \; \{y \; (X, \varphi A)\}$$

$$(X, X) \rightarrow \epsilon \; \{\epsilon\}$$

where y is the output for $A \rightarrow \Psi$ in G. Q.E.D.

We do not wish too much significance to be assigned to Theorem 4.29. It is an interesting result, but not terribly important. First of all, there are larger classes of simple translations on G which are expressible as simple translations on $T_M(G)$, but they are not so easy or natural to describe. Furthermore, the proof of Theorem 4.29 gives but one way for these translations on G to be expressed on $T_M(G)$; there are other ways in which some of them could be expressed. In particular, some simple translations on G can be expressed as simple Polish translations on $T_M(G)$, if different techniques are used for constructing the compilers P and P' of the proof of the theorem: and some simple translations on G cannot be expressed as translations on $T_M(G)$. But we shall not dwell on this topic. It is tempting to get involved in determining the precise relationship between translations expressible on G and those expressible on $T_M(G)$. But such investigations would not be germane to the course of our development. We have already shown the major result of practical significance, namely that any useful translation expressible on G can be implemented by a compiler using an LL parser for $T_M(G)$.

There is one further point that is both interesting and important: that there are some simple translation grammars based on $T_M(G)$ that cannot be expressed by any simple translation on G, nor even implemented by any compiler based on the LR(k) parser for G. The intuitive reason for this is that when $T_M(G)$ uses the rule $(X, \varphi\Psi) \rightarrow (X, \varphi A)$, which corresponds to G using $A \rightarrow \Psi$, $T_M(G)$ has some extra information that G does not: namely, that this derivation started with an X. Two similar rules, say $(X, \varphi\Psi) \rightarrow (X, \varphi A)$ and $(Y, \varphi'\Psi) \rightarrow (Y, \varphi' A)$ might have different, unrelated translation elements in $T_M(G)$; yet both rules correspond to the LR(k) machine reducing Ψ to A. This fact is not really as exciting as it first appears; the catch is, of course, that to specify such a translation, the

language designer would have to get his hands on either the cycle-free machine M from which $T_M(G)$ is derived, or on $T_M(G)$ itself. But neither of these occurrences is especially likely or desirable; and even further, the language designer would then have to express his 'context-dependent" outputs (for example, that a conditional is to be translated one way if it is part of an expression, another if it is a statement, which is just the kind of flexibility $T_M(G)$ might allow) in terms of the grammar $T_M(G)$, which may be unrecognizable to him, and which may bear little visible relationship to his original grammar G; and so he might find it difficult to use this new grammar to express his notions of the semantics of the language.

This concludes our discussion of the complex and somewhat murky area of translations. However, we have confirmed the utility of our transformation by showing that $T_M(G)$ is at least as useful as G is, in directing the compilation of programs in L(G).

4.8 Improving $T_M(G)$

We have seen how to derive, given a cycle-free MSP(k) machine M for G, an LL(k) grammar $T_M(G)$ which generates L(G). We have seen that $T_M(G)$ compares favorably with G with respect to compiling ability, and that the number of steps in an LL(k) parse using $T_M(G)$ is not, in general, much greater than the number of steps in an LR(k) parse using G. But we would like to know if there is any way to improve $T_M(G)$ or enhance its parser, that will increase the parsing speed or have other salutary effects, without destroying the translating power or the LL-ness of $T_M(G)$.

There is one very elementary enhancement that can be made to the LL(k) parser for $T_M(G)$, that will significantly increase its parsing speed. This

is a well-known trick, described in many places, including [ ], page 662. In the basic LL(k) parser for $T_M(G)$, application of the rule $(X, \varphi) \rightarrow a(X, \varphi a)$ takes place as follows: when (X, φ) is the top stack symbol, and the lookahead specifies that this rule is to be applied, (X, φ) is removed from the stack and is replaced by $a(X, \varphi a)$; in the next step, the symbol a will be on top of the stack, and perforce a will also be the first lookahead symbol; so a is removed from the input stream and also popped off the stack, exposing $(X, \varphi a)$. There is no reason why this cannot be effected in one step; with (X, φ) at the top of the stack, and a as the first symbol of the lookahead which is dictating application of $(X, \varphi) \rightarrow a(X, \varphi a)$, just replace (X, φ) by $(X, \varphi a)$ on the stack, and remove a from the input stream.

This speeded-up LL(k) parser can support any compilation activities that the original parser could, but it is somewhat faster. In particular, there will be one less step in the fast LL(k) parse for each symbol in the input string being parsed. So now the fast LL(k) parse for $T_M(G)$ begins to compare favorably with the LR(k) parse for G; the number of steps in the former will be equal to the number of steps in the latter, plus twice the number of predictions made, minus the length of the input string. In most normal circumstances the last term will be greater than the second (since predictions will rarely be made just to read two input symbols), so the fast LL(k) parse will in practice usually be faster than the LR(k) parse.

With regard to the issue of the parser for $T_M(G)$, we would also like to know if it really needs to inspect k symbols of lookahead. That is, we are assured that $T_M(G)$ is LL(k); but is there a smaller k' such that $T_M(G)$ is LL(k')? This question is not only of theoretical significance, but also impacts the size and speed of the parser needed to analyze strings generated by

$T_M(G)$. In general, however, this question has a negative answer. We have seen from our earlier study that $T_M(G)$ uses the lookahead to choose among various rules precisely in the way that M uses the lookahead to choose among various transitions. So if M at any point really needs to know all k symbols of the lookahead, and can not always get by with effectively looking at some k'-length prefix of this lookahead, then $T_M(G)$ will also need to inspect full k symbols of lookahead at some point, and so will not be LL(k') for any $k' < k$. These terms can be made more formal and easily lead to this result.

Theorem 4.31 If M is a cycle-free MSP(,) machine that uses k symbols of input, then $T_M(G)$ is not LL(k'), for any $k' < k$.

This is not to say that a parser for $T_M(G)$ will always need to inspect all k symbols of lookahead in order to determine its action, only that in some cases it will need to. In many cases, the parser will no doubt be able to get by with less, and a cleverly designed parser will take advantage of these special cases, both to save parsing time and to reduce the parser table size. But $T_M(G)$ will not really be LL(k').

There is one area in which $T_M(G)$ seems to be grossly deficient, and which adversely affects both the size and speed of the LL(k) parser for $T_M(G)$; and that is in the size of $T_M(G)$. The concept of the size of a grammar is not precisely defined, but it is clearly related to the number of nonterminals and the number of rules; and in both of these areas, $T_M(G)$ is greatly inflated over G. For example, the grammar $T_M(C)$ of Figure 4.5, which is repeated in Figure 4.15, has 24 rules and 18 nonterminals; the grammar G from which it was derived had 7 rules and 3 nonterminals. This is not just a problem of aesthetics. First of all, the size of a grammar directly affects the size of the

parsing table needed by the LL(k) parser for the grammar; more about this soon. And furthermore, all these extra nonterminals mean extra reductions to be made and more steps in the parse.

(S, ϵ) → a(S, a)	(B, ϵ) → a(B, a)	(B, aB) → (B, A)
(S, a) → (B, ϵ) (S, aB)	(B, ϵ) → b(B, b)	(B, A) → x(B, Ax)
(S, a) → d(S, ad)	(B, a) → c(B, ac)	(B, Ax) → (B, B)
(S, a) → c(S, ac)	(B, a) → d(B, ad)	(B, Ax) → y(B, Axy)
(S, aB) → (S, A)	(B, a) → (B, ϵ) (B, aB)	(B, Axy) → (B, B)
(S, ad) → (S, A)	(B, b) → (B, B)	(B, B) → ϵ
(S, ac) → (S, A)	(B, ac) → (B, A)	
(S, A) → x(S, Ax)	(B, ad) → (B, A)	
(S, Ax) → (S, S)		
(S, S) → ϵ		

Figure 4.15

It is impossible to precisely characterize the size of $T_M(G)$ (whether in terms of rules or nonterminals) purely in terms of the size of G, since the nature of $T_M(G)$ depends mainly on the structure of the cycle-free MSP(k) machine M from which it is derived. It is possible to contrive some very gross upper and lower bounds for the size of $T_M(G)$, but they are practically useless. The problem is that while any useful cycle-free machine will have a reasonable, describable structure, some cycle-free machines will not, and they are the ones that determine the bounds. We can get some crude feeling for the size of $T_M(G)$ by inspecting the machine M, without computing the whole grammar. For example, we know that there will be at least one non-

terminal for each state of M. Also, for each predictive state in M, some number of states will receive an additional name. At the very worst, the number of nonterminals in $T_M(G)$ will be proportional to the square of the number of states of M (and even this will have a constant factor less than one). The number of rules can be similarly grossly bounded by the number of nonterminals in $T_M(G)$ times the size of the vocabulary of G (terminal and nonterminal). However these bounds are not attainable, and are not even approached by any reasonable grammar and machine. But in terms of simple structural properties of M, it is difficult to obtain precise closed expressions for the size of $T_M(G)$, for it is possible for M to have a very baroque structure, causing a peculiarly shaped and sized derived grammar. We shall not try to characterize the precise degree of $T_M(G)$'s deficiency; we shall assume that in general it will be unsatisfactorily large, and shall concentrate on the problem of making it smaller.

It is not immediately obvious what the appropriate parameters are, and what their respective weights, when it comes to reducing the size of a grammar. It is possible to make arguments for each of the following characteristics of a grammar, as being of great significance in measuring its size; the number of rules; the number of nonterminals; the length of the longest rule; the sum of the lengths of the rules; the value of the lookahead which is needed to parse the grammar. Which of these should really be considered foremost depends in great measure on the reason for trying to reduce the grammar's size. In our case, we shall more or less arbitrarily select one particularly goal for reducing $T_M(G)$. We are not going to try to satisfy our aesthetic sensibilities, or minimize some complexity-oriented measure

of a grammar; nor even attempt to make the grammar more "manageable" in some vague way. We are interested in reducing the size of the parsing table that will be constructed from the grammar and that will be used in a compiler for the language. This reduction may come at the expense of increasing the maximum size of the stack that can occur during a parse; this latter figure is related to the lengths of the rules.

Thus we must consider what kind of parser we will be constructing, and what factors influence its size. The whole thrust of our work has been to create LL grammars, so we shall consider the kind of parsing table for LL grammars described by several authors [1, 13,20]. Basically, for a given strong LL(k) grammar G, the table looks as follows: the rows are labelled with the nonterminals of G, the columns with all strings in V_T^k. This table will be used by the parser in the expected way: if the topmost element of the stack is A, and the lookahead is ω, then the (A, ω)-entry of the table is looked up, and replaces A on the stack. (The other function of the parser is: if the terminal symbol a is on top of the stack and also the first symbol of the input, then it is removed from both places.) What is the size of such a table? The number of entries is $|V_N| \cdot |V_T|^k$; an (A,ω) entry itself will be the right-hand side of some A-rule. We can keep these right-hand sides in some list and make the (A, ω) entry of the table point to the appropriate list entry. Thus the amount of storage needed is something like $|V_N| \cdot |V_T|^k +$ the sum of the lengths of the rules of G.

So much for theory. In practice, however, the issues are not so sharply drawn. First of all, the second term is frequently ignored, largely because it is not clear how to get a handle on that parameter of a grammar or how to

go about making it significantly smaller. It is generally felt that there is some kind of tradeoff between the number of rules of a grammar and their individual lengths, so that the sum of the lengths will not vary greatly over grammars that are obtained from each other by minor modifications. It would seem then that the value of k, which effects the size of $|V_T|^k$, is the dominant factor in determining parser size. But this is not strictly true. In practice, it may well be worthwhile to increase slightly the value of k if that enables us to eliminate many of the nonterminals of the grammar. The reason for this is that parsing tables are not organized quite so naively as described above; frequently those are far fewer than $|V_T|^k$ entries for a given row. For example if, as often happens, all strings with τ as a prefix, where $|\tau|<k$, have the same entry for a given row, there won't be entries for each of these strings for that row, but just one, labelled by τ. Furthermore, this parsing table is usually very sparse; i.e., many of its entries are "ERROR," indicating that with that nonterminal on top of the stack, the given lookahead can not occur during a legal parse. At the expense of delaying the time at which such an error is located, it is possible to eliminate many of these entries. These remarks are just intended to convey a feeling for some of the ways of escaping from the tyrrany of the exponential in the table size. For similar discussions for LR(k) parsing, see [4], page 80, and [1], Chapter 6.

Before we go much further in deciding which parameters of $T_M(G)$ we are going to try to minimize, we should determine what kind of tools we are going to use to attain our goals. Since we want this reduction procedure to be readily applicable, we do not wish to get involved in very recondite further transformations; and since we want to maintain the LL-ness of the grammar, and preserve the class of translations which it can direct, we must not use

any transformations which do grave violence to the grammar's structure. We have decided that the simplest and most appropriate reduction technique is that of eliminating nonterminals by substitution, as described below.

The choice of this technique reinforces our choice of which grammatical parameter to focus on; for while it is natural to decrease the number of symbols and rules in a grammar by repeated substitutions, it is not possible to reduce the value of the lookahead the grammar needs, merely by substituting out some nonterminals. So the only significant factor over which we can really exercise some control is the number of nonterminals in the grammar. We shall take it as our mission to minimize by substitution, the number of nonterminals in $T_M(G)$, while keeping an eye on the effect this process has on the value of k.

We also observe that if G_2 is obtained from G_1 by substituting out some nonterminals, then the G_2-tree for a string ω will be a compact version of the G_1-tree for ω; so a G_2-parser will go through fewer configurations in recognizing ω than a similar G_1-parser, and so will be faster.

Definition 4.32 Let G be a grammar, A a nonterminal of G. Let the A-rules of G be $A \to x_1$, $A \to x_2$, ..., $A \to x_n$. Then we eliminate A from G by substitution by removing A from V_N, removing all the A-rules from the list of rules, and replacing ever rule of the form $B \to \alpha A \beta$ by the set of rules $B \to \alpha x_1 \beta$, $B \to \alpha x_2 \beta$, ..., $B \to \alpha x_n \beta$.

It is not always possible to eliminate a nonterminal from a grammar in this way.

Lemma 4.33 If A does not occur on the right-hand side of any A-rule, then A can be eliminated by substitution; the result is a grammar with one less nonterminal.

Proof Immediate from the definition. Q.E.D.

We are interested in eliminating more than one nonterminal from the grammar $T_M(G)$. Rather than try to define the concept of eliminating several nonterminals at the same time, we shall say that this is done by eliminating one nonterminal from $T_M(G)$, eliminating another from the resultant grammar, and so on. We would like to know how many (and which) of the nonterminals of $T_M(G)$ can be ultimately disposed of by repeated applications of the substitution process. Sometimes even though each member of a set of nonterminals could be eliminated as an individual, it is not possible to eliminate every one in the set.

Definition 4.34 Let $A_1, \ldots, A_n$ be a sequence of distinct nonterminals such that A_{i+1} appears on the right-hand side of an A_i-rule, A_1 appears on the right-hand side of an A_n-rule, and A_i does not appear on the right-hand side of any other A_j rules. Then $A_1, \ldots, A_n$ is a recursive sequence of nonterminals.

It is possible for any single nonterminal to appear in several recursive sequences. We are not interested in the ordering of the nonterminals in a recursive sequence; we shall call a set of nonterminals a recursive sequence if there is some ordering of the nonterminals which makes them into a recursive sequence.

Lemma 4.35 If $A_1, \ldots, A_n$ is a recursive sequence of nonterminals of G, then not all of $A_1, \ldots, A_n$ can be eliminated from G.

Proof After eliminating all nonterminals except A_j, there will be some A_j rule with A_j on its right-hand side; then A_j cannot be eliminated. Q.E.D.

The following results are trivial.

Lemma 4.36 Suppose $A_1, \ldots, A_n$ is not a recursive sequence. Then all of $A_1, \ldots, A_n$ can be eliminated from G, and the elimination can be done in any order.

Lemma 4.37 If G' is obtained from G by eliminating a nonterminal of G by substitution, then $L(G) = L(G')$.

In other words, for every different recursive sequence of nonterminals, at least one of the nonterminals cannot be removed from the grammar. However, any nonterminal not in any such sequence can always be substituted out. This leads us to formulate the following tentative plan for removing as many nonterminals as possible from an arbitrary grammar G: 1) find all distinct recursive sequences of G; 2) find a minimum set of nonterminals such that there is at least one member of the set in each recursive sequence; 3) eliminate all nonterminals other than those in this minimum set, in any order.

It is not hard to show that this procedure does work, and eliminates a maximal number of nonterminals from the grammar. However, this process is not immediately applicable to our situation, for we are trying to minimize the nonterminals of $T_M(G)$ subject to the constraint of keeping the grammar LL. We do not insist on not increasing the value of k, but we do need the grammar to remain LL(k') for some k'. Let us see what restrictions this places on which nonterminals can be eliminated from $T_M(G)$.

There are four kinds of rules in $T_M(G)$: $(X, \varphi) \rightarrow a(X, \varphi a)$, $(X, \varphi) \rightarrow (Y, \epsilon)$ $(X, \varphi Y)$, $(X, \varphi\alpha) \rightarrow (X, \varphi A)$, and $(X, X) \rightarrow \epsilon$. Looking at it from the LL(k), look-ahead point of view, the elimination of different nonterminals might have

different effects. For example, if we replaced the instance of $(X, \varphi a)$ in $(X, \varphi) \rightarrow a(X, \varphi a)$ by all the right-hand sides which $(X, \varphi a)$ goes to, we would at worst increase the lookahead for the grammar by one. That is, since $T_M(G)$ is strong LL(k), if $(X, \varphi a) \rightarrow R_1$ and $(X, \varphi a) \rightarrow R_2$, then R_1 and R_2 can be distinguished by k-lookahead; hence, aR_1 and aR_2 can be distinguished by k + 1 lookahead. Similarly, replacing $(X, \varphi A)$ in $(X, \varphi\alpha) \rightarrow (X, \varphi A)$, by all its right-hand sides, does not affect the lookahead value of the grammar at all. However, eliminating $(X, \varphi Y)$ from $(X, \varphi) \rightarrow (Y, \epsilon)\ (X, \varphi Y)$ could completely destroy the LL-ness of the grammar, if (Y, ϵ) generates an infinite set of strings. In that case, if the rules for $(X, \varphi Y)$ were $(X, \varphi Y) \rightarrow R_1$ and $(X, \varphi Y) \rightarrow R_2$, we would get $(X, \varphi) \rightarrow (Y, \epsilon)R_1$ and $(X, \varphi) \rightarrow (Y, \epsilon)R_2$; since (Y, ϵ) can generate arbitrarily long strings we would not, in general, be able to "see" past (Y, ϵ), to examine the rest of the lookahead and distinguish $(Y, \epsilon)R_1$ from $(Y, \epsilon)R_2$.

This completes the enumeration of all ways that nonterminals can be used on the right-hand side of a rule; and we have seen that the only usage that can give us trouble if eliminated is $(X, \varphi Y)$ in $(X, \varphi) \rightarrow (Y, \epsilon)(X, \varphi Y)$, if (Y, ϵ) generates an infinite set of strings. All other instances of nonterminals can be substituted out, without fear of doing violence to the LL-ness of the grammar; at worst, eliminating $(X, \varphi a)$ from $(X, \varphi) \rightarrow a(X, \varphi a)$ can increase the value of k by one. But since we do not eliminate individual "instances" of nonterminals, but expunge the nonterminal completely, we get the following tentative summation. The elimination of a nonterminal from $T_M(G)$ destroys the LL-ness of the grammar if and only if: the nonterminal is $(X, \varphi Y)$; it appears in $(X, \varphi) \rightarrow (Y, \epsilon)\ (X, \varphi Y)$; and (Y, ϵ) generates an

infinite set of strings.

There is, however, one codicil which must be added to this statement. There is one case where (X, φY) can be removed from the grammar even though it satisfies all of these conditions; and that is when there is exactly one rule with (X, φY) as its left-hand side. In such a case, no confusion can result from eliminating (X, φY); and then instead of having (X, φ) → (Y, ϵ) (X, φY) and (X, φY) → R, we get (X, φ) → (Y, ϵ)R. Now the same comments that applied to (X, φY) will apply to the first nonterminal of R; if there is only one rule with it on the left-hand side, it can be eliminated from the grammar, but otherwise it cannot be. If there is a sequence of such nonterminals, then all of the nonterminals in the sequence can be eliminated.

To make this precise, we introduce the following definition.

Definition 4.38 A string of symbols $A_1, A_2, \ldots, A_n$ is a chain in G if for each i, $1 \leq i < n$, A_i is a nonterminal that appears on the left-hand side of exactly one rule, where A_{i+1} is the first nonterminal on the right-hand side of the A_i-rule, and A_n is ϵ, if $A_{n-1} \rightarrow \epsilon$ is a rule, or otherwise a nonterminal that appears on more than one left-hand side. We say that A_1 is the head of the chain.

If (X, φ) → (Y, ϵ)(X, φY) is a rule, and $A_1, \ldots, A_n$ is a chain headed by (X, φY), then we can eliminate all nonterminals in the chain if $A_n = \epsilon$; otherwise, we can eliminate all but one.

