PRACTICAL TRANSLATORS FOR LR(k) LANGUAGES

by

FRANKLIN LEWIS DeREMER

S. B ., Massachusetts Institute of Technology
(1965)

S. M., Massachusetts Institute of Technology
(1966)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
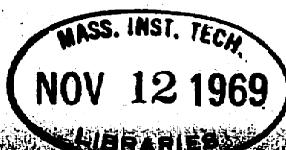
September, 1969

Signature of Author _____
Department of Electrical Engineering, August 25, 1969

Certified by _____
Thesis Supervisor

Accepted by_____
Chairman, Departmental Committee on Graduate Studies

PRACTICAL TRANSLATORS FOR LR(k) LANGUAGES

by

FRANKLIN LEWIS DeREMER

Submitted to the Department of Electrical Engineering on
August 25, 1969 in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science.

ABSTRACT

A context-free syntactical translator (CFST) is a machine which defines
a translation from one context-free language to another. A transduction
grammar is a formal system based on a context-free grammar and it
specifies a context-free syntactical translation. A simple suffix trans-
duction grammar based on a context-free grammar which is LR(k)
specifies a translation which can be defined by a deterministic push-down
automation (DPDA).

A method is presented for automatically constructing CFSTs (DPDAs)
from those simple suffix transduction grammars which are based on the
LR(k) grammars. The method is developed by first considering grammatical
analysis from the string-manipulation viewpoint, then converting the
resulting string-manipulation algorithms to DPDAs, and finally considering
translation from the automata-theoretic viewpoint.

The results are relevant to the automatic construction of compilers from
formal specifications of programming languages. If the specifications
are, at least in part, based on LR(k) grammars, then corresponding
compilers can be constructed which are, in part, based on CFSTs.

Thesis Supervisor: Arthur Evans, Jr.
Title: Assistant Professor of Computer Science

ACKNOWLEDGEMENTS

For their contributions to the successful completion of this dissertation I express my gratitude to my thesis advisor, Professor Arthur Evans, Jr., and to my readers, Professors Thomas E. Cheatham, Jr. (of Harvard University), Chung L. Liu, and John M. Wozencraft. I especially thank Professors Wozencraft and Cheatham for their special interests in and contributions to the research.

Thanks go to my wife, Sherry, for suffering through both the typing of the preliminary versions of the thesis and the rather restricted social life of the last three months.

Finally, I thank Miss Andrea Bickell for the cheerful and excellent typing service rendered with regard to the final copy.

TABLE OF CONTENTS

PAGE REFERENCES

Chapter 1

INTRODUCTION

1.1 Subject

The general subject of interest in this dissertation is "programming linguistics", which we consider to be a science concerning the design and specification of programming languages and the translation and subsequent evaluation and execution of programs. In particular, we are primarily interested in the problem of automatically generating translators from formal specifications of translations based on context-free (CF) grammars.

1.2 Languages, Translations

In the sequel we use the two words, language and translation (also translator), in both the formal and informal sense. The proper sense in each case is always clear from context. A language is defined formally in Chapter 2 to be a set of strings. However, when we say "programming language" or "language designer", we have in mind a more intuitive notion. For instance, when we refer to the "language" ALGOL 60, we mean the syntax and semantics, the set of strings and their meanings, the lexicon and the grammar, operator precedences and associativities, scopes of variables, etc. Similarly, our formal definition in Chapter 5 of translations limits them to mappings from one set of strings to another, but we also use the term to mean a mapping from one set of things, of any sort, to another, of any sort.

1.3   Viewpoint: TWSs, Modular Compilers

For purposes of discussion we picture ourselves throughout the dissertation as a subcontractor to a language designer. The designer has a contract to design and implement a practical algorithmic programming language, and he has subcontracted to us the task of implementing a compiler for his language.

We desire to automate our implementation procedures for three reasons: (1) the designer is likely to want to experiment to some extent to determine the effect of various design decisions, and he would like fairly short response time, (2) we expect to receive more such contracts in the future, and (3) implementing a compiler usually requires many man-hours of expensive programmer time. The embodiment of such an automation is called a translator writing system (TWS) (see the survey (F&G 68)). It is a system which takes as input the specification of the syntax and translation of a language and which produces as output a compiler for that language.