We can combine this new condition with our previously described method for eliminating the maximum number of nonterminals from a grammar. We now have what amounts to an additional contraint, that no chain headed by such an (X, φY) can be wholly eliminated. We can identify each of those (X, φY) nonterminals,

and compute the chain which it heads; we then must find a minimal set of nonterminals which contains a representative from each such chain as well as from each recursive sequence. This minimization can be addressed by constructing a Boolean expression in conjunctive normal form, one term for each recursive sequence and for each appropriate chain; a term will be the disjunction of the symbols in the set. We then must find the minimal cover of this expression, the smallest (fewest literals) expression $A \wedge B \wedge C \ldots$, which logically implies the original expression. There are standard methods for computing such a cover. We note that there may be several different minimal covers of the same size; presently we shall give some criteria for choosing among them.

We now present a technique for making this grammatical minimization process a little more structured and manageable.

Definition 4.39 For a given grammar $T_M(G)$, the nonterminal graph of $T_M(G)$ is a directed graph constructed as follows: the nodes of the graph are the nonterminals of $T_M(G)$ and the symbol ϵ; and there is an edge from nonterminal A_1 to symbol A_2 if and only if A_2 appears on the right-hand side of a rule for A_1.

For example, the graph for $T_M(G)$ of Figure 4.15 is given in Figure 4.16.

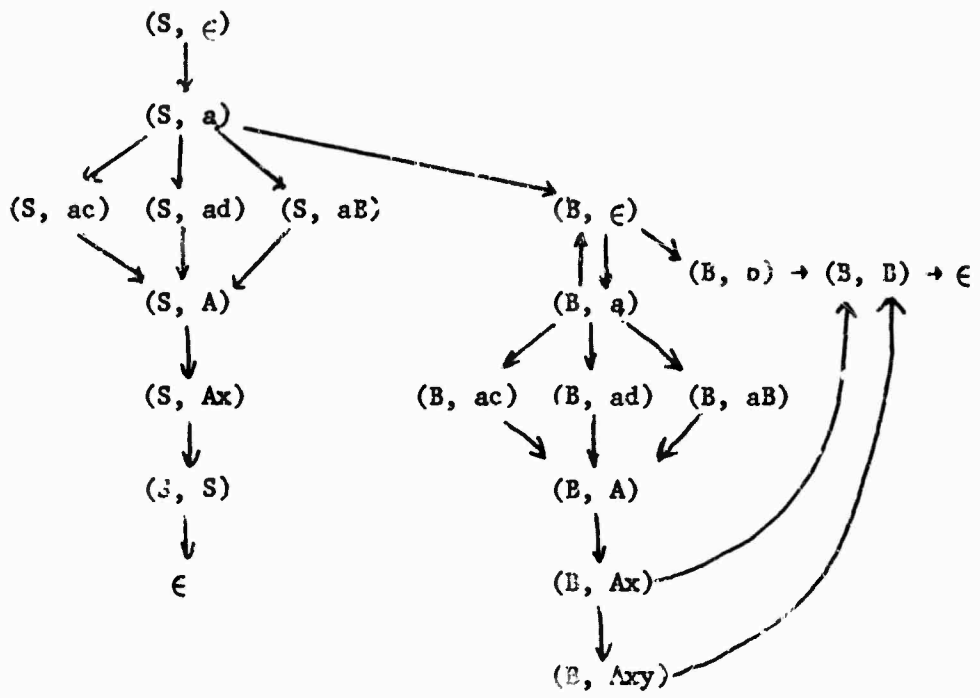


Figure 4.16

We can use this nonterminal graph for a variety of purposes. First of all, to determine those (Y, ϵ) nonterminals which generate infinite strings. It is obvious that if A generates an infinite set of strings, there must be some nonterminal B such that $A \overset{*}{\Rightarrow} xBy$ and $B \overset{*}{\Rightarrow} uBv$. But such a recursive nonterminal B would occur in a cycle in the nonterminal graph, and can thus be detected. Thus (Y, ϵ) generates an infinite set if and only if there is some nonterminal (X, φ) accessible from (Y, ϵ), such that (X, φ) is contained in some cycle in the graph.

The cycles in this graph also immediately give us all recursive sequences in $T_M(G)$, since a set of nonterminals form a recursive sequence if and only if they form a cycle in the graph. So our plan should be to first locate all cycles in this graph. This immediately gives us all the recursive sequences of $T_M(G)$, and enables us to identify those nonterminals of the form

(Y, ϵ) which generate infinite sets. Once we have determined these (Y, ϵ), we are in a position to pick some chains which cannot be eliminated from the grammar: namely, those headed by nonterminals of the form (X, φY).

In this example, there is only one cycle in the graph, consisting of the nonterminals (B, ϵ) and (B, a). So at least one of these must not be eliminated from the grammar. Furthermore, this means that (B, ϵ) generates an infinite set of strings. Hence we must consider the nonterminals (S, aB) and (B, aB). The first of these heads the chain (S, aB), (S, A), (S, Ax), (S, S), ϵ; since ϵ is in the chain, none of the nonterminals in it need remain in the grammar. However, either (B, aB) or (B, A) or (B, Ax) must remain in the grammar, since (B, aB), (B, A), (B, Ax) forms a chain. Hence the logical expression describing which nonterminals must be retained in the grammar, is ((S, ϵ)) $\wedge$ ((B, ϵ) $\vee$ (B, a)) $\wedge$ ((B, aB) $\vee$ (B, A) $\vee$ (B, Ax)). There are a variety of minimal covers for this expression, but each of them contains three nonterminals. So 15 of the 18 nonterminals of this grammar can be eliminated, and still leave the result LL. Two possible covers for this expression are (S, ϵ) $\wedge$ (B, ϵ) $\wedge$ (B, aB) and (S, ϵ) $\wedge$ (B, a) $\wedge$ (B, A). The two grammars corresponding to these choices are given in Figure 4.17.

$(S, \epsilon) \rightarrow a(B, \epsilon)x$	$(S, \epsilon) \rightarrow acx$
$(S, \epsilon) \rightarrow acx$	$(S, \epsilon) \rightarrow adx$
$(S, \epsilon) \rightarrow adx$	$(S, \epsilon) \rightarrow abx$
$(B, \epsilon) \rightarrow acx$	$(S, \epsilon) \rightarrow aa(B, a)x$
$(B, \epsilon) \rightarrow acxy$	$(B, a) \rightarrow c(B, A)$
$(B, \epsilon) \rightarrow$ [illegible]	$(B, a) \rightarrow d(B, A)$
$(B, \epsilon) \rightarrow adxy$	$(B, a) \rightarrow b(B, A)$
$(B, \epsilon) \rightarrow a(B, \epsilon)(B, aB)$	$(B, a) \rightarrow a(B, a)(B, A)$
$(B, \epsilon) \rightarrow b$	$(B, A) \rightarrow x$
$(B, aB) \rightarrow x$	$(B, A) \rightarrow xy$
$(B, aB) \rightarrow xy$	

Figure 4.17

Let us create a little terminology to make it easier to talk about these new grammars.

Definition 4.40 Let $T_M(G)$ be a grammar derived from a cycle-free MSP(k) machine M ϵ G. Then a minimizing term of $T_M(G)$ is defined as follows: 1) if $A_1, A_2, \ldots, A_n$ is a recursive sequence of nonterminals of $T_M(G)$, then $(A_1 \vee A_2 \vee \ldots \vee A_n)$ is a minimizing term; 2) if (Y, ϵ) generates an infinite set of strings, and if $(X, \varphi) \rightarrow (Y, \epsilon)\ (X, \varphi Y)$ is a rule of $T_M(G)$, and if $(X, \varphi Y), A_1, A_2, \ldots, A_n$ is a chain, then $((X, \varphi Y) \vee A_1 \vee A_2 \vee \ldots \vee A_n)$ is a minimizing term. If $X_1, X_2, \ldots, X_m$ are all the minimizing terms for $T_M(G)$ then $X_1 \wedge X_2 \wedge \ldots \wedge X_m$ is the minimizing expression for $T_M(G)$. If $\{A_1, A_2, \ldots, A_n\}$ is a set of nonterminals of $T_M(G)$, then it is a minimal cover of the minimizing expression if $A_1 \wedge A_2 \wedge \ldots A_n \Rightarrow X_1 \wedge X_2 \wedge \ldots X_m$, and if there is

no smaller set $\{B_1, \ldots, B_p\}$ such that $B_1 \wedge B_2 \wedge \ldots \wedge B_p \Rightarrow X_1 \wedge X_2 \wedge \ldots \wedge X_m$. (Here, $\Rightarrow$ is used in the logical sense of implication.)

Definition 4.41 Let $T_M'(G)$ be a grammar obtained by eliminating some nonterminals of $T_M(G)$ by substitution. If the nonterminals of $T_M'(G)$ form a minimal cover of the minimizing expression of $T_M(G)$, then $T_M'(G)$ is called a minimal version of $T_M(G)$.

These definitions merely formalize our preceding discussion.

Lemma 4.42 Let G be a strong LL(k) grammar, G' the grammar obtained by eliminating the nonterminal A from G by substitution. Then G' is strong LL(k) for some $k' \geq k$, unless both of the following conditions hold:

i) There is a rule of G of the form $B \rightarrow \alpha A \beta$, where α generates an infinite set of strings.

ii) There is more than one A-rule.

Proof Suppose that G' is not strong LL(k') for any k'. Then for some B and terminal string ω, $B \overset{*}{\underset{L}{\Rightarrow}} \omega$ beginning with rule p_1 and $B \overset{*}{\underset{L}{\Rightarrow}} \omega$ beginning with rule p_2, in G'. But if p_1 is $B \rightarrow \Psi_1$ and p_2 is $B \rightarrow \Psi_2$, and if p_1 and p_2 were both rules of G, then $\text{FIRST}_k(\Psi_1\ \text{FOLLOW}_k(B)) \cap \text{FIRST}_k(\Psi_2\ \text{FOLLOW}_k(B)) = \phi$, since G was strong LL(k). Therefore at least one of the rules must be new to G'. That is, p_1 is $B \rightarrow \alpha\varphi_1\beta$, where $L \rightarrow \alpha A \beta$ and $A \rightarrow \varphi_1$ were rules of G. But then $\text{FIRST}_k(\alpha\varphi_1\beta) \subset \text{FIRST}_k(\alpha A \beta)$; yet $\text{FIRST}_k(\alpha A \beta\ \text{FOLLOW}_k(B)) \cap \text{FIRST}_k(\varphi_2\ \text{FOLLOW}_k(B)) = \phi$ if p_2 is a rule of G. Therefore p_2 is a new rule of G' as well, of the form $B \rightarrow \alpha\varphi_2\eta$, where $B \rightarrow \gamma A \eta$ and $A \rightarrow \varphi_2$ were rules of G. But $\text{FIRST}_k(\alpha\varphi_1\beta) \subset \text{FIRST}_k(\alpha A \beta)$ and $\text{FIRST}_k(\gamma\varphi_2\eta) \subset \text{FIRST}_k(\gamma A \eta)$, while $\text{FIRST}_k(\alpha A \beta\ \text{FOLLOW}_k(B)) \cap \text{FIRST}_k(\gamma A \eta\ \text{FOLLOW}_k(B)) = \phi$ if $B \rightarrow \alpha A \beta$ and $B \rightarrow \gamma A \eta$ are different rules

of G, since G was strong LL(k). Hence $B \rightarrow \alpha A\beta$ and $B \rightarrow \gamma A\eta$ were the same rule, say $B \rightarrow \alpha A\beta$. Hence if p_1 and p_2 are different rules, then $\varphi_1 \neq \varphi_2$, so there are two different A-rules. Now suppose that α did not generate an infinite set; let the length of the longest terminal string it generates be k_1. We know that $FIRST_k(\varphi_1) \cap FIRST_k(\varphi_2) = \phi$, since they are both right-hand sides of A-rules; hence if we set $k' = k + k_1$, we have $FIRST_{k'}(\alpha\varphi_1) \cap FIRST_{k'}(\alpha\varphi_2) = \phi$. Therefore G' will be strong LL(k'), which contradicts the hypothesis. Hence α generates an infinite set, and both required conditions hold.

If both conditions do hold, then $B \rightarrow \alpha\varphi_1 y$ and $B \rightarrow \alpha\varphi_2 y$ are both rules of G', where α generates indefinitely long terminal strings. Hence G' is not strong LL(k') for any k'. Q.E.D.

Theorem 4.43 If $T_M'(G)$ is a minimal version of $T_M(G)$, then $L(T_M'(G)) = L(T_M(G))$, and $T_M'(G)$ is LL(k') for some k'.

Proof By the definition of minimal version and the preceding lemmas. Q.E.D.

Theorem 4.44 Let $T_M'(G)$ be a minimal version of $T_M(G)$, and let G' be obtained by eliminating some nonterminal of $T_M'(G)$ by substitution. Then G' is not LL(k') for any k'.

Proof The nonterminals of $T_M'(G)$ form a minimal cover of the minimizing expression of $T_M(G)$. Since G' is obtained by eliminating a nonterminal of $T_M'(G)$, its nonterminals do not form such a cover. So there is some minimizing term of $T_M(G)$ which does not contain any nonterminal of G'. By Lemma 4.35, since G' is well-defined, this term is not formed from a recursive sequence of nonterminals of $T_M(G)$. Hence this term is that constructed for some chain of $T_M(G)$. Thus if the nonterminal being eliminated from $T_M'(G)$ is A, there are

rules $B \rightarrow \alpha A \beta$, $A \rightarrow \varphi_1$, and $A \rightarrow \varphi_2$ in $T_M'(G)$, where α generates an infinite set of strings. Hence by Lemma 4.42, G' is not LL(k') for any k'. Q.E.D.

Lemma 4.45 Let G' be obtained by eliminating by substitution some nonterminal of G. Then any simple translation on G has an equivalent simple translation on G'.

Proof The new translation is constructed as follows. The rules of G that are also in G' retain their translation elements. If $B \rightarrow \alpha\varphi\beta$ is a new rule of G', created by substituting φ for A in $B \rightarrow \alpha A \beta$, consider the translation elements for these rules: $B \rightarrow \alpha A \beta \{\alpha' A \beta'\}$ and $A \rightarrow \varphi\{\varphi'\}$. Then the translation element for $B \rightarrow \alpha\varphi\beta$ will be $\{\alpha'\varphi'\beta'\}$. It is straightforward that this new translation is simple and is equivalent to the original one. Q.E.D.

Theorem 4.46 If $T_M'(G)$ is a minimal version of $T_M(G)$, then any simple translation on $T_M(G)$ has an equivalent simple translation on $T_M'(G)$.

Proof By preceding lemma. Q.E.D.

Using the same principle, we get:

Theorem 4.47 Let P be any compiler based on the LL(k) parser for $T_M(G)$. Then there is an equivalent compiler based on the LL(k') parser for $T_M'(G)$, where $T_M'(G)$ is a minimal version of $T_M(G)$ which is LL(k').

In summary then, if $T_M'(G)$ is a minimal version of $T_M(G)$, then it generates the same language that $T_M(G)$ does, and it is also LL(k') for some k'; it also supports the same compilations that $T_M(G)$ does. Furthermore, it really is minimal in the sense that any further nonterminal elimination is either impossible or destroys the grammar's LL-ness. We can compute all minimal versions

of $T_M(G)$ by constructing the nonterminal graph for $T_M(G)$, using the graph to find the minimizing expression, and then finding all minimal covers for this expression. Each minimal cover gives rise to a different minimal version, with nonterminals being those of the cover.

How shall we select among the various different minimal versions of a particular $T_M(G)$? Let us consider the two different minimal versions of Figure 4.17. They each have three nonterminals; the first has eleven productions, the second has ten. But most important, the first grammar is only LL(4) (because of the rules $(B,\epsilon) \rightarrow acx$ and $(B,\epsilon) \rightarrow acxy$), while the second is LL(2). By any reasonable standards then, the second is far superior. Is there any way we could have been guided to select that grammar, short of constructing all possible minimal versions of $T_M(G)$ and then choosing the one with the smallest lookahead value?

Let us first recall how the lookahead value of a grammar can increase during the process of eliminating nonterminals. As we have seen, this value can be increased by one when two possible right-hand sides for $(X, \varphi a)$ replace it in $(X, \varphi) \rightarrow a(X, \varphi a)$, resulting in $(X, \varphi) \rightarrow ab(X, \varphi ab)$ and $(X, \varphi) \rightarrow ac(X, \varphi ac)$. Repeated substitutions like this result in rules like $(X, \varphi) \rightarrow abcd\Psi_1$ and $(X, \varphi) \rightarrow abcd\Psi_2$, increasing the value of the needed lookahead even further. We can anticipate situations like this by examining the nonterminal graph for $T_M(G)$. A <u>path</u> through this graph is a sequence of vertices, such that there is an edge connecting each element of the sequence to the next one. It is easy to see that if p is a path in the graph connecting one element of the chosen minimal cover to another element, or to the vertex for ϵ, with no intervening vertices from the cover in the path, then there will be a rule in the minimal version of the grammar derived from p; in particular, if p con-

nects (X, φ_1) to (X, φ_2), the rule will be $(X, \varphi_1) \rightarrow \Psi(X, \varphi_2)$, where the nature of Ψ depends on the vertices the path passes through. Consequently, if there are paths from (X, φ_1) to (X, φ_2) and to (X, φ_3) which are coincident for their first parts, we will have rules of the form $(X, \varphi_1) \rightarrow \Psi\Psi_1(X, \varphi_2)$ and $(X, \varphi_1) \rightarrow \Psi\Psi_2(X, \varphi_3)$. For example, consider the graph of Figure 4.16. If the nonterminals of the minimum cover are (S, ϵ), (B, ϵ) and (B, aB), then there are the following paths through the graph: (B, ϵ), (B, a), (B, ac), (B, A), (B, Ax), (B, B), ϵ; and (B, ϵ), (B, a), (B, ac), (B, A), (B, Ax), (B, Axy), (B, B), ϵ. These paths are identical for a while, and give rise to rules $(B, \epsilon) \rightarrow acx$ and $(B, \epsilon) \rightarrow acxy$, which make the grammar LL(4). On the other hand, if the nonterminals are (S, ϵ), (B, a), and (B, A), the above two sequences are not paths, since the nonterminal (B, A) is in the interior of each of them.

The significance of all this is that we should be sensitive to this issue and attempt to prevent the existence of such paths through the graph. This can be done by trying to choose the nonterminals of the minimal cover so that there is a nonterminal closely "in front of" each branching in the graph. Then it will be impossible for there to be two long paths that start at the vertex for a nonterminal of the cover, that are identical for a long time, and that then diverge at a branching; impossible, because both these paths would run into the vertex of some other nonterminal of the cover before they reached the branching. Then the early part of the two sequences would be one path, giving rise to only a single rule; while the two paths which do eventually diverge are not coincident for very long, thus not increasing the lookahead greatly.

Admittedly these are only qualitative arguments, but they give us some handle for dealing with the choice of a minimal version of $T_M(G)$ that increases the lookahead value the least. We now have some criteria for compar-

ing the attractiveness of different minimal covers for the same grammar, the idea being to choose that one whose nonterminals are "closest" to branches in the graph. In the example we have been considering, this approach would have correctly led us to choose the minimal version with nonterminals (B, ϵ), (B, a), and (B, A), as opposed to other possibilities.

We can consider extending this approach, to creating a quasi-minimal version of $T_M(G)$, that contains more nonterminals than are strictly necessary for a minimal version, but which are carefully chosen so as to keep down the lookahead value of the resulting grammar. That is, it may be that any truly minimal version of $T_M(G)$ has an unsatisfactorily large lookahead; but by judiciously adding some extra nonterminals, it may be possible to reduce dramatically this lookahead value and hence the size of the parsing table. These additional nonterminals will be chosen to be "near" the branchings in the nonterminal graph for $T_M(G)$. In our example, if we chose (S, ϵ), (B, ϵ), and (B, Ax) as the minimal cover, and included the additional nonterminal (S, a), because it is situated at a branching, we would get the grammar of Figure 4.18. Though this grammar contains four nonterminals, which is one more than is strictly necessary for a minimal version of $T_M(G)$, it is also LL(1), which is better than any really minimal version of $T_M(G)$. We mention this as being of possible practical interest; we do not have any formal insight into this process of including superfluous nonterminals, or a precise analysis of when and how it might best be done.

$(S, \epsilon) \rightarrow a(S, a)$

$(S, a) \rightarrow cx$

$(S, a) \rightarrow dx$

$(S, a) \rightarrow a(B, a)x$

$(B, a) \rightarrow cx(B, Ax)$

$(B, a) \rightarrow dx(B, Ax)$

$(B, a) \rightarrow bx(B, Ax)$

$(B, a) \rightarrow a(B, a)\ (B, Ax)$

$(B, Ax) \rightarrow y$

$(B, Ax) \rightarrow \epsilon$

Figure 4.18

CHAPTER 5

CONSTRUCTING CYCLE-FREE MACHINES

5.1 Transformable Grammars

We have carefully explored the features of a new transformation which creates, for a non-LL(k) grammar G, an equivalent LL(k) grammar $T_M(G)$. We have studied various properties of the transformed grammar, and have also seen how it may be drastically reduced in size in order to bring it to more manageable proportions for use in a compiler. But now we must return to an earlier problem, because the point of departure for this transformation is not the original grammar G, but rather a cycle-free MSP(k) machine for G. The transformed grammar is even denoted $T_M(G)$, since it is derived from a particular cycle-free machine for G, M. This state of affairs immediately raises a number of important questions. How do we find a cycle-free MSP(k) machine for a given grammar G? How do we know whether or not there even exists such a machine? And if there are several cycle-free MSP(k) machines for G (and there often will be), what criteria should we use for determining which of these machines is most appropriate for application of the transformation procedure? Even further, could we be directed in our construction of cycle-free MSP(k) machines so that only the "best" such machine is constructed? It is to issues such as these that we turn our attention in this chapter.

Definition 5.1 A grammar G is k-transformable if there exists a cycle-free MSP(k) machine for G.

This definition enables us to couch our discussions in terms of grammars rather than in terms of particular machines for those grammars. A k-transformable grammar is suitable for application of the transformation process - we can try to find a cycle-free machine for it and then read a derived grammar off the machine. We shall try to get some feeling for this class of grammars.

Theorem 5.2 For an LR(k) grammar G, there are only finitely many MSP(k) machines for G.

Proof Since there are only finitely many LR(k)-items over G, there are only a finite number of possible states for an MSP(k) machine for G. Then since any machine has only finitely many states, all different, with only a finite number of connections among the states, the result is immediate. Q.E.D.

Theorem 5.3 It is decidable whether or not an LR(k) grammar G is k-transformable.

Proof The decision process is very straightforward: it consists of starting with the canonical LR(k) machine for G, constructing all possible MSP(k) machines for G from it, and then seeing whether or not any of these machines is cycle-free. By Theorem 3.52, every MSP(k) machine for G can be obtained by starting with the LR(k) machine and performing a sequence of state-splittings, none of which interfere with any of the previous ones. We also know, from Corollary 3.19, that it is possible to compute all possible state-splittings of any given state. Thus we have an effective procedure to compute all possible MSP(k) machines for an LR(k) grammar G. We begin with the LR(k) machine for G; at each stage, we will have some MSP(k) machine for G, and we pick some intermediate state of that machine which does not dominate any base state, choose

some splitting of that state, and replace the state by that splitting. If there is no such intermediate state, or no splitting of it, we backtrack and try another choice. In this way it is possible to exhaust all possible MSP(k) machines. Since at each stage, the current MSP(k) machine has one more base state than the machine of the preceding stage, we never get caught in a loop which constructs the same machines over and over. Once this construction phase is completed, we can examine each of the MSP(k) machines and see whether some one is cycle-free, and thus complete our decision procedure.

Q.E.D.

This decision procedure is not quite as unsatisfactory as it seems at first. Although it does blindly create all MSP(k) machines for G (and some particular machines possibly several times), we have given deterministic non-heuristic algorithms for determining splittings of a state and replacing a state by a splitting. This decision procedure would be suitable for execution by a computer, especially since it would only have to be done once, when trying to construct an LL(k) grammar for the language under consideration. Furthermore, in later sections we shall see how this procedure might be greatly improved by its understanding of how cycles may be removed from a machine by state-splitting.

Our first order of business, however, is to get a feeling for the extent of the class of k-transformable grammars. Our first major result will show that they include all the LL(k) grammars.

A word about notation. The lemmas that follow are very technical and their proofs are tedious; in the interests of simplifying the symbology and making the proofs more readiable, we have omitted explicit mention of the

right context $\dashv^k$. It is to be assumed that it is present however, so that FIRST_k of a string will always have k symbols in it.

Lemma 5.4 If $A \to \alpha_1.\alpha_2(\tau)$ and $B \to \beta_1.\beta_2(\omega)$ are essential items of the same state of the LR(k) machine for G, then for some $u \in V_T^*$, $S \overset{*}{\underset{L}{\Rightarrow}} u\,\alpha_2\Psi_1$ and $S \overset{*}{\underset{L}{\Rightarrow}} u\beta_2\Psi_2$, such that $\tau \in \text{FIRST}_k(\Psi_1)$ and $\omega \in \text{FIRST}_k(\Psi_2)$.

Proof Every state of the LR(k) machine is a β-successor of the initial state for some β. We proceed by induction on the length of β.

If $|\beta| = 0$, then there are no essential items in q, so to start the induction off safely we must consider the case where $|\beta| = 1$. Then q is a σ-successor of the initial state, and the items in question are $A \to \sigma.\alpha_2(\tau)$ and $B \to \sigma.\beta_2(\omega)$. Then $A \to .\sigma\alpha_2(\tau)$ and $B \to .\sigma\beta_2(\omega)$ are items of the initial state, and are each descended from S-items. Then we have $S \overset{*}{\underset{L}{\Rightarrow}} A\,\Psi_1$ and $S \overset{*}{\underset{L}{\Rightarrow}} B\,\Psi_2$, with $\tau \in \text{FIRST}_k(\Psi_1)$ and $\omega \in \text{FIRST}_k(\Psi_2)$; then if we choose u such that $\sigma \overset{*}{\underset{L}{\Rightarrow}} u$, we have $S \overset{*}{\underset{L}{\Rightarrow}} A\,\Psi_1 \underset{L}{\Rightarrow} \sigma\,\alpha_2\,\Psi_1 \overset{*}{\underset{L}{\Rightarrow}} u\,\alpha_2\,\Psi_1$ and $S \overset{*}{\underset{L}{\Rightarrow}} B\,\Psi_2 \underset{L}{\Rightarrow} \sigma\beta_2\,\Psi_2 \overset{*}{\underset{L}{\Rightarrow}} u\,\beta_2\,\Psi_2$, as desired.

Now say $|\beta| = n+1$. Then $\beta = \alpha\sigma$, where $|\alpha| = n$; and q is the σ-successor of q', which is some α-successor of the initial state. Then the items of q in question are $A \to \alpha_1\sigma.\alpha_2(\tau)$ and $B \to \beta_1\sigma.\beta_2(\omega)$; and $A \to \alpha_1.\sigma\alpha_2(\tau)$ and $B \to \beta_1.\sigma\beta_2(\omega)$ are items of q'. Now either both, one, or neither of these items is an essential item of q'. If both are, then by induction we have $S \overset{*}{\underset{L}{\Rightarrow}} u\,\sigma\alpha_2\Psi_1$ and $S \overset{*}{\underset{L}{\Rightarrow}} u\,\sigma\beta_2\Psi_2$; if we choose u_1 such that $\sigma \overset{*}{\underset{L}{\Rightarrow}} u_1$ and set $u_2 = uu_1$, we have $S \overset{*}{\underset{L}{\Rightarrow}} u_2\alpha_2\Psi_1$ and $S \overset{*}{\underset{L}{\Rightarrow}} u_2\beta_2\Psi_2$, with $\tau \in \text{FIRST}_k(\Psi_1)$ and $\omega \in \text{FIRST}_k(\Psi_2)$ as required. If $A \to \alpha_1.\sigma\alpha_2(\omega)$ is essential in q' while $B \to .\sigma\beta_2(\omega)$ is not, then the latter is descended from $C \to \gamma_1.D\gamma_2(\rho)$ which is essential in q'. Then $D \overset{*}{\underset{L}{\Rightarrow}} \sigma\beta_2\varphi$, such that $\omega \in \text{FIRST}_k(\varphi\gamma_2\rho)$. By induction,

$S \overset{*}{\underset{L}{\Rightarrow}} u\sigma\alpha_2\Psi_1$ and $S \overset{*}{\underset{L}{\Rightarrow}} uD\gamma_2\Psi_2$, with $\rho \in FIRST_k(\Psi_2)$. Then $S \overset{*}{\underset{L}{\Rightarrow}} uD\gamma_2\Psi_2 \overset{*}{\underset{L}{\Rightarrow}} u\sigma\beta_2\varphi\gamma_2\Psi_2$, with $\omega \in FIRST_k(\varphi\gamma_2\Psi_2)$. If we define u_1 and u_2 as before, then $S \overset{*}{\underset{L}{\Rightarrow}} u_2\alpha_2\Psi_1$ with $\tau \in FIRST_k(\Psi_1)$ and $S \overset{*}{\underset{L}{\Rightarrow}} u_2\beta_2\varphi\gamma_2\Psi_2$, with $\omega \in FIRST_k(\varphi\gamma_2\Psi_2)$, which was the required result. The final case, where $A \rightarrow .\sigma\alpha_2(\tau)$ and $B \rightarrow .\sigma\beta_2(\omega)$ are both non-essential items of q', follows from an identical analysis. Q.E.D.

We note that we could have used essentially the same proof to prove the following more general result.

Lemma 5.5 If $A \rightarrow \alpha_1.\alpha_2(\tau)$ and $B \rightarrow \beta_1.\beta_2(\omega)$ are any two items of the same state of the LR(k) machine for G, then for some $u \in V_T^*$, $S \overset{*}{\underset{L}{\Rightarrow}} u\alpha_2\Psi_1$ and $S \overset{*}{\underset{L}{\Rightarrow}} u\beta_2\Psi_2$, where $\tau \in FIRST_k(\Psi_1)$ and $\omega \in FIRST_k(\Psi_2)$.

The proof of this lemma follows the same inductive structure as that of the previous one, and the details are only slightly different. We chose to prove explicitly the more restricted result for two reasons: the proof techniques employed there will be used again; and primarily because it has an important analogue, Lemma 5.15, which will be used later, and which does not have a more general form analogous to Lemma 5.5.

Definition 5.6 The core of an item $A \rightarrow \alpha_1.\alpha_2(\tau)$ is $A \rightarrow \alpha_1.\alpha_2$.

Theorem 5.7 Let G be an LL(k) grammar, q any state of the LR(k) machine for G. If I_1 and I_2 are essential items of q with different cores, then $FIRST_k(I_1) \cap FIRST_k(I_2) = \phi$.

Proof. Every state of an LR(k) machine is a β-successor of the initial state for some β. We proceed by induction on the length of β. If $|\beta| = 0$, then q is the initial state; since there are no essential items in the initial state, the statement is vacuously true. (If this is not satisfying,

we can explicitly apply the techniques described below, to start the induction for $|\beta| = 1$.)

Now suppose the theorem is true if $|\beta| = n$; we shall show it true for $|\beta| = n+1$. Now if $|\beta| = n+1$, we can write β as $\alpha\sigma$, where $|\alpha| = n$ and $\sigma \in V_N \cup V_T$. Then q is the σ-successor of q', which is an α-successor of the initial state. Thus every essential item of q has σ just before the dot. Let $A \rightarrow \alpha_1\sigma.\alpha_2(\tau)$ and $B \rightarrow \beta_1\sigma\ \beta_2(\omega)$ be any two essential items I_1 and I_2 of q; then $A \rightarrow \alpha_1.\sigma\alpha_2(\tau)$ and $B \rightarrow \beta_1.\sigma\beta_2(\omega)$ are items of q'. We assume that $x \in FIRST_k(I_1) \cap FIRST_k(I_2)$, and will prove a contradiction.

There are a variety of possibilities for the items $A \rightarrow \alpha_1.\sigma\alpha_2(\tau)$ and $B \rightarrow \beta_1.\sigma\beta_2(\omega)$ of q': they may both be essential items of q'; exactly one may be an essential item; or neither may be. In any event, since $x \in FIRST_k(\beta_2\omega) \cap FIRST_k(\alpha_2\tau)$, there is certainly some y such that $y \in FIRST_k(\sigma\beta_2\omega) \cap FIRST_k(\sigma\alpha_2\tau)$.

If they are both essential items, then they have different cores, since $A \rightarrow \alpha_1\sigma.\alpha_2(\tau)$ and $B \rightarrow \beta_1\sigma.\beta_2(\omega)$ have different cores. But then y will be in $FIRST_k$ of two essential items of q' with different cores, which contradicts the induction hypothesis, since q' is an α-successor of the initial state where $|\alpha| = n$.

Next assume $A \rightarrow \alpha_1.\sigma\alpha_2(\tau)$ is an essential item of q', while $B \rightarrow .\sigma\beta_2(\omega)$ is not. Then $B \rightarrow .\sigma\beta_2(\omega)$ is not the descendant of any item with the same core as $A \rightarrow \alpha_1.\sigma\alpha_2(\tau)$, since that would make σ left recursive, which is impossible in an LL(k) grammar. Thus $B \rightarrow .\sigma\beta_2(\omega)$ is a descendant of $C \rightarrow \gamma_1.\gamma_2(\rho)$, an essential item of q' with a different core from that of $A \rightarrow \alpha_1.\sigma\alpha_2(\tau)$. Therefore $y \in FIRST_k(\gamma_2\rho)$, which means that y is in $FIRST_k$ of both $C \rightarrow \gamma_1.\gamma_2(\rho)$ and $A \rightarrow \alpha_1.\sigma\alpha_2(\tau)$, which violates the induction hypothesis.

Finally, suppose $A \to .\sigma\alpha_2(\tau)$ and $B \to .\sigma\beta_2(\omega)$ are both non-essential items of q'. We shall refer to these items as I_3 and I_4. If I_3 and I_4 are descended from essential items of q' with different cores, then y is in $FIRST_k$ of each of these items, which contradicts the induction hypothesis. Thus I_3 and I_4 are descended from items $C \to \gamma_1 \cdot \gamma_2(\rho_1)$ and $C \to \gamma_1 \cdot \gamma_2(\rho_2)$. We observe that neither of I_3 and I_4 can be a descendant of the other, for that would make σ left recursive. Thus there are chains of items of q', $J_0, J_1, \ldots J_n$ and $K_0, K_1, \ldots, K_m$, such that $J_0 = C \to \gamma_1 \cdot \gamma_2(\rho_1)$, $K_0 = C \to \gamma_1 \cdot \gamma_2(\rho_2)$, $J_n = I_3$, and $K_m = I_4$. Let j be the smallest number such that the core of J_j is not the same as the core of K_j; since I_3 and I_4 have different cores, and since neither chain is a prefix of the other, there must be such a $j > 0$. Then J_j is $E \to .\Psi_1(\pi_1)$ and K_j is $E \to .\Psi_2(\pi_2)$; the left-hand sides are the same since J_{j-1} and K_{j-1} had identical cores. Then $E \to \Psi_1$ and $E \to \Psi_2$ are different rules of G. Since I_3 is a descendant of $E \to .\Psi_1(\pi_1)$ and since $y \in FIRST_k(I_3)$, it means that $\Psi_1 \overset{*}{\underset{L}{\Rightarrow}} z_1$ such that $y = z_1\pi_1/k$; and similarly for I_4. Thus for some u, we have $S \overset{*}{\underset{L}{\Rightarrow}} uE\pi_1v_1 \underset{L}{\Rightarrow} u\Psi_1\pi_1v_1 \overset{*}{\underset{L}{\Rightarrow}} uz_1\pi_1v_1$ and $S \overset{*}{\underset{L}{\Rightarrow}} uE\pi_2v_2 \underset{L}{\Rightarrow} u\Psi_2\pi_2v_2 \overset{*}{\underset{L}{\Rightarrow}} uz_2\pi_2v_2$ for some v_1 and v_2, where $z_1\pi_1/k = z_2\pi_1/k$; and this violates the definition of LL(k) grammars.

This is the final case and completes the induction. Q.E.D.

In proving the results of this section, we shall use a slightly more general definition of a legal state-splitting and of MSP(k) machine than we have been using. The change is so minor however, that all our important results are valid for the new definition and require only the most trivial alterations in their proofs. The change consists of changing $L_k(q, A_i)$ to $L_k(B, A_i)$ throughout the definition of a state splitting; and specifying in the definition of

an MSP(k) machine, that if q is a base state and q_i is an image of q under g_1, then $\{x \mid g_1(q, x) = q_i\} = L_k(q, g_2(q_i))$. The meaning of this revision is easily explained. Originally we required that there be some $.A_i$ item in each $L_k(q', A_i)$ chain, and that the set of strings, which were to occasion the prediction of an A_i from the base of a splitting of q', was to be equal to the set of all lookaheads that q' might have seen and that could have come from A_i (i.e., $L_k(q', A_i)$). But now we relax this requirement somewhat. We now say that a state-splitting is legal if the possible lookaheads from those A_i's which are being predicted (i.e, those A_i's which are the post-dot component of some item in the base state) always ensure that a $.A_i$ will be found; that is, we only need to break $L_k(B, A_i)$ chains in order to define the splitting. It is not necessary for a lookahead that comes from any A_i to indicate the presence of A_j in the state, but only for those lookaheads that come from an A_i in the base state; it is this set of lookaheads which will cause an MSP(k) machine containing the splitting to make the predictive transition. Conceptually, this is an extension of a similar concept that occurred in our earlier definition, in the case of a left recursive A_i.

Making the stipulated changes in the definitions is completely inessential to the main course of our development. All major results need only have the same changes made in their statements or proofs in order to maintain their validity. The only results that cannot be so easily updated are 3.31 through 3.34. These results were not very important and were only provided to give some feeling as to what constituted a legal state-splitting; slightly different versions of them can be obtained for the revised definition of a splitting. The operation of the MSP(k) machine is unaltered, as is the

grammatical derivation procedure from cycle-free machines; and the transformed grammar has all the properties we proved in Chapter 4. From now on we shall be using these new definitions of state splitting and MSP(k) machine; and a k-transformable grammar means one for which such a cycle-free MSP(k) machine exists. It is true that the class of k-transformable grammars under the new definition is larger than the class using the old definition, for the changes we have made do enlarge the class of legal state-splittings.

Theorem 5.8 If q is a non-final state of the LR(k) machine for an LL(k) grammar G, then there is a splitting (B, Q) of q such that B consists of precisely the essential items of q.

Proof We directly define the splitting (B, Q). If A is a non-terminal post-dot component of some essential item of q, then there is an initial state associated with A, consisting of all descendants of all .A-essential items of q. An initial state P_i is created for each such nonterminal A_i, and B is

defined as precisely the essential items of q.

We must show that this proposed splitting is indeed legal. First suppose that $L_k(B,A_i) \cap L_k(B,A_j) \neq \emptyset$, for some i and j. Then there is some essential $.A_i$ item I_1, and some essential $.A_j$ item I_2, such that $FIRST_k(I_1) \cap FIRST_k(I_2) \neq \emptyset$. But since $A_i \neq A_j$, I_1 and I_2 must have different cores, so by the preceding result, $FIRST_k(I_1) \cap FIRST_k(I_2) = \emptyset$.

Next, since G is LL(k), there can be no $.A_i$ items in P_i, for that would make A_i left-recursive. Hence $FOLLOW_k(A_i,B) \cap FOLLOW_k(A_i,P_i) = \emptyset$.

Finally, consider any $L_k(B,A_i)$ chain c. Then this chain must begin with a $.A_i$-essential item, for otherwise, there would be two essential items with different cores but with intersecting $FIRST_k$'s. We define $H_1(c)$ to be the first item in c, the essential item, and $H_2(c)$ to be the rest of c. We have to show that these definitions of H_1 and H_2 satisfy the defining equations for B and P_i. First of all, $H_1(c) \subset B$ for any c, by definition of B as all essential items of q. Then if I is such that $FIRST_k(I) \not\subset \bigcup_i L_k(B,A_i)$, I must be a terminal essential item, in which case it will be in B. On the other hand, an item I is in B if and only if it is an essential item; if it is not a terminal essential item, then it is in $H_1(c)$ for some c; while if it is a terminal essential item, then $FIRST_k(I)$ does not intersect $FIRST_k(I')$, for any non-terminal essential item I', by the previous lemma, and thus is not contained in $\bigcup_i L_k(B,A_i)$. Thus the defining equation for B is satisfied.

If I is any item in P_i, then I must be a descendant of some essential $.A_i$-item, and hence will be in $H_2(c)$ for some c. On the other hand, if I is in $H_2(c)$ for some $L_k(B,A_i)$-chain c, then c must begin with an essential

.A_i-item by the previous lemma, and so I will be a descendant of some essential .A_i-item. Therefore the defining equation for P_i is satisfied, and we are done. Q.E.D.

Theorem 5.9 Let G be an LL(k) grammar, and q be any state of the LR(k) machine for G. If q' is any state whose essential items are a subset of the essential items of q, then there is a splitting of q' such that the base state is precisely the essential items of q'.

Proof We define the splitting in this case the same way as we did in the preceding case. If I_1 and I_2 are essential items of q' with different cores, then they are also essential items of q with different cores, so $FIRST_k(I_1) \cap FIRST_k(I_2) = \phi$. Furthermore, since no essential .A_i-item in q has any .A_i-items as descendants in q, the same holds true in q'. These are the only two facts needed to make the preceding proof go through for q' as well.

Q.E.D.

Theorem 5.10 Let M_0 be the LR(k) machine for the LL(k) grammar G. Then there exists a cycle-free MSP(k) machine M for G.

Proof We shall give a sure-fire, though somewhat mindless, algorith for constructing M from M_0 by a sequence of state-splittings. In later sections we shall see how this procedure could be refined.

1) Set n = 1.
2) Find an intermediate state q such that q is accessible from the starting state by α, where $|\alpha| = n$; if there is no such q, go to step 4
3) Replace q by a splitting (B,Q) of q, such that B is precisely the essential items of q; go to step 2

4) If there are any intermediate states still unsplit in the machine, set n = n+1 and go to step 2; otherwise, stop.

This procedure creates a sequence of MSP(k) machines for G, starting with M_0, such that each is obtained from the preceding one by the performance of a state-splitting. We claim that this procedure is well-defined, that it terminates, and results in a cycle-free machine.

First of all, we recall that any state of an MSP(k) machine for G is contained in some state of the LR(k) machine for G, so that any such intermediate MSP(k) state can indeed be split in such a way that the base of the splitting is just the essential items of the state; this is by the previous result. So at each stage the procedure is well-defined.