The questions, then, which confront us are: how do we specify programming languages and their translations, and how can we map these specifications into compilers? We choose a modular approach which is a combination of some of the notions of Cheatham (Che 67) and Landin (Lan 66). We find it convenient, even natural, to section our specifications into components. For instance, we might specify separately the lexicon, the

context-free syntax, and the context-sensitive syntax. (We discuss briefly in Section 1.4 and extensively in Chapter 5 our reasons for sectioning the specifications in certain ways.) Further, we find it convenient to base some aspects of our translation specification on these different components. It is reasonable, then, to view a compiler, conceptually at least, as a concatenation of several corresponding subtranslators; i. e., as modularized.

The adoption of this viewpoint results in three significant advantages relative to a less modular approach. First, the otherwise complex task of compiling is viewed as broken into several relatively simple components, each of which may be analyzed virtually independently of the others. Second, the task of a TWS is viewed as the separate generation of several subtranslators, followed by their optimal combination to form a compiler. Third, because the specifications of some of the subtranslations can be naturally and conveniently based on formal grammars, the abundant results of both formal-grammar theory and automata theory are relevant to the corresponding translators and their automatic generation. We consider the theoretical underpinnings which accrue from the latter to be important because (1) they allow us to make provable statements regarding the efficiency, execution time, size, etc., of our translators, (2) they allow us to modify our translators in a rational way to get an optimal compromise between time and space, (3) they help us avoid ad hoc, ill-understood modifications which make the subsequent combination of translators difficult, if not impossible or incorrect, and (4) they add a certain degree of "cleanliness"

A possible criticism cf our approach is that the separate analyses of the components may result in translation methods or devices which, where combined, will form a compiler with gross redundancies, such as repeated building and scanning of data structures, which cannot be eliminated by any reasonably simple procedure. We do not believe that this will be the case, but we shall not go so far as to make this belief a thesis to be proved here. The results of ourselves and others are, however, steps in that direction

One existing result in this vein is presented in (Joh 68). It is a method of automatically generating practical "lexical analyzers", really "lexical translators", from a specification based on regular expressions. The technique is based directly on some rudimentary notions of finite-state machine theory. It is our desire to get similar results for "CF syntax analyzers", really "CF syntactical translators (CFSTs)".

1.4   The Role of CF Grammars

Another belief which is fundamental to our work is that CF grammars can be used in a natural and convenient way as bases for the specifications of significant portions of the syntax and translation of programming languages, and we believe that this includes useful languages in which highly readable programs can be written. Furthermore, we find that a well designed CF grammmer makes a concise, readable, and useful syntactical reference for a language, a reference from which operator precedences and associativities,

scopes of definitions, and other such "structural properties", can be quickly

and easily determined.

Having stated our view in positive terms, we now add some disclaimers.
(1) We do not contend that it is obvious how to design CF grammars so that

they exhibit the above stated properties. For instance, we do not think that

the (probably) best known CF grammar, that of ALGOL 60 (Nau 63), is an

example of a good syntactical reference; it seems more complex than it

needs to be. However, we illustrate in Chapter 7 a grammar which partially

specifies a language comparable to ALGOL in many respects, and which, we

think, is a reference with the desired properties. Unfortunately, the value

of our results is somewhat limited until this grammar design problem is

better understood. We have pursued our research, then, on the hope that

some results relating to this problem are forthcoming. (2) We do not contend

that programming languages should be CF. We merely believe that much of

their syntax can be easily defined via CF grammars and that the remaining

syntax, e. g. , "context-sensitive features", can then be defined in other ways,

probably related to the CF grammars. See for example(Knu 66). (3) Neither

do we contend that CF grammars are a panecea with respect to language

specification. Indeed, they are woefully inadequate for indicating nonasso-

ciative operators, for instance; and there are certainly other ways (see Chapter

8) in which their usefulness would be enhanced if they could be extended. We

merely believe that they are the most useful devices currently available for specifying many of the "structural properties" of languages.