Now to show that it terminates. First of all, since there are only finitely many MSP(k) machines for G, there is some value N such that every state of every MSP(k) machine for G is accessible from the starting state by a path of length less than N. Furthermore, if at some point in the procedure, the test indicates that there are no more unsplit states accessible by a path of length m, then there are never any suchunsplit states later in the procedure. Thus if at some stage, there are unsplit states, they are accessible by paths longer than the current value of n and shorter than N, and so will be considered later. Thus this procedure terminates at the latest when n reaches the value of N; and when it does terminate, the resulting MSP(k) machine consists only of initial states and of base states with only essential items. We claim that this machine is cycle-free. If there were a cycle in this machine, it would have to contain only base states, since an initial state is not a successor of any other state.

Now if there were some cycle involving base states, let q be any base state in such a cycle. Recall that q and all other states in the cycle contain only essential items. If the length of the cycle in which q is involved is of length m, then it must be the case that the essential items of q can be obtained from themselves by moving the dot in each m places to the right. But this is absurd, so there can be no such cycle, and we are done. Q.E.D.

Corollary 5.11 If G is LL(k), then G is k-transformable.

This result has interesting consequences on a number of different levels. First of all, it enhances the respectability of our definitions. We have gone to great pains to describe a class of grammars that can be transformed into LL(k) form, and it is reassuring to know that this class at least includes all those grammars which are already LL(k). But there are more substantive issues as well.

Corollary 5.12 The class of languages recognized by the class of cycle-free MSP(k) machines is precisely the LL(k) languages.

In other words, for every LL(k) language $\mathcal{L}$, there is a grammar G such that L(G) = $\mathcal{L}$ and such that there is a cycle-free MSP(k) machine for G.

Next we wish to demonstrate an even further extent of the k-transformable grammars and the attractiveness of our transformation. There is one large decidable class of grammars which are not LL(k) and which can be transformed into LL(k) form by the application of a precise algorithm: these are the LC(k) grammars of Rosenkrantz and Lewis. Intuitively, these are grammars that can be parsed in a mixed hybrid of bottom-up and top-down

parsing. The basic idea is that the identity of a rule used in a derivation can be established from the left corner of the rule (the first symbol on its right hand side) and from the lookahead following the left corner. The details are complex, and a complete discussion is given in [19]. It is our goal to show that the k-transformable grammars strictly include the class of LC(k) grammars.

We review the definition of LC(k) grammars.

Definition 5.13 The left corner of the production $A \to \Psi$ is the first symbol of Ψ.

Definition 5.14 If A is a nonterminal, we say that $S \overset{*}{\underset{LC}{\Rightarrow}} uA\Psi$ where $u \in V_T^*$ and $\Psi \in (V_N \cup V_T)^*$ if $S \overset{*}{\underset{L}{\Rightarrow}} uA\Psi$ and if A is not the left corner of the last production used in the derivation.

Lemma 5.15 If $A \to \alpha_1.\alpha_2(\tau)$ and $B \to \beta_1.\beta_2(\omega)$ are essential items of the same state of the LR(k) machine for G, then for some $u \in V_T^*$, $S \overset{*}{\underset{LC}{\Rightarrow}} u\alpha_2\Psi_1$ and $S \overset{*}{\underset{LC}{\Rightarrow}} u\beta_2\Psi_2$ such that $\tau \in FIRST_k(\Psi_1)$ and $\omega \in FIRST_k(\Psi_2)$.

Proof The proof of Lemma 5.4 can be interpreted as a proof of this lemma as well. Q.E.D.

We state without proof the following elementary result.

Lemma 5.16 If $B \to .\beta(\omega)$ is descended from $A \to \alpha_1.\alpha_2(\rho)$, then $C \overset{*}{\underset{R}{\Rightarrow}} B\eta$ for some $\eta \in V_T^*$ such that $\omega \in FIRST_k(\tau\alpha_2\rho)$

Definition 5.17 G is LC(k), $k > 0$, if the following three conditions hold:

1) given $t \in V_T^*$, $x \in V_T^k$, $A \in V_N$, there is at most one production of the form $B \to \varphi$ where φ begins with a terminal symbol or is ϵ such that $S \underset{LC}{\overset{*}{\Rightarrow}} t A \Psi$ and $A \underset{R}{\overset{*}{\Rightarrow}} B \rho$ for some $\Psi \in (V_N \cup V_T)^*$ and $\rho \in V_T^*$, where $x \in FIRST_k(\varphi\rho\Psi)$

2) given $t \in V_T^*$, $x \in V_T^k$, $A \in V_N$, and $C \in V_N$, there is at most one production of the form $B \to C\varphi$ such that $S \underset{LC}{\overset{*}{\Rightarrow}} uA\Psi$ and $A \underset{R}{\overset{*}{\Rightarrow}} B\rho$ for some u, $\rho \in V_T^*$ and $\Psi \in (V_N \cup V_T)^*$, such that $u\, C \overset{*}{\Rightarrow} t$ and $x \in FIRST_k(\varphi\rho\Psi)$.

3) if in the case where $C = A$, there is a production $B \to A\varphi$ satisfying 2), then there do not exist u' and Ψ' such that $S \underset{LC}{\overset{*}{\Rightarrow}} u'A\Psi'$ and such that $u'A \overset{*}{\Rightarrow} t$ and $x \in FIRST_k(\Psi')$.

We have slightly altered the notation of this definition from how it appears in [19], in order to make it easier for us to handle, but we have not made any significant changes.

Theorem 5.18 Let G be any LC(k) grammar, q any state of the LR(k) machine for G. If I_1 and I_2 are essential items of q with different cores, then $FIRST_k(I_1) \cap FIRST_k(I_2) = \phi$.

Proof The structure of the proof is identical to the proof of Theorem 5.7, and most of the details are the same as well. We shall not recopy that proof, but shall only discuss those parts of the other proof that utilized the fact that G was LL(k) in that case. There are two such cases, where it was argued that $B \to .\sigma\beta_2(\omega)$ could not be descended from $A \to \alpha_1.\alpha\alpha_2(\rho)$, and where it was argued that $A \to .\alpha\alpha_2(\tau)$ and $B \to .\sigma\beta_2(\omega)$ could not be descended from essential items of q with the same core. We shall reconsider these two cases. Recall that the assumption was that $x \in FIRST_k(\alpha_2\tau) \cap FIRST_k(\beta_2\omega)$, and that $y \in FIRST_k(\sigma\beta_2\omega) \cap FIRST_k(\alpha\alpha_2\tau)$; this was to lead to a contradiction.

Suppose that $A \to \alpha_1 . \sigma\alpha_2(\tau)$ is an essential item and that $B \to .\sigma\beta_2(\omega)$ is not. If $B \to .\sigma\beta_2(\omega)$ is descended from an item with a different core from that of $A \to \alpha_1 . \sigma\alpha_2(\tau)$, we immediately reach a contradiction, as we did in the proof of Theorem 5.7. Assume then that $B \to .\sigma\beta_2(\omega)$ is descended from an item of the form $A \to \alpha_1 . \sigma\alpha_2(\rho)$ in q'. Then σ is a nonterminal; call it C.

Since $A \to \alpha_1 . C\alpha_2(\rho)$ is an essential item of q', we have $S \overset{*}{\underset{LC}{\Rightarrow}} uC\alpha_2\Psi$, with $\rho \in FIRST_k(\Psi)$; since $B \to .C\beta_2(\omega)$ is descended from $A \to \alpha_1 . C\alpha_2(\rho)$, we have $C \overset{*}{\underset{R}{\Rightarrow}} B\eta$, with $\omega \in FIRST_k(\eta\alpha_2\Psi)$. Then since $x \in FIRST_k(\beta_2\omega)$, we have $x \in FIRST_k(\beta_2\eta\alpha_2\Psi)$. Thus for t, x, and C, there is a rule $B \to C\beta_2$ such that $S \overset{*}{\underset{LC}{\Rightarrow}} uC\gamma$, $C \overset{*}{\underset{R}{\Rightarrow}} B\eta$, and $uC \overset{*}{\Rightarrow} t$, where $x \in FIRST_k(\beta_2\eta\gamma)$. On the other hand, since $A \to \alpha_1 . C\alpha_2(\tau)$ is an essential item of q', we have $S \overset{*}{\underset{LC}{\Rightarrow}} uC\alpha_2\Psi'$, with $\tau \in FIRST_k(\Psi')$; since $x \in FIRST_k(\alpha_2\tau)$, this means that $S \overset{*}{\underset{LC}{\Rightarrow}} uC\gamma'$, with $uC \overset{*}{\Rightarrow} t$ and $x \in FIRST_k(\gamma')$. But these two statements contradict the third clause of the definition of LC(k) grammars.

To try to make the foregoing a little clearer, consider the two trees of Figure 5.1. They illustrate the two situations just delineated, and do indeed violate the definition of LC(k)-ness.

Now, we must consider the case where $A \to .\sigma\alpha_2(\tau)$ and $B \to .\sigma\beta_2(\omega)$ are both non-essential items of q'. If they are descended from items with different cores, then y is in $FIRST_k$ of each of those items and we are done. Assume then that they are descended from essential items with the same core, namely $C \to \gamma_1 . D\gamma_2(\rho_1)$ and $C \to \gamma_1 . D\gamma_2(\rho_2)$ respectively. Since these two items are in q', we have $S \overset{*}{\underset{LC}{\Rightarrow}} uD\gamma_2\Psi_1$ and $S \overset{*}{\underset{LC}{\Rightarrow}} uD\gamma_2\Psi_2$, with $\rho_i \in FIRST_k(\Psi_i)$. If σ is a non-terminal, then for a given t, x, D, and σ, we have two productions

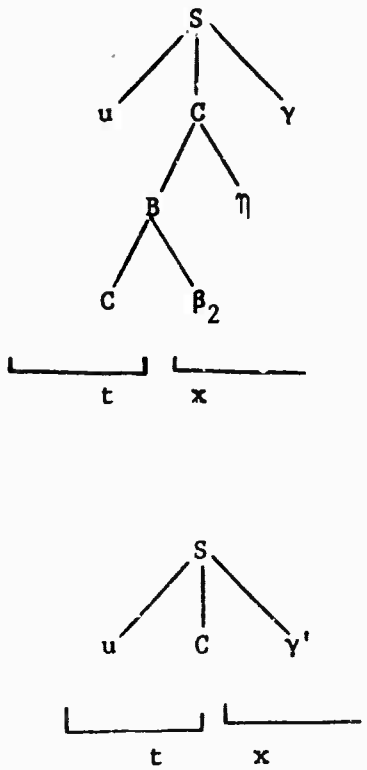
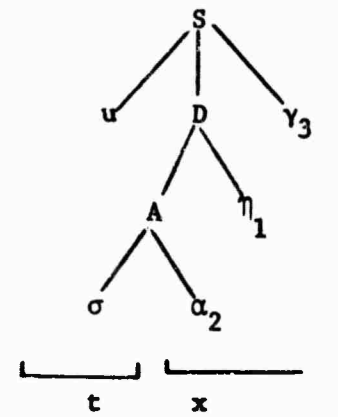


Figure 5.1

$A \to \alpha\alpha_2$ and $B \to \sigma\beta_2$, such that $S \underset{LC}{\overset{*}{\Rightarrow}} uD\gamma_3$, $D \underset{R}{\overset{*}{\Rightarrow}} A\eta_1$ where $u\sigma \overset{*}{\Rightarrow} t$ and $x \in FIRST_k(\alpha_2\eta_1\gamma_3)$ and such that $S \underset{LC}{\overset{*}{\Rightarrow}} uD\gamma_4$, $D \underset{R}{\overset{*}{\Rightarrow}} B\eta_2$, where $u\sigma \overset{*}{\Rightarrow} t$ and $x \in FIRST_k(\beta_2\eta_2\gamma_4)$, which violates the second clause of the LC(k) definition. Once again we include an illustrative figure, Figure 5.2.



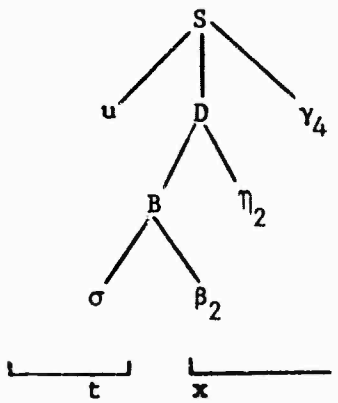
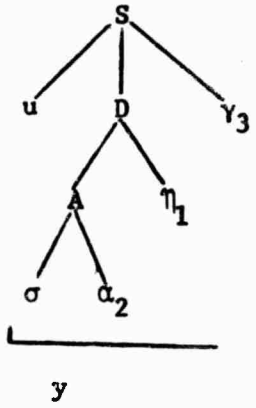


Figure 5.2

Finally, we must consider the case where $A \rightarrow .\sigma\alpha_2(\tau)$ and $B \rightarrow .\sigma\beta_2(\omega)$ are descended from essential items of q' with the same core, and σ is a terminal symbol. Then as in the preceding case, $S \overset{*}{\underset{LC}{\Rightarrow}} uD\gamma_3$, $D \overset{*}{\underset{R}{\Rightarrow}} A\eta_1$ with $\tau \in FIRST_k(\eta_1\gamma_3)$, and $S \overset{*}{\underset{LC}{\Rightarrow}} uD\gamma_4$, $D \overset{*}{\underset{R}{\Rightarrow}} B\eta_2$ with $\omega \in FIRST_k(\eta_2\gamma_4)$. But since $y \in FIRST_k(\sigma\beta_2\omega) \cap FIRST_k(\sigma\alpha_2\tau)$, this means that given u, y, and D, there are two productions beginning with terminals, namely $A \rightarrow \sigma\alpha_2$ and $B \rightarrow \sigma\beta_2$, such that $S \overset{*}{\underset{LC}{\Rightarrow}} uD\gamma_3$, $D \overset{*}{\underset{R}{\Rightarrow}} A\,\eta_1$, with $y \in FIRST_k(\sigma\alpha_2\eta_1\gamma_3)$, and such that $S \overset{*}{\underset{LC}{\Rightarrow}} uD\gamma_4$, $D \overset{*}{\underset{R}{\Rightarrow}} A\,\eta_2$, with $y \in FIRST_k(\sigma\beta_2\eta_2\gamma_4)$. But this contradicts the

first clause of the LC(k) definition. Figure 5.3 is included for illustrative purposes.



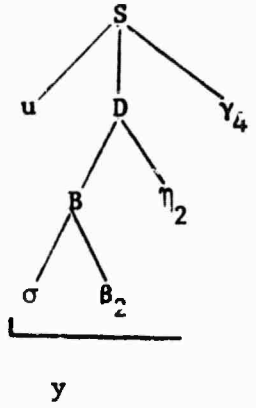


Figure 5.3

This concludes the enhancement of the proof of Theorem 5.7 to make it apply to LC(k) grammars as well, and so we are done. Q.E.D.

We now proceed to establish the analogue to Theorem 5.8.

Theorem 5.19 If q is a non-final state of the LR(k) machine for an LC(k) grammar G, then there is a splitting (B,Q) of q such that B consists of precisely the essential items of q.

Proof We define (B,Q) as we did in the earlier theorem. Namely, if there is an essential .A-item in q, then there is a predictive state for A, consisting of the descendants of all essential .A-items in q.

The proof of the earlier theorem works almost exactly here, because of our preceding result. The only thing we have to demonstrate differently in this case is that $FOLLOW_k(A_i,B) \cap FOLLOW_k(A_i,P_i) = \emptyset$, because here $FOLLOW_k(A_i,P_i)$ might not be empty. But this fact follows from the LC(k)-ness of G, as follows.

If $x \in FOLLOW_k(A_i,B) \cap FOLLOW_k(A_i,P_i)$, then there is some essential $.A_i$-item $C \rightarrow \alpha_1.A_i\alpha_2(\tau)$ with $x \in FIRST_k(\alpha_2\tau)$, and some non-essential item $D \rightarrow .A_i\beta_2(\omega)$ with $x \in FIRST_k(\beta_2\omega)$ such that the latter is a descendant of some essential $.A_i$-item $E \rightarrow \gamma_1.A_i\gamma_2(\rho)$.

Then we have $S \underset{LC}{\overset{*}{\Rightarrow}} uA_i\gamma_2\Psi$, with $\rho \in FIRST_k(\Psi)$, and $S \underset{LC}{\overset{*}{\Rightarrow}} uA_i\alpha_2\Psi'$ with $\tau \in FIRST_k(\Psi')$. Further, we know that $A_i \underset{R}{\overset{*}{\Rightarrow}} D\eta$, with $\omega \in FIRST_k(\eta\alpha_2\Psi')$; hence $x \in FIRST_k(\beta_2\eta\alpha_2\Psi')$. Thus for a given t, x, and A_i, we have a rule $D \rightarrow A_i\beta_2$ with $S \underset{LC}{\overset{*}{\Rightarrow}} uA_i\gamma$ and $A_i \underset{R}{\overset{*}{\Rightarrow}} D\eta$ such that $uA_i \overset{*}{\Rightarrow} t$ and $x \in FIRST_k(\beta_2\eta\gamma)$. But at the same time, we have $S \underset{LC}{\overset{*}{\Rightarrow}} uA_i\alpha_2\Psi'$ such that $uA_i \overset{*}{\Rightarrow} t$ and $x \in FIRST_k(\alpha_2\Psi')$. This contradicts the third clause of the LC(k) definition, and so we are done. Q.E.D.

We illustrate this argument in Figure 5.4.

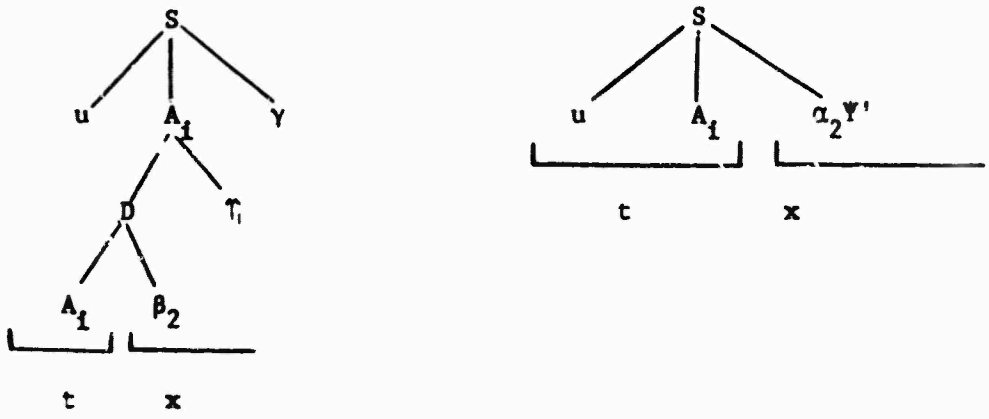


Figure 5.4

Theorem 5.20 If G is LC(k), then G is k-transformable.

Proof The algorithm of Theorem 5.10, for constructing a cycle-free MSP(k) machine for an LL(k) grammar G, only used the property that any state of the LR(k) machine for G could be split in such a way that the base of the splitting contained only essential items. Since we have just shown that this property applies as well to LC(k) grammars, that algorithm will work for LC(k) grammars as well. Q.E.D.

This means that our transformation is at least as effective as the one in [19]. Actually, since every LL(k) grammar is LC(k), Corollary 5.11 follows from Theorem 5.20, but we felt that the former proof was more understandable than the latter, so both were included.

Consider the LR(0) grammar S → bAc, A → ABx, A → ABy, A → a, B → Bd, B → d. This grammar can be shown not to be LC(k) for any k; intuitively, the reason is that even after seeing past the corners of A → ABx and A → ABy, no amount of lookahead can distinguish these two rules. However, this grammar

is 0-transformable; its LR(0) machine is given in Figure 5.5, and is obviously cycle-free.

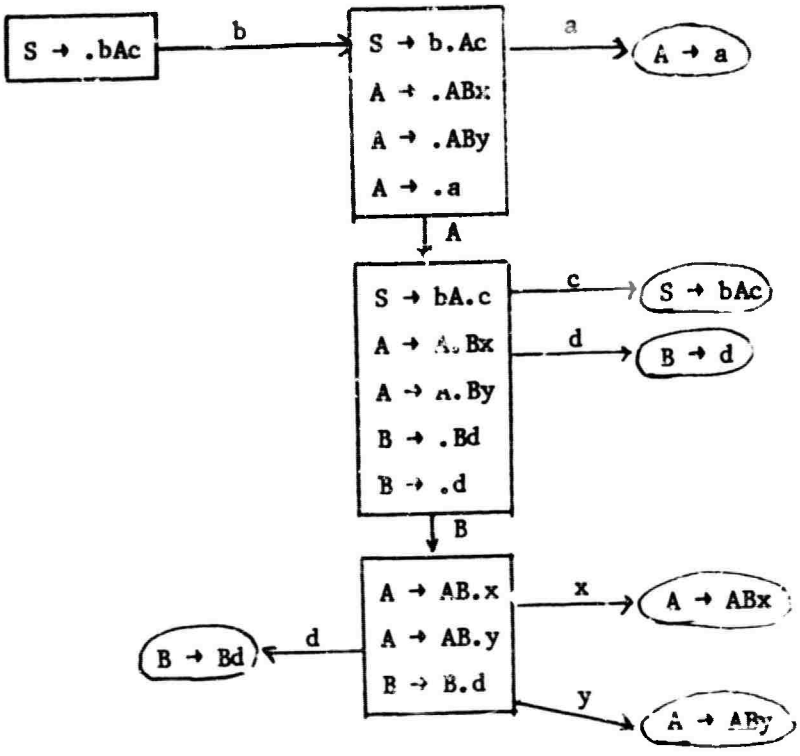


Figure 5.5

The grammar derived from this machine is given in Figure 5.6 as another illustrative example.

(S,ϵ) → b(S,b)
(S,b) → a(S,ba)
(S,ba) → (S,bA)
(S,bA) → c(S,bAc)
(S,bA) → d(S,bAd)
(S,bAc) → (S,S)
(S,bAd) → (S,bAB)
(S,S) → ϵ
(S,bAB) → d(S,bABd)
(S,bAB) → x(S,bABx)
(S,bAB) → y(S,bABy)
(S,bABd) → (S,bAB)
(S,bABx) → (S,bA)
(S,bABy) → (S,bA)

Figure 5.6

This grammar is LL(1), as advertised. In order to reduce its size, we construct its nonterminal graph.

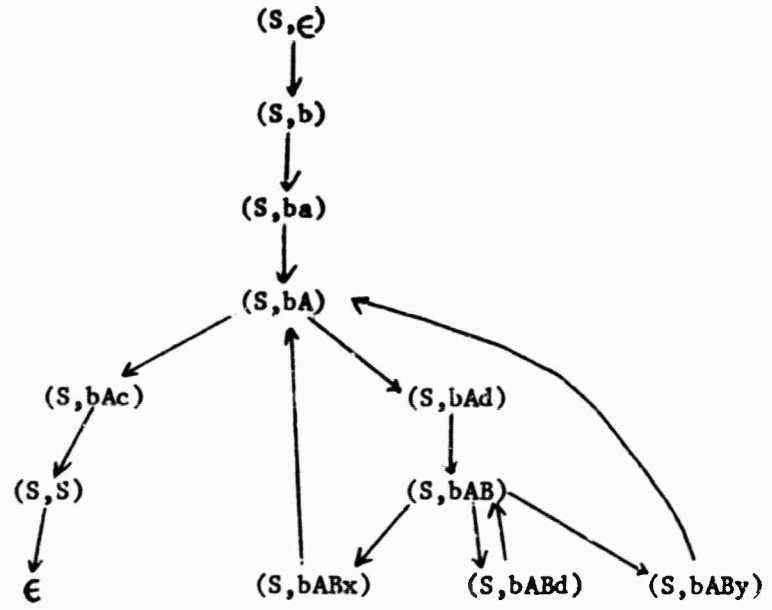


Figure 5.7

For absolute minimality of a minimal version, we eliminate all nonterminals other than (S,∈) and (S,bAB), resulting in an LL(3) grammar with two nonterminals and seven productions. However, if we additionally include (S,bA), the result is an LL(1) grammar with three nonterminals and six productions. ((S,bA) was chosen because of its proximity to a branching in the graph.) Renaming the nonterminals as S, A, and B, this grammar is S → baA, A → c, A → dB, B → dB, B → xA, B → yA.

In any event, since we have seen a 0-transformable grammar which is not LC(k) for any k, and since it is clear that a 0-transformable grammar is k-transformable for any k, we have the following:

Theorem 5.21 The class of k-transformable grammars strictly includes the class of LC(k) grammars.

Looking back at our proofs that LC(k) and LL(k) grammars are k-transformable, we see that the critical point in each case was that any state of the LR(k) machine for an LL(k) or LC(k) grammar can be split in such a way that the base of the splitting consists precisely of the essential items of the state being split. We might define a class of grammars which satisfy this property, namely that any state of the LR(k) machine for a grammar in the class can be split in such a way. Such a class would fall between the LC(k) and k-transformable grammars; in fact, this class has been proposed by Rosenkrantz, Lewis, and Stearns [15] under the jocular name of "fingernail grammars". Such grammars have the property that if a prefix of the right-hand side of some rule has been recognized, then k symbols of lookahead will identify what the next symbol of the rule must be; or when the rule is over, what its left-hand side is. They can be thought of as a generalization of LC(k) grammars, and the k-transformable grammars are a generalization of this class. This was the original genesis of the ideas in this thesis, which led in a roundabout way to the work reported herein.

5.2 Strategies for State-Splitting and Cycle Breaking

As we have stated a number of times, a possible application of the transformation we have developed would be in an early stage of an automatic compiler writing system, where a human-defined programming language grammar would be transformed into a version more suitable for actual use in a compiler. But the point of departure for the application of our transformation must always be some particular cycle-free MSP(k) machine for the original grammar. While we know that every cycle-free MSP(k) machine for a grammar can be obtained by starting with the LR(k) machine and performing some sequence of state splittings, such assurances are a little too vague to be of much practical value. We now turn our attention to the problem of how to split states in an intelligent manner in order to obtain a cycle-free machine, and specifically, how to get a cycle-free machine whose derived grammar exhibits a variety of desirable properties. The development of this section will be somewhat less formal than the foregoing; we shall highlight what seem to us to be the key issues of the problem. Our goal is to develop a feeling for good approaches to this area, which could be incorporated in a heuristic (rather than fully algorithmic) way into the hypothetical compiler-compiler we have mentioned.

As we saw in the previous section, every intermediate state of an MSP(k) machine for an LL(k) grammar G, can be split in such a way that the base of the splitting consists precisely of the essential items of the state being split. Then we saw now by starting with the LR(k) machine for G, and by blindly splitting states in this way, we could be assured of eventually ending up with a cycle-free MSP(k) machine. This is precisely the sort of magic which

we wish to avoid. We want to develop some deterministic strategies whereby the cycles of a machine may be deliberately destroyed, rather than accidentally as they were in the preceding section. So let us start at the beginning, and ask: how is it possible for the performance of a state-splitting to cause the elimination of a cycle in an MSP(k) machine, and how can we choose such a splitting?

Let us consider a simple stylized cycle, such as the one in Figure 5.8. Two states might occur in this configuration in an MSP(0) machine. How can we split either of these states to destroy this cycle? Since q_2 is the σ_1-successor of q_1, it means that the essential items of q_2 are obtained from some of the $.\sigma_1$ items of q_1, by moving the dot one place to the right.

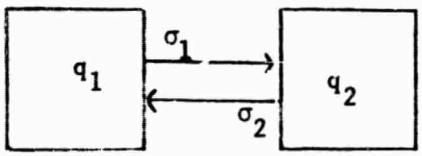


Figure 5.8

Now suppose we were able to split q_1 into a bipartite state-splitting, such that none of the $.\sigma_1$ items of q_1 were in the base state of the splitting. Then they would all have to be in the initial state, and there would be no σ_1-transitions out of the base state. However, the essential items of the base of this splitting would be the same as the essentials of q_1, so if we replaced q_1 by this splitting, the base would become the σ_2-successor of q_2. The picture that would result is given in Figure 5.9, and presto the cycle is gone.

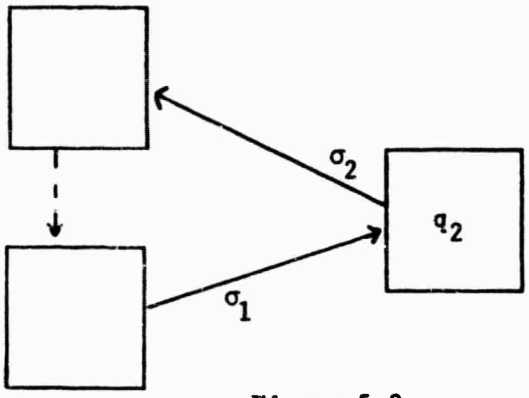


Figure 5.9

This process is known as breaking a cycle by performing a state-splitting. The concept involved is simple enough. If q_2 is the σ_1-successor of q_1 in a cycle, then the essential items of q_2 are derived from some of the items of q_1; if q_1 can be split so that these items do not appear in the base state of the splitting, q_2 will no longer be the successor of the state holding q_1's place in the machine, and the cycle is broken.

This concept extends immediately to larger cycles. If state q_2 is the σ-successor of q_1 and q_1 is the α-successor of q_2, then q_1 is its own $\sigma\alpha$-successor, and there is a cycle in the machine. If we split q_1 into a bipartite splitting with $.\sigma$ items only in the predictive state, then q_2 will be the σ-successor of the initial state, and the cycle no longer exists. Thus we have a conscious and deliberate method for eliminating a cycle by performing a state-splitting. We do not claim that the only way to destroy a cycle is to split one of its states in such a way, but it is by far the best and most direct way, and we shall concentrate our attention on it.

But is it necessary to have a bipartite splitting of q ? And is it necessary to eliminate all $.\sigma$ items from the base state of the splitting? The answer to both these questions is no, but there are some complications involved

ved. Let us consider the second one first. Obviously, we do not need to exclude from the base state those items of the form $A \to \alpha.\sigma\ (\tau)$, because they do not contribute to essential items of q_2. But even among the items of the form $A \to \alpha.\sigma\ \beta(\tau)$, some are more important than others, because there may be differences among the essential items of q_2.

Referring again to the simple cycle of Figure 5.8, we remark that every essential item of q_2 is obtained from some $.\sigma_1$-item of q_1 by moving the dot. But every item of q_1 is a descendant of some essential item of q_1; and every essential item of q_1 is obtained by moving the dot on some $.\sigma_2$-item of q_2. But now we have come full circle, because every item of q_2 is a descendant of some essential item of q_2. In other words, every essential item of q_2 causes some items (its descendants) to be in q_2; some of these items may cause the appearance of an essential item in q_1; and the process continues. That is, each essential item of q_1 causes the appearance of some essentials in q_2, and vice versa. This is the essence of there being a cycle consisting of these two states: the essential items of q_1 cause the essential items of q_2 to be in the σ_1-successor of q_1, while these same essential items of q_2 cause the essentials of q_1 to be those of the σ_2-successor of q_2. Thus the essential items of q_1 cause their own reappearance in the $\sigma_1\sigma_2$-successor of q_1, causing there to be a cycle. Let us consider an essential item I_1 of q_1 which causes the appearance of essential item I_2 in q_2, which in turn causes the appearance of I_1 in q_1. Then I_1 is a very dangerous item indeed; it is enough to start a cycle all by itself. That is, any state that has I_1 in it, will also have I_1 in its $\sigma_1\sigma_2$-successor; and since there are only finitely many items containing I_1, this means there is going to be a cycle

somewhere, unless there are some state-splittings along the way. Similar comments apply to the item I_2 of q_2. Such items we call seed items of the cycle, because they contain the seed of a cycle in and of themselves; if such an item is planted in any state, some of the successors of that state will be involved in a cycle. The seed items of q_2 are the truly critical essential items of q_2, which at all costs must not appear in the σ_1-successor of the base state of a splitting of q_1, if we desire to break the cycle by splitting q_1.

We can put this in other words as follows. If $A \rightarrow \alpha\sigma.\beta(\tau)$ is a seed item of q_2, then we call $A \rightarrow \alpha.\sigma\beta(\tau)$ a vital item of q_1. The goal of splitting q_1 is to replace q_1 by a splitting such that no vital items of q_1 are in the base of the splitting.

However, there may be essential items of q_2 which do not cause their own appearance in q_2; rather they just happen to pop up in q_2 because of some of the other essential items. The disposition and fate of these items is not so critical; no harm is done if they do appear in the σ_1-successor of the base of the splitting of q_1.

As an illustration of these concepts, consider the single state of Figure 5.10, which is a cycle all by itself. There are two essential items in this state. By our first ideas, in order to break this cycle it would be

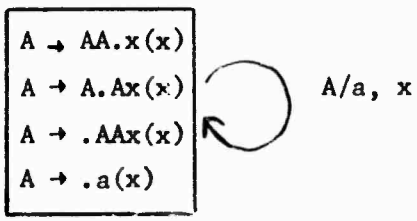


Figure 5.10

necessary to split this state so that neither A → A.Ax(x) nor A → .AAx(x) would appear in the base of the splitting, for these are the items from which the essential items are obtained. But that is manifestly impossible, since A → A.Ax(x) is an essential item and must be present in the base state of any splitting. Let us see then how our new concepts would aid us here.

The essential item A → A.Ax(x) is a seed item of this cycle, since A → .AAx(x) is its descendant, and A → A.Ax(x) can be obtained from this latter item by moving the dot one place. However the item A → AA.x(x) is not a seed item, since it is not obtained from any of its descendants, but rather from the other essential item (by moving the dot). Thus A → .AAx(x) is the only vital item, the only item that must not appear in the base state of the splitting for us to successfully break this cycle. This can be arranged by a simple bipartite state-splitting of the state; the result of doing this splitting is shown in Figure 5.11. And indeed there no longer is a cycle on A.

We do note that the base state is not the A-successor of the predictive state, but its AA-successor, even though the original state was its own A-successor. This occurs because not all of the essential items were seed items.

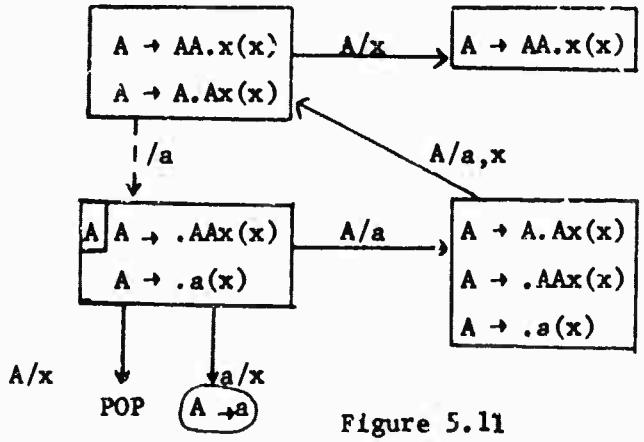


Figure 5.11

The computation of seed nodes can sometimes be a little more subtle than the foregoing indicates. For example, in the state of Figure 5.12, both essential items are seed items, though it may initially seem more as though neither one is. For example, $B \rightarrow .aAy$ is a descendant of $A \rightarrow a.Bx$; $B \rightarrow a.Ay$ can be obtained from $B \rightarrow .aAy$ by moving the dot; $A \rightarrow .aBx$ is a descendant of

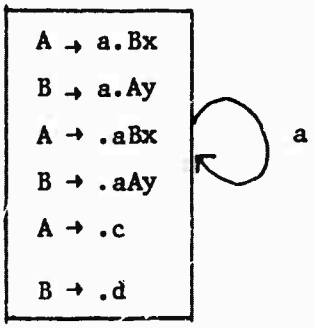


Figure 5.12

$B \rightarrow a.Ay$; and we finally get back to the essential item we started with by moving the dot on $A \rightarrow .aBx$.

We now define these notions formally in order to demonstrate some useful facts.

Definition 5.22 Item I' is an ϵ-derivative of item I if I' is a descendant of I; I' is a σ-derivative of I if I is $A \rightarrow \alpha_1 . \sigma\alpha_2(\tau)$ and I' is $A \rightarrow \alpha_1 \sigma . \alpha_2(\tau)$. I' is an α-derivative of I if there is a sequence of items $I = I_0, I_1, \ldots, I_n = I'$, such that I_{i+1} is a σ_i-derivative of I_i, and $\alpha = \sigma_0\ \sigma_1 \cdots \sigma_{n-1}$. The α-derivative of a set of items is the set of α-derivatives of all items in the set.

The following is a restatement of the way successors are computed in LR(k) machines.

Lemma 5.23 If q' is the non-final α-sucessor of q in the LR(k) machine M, then the items of q' are the set of α-derivatives of the items of q.

The situation is not so simple in a more general MSP(k) machine, because there may be split states, and the α-derivatives of an earlier state may be scattered. However we do know the following.

Lemma 5.24 If q' is a non-final α-successor of q in an MSP(k) machine, then the items of q' are a subset of the α-derivative of the items of q.

In particular, this means that every essential item of q' is an α-derivative of some essential item of q.

Now suppose we had a cycle in an MSP(k) machine, where q is some state in the cycle, and q is its own α-successor. Then every essential item of q is an α-derivative of some essential item of q.

Definition 5.25 Let q be a state of an MSP(k) machine such that q is its own α-successor. If an essential item of q is its own α^n-derivative, for some n, then that item is a seed item of q with respect to the cycle on α.

Lemma 5.26 If q is its own α-successor in an MSP(k) machine, then there exist some seed nodes of q with respect to the cycle on α, and every essential item of q is an α^i-derivative of some seed node, for some i.

Proof Pick any essential item I_0 of q. Then I_0 is an α-derivative of some essential item I_1 of q; similarly, I_1 is an α-derivative of some I_2, and so on. Thus we get a sequence I_0, I_1, I_2, ... such that I_i is an α-derivative of I_{i+1}, and all are essential items of q. Since there are only finitely many essential items of q, there must be some repetition in this sequence.

So for some I_i and I_{i+j}, we have $I_i = I_{i+j}$. Then I_i is its own α^j-derivative, and so is a seed item. And also I_0, which was arbitrary, is an α^i-derivative of I_i. Q.E.D.

Lemma 5.27 If $B \rightarrow \beta_1 . \beta_2(\omega)$ is an α-derivative of $A \rightarrow \alpha_1 . \alpha_2(\tau)$, then $\alpha_2 \stackrel{*}{\Rightarrow} \alpha\beta_2\Psi$ for some Ψ.

Proof We proceed by induction on the length of α. If $|\alpha| = 0$, then $B \rightarrow . \beta_2(\omega)$ is a descendant of $A \rightarrow \alpha_1 . \alpha_2(\tau)$, and the result is immediate. Suppose the statement is true for $|\alpha| = n$; then consider the case where $|\alpha| = n+1$. Suppose $B \rightarrow \beta_1 . \beta_2(\omega)$ is an essential item. Then $\beta_1 = \beta_3\sigma$, where $\alpha = \varphi\sigma$. Then $B \rightarrow \beta_3 . \sigma\beta_2(\omega)$ is a φ-derivative of $A \rightarrow \alpha_1 . \alpha_2(\tau)$, where $|\varphi| = n$. Therefore $\alpha_2 \stackrel{*}{\Rightarrow} \varphi\sigma\beta_2\Psi$ by induction; since $\alpha = \varphi\sigma$, this gives us $\alpha_2 \stackrel{*}{\Rightarrow} \varphi\beta_2\Psi$ as required If $B \rightarrow . \beta_2(\omega)$ is not essential, then it is the descendant of some essential item which in an α-derivative of $A \rightarrow \alpha_1 . \alpha_2(\tau)$, and the result follows immediately. Q.E.D.

Theorem 5.28 If $A \rightarrow \alpha_1 . \alpha_2(\tau)$ is a seed item of q with respect to the cycle on α, then $\alpha_2 \stackrel{*}{\Rightarrow} (\alpha)^i \alpha_2\Psi$ for some Ψ and i.

Proof If $A \rightarrow \alpha_1 . \alpha_2(\tau)$ is a seed item, then it is its own α^i-derivative for some i. The result then follows from the preceding lemma. Q.E.D.

We shall find this result very useful in determining good strategies for breaking a cycle by state-splitting.

Definition 5.29 Suppose q' is a σ-successor of q and q is an α-successor of q', in the MSP(k) machine M. If $A \rightarrow \alpha_1\sigma . \alpha_2(\tau)$ is a seed item of q' with re-

spect to the cycle on $\alpha\sigma$, then $A \rightarrow \alpha_1 . \sigma\alpha_2(\tau)$ is a <u>vital item</u> of q with respect to the cycle.

With these ideas in hand, we proceed to consider the issues of state-splittings which keep vital items out of the base state.

To return to a question we asked earlier, is it necessary to perform a bipartite splitting of a state in a cycle in order to break the cycle? Intuitively, we would guess that the answer should be no, that the only important issue is to somehow keep the vital items of the state out of the base of the splitting, with the number of initial states being irrelevant. This is the case, but this fact is not very important, since in any conceivable case a bipartite splitting will suffice. Or put another way, under any realistic circumstances, there can be no worthwhile multi-partite splitting. Suppose q is its own $\alpha\alpha$-successor, and suppose there is a splitting of q with vital items $A_1 \rightarrow . \alpha\alpha_1(\tau_1)$ and $A_2 \rightarrow . \sigma\alpha_2(\tau_2)$ in different initial states. Then by the laws of legal state-splitting, $FIRST_k(\sigma\alpha_1\tau_1) \cap FIRST_k(\sigma\alpha_2\tau_2) = \phi$. But by the definition of vital item, $A_1 \rightarrow \sigma.\alpha_1(\tau_1)$ and $A_2 \rightarrow \sigma.\alpha_2(\tau_2)$ will be seed items; therefore, $\alpha_1 \overset{*}{\Rightarrow} (\alpha\sigma)^i \alpha_1\Psi_1$ and $\alpha_2 \overset{*}{\Rightarrow} (\alpha\sigma)^j \alpha_2\Psi_2$ for some i, j, Ψ_1, and Ψ_2, by Theorem 5.28. Therefore, for any m at all, $\alpha_1 \overset{*}{\Rightarrow} (\alpha\sigma)^m \varphi_1$ and $\alpha_2 \overset{*}{\Rightarrow} (\alpha\sigma)^m \varphi_2$. Thus unless the only string $\alpha\sigma$ generates is ϵ, $FIRST_k(\sigma\alpha_1)$ will intersect with $FIRST_k(\sigma\alpha_2)$. It is safe to say that in any practical situation the label of a cycle in an MSP(k) machine will not be a sequence of symbols that generates only ϵ. So in any interesting situation, there can not be any multi-partite splitting of a state such that vital items occur in different initial states of the splitting; thus there seems to be no practical use for multi-partite splittings, and we can restrict our attention to bi-

partite splittings for the time being.