LR(k) grammars. Actually, we do not intend to cover all of the CF grammars here. Our experience is that, if a designer sets out to design an unambiguous CF grammar to specify the "structural properties" of a language, his result will be an LR(k) grammar (Knu 65); i. e. , a grammar whose sentences can be analyzed (parsed) during a single, deterministic scan from left to right. Intuitively, we feel that this situation obtains because the language is presumably designed to be written and read by humans, and humans, at least those who are used to reading natural languages from left to right, would probably find programs quite unreadable if they could not be syntactically analyzed during a single scan from left to right.

Thus, to the extent that unambiguity is a desirable characteristic of a syntactical reference, anyway, our results should be as useful as if they covered all CF grammars. We do not find the restriction to unambiguity bothersome.

The reason we choose the LR(k) grammars, in particular, is that they form the largest set of CF grammars whose sentences can be analyzed quickly by a deterministic, left-to-right automaton, as we show. We can therefore automatically generate at least part of a compiler for any language whose specification is, in part, based on an LR(k) grammar, and we can expect that part of the compiler to be fast.

Translators. Finally, we emphasize that we are really interested in
translators rather than just parsers, for reasons which we discuss exten-
sively in Chapter 6. As a method for specifying CF syntactical translations
we have chosen the "transduction grammars" of Lewis and Stearns (L&S 68).
In fact, we use only the "simple suffix" transduction grammars (SSTGs)
(see Chapter 6). Again our choice was based on the fact the method seems
both natural and convenient for our purposes and on the fact it has strong
ties with automata theory.

1.5   Thesis

It is our thesis that by applying some rudimentary notions of
automata theory we can develop a practical method of <u>automatically generating</u>
<u>CFSTs from</u> those <u>SSTGs</u> which are <u>based on</u> the <u>LR(k) grammars</u>. Further-
more, if the SSTGs in question are used to specify the CF syntactical
translations of useful, readable programming languages, the resulting
CFSTs will be of practical size and speed.

By a "practical" method or CFST we mean one which is competitive
with the methods or "recognizers" of section II. B of (F&G 68); i. e., ones
which have actually been used in the construction of compilers. Our aim is
not so much to improve on the size and speed of CFSTs as it is to provide the
language designer with <u>flexibility</u>. With existing methods the designer usually
has to modify his grammar substantially before it is acceptable to the method.
By covering all the LR(k) grammars we, hopefully, get a method which will accept

grammars as they are designed as syntactical references for languages, with no modifications. If the grammars are unambiguous and if all their sentences can be parsed deterministically, during a single scan from left to right, the latter will be true.

1.6   Approach

Our approach to this problem is basically inspired by and quite similar to Knuth's. However, we draw even more heavily on automatic theory than he did, at least with respect to getting practical results, and we treat translation rather than just parsing. We treat parsers first because they provide a convenient basis from which we can develop translators. This follows from the fact that the specifications of our translations are based on CF grammars.

We begin in Chapter 2 by discussing parsing from the string-manipulation viewpoint, as is typical when working with formal grammars. We present a particular parser, described as a string-manipulation algorithm, and motivate our own definition of the LR(k) grammars.

In Chapter 3 we develop a foundation by treating only the LR(0) grammars. We draw on finite-state machine (FSM) theory to develop a machine for making basic string-manipulation (parsing) decisions. Then we shift entirely to automata theory by deriving from our string-manipulation algorithm, plus FSM, a deterministic push-down automaton (DPDA). That is, we get DPDAs as parsers for LR(0) grammars.

In Chapter 4 we find that a large and useful subset of the LR(k) grammars, which we call the "Simple LR(k)" grammars, can be covered by first constructing an FSM as in the case of an LR(0) grammar, then adding to the machine some "look-ahead" information computed in a simple way, and finally converting the string-manipulation algorithm and FSM to a DPDA with "look-ahead".

We generalize to cover all LR(k) grammars in Chapter 5. We find that parsers for some of these grammars can be constructed just as are those for Simple LR(k) grammars, if more complex methods for computing "look-ahead" information are employed. In general, however, we find that some state-splitting operations must be applied to the FSMs along with the more complex computations of "look-ahead". Our development in Chapter 5 is in two phases. We first cover a set of grammars of the "bounded context" variety and then we generalize to cover all LR(k) grammars.

Our result going into Chapter 6, then, is a parser-constructing technique which grows in complexity as it discovers the complexity of the grammar at hand.