It is not a difficult matter to compute the seed items of a state with respect to a given cycle, and hence we can obtain the vital items of any state in a cycle. But such a computation will frequently be unnecessary. The vital items of a state in a cycle form the minimum set of items which must go into the predictive state of a splitting in order to destroy the cycle. If q is its own σ-successor, then the vital items of q are all σ-items. It would be reasonable to first attempt to find a splitting of q such that all $.\sigma$-items are in the predictive state; if no such splitting can be found, then we could actually compute precisely which of the $.\sigma$-items are vital, and try to find some splitting such that at least these items are in the predictive state. Very often, if there is a splitting which puts the vital items in the initial state, then it also places the other $.\sigma$-items there as well. This will be the case whenever σ generates a string longer than k, as is easy to demonstrate.

How do we go about finding a useful splitting of a state in a cycle? First of all, we determine the vital items of the cycle in that state (or whatever set of items we want to keep out of the base state). Then we want to find a splitting of the state that leaves all these items out of the base of the splitting. We recall that a splitting is defined by the functions H_1 and H_2 on the simple chains through the state; so our next step is to form all these chains. We isolate those simple chains that contain a vital item -- we call these vital chains. We must find a splitting which breaks every vital chain "above" all the vital items in it, and breaks the other simple chains in any way that makes the splitting legal. As we have seen, for use-

ful cases we need concern ourselves only with bipartite splitting, so the only candidates for the predictive nonterminal of the initial state of the splitting are those that appear on the left-hand side of an item above every vital item in any vital chain. We could select each possible candidate, try all its state-splittings, and see if any of them are both legal and put all vital items into the predictive state. However, we can restrict this inefficient procedure and give it some guidance in the following way.

For any vital item I, consider $FIRST_k(I)$. We know that, in order for I to be in the predictive state and not in the base state, any lookahead from $FIRST_k(.)$ will have to occasion the prediction to be made. So the union of $FIRST_k(I)$, over all vital items I, will have to be contained in the predictive language that occasions the prediction to be made. We can define the language of a chain as being $FIRST_k$ of the terminal item at the end of a chain. We then perform the following procedure:

1) Compute the language of every vital chain.

2) Designate any other chain whose language intersects the language of some vital chain as being vital as well.

3) Iterate this process until no new vital chains are found.

Then we have a class of vital chains, some of which contain vital items and some which do not.

The reason for performing this computation is simple. It is possible to compute the languages of the vital items of a state by taking the union of the languages of all chains in which some vital item appears. It is essential that a prediction be made whenever one of these lookaheads is espied upon entry to the base state, if the vital items are to be in the predictive state. However, if some string, which might be a lookahead of a vital item, is also in the

language of some other chain, then for whatever nonterminal A is being predicted, there had better be a .A item in that non-vital chain as well. So all strings in the language of that other chain will occasion a prediction to be made, so that chain may as well be considered vital; and the computation continues. The result is a set of chains such that any string in the language of any of these chains will cause a prediction to be made. Thus the first step in computing a splitting is to find some nonterminal A such that there is a .A item in each of these chains, above any vital items that may be in a chain. The plan is to find a distinguished set of .A-items; the A's in these items will be the predicted nonterminal. The idea is that these distinguished .A-items will serve to define H_1 and H_2 on the vital chains. That is, H_1 of a chain will be all of the chain from the top through the selected .A-item, while H_2 will be the rest of the chain.

Locating a nonterminal A which has some .A-items in the appropriate places in the vital chains, is not by itself the end of the story. First of all, if A is left recursive there may be several .A-items to choose from in some vital chains. The choice of certain of these items as the place to break the chain may violate the condition of disjointness of the follow sets of A in the base and predictive states. Clearly, caution must be exercised in such a case. Furthermore, even if A is not left recursive, the chosen .A-items may occur in other, non-vital chains. Then the strings in the languages of these chains will also have to be in the predictive language; therefore any chain which has one of these strings in its language will have to contain one of the distinguished .A-items. The functions H_1 and H_2 will be defined on these chains in the usual way: generally, it is best to break a chain at the lowest pos-

sible .A-item, so as to minimize the possibilities of a follow set conflict.

As an example of how this procedure is applied, consider the LR(2) state of Figure 5.13; this state is its own a-successor. Inspection shows us that the only seed item of this state is the item A → a.B(bb), and so the only

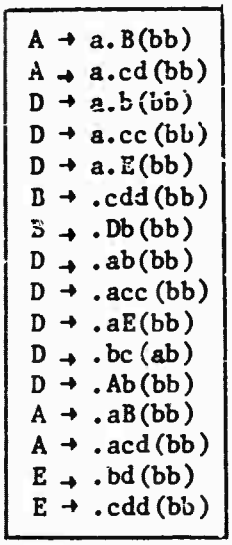


Figure 5.13

vital item is A → .aB(bb); we want to find a bipartite splitting of this state so that A → .aB(bb) is not in the base state.

The chains through this state are given in Figure 5.14. Each chain is organized vertically and is numbered for convenience; the language of each chain is given beneath it in braces.

1	2	3
A → a.cd(bb) {cd}	D → a.b(bb) {bb}	D → a.cc(bb) {cc}
4	**5**	**6**
D → a.E(bb) E → .bd(bb) {bd}	D → a.E(bb) E → .cdd(bb) {cd}	A → a.B(bb) B → .cdd(bb) {cd}
7	**8**	**9**
A → a.B(bb) B → .Db(bb) D → .ab(bb) {ab}	A → a.B(bb) B → .Db(bb) D → .acc(bb) {ac}	A → a.B(bb) B → .Db(bb) D → .aE(bb) {ab, ac}
10	**11**	**12**
A → a.B(bb) B → .Db(bb) D → .Ab(bb) A → .aB(bb) {aa, ac}	A → a.B(bb) B → .Db(bb) D → .Ab(bb) A → .acd(bb) {ac}	A → a.B(bb) B → .Db(bb) D → .bc(bb) {bc}

Figure 5.14

As we have said, A → .aB(bb) is the only vital item; since the only chain in which it appears is chain 10, chain 10 is the only vital chain we begin with. Thus the first version of the predictive language is {aa, ac}. The string ac is in the language of chains 8, 9, and 11 as well, so they too are vital chains. Furthermore, since ab is in the language of chain 9, it is also part of the predictive language, and this makes chain 7 into a vital chain. Therefore chains 7-11 are vital; and we must find some σ such that there are .σ items in each of these chains, which is above the vital item in chain 10. By inspection, we see that there are two possibilities for such a σ, namely B and D. The item A → a.B(bb) is

in each of these chains, as is the item B → .Db(bb). However, there is one more step to check; namely, we must determine whether the other lookaheads of these items can safely occasion their prediction. For the case $\sigma = B$, this fails to be the case. The item A → a.B(bb) appears in chain 6, and cd is in the language of chain 6; but cd is also in the languages of chains 1 and 5, and A → a.B(bb) appears in neither of them. Things work out better for predicting a D. The only non-vital chain in which B → .Db(bb) appears is chain 12; the language of chain 12 is {bc}, and bc occurs in the language of no other chain. Hence the D in B → .Db(bb) can always be safely predicted upon sighting one of its lookaheads. The splitting induced by this prediction is given in Figure 5.15; the cycle has indeed been broken, and the base state is the a-successor of the predictive state.

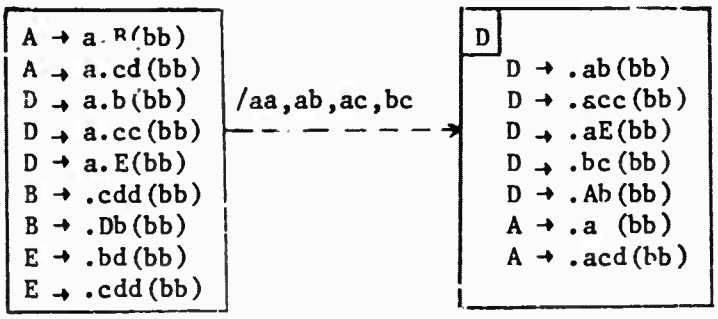


Figure 5.15

Of course there are any number of places where this procedure we have been describing may fail. First of all, there may be no predictable nonterminal which induces a splitting in which all vital items are excluded from the base of the splitting. That is, there might not be any nonterminal A, such that there is a .A-item in the appropriate place in each vital chain. Then even if there is such an A, the wrong choice of distinguished .A-items may have been

made. That is, if A is left recursive, $FOLLOW_k(B, A) \cap FOLLOW_k(P, A)$ may not be empty, if B and P are defined as the choice of .A-items would dictate. This might necessitate going back and choosing other distinguished .A-items to define H_1 and H_2, and seeing whether or not any problems result. It might be necessary to increase the number of distinguished .A-items over those that are absolutely necessary to keep the vital items out, in order to avoid disjointness problems. But these are fine tuning issues with respect to the choice of a particular A as the predictive nonterminal. It always suffices just to compute all possible legal bipartite splittings where A is the predictive nonterminal, and see whether any of them have the vital items in the predictive state. Of course, there may be no such legal splitting based on an A, so we might have to backtrack to find another suitable predictive nonterminal and some splitting based on it. And in fact there may not be any splitting at all of the state in question which has the desired structure; exhaustion of the list of all possible predictive nonterminals might show that no splitting of this state could break the cycle. Then we could attempt to break the cycle by splitting some other of its states, and apply this procedure all over again to that state.

The foregoing discussion and algorithms provide us with a structured approach to the problem of breaking a particular cycle in an MSP(k) machine by splitting one of its states. Now we have made no formal claim that if there is some way to eliminate a cycle from an MSP(k) machine, then the foregoing procedure will succeed and find a state in the cycle which can be split in a bipartite splitting, with all vital items in the predictive state. The possibilities for the effect of a sequence of arbitrary state-splittings

on a machine are too bizarre to allow any such wide-sweeping generalization. That is, it may happen that by splitting some other state which dominates all the states in the cycle, to eliminate from the machine the exact states which form the cycle in question. But we do not really need such a general assurance, for we are not looking for a decision procedure; we already know that the class of k-transformable grammars is decidable. Rather we are interested in heuristics, in good procedures that work very often, and especially in most useful cases.

The obvious next step then, is to go from the problem of eliminating any single cycle from an MSP(k) machine, to eliminating all the cycles from an MSP(k) machine. We are especially interested in one particular version of the latter task, namely where the MSP(k) machine is the LR(k) machine for the grammar we start out with. That, after all, is what we're really trying to do: eliminate all the cycles from the LR(k) machine.

In the foregoing, we have examined the problem of eliminating a single cycle from an MSP(k) machine by splitting one of the states in the cycle. Can we similarly propose strategies for removing a number of cycles from a machine by sequentially splitting a number of states? Is there more to breaking a number of cycles than just breaking each one individually?

In general, the answer to this last question is no, there is not any more to eliminating a set of cycles than just eliminating each one of them; but there are a few possible complications which we shall discuss briefly.

First of all, we know that by splitting a state in a cycle so that all of its vital items with respect to that cycle are to be found only in the predictive state(s) of the splitting, we do indeed remove that particular cycle

from the machine. That is, if q is its own α-successor, and we replace q by a bipartite splitting with all the vital items in the predictive state P, then the base state B will be the α^i-successor of P, and the cycle is gone. But is it possible for other cycles in the machine to be affected by this state-splitting in such a way that although they may have formerly been eliminable, they no longer are? Or can new cycles be introduced into the machine as a result of the state-splitting, new cycles which cannot be removed by state-splitting?

Offhand, it appears that the first of these questions is not a very sensible one. It seems that splitting a state of one cycle can have only very limited effects on any other cycle. Either some state of the other cycle is dominated by the state being split and disappears after it is split, causing the other cycle to vanish along with it; or else the other cycle will not be effected by the state-splitting. However, upon reflection we see that there is one circumstance in which breaking one cycle by splitting one of its states can adversely affect the prospects for breaking another cycle: and that is where the state being split to break one cycle is also a state of the other cycle. Then it might be that by splitting the state in question so as to break the first cycle, the vital items of the other cycle will be put in the base state of the splitting; and if no other states of this second cycle can be broken in a useful way, then there will be no way to break the cycle, even though some other splitting of the state in question might have broken the other cycle.

For example, consider the LR(0) states of Figure 5.16; there are two cycles in this diagram, one on a and the other on Aa. If we split the state

on the left by predicting an A, we can break one of these cycles as shown

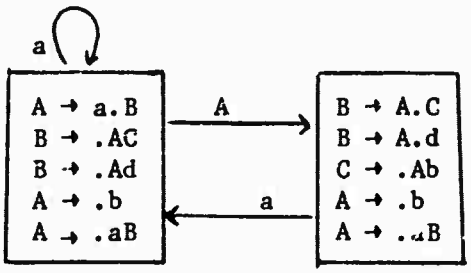


Figure 5.16

in Figure 5.17. But as we see from inspection of the result of this state-splitting, it is now impossible to break the cycle on Aa, because one of the

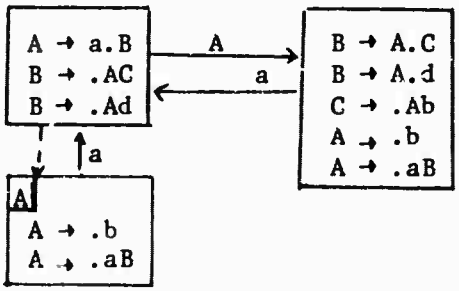


Figure 5.17

states in the cycle is a base state which cannot be split; while there are no nonterminals that can be predicted in the other state, and hence that can't be split either. However, had we made the original split by predicting B rather than an A, we would have the configuration of Figure 5.18, in which both cycles have been broken. The moral of this story is that in splitting a state through which several cycles pass, we must exercise caution in choosing the splitting, so as to break all cycles if possible. Furthermore, it may happen that while two cycles in a given machine can each individually be

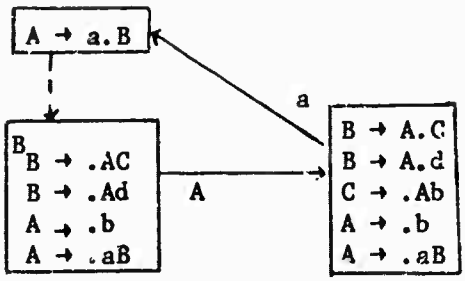


Figure 5.18

broken by splitting the same state of the machine, two different splittings are used to break the two different cycles. But it is not hard to see that except in the most contrived circumstances, if there is one bipartite splitting of q which excludes all of one set of vital items from the base, and another bipartite splitting of q which excludes all of another set of vital items from its base, then there is some splitting of q (possibly multi-partite) which excludes both sets of vital items from the base. (This is one case where bipartite splittings may not suffice and the general theory is needed.) Thus we do have one caveat to heed while proceeding to break a set of cycles by splitting a state from each, one at a time: namely, to be on the lookout for states which can be split so as to break several cycles. Pursuance of this course will also help us avoid a potential pitfall. It might happen, if we are not careful, that all the states in some cycle are split (so as to break other cycles), but the cycle is left intact, consisting of the base states of these splittings. Caution with respect to the issues just discussed can forestall this occurrence.

The question of the introduction of new cycles as the result of performing a state-splitting seems to be a serious difficulty, but a little reflection shows it not to be a dangerous problem. First of all, suppose that a splitting obeys the following property: if some .σ-item is in a component of the splitting, then all .σ-items are. If a state in a machine is

replaced by such a splitting, then no new states (other than the components themselves) are introduced into the machine as a result of the splitting; the successors of the components are the same as the successors of the state they are replacing. A great many state-splittings do obey this condition; since replacing a state by such a splitting does not add any new states to the machine, it cannot cause any new cycles either. But even if this happy state of affairs does not obtain, and we use a less benign splitting, the situation of the machine will not be degraded by performing the state-splitting. To be more precise, we make the following claim: suppose M' is obtained from M by splitting some state of M; then if every cycle of M can be broken, so can every cycle of M' be broken. In other words, though the performance of a state-splitting may introduce new cycles into a machine, these cycles will not be of a new and difficult character, and they by themselves will not prevent us from realizing our goal of a cycle-free machine; one of the new cycles will be impossible to remove only if some old cycle was also unbreakable.

The argument for our claim is as follows. Suppose there is a new unbreakable cycle in the machine after a state-splitting has been done. Let q be some state of this cycle, where q is its own α-successor. Then q is a substate of some state q', which was in the machine before the splitting was done. For any φ, the φ-successor of q in its cycle is contained in some φ-successor of q'. In particular, q is contained in some α^i-successor of q', for each i. This means that some α^i-successor of q' is its own α^j-successor. Thus there is some cycle of successors of q', where each state in this cycle contains some state in the cycle in which q is involved. If some state in

this former cycle could be split so as to break the cycle, the same would be true for the state it contains in the latter cycle. So if the latter cycle is unbreakable, so is the former, and our claim is established. It is easy to formalize the foregoing discussion and make this assertion rigorous. The upshot of this argument is that no disasters will occur as the result of a state-splitting: no new unbreakable cycles can crop up to haunt us.

Thus a simplistic approach to the problem of ridding a machine of all its loops turns out to be relatively attractive. In summary, the strategy is to pick any cycle and split one of its states so that no vital items are in the base of the splitting; this will break that particular cycle. Then we pick some other cycle in the new machine and apply this process again; and continue iterating this procedure as long as necessary -- always watching out for states that can break two cycles at the same time. If the result of this process is a cycle-free machine, then we can apply the grammatical transformation of Chapter 4 and construct the desired LL(k) grammar. We stress that this approach will not necessarily work for the full class of k-transformable grammars, but that it will be useful for any reasonable grammar that is not laden with a variety of peculiar features. If this systematic approach does not succeed in constructing a cycle-free machine for a given grammar, we can always resort to the haphazard procedure of constructing all possible MSP(k) machines for the grammar and seeing if any one of them is cycle-free.

In particular, there is one large class of grammars, a subset of the k-transformable grammars, for which these techniques will work, and the outstanding feature of this class is that its decision procedure is much better than the general one for k-transformable grammars. We can define this class as those grammars whose LR(k) machines satisfy the following property: there

is a set of states of the machine and a splitting of each state, in the set such that for any cycle in the machine, one of these states is in the cycle, and the splitting of that state breaks the cycle. (This is almost the same as the simpler statement that each cycle in the machine can be broken by splitting one of its states. There is a difference of quantification between the two statements, because in the second statement the same state might be split in different ways in order to break different cycles, while this cannot be the case in the more precise definition we are using.) By our previous discussion, it will be possible to construct a cycle-free MSP(k) machine for one of these grammars. The construction procedure will consist of breaking any cycle in the machine by splitting the appropriate state, and computing the resultant machine. Any cycle in this new machine will be breakable because any cycle in the original machine was; the splittings of states in the original machine can serve as models for splitting states in the new machine. So we choose some cycle and break it by a state-splitting as before. Again, any cycle in this resulting machine will be breakable, and so we can continue on in this way, until we have eliminated all the cycles and are left with a cycle-free machine. Though tedious to do, all of this can be made rigorous.

It is apparent that it is relatively straightforward to determine if a grammar satisfies the above condition; it is only necessary to see if each cycle in the LR(k) machine can be broken by a state-splitting. This is at least a deterministic problem which we can attack in a structured way, as opposed to the random decision procedure we have for the general class of k-transformable grammars. Furthermore, this test is one that can be applied directly to the LR(k) machine for the grammar; we do not need to construct

a whole series of auxiliary machines just to determine if the grammar interests us or not.

With the preceding discussion we have achieved at least a limited version of our goal of rationalizing the process of constructing cycle-free machines and bringing the transformation procedure into the realm of practicality. Now we wish to provide some additional suggestions, some second-order heuristics, to guide the program or person going through the procedure of eliminating the cycles from an $\mathcal{R}(k)$ machine by repeated state-splittings. These observations are intended to assist him in producing a cycle-free machine which is attractive when compared with other cycle-free machines that he might otherwise create, with respect to features that influence the character of the derived grammar.

First of all, we know that the size of the derived grammar is heavily influenced by the number of predictive states in the machine from which it is derived. So an effort should be made to keep down the number of state splittings performed and the number of predictive states created. Whenever possible, a bipartite splitting is to be preferred to a multipartite one; and it becomes even more attractive to break several cycles with one splitting, whenever such a feat is possible. Furthermore, it is recommended to try to introduce as few as possible new states into the machine as a result of a state-splitting; this can be best effected by choosing a splitting (whenever possible) which has all $.\sigma$ items in the same component of the splitting, for each σ. If this is done, the successors of the components will be states already in the machine. This minimization of new states is a desirable goal because it forestalls the creation of new cycles, which, though related to

existing cycles and hence breakable, will nonetheless require additional state-splittings to accomplish this breaking. In general, if there are several bipartite splittings of a given state that will suffice to break a cycle in which that state occurs, the preferred splitting is that which places the fewest items from the state in the base of the splitting. Pursuance of this strategy, particularly in conjunction with the preceding recommendations, will have a beneficial effect on the size of the grammar derived from the constructed cycle-free machine. The reasoning behind this is as follows. Suppose q is a state which is its own α-successor, and which can be split in two ways, each of which puts all the vital items into the predictive state; but suppose in the first splitting only, all $.\sigma$ items are in the predictive state. Now imagine two cycle-free machines, derived from the original machine, one in which q has been replaced by the first splitting and the other in which it has been replaced by the second. Let us then consider the grammars derived from these two machines. Suppose that the base state of the splitting has the name (X, φ) in each machine. If the names of the predicted nonterminals of the two splittings are Y_1 and Y_2, then since the base will be the α-successor of the predictive in either machine, the base will also have the name (Y_1, α) in the first machine and (Y_2, α) in the second. In the first machine, the base state of the splitting has $.\sigma$ items, so it has a σ-successor; this successor will have two names, $(X, \varphi\sigma)$ and $(Y_1, \alpha\sigma)$. In the second machine however, the predictive state has the $.\sigma$-items, and a σ-successor; its only name will be (Y_2, σ). Thus in the former case the σ-transition contributes two nonterminals to the derived grammar, while in the latter it gives rise to only one.

For example, consider the LR(0) grammar S → A, A → aB, A → b, B → Ab. The LR(0) machine for this grammar is shown in Figure 5.19, and has one cycle

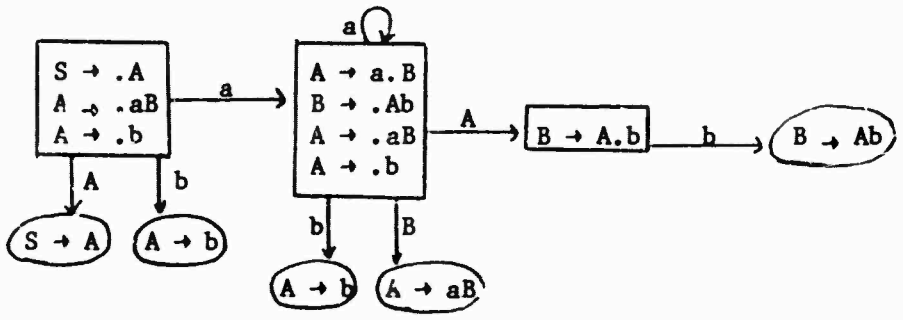


Figure 5.19

in it. The cycle can be broken either by predicting a B or by predicting an A in the state of the cycle; in either case the vital item A → .aB is not in the base state. The two resulting machines are shown in Figure 5.20 and Figure 5.22. According to our argument, the second of these machines is to be preferred, because it has the smaller base state.

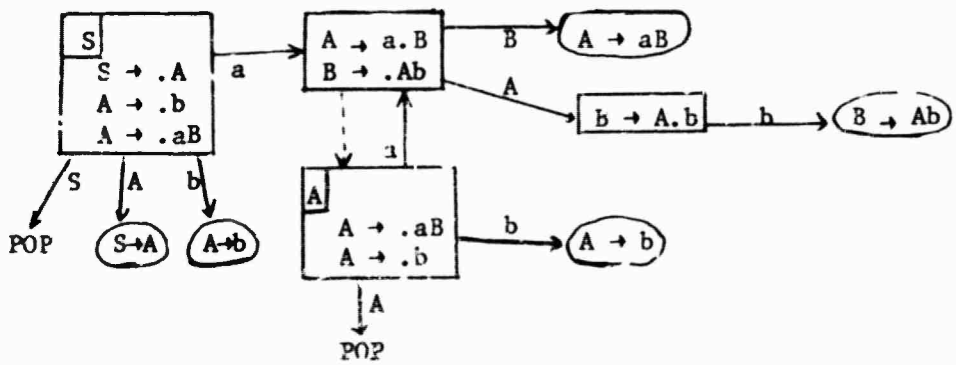


Figure 5.20

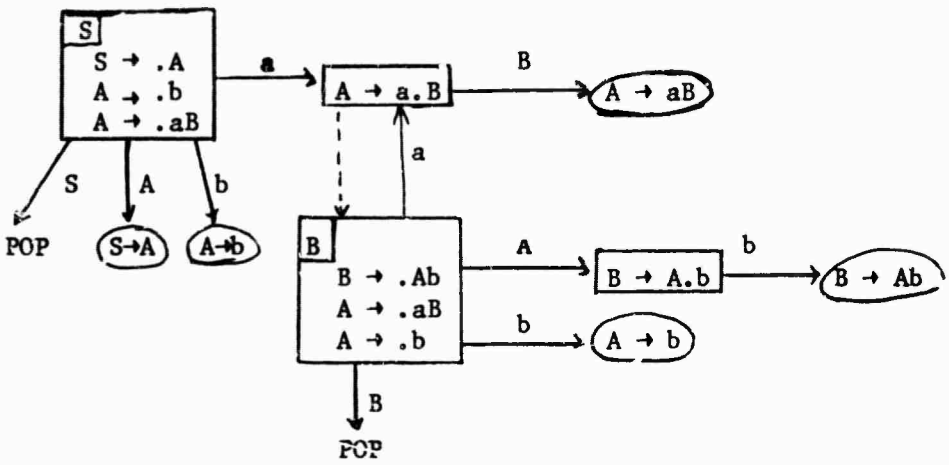


Figure 5.21

To check this we compute the derived grammars for these two machines in Figure 5.22.

(S, ∈) → a(S, a)
(S, ∈) → b(S, b)
(S, b) → (S, A)
(S, a) → (A, ∈)(S, aA)
(S, aA) → b(S, aAb)
(S, aAb) → (S, aB)
(S, aB) → (S, A)
(S, A) → (S, S)
(S, S) → ∈
(A, ∈) → a(A, a)
(A, ∈) → b(A, b)
(A, a) → (A, ∈)(A, aA)
(A, aA) → b(A, aAb)
(A, aAb) → (A, aB)
(A, aB) → (A, A)
(A, b) → (A, A)
(A, A) → ∈

(S, ∈) → a(S, a)
(S, ∈) → b(S, b)
(S, b) → (S, A)
(S, a) → (B, ∈)(S, aB)
(S, aB) → (S, A)
(S, A) → (S, S)
(S, S) → ∈
(B, ∈) → a(B, a)
(B, ∈) → b(B, b)
(B, a) → (B, ∈)(B, aB)
(B, aB) → (B, A)
(B, A) → b(B, Ab)
(B, Ab) → (B, B)
(B, b) → (B, A)
(B, B) → ∈

Figure 5.22

Inspection of the e two grammars verifies that the second machine is preferable to the first. The first derived grammar has 15 nonterminals and

17 rules, while the second has 13 nonterminals and 15 rules. Not a drastic improvement perhaps, but suggestive. We could have predicted this difference by examining the machines. The first machine has two more states as successors of the base state than the second machine; these states thus have one extra name each in the first machine, meaning two more nonterminals in the first grammar than in the second.

To conclude this section, we make some remarks about some possible generalizations of our model of MSP(k) machines. As we have defined and developed the theory, an MSP(k) machine is basically a canonical LR(k) machine that is allowed to make predictions based on the inspection of k symbols of lookahead. We put a number of restrictions on our model that were not critical, but which just made the formalisms more tractable; now we shall remove some of these conditions. We shall not redo the entire development for these more general machines; it will be fairly clear that no great dislocations occur under the proposed modifications. We include these revised models here because they will frequently be of practical utility; it will often be the case that a smaller cycle-free machine can be found if these restrictions are lifted than if they are not.

First, we recall the more or less arbitrary restriction that any initial state of an MSP(k) machine could be associated with only one base state of the machine. This was done so as to ease the proof that the derived grammar generates the same language as that recognized by the machine. However, it is possible to relax this restriction in some circumstances, without invalidating the result. Namely, if two identical initial states are associated with two different base states and the predictive languages associated with the two predictions are also identical, then the two initial states can

really be considered as one. This is a most valuable feature for it can significantly reduce the number of initial states in a machine and enhance the size of the derived grammar. The relaxation of this restriction obviously affects our strategies for state-splitting; it now becomes most desirable to split two states in such a way that the splittings share a common initial state.

For example, consider the states of Figure 5.23, which are involved in three cycles. It is possible to break these cycles in several different ways,

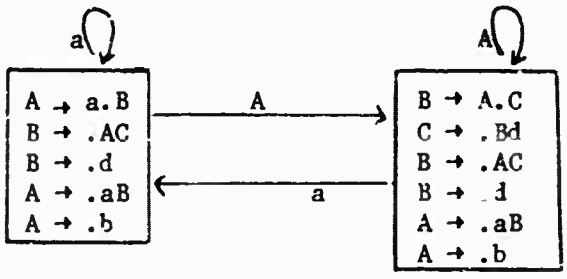


Figure 5.23

including predicting B in the first state and C in the second, which would follow our earlier stated goal of keeping as little as possible in the base states. However, it is also possible to break these cycles by predicting a B in both states, and sharing the predictive state between the two splittings, as shown in Figure 5.24.

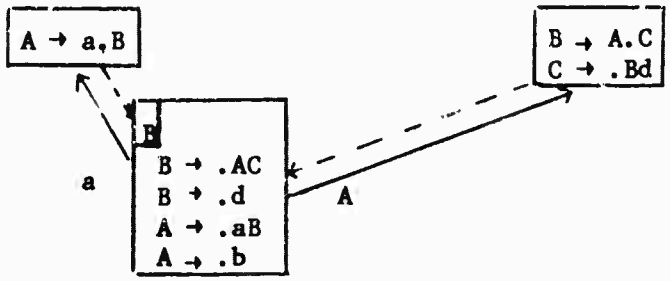


Figure 5.24

This machine section thus has only one predictive state rather than two, which has a beneficial effect on the derived grammar. Let (X, φ) be any name for the leftmost base state in the machine at large. Then the machine section of Figure 5.24 will give rise to the rules of Figure 5.25.

(X, φ) → (B, ϵ)(X, φB)
(B, ϵ) → a(B, a)
(B, ϵ) → b(B, b)
(B, ϵ) → d(B, d)
(B, d) → (B, B)
(B, b) → (B, A)
(B, a) → (B, ϵ)(B, aB)
(B, aB) → (B, A)
(B, A) → (B, ϵ)(B, AB)
(B, AB) → d(B, ABd)
(B, ABd) → (B, AC)
(B, AC) → (B, B)
(B, B) → ϵ

Figure 5.25

This compares very favorably with the grammar that would result if we split one of the states by predicting B and the other by predicting C. In that case, this section of the machine would generate 18 nonterminals and 20 rules, for each name (X, φ) of the first base state. This illustrates why the process of choosing a good set of state-splittings is partly an intuitive and heuristic procedure, since there are potentially conflicting standards for what de-

fines a good splitting. On the one hand, we like to keep as little as possible in the base state; on the other, we like it when two splittings can share a predictive state. Anyone trying to construct a "good" cycle-free machine by performing a sequence of state-splittings, must keep these and other criteria in mind, and strive to strike a balance among them.

Another restriction we made very early on was that only lookaheads of length k could be inspected in order to make predictions in machines whose states were composed of LR(k) items. This was to ensure that every item generated some lookahead strings of the appropriate length; that made all the definitions very convenient and easy to state. But there is a drawback to this simplifying assumption. In general, if $k_1 > k_2$, then there are many more LR(k_1) items for a given grammar than there are LR(k_2) items. So frequently the LR(k_1) machine for a grammar will be much larger than the LR(k_2) machine and therefore a cycle-free machine obtained from the former may well have more states than one obtained from the latter. Hence we would prefer to construct a cycle-free MSP(k_2) machine. However, it may be the case that k_2 symbols do not provide sufficient lookahead in order to split states in the LR(k_2) machine, while k_1 symbols would suffice. We would like to use k_1 symbols of lookahead to make a prediction in a machine whose states are sets of LR(k_2) items. Even though this violates one of our restrictions, there is nothing wrong with this in principle, provided we are careful with the identity of the lookahead set.

For example, consider the LR(1) state of Figure 5.26. It is impossible to break this cycle using only one symbol of lookahead, since there is an essential .a-item and a must be in the predictive language.

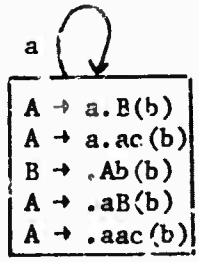


Figure 5.26

However, we can make a prediction based on two symbols of lookahead; namely predict B upon seeing aa. The splitting is shown in Figure 5.27.

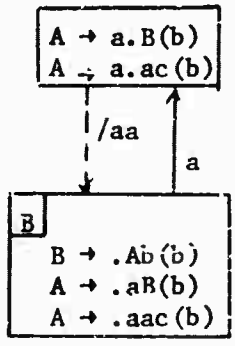


Figure 5.27

In order to describe what we are doing in cases like this, we want to modify our definition of prediction and state-splitting so that it becomes meaningful to predict an A in an LR(k_2) state upon seeing appropriate k_1-length lookaheads. We could try to modify the definitions in the moststraightforward way: namely, by changing expressions like $FIRST_{k_2}(A, B)$ to $FIRST_{k_1}(A, B)$ in the definition of the predictive language. The problem with this is that if B is a base state consisting of LR(k_2) items, then $FIRST_{k_1}(A, B)$ may not be defined. Recall that $FIRST_{k_1}(A, B)$ is the union of $FIRST_{k_1}$ for all .A items in B; but if A generates short strings, then an item like $C \rightarrow \alpha.A(\omega)$ where $\omega \in V_T^{k_2}$, may not have a defined $FIRST_{k_1}$; A may not generate any strings

of length greater than k_1. That was the advantage of using LR(k)-items to do k-lookahead prediction; every such item was assured of generating k-length strings. So the question becomes, how do we go about defining $FIRST_{k_1}$ on $LR(k_2)$ items where $k_1 > k_2$?

There are a variety of ways of varying accuracy in which we can do this. Any of these approaches will work in the sense that machines and grammars constructed using any of these definitions will exhibit the desired properties. Some ways are more precise than others, however, in capturing the precise meaning of $FIRST_{k_1}$ of an $LR(k_2)$ item.

As a first attempt, we could simply define $FIRST_{k_1}(A \rightarrow \alpha_1.\alpha_2(\omega))$ as $FIRST_{k_1}(\alpha_2\ FOLLOW_{k_1}(A))$; in other words, we are defining the lookahead of an item in terms of what it can produce, without paying much attention to the particular context in which it appears. That is, ω is really the precise context of the core of this item, and is a particular member of $FOLLOW_{k_2}(A)$; we might choose to forget this bit of information and just consider the full set of strings that can follow A as the context for this item.

A little more accurately, if $A \rightarrow .\alpha(\omega)$ is an immediate descendant of $B \rightarrow \beta_1.A\beta_2(\tau)$, we might define $FIRST_{k_1}(A \rightarrow .\alpha(\omega))$ as being $FIRST_{k_1}(\alpha\beta_2 FOLLOW_{k_1}(B))$; that is, we recognize that the A on the left-hand side of this item is not any A, but the A in $B \rightarrow \beta_1.A\beta_2(\tau)$; so the context for this A is anything generated by β_2, followed by anything that can follow a B.

But we have still not captured the information that whatever the k_1-context of this item may be, its k_2-prefix is known to be ω. This can be accounted for by defining $FIRST_{k_1}(A \rightarrow .\alpha(\omega))$ as $FIRST_{k_1}(\alpha X)$, where $X = \{x \mid x \in FIRST_{k_1}(\beta_2\ FOLLOW_{k_1}(B))$ and $x/k_2 = \omega\}$. However, we are still being

careless about the context of the B, using its full follow set, even though we should be restricting it to a more specific context, namely τ. We could begin to remedy this by using just those elements of $FOLLOW_{k_1}(B)$ that begin with τ, but that is only part of the solution. To get really precise we should make the following definition.

Definition 5.30 The k_1-context of an $LR(k_2)$ item, where $k_1 > k_2$, is defined as follows:

i) If the item is $A \rightarrow \alpha.\beta(\omega)$, its k_1-context is the same as $A \rightarrow .\alpha\beta(\omega)$.

ii) If the item is $A \rightarrow .\alpha\ (\dashv^{k_2})$, its k_1-context is $\dashv^{k_1}$.

iii) If $A \rightarrow .\alpha(\omega)$ is an immediate descendant of $B \rightarrow \beta_1.A\beta_2(\tau)$, then the k_1-context of $A \rightarrow .\alpha(\omega)$ includes all strings y in $FIRST_{k_1}(\beta_2 X)$, where X is the k_1-context of $B \rightarrow \beta_1.A\beta_2(\tau)$, such that $y/k_2 = \omega$.

Then if we talk about $FIRST_{k_1}$ of an item $A \rightarrow .\alpha(\omega)$, we mean $FIRST_{k_1}(\alpha X)$, where X is the k_1-context of the item. But even this can be refined further. For purposes of state-splitting, we are not really interested in the k_1-context of an item, but only its k_1-context with respect to that state; namely the possible lookaheads that might be found after the core of this item is recognized by this state of the machine. This will generally be a strict subset of the full k_1-context of the item; the previous definition can be modified to reflect this additional restriction.

We repeat that any of these definitions of $FIRST_{k_1}$ could be used in order to define a state-splitting of an $LR(k_2)$ state using k_1 symbols of lookahead, and the results derived in the earlier chapters will still pertain. That is, we can choose to predict an A on seeing ω, provided there is a .A

item in every chain that causes a k_1-lookahead of ω, even though in this state there will never really be a lookahead of ω, and that some chains seem to be causing lookahead of ω only because of the inexactness of the computation procedure for what chains do cause as lookahead. In other words, there is no harm in making predictions on spurious lookaheads, so long as the appropriate action is taken for real lookaheads. The only problem with using the grosser definitions of $FIRST_{k_1}$ is that they might prevent us from splitting states that might legitimately be split using a more accurate definition.

We can take this idea one step further, and use k_1 symbols of lookahead not only to splits states of an $LR(k_2)$ machine, but also the states of an $SLR(k_2)$ machine. [4] The states of an SLR(k) machine are composed entirely of LR(0) items; k symbols of lookahead, computed in various ways resembling our computation of $FIRST_{k_1}$ of $LR(k_2)$ items, may be used to determine which state to transfer to during the parse. We can easily add on to this model the additional concept of consulting some symbols of lookahead for predictive purposes. As before, the set of strings that are to occasion prediction of an A are to include at leas all strings of length k that might be sighted upon entry to the state and that might have some prefix derived from an A. We can use the previously described methods to do such computation. If a valid prediction can be made on such lookaheads, then the result of replacing a state of an SLR(k) machine by a splitting induced by such a prediction is a variant sort of MSP(k) machine, which will operate in the usual manner; if a sequence of such splittings produces a cycle-free machine, a grammar can be derived from this machine and it will have all the customary desirable properties. This derived grammar will have one additional powerful attraction:

its size. Since an SLR(k) machine for G (if it exists) is usually much smaller than the LR(k) machine for G, the sizes of the grammars derived from MSP(k) machines constructed from these two machines bear a similar relationship, frequently to a significant degree.

CHAPTER SIX
TOPICS FOR FURTHER RESEARCH

In this thesis, we defined and studied a new mode of parsing, which was a very general hybrid of bottom-up and top-down deterministic parsing schemes. We developed a model for this method of parsing in the form of MSP(k) machines, and then focused our attention on a particular kind of MSP(k) machine, namely those which contained no cycles. We showed that if a grammar G could be parsed by a cycle-free MSP(k) machine M, then we could derive an LL(k) grammar $T_M(G)$, whose nature depended on M and G, and which was equivalent to G. We examined the properties of $T_M(G)$, especially the kinds of translations it could support, and found ways to make it a more manageable size. Finally, we tried to get some feeling for the class of grammars that can be so transformed, and gained some insight into the problem of actually finding the cycle-free MSP(k) machine for such grammars. While we undoubtedly left some questions unanswered, we feel that we have in large measure succeeded at the task which we set ourselves, namely to discover and study a new transformation to convert non-LL(k) grammars into LL(k) form. But in a sense, the most satisfying and exciting part of this research has been the fact that a large number of collateral issues have been raised in the course of the preparation of this work. It is sometimes said that an important measure of a piece of reasearch is not the number of old questions it answers, but the number of interesting new questions which it asks. We shall now consider some unanswered questions that are related to the work reported in this thesis. These range from speculations on mild generalizations of our work to rather extensive modifications and expansions of our basic ideas.

CHAPTER SIX
TOPICS FOR FURTHER RESEARCH

In this thesis, we defined and studied a new mode of parsing, which was a very general hybrid of bottom-up and top-down deterministic parsing schemes. We developed a model for this method of parsing in the form of MSP(k) machines, and then focused our attention on a particular kind of MSP(k) machine, namely those which contained no cycles. We showed that if a grammar G could be parsed by a cycle-free MSP(k) machine M, then we could derive an LL(k) grammar $T_M(G)$, whose nature depended on M and G, and which was equivalent to G. We examined the properties of $T_M(G)$, especially the kinds of translations it could support, and found ways to make it a more manageable size. Finally, we tried to get some feeling for the class of grammars that can be so transformed, and gained some insight into the problem of actually finding the cycle-free MSP(k) machine for such grammars. While we undoubtedly left some questions unanswered, we feel that we have in large measure succeeded at the task which we set ourselves, namely to discover and study a new transformation to convert non-LL(k) grammars into LL(k) form. But in a sense, the most satisfying and exciting part of this research has been the fact that a large number of collateral issues have been raised in the course of the preparation of this work. It is sometimes said that an important measure of a piece of research is not the number of old questions it answers, but the number of interesting new questions which it asks. We shall now consider some unanswered questions that are related to the work reported in this thesis. These range from speculations on mild generalizations of our work to rather extensive modifications and expansions of our basic ideas.

One piece of unfinished business is, of course, the development, implementation, and analysis of an actual compiler-compiler containing a transformation phase embodying the ideas we have developed. Such an effort could lead to an evaluation of the practicality and utility of the work we have done. Construction of such a system would also provide the impetus for a deeper understanding of the process of constructing a cycle-free MSP(k) machine for a grammar; faster, more efficient algorithms for selecting state-splittings and for breaking cycles would have to be developed. Also, more precise information would be needed about the trade-offs involved in reducing the size of a derived grammar. In a sense, these issues are loose ends of our work which would become more interesting in the context of an implementation effort.