In Chapter 6 we motivate the abstraction of a string-to-string translation from the compilation process. Then we define transduction grammars for use in specifying these translations and show how to convert our parsers to translators. Finally, we show how we envision our translators fitting into compilers, via an explicit model, and we discuss the relevance of our results to the design and specification of languages, translations and compilers.

We illustrate in Chapter 7 the practicability of our scheme. We first summarize our translator constructing technique as a whole. Then we propose a method of implementing the translators, apply the method to a particular, practical transduction grammar, and show that our scheme compares favorably with an existing, practical technique.

We end the dissertation with Chapter 8 in which we note some developments which are desirable before our scheme is incorporated in a TWS, state some conclusions, and pose some question for future research.

1.7   Efficiency, Complexity, Recognizers

Several more informal definitions are in order before we proceed.

In the sequel we frequently refer to the "efficiency" of our translators. By "time-efficiency" we mean the ability to effect a translation using a minimum number of "machine operations", and therefore time. In Chapter 4 we give a specific definition in terms of an ideal machine. By "space-efficiency" we mean the ratio of the amount of space necessary to store the specification of a translation to that necessary to store the corresponding translator. We define this more precisely in Chapter 7.

The "size" of a grammar is the number of symbols required to write down all the left and right parts of the productions. By "grammatical complexity" we mean a measure of the time required to construct a parser for a grammar when using our technique. Although this definition may seem

to reflect some egotism, we use it for lack of a better choice. It does, however, seem to correspond to the intuitive notion fairly well. Our measure depends both on the size of the grammar and on the "complexity' of the functions which must be employed to compute "look-ahead" and state-splitting.

Finally, we use the word "recognizer" in a more technical sense than it was used in (F&G 68). We adopt the automata-theoretic notion that a recognizer is a machine which reads a string and either accepts or rejects it, as far as its being in a given language is concerned. Our parsers and translators output considerably more information than is contained in a simple "yes" or "no" from a recognizer.

Chapter 2

PRELIMINARIES

2.1 Notation, Preliminary Definitions

We begin by defining terms and notation. We assume the reader is familiar with the properties of <u>symbols</u>, <u>strings</u> of symbols, <u>regular expressions</u>, and <u>languages</u>, <u>finite state machines</u> (FSMs), <u>formal grammars</u>, and both <u>deterministic</u> and <u>nondeterministic</u> <u>pushdown automata</u> (DPDAs and NPDAs).

A <u>context-free</u> (CF) <u>grammar</u> is a quadruple (V_T, V_N, S, P) where V_T is a finite set of symbols called <u>terminals</u>, V_N is a finite set of symbols distinct from those in V_T called <u>nonterminals</u>, S is a distinguished member of V_N called the <u>starting symbol</u>, and P is a finite set of pairs called <u>productions</u>. Each production is written $A \rightarrow \omega$ and has a <u>left part</u> A in V_N and <u>right part</u> ω in V^* where $V = V_N \cup V_T$. V^* denotes the set of all strings composed of symbols in V, including the empty string.

Without loss of generality we conventionalize that (i) the productions are arbitrarily numbered from 0 to s, and (ii) the zeroth production is of the form $S \rightarrow \vdash S' \dashv$, where S' is sort of a subordinate starting symbol and S and the terminal "pad" symbols $\vdash$ and $\dashv$ appear in none of the other productions.

We use Latin capitals to denote nonterminals, lower case Latin letters, digits and special symbols (e.g., +, *, :, etc.) to denote terminals, and

lower case Greek letters to denote strings. An exception is that we reserve ϵ to denote the empty string. We use $|\beta|$ to denote the length of (number of symbols in) the string β, and $k{:}\beta$ to denote the first k symbols of β if $|\beta| \geq k$ and β otherwise. If $\alpha = \varphi\beta$ is a string, then φ is a prefix and β is a suffix of α, and α is the concatenation of φ and β.