A more theoretical question concerns the extent of the k-transformable grammars. Is there some other way of characterizing this class of grammars? We have seen that the class of k-transformable grammars strictly includes the class of LC(k) grammars. In other words, we know that if the Rosenkrantz transformation can convert a grammar into LL(k) form, then that grammar is k-transformable. Does this result extend to other conventional transformations? In particular it would be good if we could relate the class of k-transformable grammars to the class of grammars that can be converted into LL(k) form by the application of some sequence of transformations drawn from a collection of standard transformations. A canonical "bag of tracks" might include substitution and left factoring, as well as the Rosenkrantz transformation; these are the most common techniques currently employed in heuristic efforts to achieve an LL(k) form for a grammar [20]. A result relating the k-transformable grammars to this class of grammars might eliminate the need

for the haphazard application of conventional transforming tricks.

There is another conjecture about the extent of the k-transformable grammars, suggested by R. E. Stearns, which effectively suggests that the k-transformable grammars are precisely the class of grammars that it pays to transform into LL(k) form. It may be stated in the following way. Let us mean by the simple Polish punctation of a grammar, a translation grammar where the translation element of any rule consists of the nonterminals of the rule followed by a terminal symbol unique to the rule. (One can think of this symbol as the name of the action routine to be called when this rule is recognized.) The conjecture states that if G is an LR(k) grammar and if there exists an LL(k) grammar G', equivalent to G, with some simple translation grammar for G' equivalent to the simple Polish punctation of G, then G is k-transformable. This conjecture says that if there is some LL(k) grammar which is as useful for compilation as G, then that grammar can be found by application of our transformation.

Clearly, in order to make any satisfactory progress in this area, we must have the appropriate definition for k-transformable grammar. It may be necessary to further generalize our definitions of prediction, state-splitting, and MSP(k) machine, and refine our grammatical derivation process, in order to achieve desirable results. We have already seen several different versions of our basic ideas; for example, at first, prediction had to be of all A's in a state, and then we allowed it to be of just certain specific A's. It may be useful to try a further generalization, and allow an A to be predicted by some subset of its lookahead language. This will clearly expand the class of permissible state splittings and hence the class of k-transformable gram-

mars.

For example, consider the LR(1) state of Figure 6.1, which is its own a-successor. According to our conventional definition of prediction, this

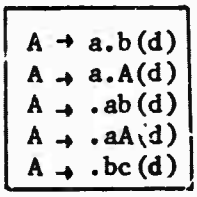


Figure 6.1

state cannot be split using one symbol of lookahead, since $b \in FIRST_1(A)$ but b does not indicate the presence of A . But the symbol a does indicate the presence of A, and induces the splitting of Figure 6.2. There are numerous

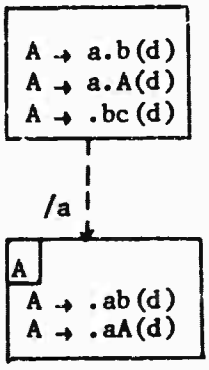


Figure 6.2

and nontrivial technical difficulties in trying to generalize the notion of state-splitting in this way, particularly in computing the successors of components of the splitting; but further research might provide solutions to the problems.

It might be interesting to consider even more radical variations of our notion of prediction. Consider a machine model that operates as follows. Upon entry to a base state and inspection of the lookahead, a prediction can

be made and transfer effected to a submachine.
Upon fulfillment of the prediction by the submachine, control is returned to the calling base state; but rather than immediately passing to a successor, the base state can again inspect the lookahead and then possibly make another prediction. In other words, a base state can have several predictive states associated with it, which may be transferred to sequentially. The potential value of such a generalization is clear. It might happen upon entry to a state that with the lookaheads available, only a relatively "low-level" prediction can be made, not enough to break a cycle in which the state occurs; but after this first prediction is fulfilled, a more useful prediction can then be made.

Of course, the grammatical derivation procedure would have to be altered to accommodate this revised machine model. This might be done as follows. Suppose some base state can make a prediction of a Y; after the Y has been found, the base state can predict a W or a Z, depending on the lookahead; and after either of these predictions is fulfilled the base state makes no more predictions, but relinquishes control to a successor state. Then if (X, m) is a name of the base state, the derived grammar will have rules $(X, \varphi) \rightarrow (Y, \epsilon)(X, \varphi)'$; $(X, \varphi)' \rightarrow (Z, Y)(X, \varphi Z)$; $(X, \varphi)' \rightarrow (W, Y)(X, \varphi W)$. Here $(X, \varphi)'$ is a new nonterminal, which holds a place in the derivation until it is determined whether W or Z is really going to be found. The prediction of the Y is just an auxiliary to the later prediction. When the later prediction is made, it is not the nonterminal (Z, ϵ) or (W, ϵ) that gets introduced into the derivation, for Y has already been found; (W, Y) is introduced, to indicate that we are rather belatedly predicting W, after already finding Y, and that we now have to find the rest of W after Y.

Such a notion of retroactive prediction introduces numerous complications into the machine model, and requires substantial revision of our formal development.

But some generalization along these lines will be necessary to carry our concept of prediction in bottom-up parsing to its natural conclusion, and thus define the largest possible class of k-transformable grammars.

Consider the grammar of Figure 6.3; part of its LR(0) machine is shown in Figure 6.4.

S → A	B → (B)
S → C	B → b
A → BX	X → BA
A → BY	Y → BC
C → Bd	

Figure 6.3

There is a cycle in this part of the machine; and clearly no amount of lookahead could possibly be used to make a prediction that would split a state in such a way that would break the cycle. In state 1, the vital items are X → .BA and Y → .BC; but no amount of lookahead can distinguish X from Y, since they both begin with B, which can generate indefinitely long strings.

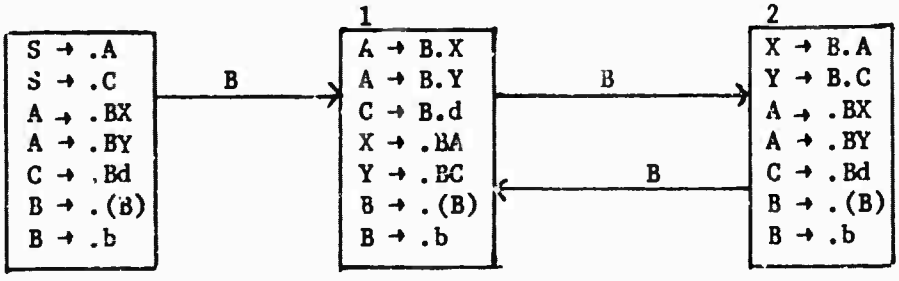


Figure 6.4

Similar comments apply to A and C in state 2. It is easy to see that nothing would be gained by constructing LR(k) machines for $k > 0$, and trying to use lookahead to break the cycles in those machines, since the same situation will obtain there. Clearly this grammar is not k-transformable for any k.

However, the cycle could be broken using the hierarchical prediction scheme just described. Upon entry to state 2, we can safely predict a B. After the B has been found, we can look ahead and predict once again; if the lookahead is the symbol d, then the B we have just found was the first part of a C, otherwise it was the beginning of an A. The grammar rules that would be derived from a splitting based on such a prediction would be (S, BB) → (B, ε)(S, BB)′; (S, BB)′ → (C, B)(S, BBC); (S, BB)′ → (A, B)(S, BBA); these could well be part of an LL(k) grammar.

It is significant that the grammar of Figure 6.3 can be transformed into LL(1) form by other means. By doing some substitutions, a left factoring, and then applying the Rosenkrantz transformation, the result is LL(1). Thus in order for our transformation to eliminate the need for all others, some generalization along the lines just described will be necessary.

We might go very far afield in another direction, and drastically revise our notion of what constitutes a legal state-splitting. On the one hand, we can consider allowing the predictive languages of two initial states of a splitting to intersect, while keeping the rest of the transformation procedure the same. The only effect of this alteration will be to destroy the LL(k)-ness of the derived grammar. This might prove useful, in enabling us to apply the transformation to an LR(k) grammar for a non-LL(k) language; the transformed grammar will generate the same language as the original grammar, but

of course it will not be LL(k). However, it might exhibit other useful and interesting properties.

Another possibility would be to construct a predictive state not for a single nonterminal but for a whole collection of them. Then the derived grammar will indeed be LL(k), but it will not necessarily generate the same language as the original grammar from which the machine was constructed. In this way we can approach the idea of an LL(k) approximation to a non-LL(k) language. This could prove to be a very attractive concept. We could transform a non-LL(k) grammar into an LL(k) grammar for a different but closely related language, by constructing an MSP(k) machine that doesn't make very precise predictions, and deriving a grammar from that machine. Then hopefully we would have an LL(k)-driven compiler that correctly parses any legal program, though it might possibly accept some illegal programs in addition. We might catch these illegal programs with a preprocessor at an earlier stage, paying the price of this two-stage complexity for the advantage of having a well-designed and efficient compiler.

As an example of these ideas, consider the grammar of Figure 6.5. Although this grammar is LR(0), the language that it generates in $\{a^n b^n \cup a^n c^n\}$, which is known not to be LL(k) for any k. The LR(0) machine for this grammar

S → A	A → ab
S → B	B → aBc
A → aAb	B → ac

Figure 6.5

is given in Figure 6.6.

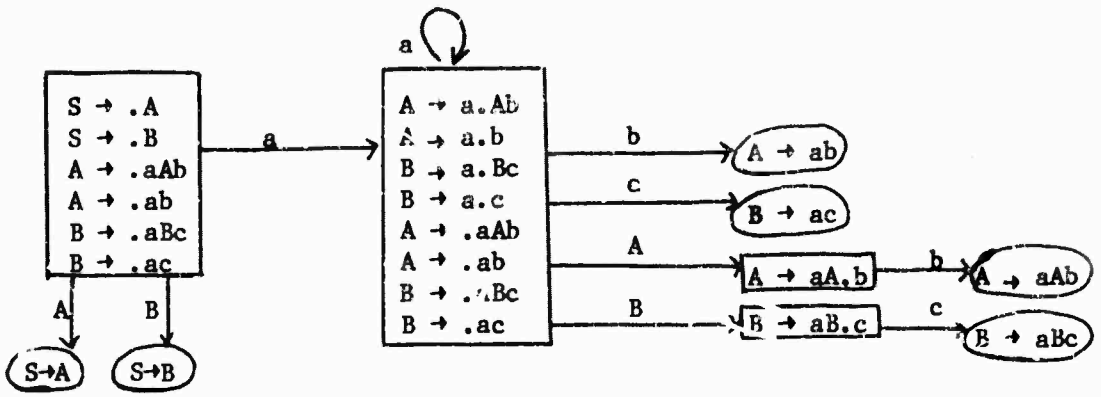


Figure 6.6

The first approach just described would have us split the state as shown in Figure 6.7. The grammar generated from the machine containing this split-

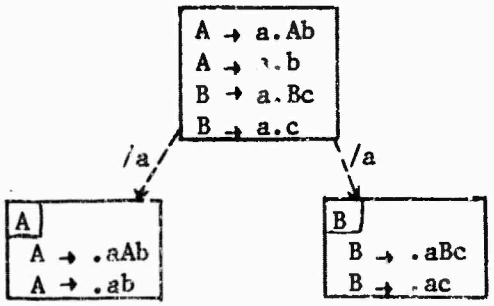


Figure 6.7

ting would contain the rules (S, a) → (A, ∈)(S, aA) and (S, a) → (B, ∈)(S, aB). This would prevent the resulting grammar from being LL(k), because A and B both generate indefinitely long strings of a's.

Another possibility would be to create a splitting as shown in Figure 6.8. The name of this predictive state will be neither (A, ∈) nor (B, ∈), but some new name (X, ∈). The derived grammar will have rules (S, a) → (X, ∈)(S, aX);

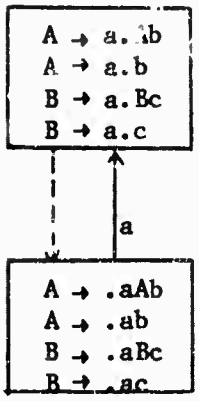


Figure 6.8

$(X, \epsilon) \rightarrow a(X, a)$; $(X, a) \rightarrow (X, \epsilon)(X, aX)$; $(X, a) \rightarrow b(X, ab)$; $(X, ab) \rightarrow (X, A)$; $(X, A) \rightarrow \epsilon$; and so on. In other words, X stands either for A or for B, whichever is appropriate in the context, This derived grammer will be LL(1), but it will generate a new language, namely $a^n(b \cup c)^n$, which is different from, but closely related to, the original language. We note that $a^nb^n \cup a^nc^n = a^n(b \cup c)^n \cap (a^*b^* \cup a^*c^*)$; so a finite-state machine preprocessor, letting through only strings in the set $a^*b^* \cup a^*c^*$, followed by the derived LL(1) parser, will correctly process the language $a^nb^n \cup a^nc^n$.

Another exciting prospect is that of applying some variant of our transformation procedure to non-LR(k) grammars. Here of course we could not start with an LR(k) machine for the grammar and try to achieve a cycle-free machine by state-splitting. However, we could attempt to construct an LR(k) machine for the grammar; the result will be a bottom-up nondeterministic parser. If we can split states and get a cycle-free version of this machine, we can proceed to read off a grammar. Of course this grammar will not be LL(k), reflecting the nondeterminacy of the machine's operation; but we have discussed a similar notion in nondeterministic prediction above. However, it may well be

that the grammar we derive from this nondeterministic MSP(k) machine will be LR(k) for some k.

For example, consider the grammar S → ADx, S → BDy, A → a, B → a, D → Dd, D → d. This grammar is not LR(k) for any k, since whether to reduce a to A or B can only be decided by looking past the D, which cannot be done with finite lookahead. The quasi-LR(0) machine for this grammar is shown in Figure 6.9.

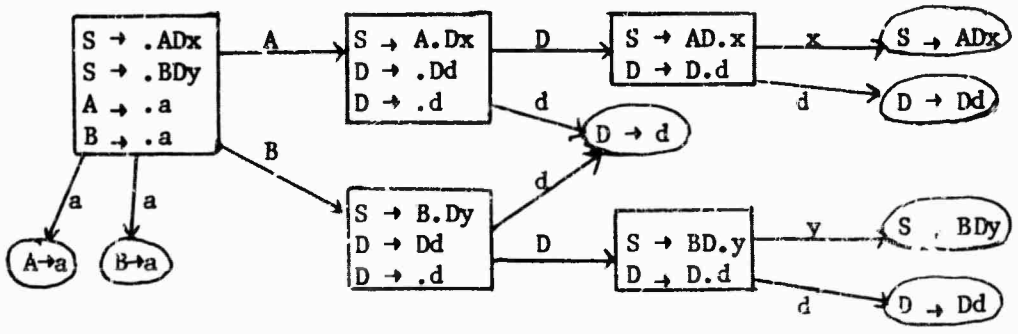


Figure 6.9

While this machine may not be deterministic, it is cycle-free, and we can derive a grammar from it, which is shown in Figure 6.10.

(S, ∈) → a(S, a)
(S, a) → (S, A)
(S, a) → (S, B)
(S, A) → d(S, Ad)
(S, B) → d(S, Bd)
(S, Ad) → (S, AD)
(S, Bd) → (S, BD)
(S, S) → ∈

(S, AD) → x(S, ADx)
(S, AD) → d(S, ADd)
(S, BD) → y(S, BDy)
(S, BD) → d(S, BDd)
(S, ADx) → (S, S)
(S, ADd) → (S, AD)
(S, BDy) → (S, S)
(S, BDd) → (S, BD)

Figure 6.10

This grammar is not LL(k), because of the rules (S, a) →(S, A) and (S, a) → (S, B); this reflects the non-determinacy of the machine's operation. However it is LR(0). It would be extraordinarily useful if we could find a way

to convert certain non-LR(k) grammars for LR(k) languages into LR(k) form, for that would free a programming language designer from the constraint of having to make his original language specification in terms of an LR(k) grammar. Admittedly this is not a very onerous restriction, and removing it risks the possibility of the designer coming up with an ambiguous grammar, but the concept is appealing nonetheless.

There is also the possibility of altering the procedure by which the derived grammar is obtained from a machine. Our current approach requires the construction of a cycle-free machine for the process to go forward. Strictly speaking, the grammar derivation procedure could read rules off a machine with cycles just as easily as from a cycle-free machine; the cycle-free requirement is only to ensure that there will be finitely many nonterminals in the grammar. It might be possible to devise a state-naming procedure for certain machines with cycles, such that each state would get only finitely many names; hence a well-defined grammar could be constructed from the machine.

The possibilities of extending our transformation raise numerous questions about successive applications of the transformation. We might consider a hierarchy of classes of grammars, the nth level being those that can be transformed into LL form by n applications of the transformation. The extent of the hierarchy, relationship between different levels in it, and other such questions would be of great theoretical interest with considerable practical significance.

Finally, it might prove interesting to conduct a theoretical investigation of a rather abstract model of MSP(k) machines. We could define an automaton called a recursive finite state machine (RFSM), consisting of a set of

finite state machines which have the added feature that they can call each other. Such calls are effected by inspection of lookahead symbols; upon entry to one of its final states, a submachine returns control to the state that called it. For each value of k, we could define the class of k-RFSMs as being those that inspect k symbols of lookahead in deciding which submachine to call. A similar model has been proposed by several authors including Tixier [21], and has been studied extensively by Lomet[16]; but in Lomet's model, a submachine is capable of transmitting information to its caller when it returns. Given this additional feature, it can be shown that such machines can accept the full class of LR(k) languages It would be interesting to know what class of languages our more restricted RFSMs accept, for our model seems very similar to the canonical pushdown machines of Rosenkrantz and Stearns [18] (basically 1-state PDA's with lookahead); these in turn are excellent models for the construction of parsers, being compact, efficient, and easy to implement.

The foregoing discussion gives a feeling for some of the issues that have been raised during the course of our research and which remain unanswered. It is hoped that whatever limited success we have achieved, and the potential utility of these investigations, will encourage others to take up where we have left off.

BIBLIOGRAPHY

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

2. Aho, A.V., and Ullman, J. D., "LR(k) Syntax Directed Translation," Unpublished manucript, Bell Laboratories, Murray Hill, N.J., 1972.

3. Cheatham, T. E. Jr., The Theory and Construction of Compilers, Computer Associates, Wakefield, Mass., 1967.

4. DeRemer, F. L., "Practical Translators for LR(k) Languages," Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Mass., 1969.

5. DeRemer, F. L., "Simple LR(k) Grammars," CACM 14, 453-460 (1971)

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

7. Griffiths, T. V., and Petrick, S. R., "On the Relative Efficiency of Context-Free Grammar Recognizers," CACM 8, 289-300 (1965)

8. Griffiths, T. V. and Petrick, S. R., "Top-Down Versus Bottom-Up Analysis," IFIP Congress 68, Software Booklet B, 80-85.

9. Irons, E. T., "A Syntax Directed Compiler for ALGOL 60," CACM 4, 51-55 (1961)

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

11. Knuth, D. E., "Top-Down Syntax Analysis," Acta Informatica 1, 79-110 (1971)

12. Kurki-Suonio, R., "Notes on Top-Down Languages," BIT 9, 225-238 (1969)

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

14. Lewis, P.M. II, and Rosenkrantz, D.J., "An ALGOL Compiler Designed Using Automata Theory," Proceedings of the Symposium on Computers and Automata, Microwave Research Institute Symposia Series, Volume XXI, Polytechnic Press, Brooklyn, N.Y., 75-88 (1971)

15. Lewis, P.M. II, Rosenkrantz, D.J., and Stearns, R.E. Private communication.

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

17. Rosenkrantz, D. J., "Matrix Equations and Normal Forms for Context-Free Grammars," JACM 14, 501-507 (1967)

18. Rosenkrantz, D. J., and Stearns, R. E., "Properties of Deterministic Top-Down Grammars," Information and Control 17, 226-256 (1970)

19. 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.

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

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

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

23. Wood, D., "A Note on Top-Down Deterministic Languages," BIT 9, 387-399 (1969).

24. Wood, D., "A Note on the Syntax Improvement Device with Improvements," Unpublished memorandum, 1969.

CHAPTER SIX
TOPICS FOR FURTHER RESEARCH

In this thesis, we defined and studied a new mode of parsing, which was a very general hybrid of bottom-up and top-down deterministic parsing schemes. We developed a model for this method of parsing in the form of MSP(k) machines, and then focused our attention on a particular kind of MSP(k) machine, namely those which contained no cycles. We showed that if a grammar G could be parsed by a cycle-free MSP(k) machine M, then we could derive an LL(k) grammar $T_M(G)$, whose nature depended on M and G, and which was equivalent to G. We examined the properties of $T_M(G)$, especially the kinds of translations it could support, and found ways to make it a more manageable size. Finally, we tried to get some feeling for the class of grammars that can be so transformed, and gained some insight into the problem of actually finding the cycle-free MSP(k) machine for such grammars. While we undoubtedly left some questions unanswered, we feel that we have in large measure succeeded at the task which we set ourselves, namely to discover and study a new transformation to convert non-LL(k) grammars into LL(k) form. But in a sense, the most satisfying and exciting part of this research has been the fact that a large number of collateral issues have been raised in the course of the preparation of this work. It is sometimes said that an important measure of a piece of research is not the number of old questions it answers, but the number of interesting new questions which it asks. We shall now consider some unanswered questions that are related to the work reported in this thesis. These range from speculations or mild generalizations of our work to rather extensive modifications and expansions of our basic ideas.