In the sequel, we often use for examples the grammar

$$G_1 = (\{(,), i, \uparrow, +, \vdash, \dashv\}, \{S, E, T, P\}, S, P_1)$$

where P_1 consists of the following productions:

(0)	$S \rightarrow \vdash E \dashv$		(4)	$T \rightarrow P$
(1)	$E \rightarrow E + T$		(5)	$P \rightarrow i$
(2)	$E \rightarrow T$		(6)	$P \rightarrow (E)$
(3)	$T \rightarrow P \uparrow T$			

If $A \rightarrow \omega$ is a production, an immediate derivation of one string $\alpha = \rho\omega\beta$ from another $\alpha' = \rho A \beta$ is written $\alpha' \rightarrow \alpha$. We say α is immediately derivable from α' via application of the production $A \rightarrow \omega$ to a particular occurrence of A in α'. The transitive completion of this relation is a derivation and is written $\alpha' \overset{*}{\rightarrow} \alpha$, which means there exist strings $\alpha_0, \alpha_1, \ldots, \alpha_n$ such that $\alpha' = \alpha_0 \rightarrow \alpha_1 \rightarrow \ldots \rightarrow \alpha_n = \alpha$ for $n \geq 0$. A right derivation, written $\alpha' \overset{*}{\underset{R}{\rightarrow}} \alpha$, is one in which for $i = 1, 2, \ldots, n$ each α_i is immediately derivable from

α_{i-1} via application of a production to the rightmost nonterminal in α_{i-1}.
We choose the right derivation as our <u>canonical</u> <u>derivation</u>.

A <u>terminal</u> <u>string</u> is one consisting entirely of terminals. A
<u>sentential</u> <u>form</u> is any string derivable from S. A <u>sentence</u> is any terminal
sentential form. The <u>language</u> L(G) generated by G is the set of sentences;
i.e., $L(G) = \{\eta \in V_T^* \mid S \to^* \eta\}$. A <u>right</u> <u>sentential</u> <u>form</u>, which we choose as
our <u>canonical</u> <u>form</u>, is any string canonically derivable from S.

An example of a canonical derivation of a string η_1 in $L(G_1)$ follows,
where in each canonical form we underline the rightmost nonterminal and
indicate the production used to derive the next form.

Canonical Form	Production	
$\underline{S}$	(0)	$S \to \vdash E \dashv$
$\vdash \underline{E} \dashv$	(1)	$E \to E + T$
$\vdash E + \underline{T} \dashv$	(4)	$T \to P$
$\vdash E + \underline{P} \dashv$	(5)	$P \to i$
$\vdash \underline{E} + i \dashv$	(2)	$E \to T$
$\vdash \underline{T} + i \dashv$	(3)	$T \to P \uparrow T$
$\vdash P \uparrow \underline{T} + i \dashv$	(4)	$T \to P$
$\vdash P \uparrow \underline{P} + i \dashv$	(5)	$P \to i$
$\vdash \underline{P} \uparrow i + i \dashv$	(5)	$P \to i$
$\vdash i \uparrow i + i \dashv = \eta_1$		

Note that a canonical derivation is a strictly right-to-left process since we
always replace the rightmost nonterminal.

We assume that grammar G has no <u>useless</u> productions; i. e., we
assume that for each production $A \to \omega$ there exists a derivation
$S \to^{*} \sigma\delta\beta$ where σ, δ, and β are terminal strings. Presumably our language
designer has made an error if there are useless productions in the grammar.
Fortunately, well known methods exist for detecting such errors (see(Gin 66),
section 1. 4).

Loosely speaking, a <u>parse</u> of a string is some indication of how that
string was derived. In particular, a <u>canonical parse</u> of a sentential form
α is the reverse of the sequence of productions (or equivalently, the numbers
thereof) used in a canonical derivation of α. We refer to the action of
determining a parse as <u>parsing,</u> the determination constitutes a <u>grammatical</u>
<u>analysis</u>, and a parsing algorithm is called a <u>parser.</u>

Being interested for the present in grammatical analysis, we view a
grammar G as serving two purposes: (i) it is a set of rules for generating
the sentences in L(G), and (ii) it defines the input/output relations of any
corresponding canonical parser; i. e., if the input to the parser is a string
η in L(G), the output should be a canonical parse of η. However, because
the latter is ill defined in the case that η has several canonical parses and
because we desire ultimately to generate a unique translation of η from a
unique canonical parse, we are led to the following definition. A grammar

G is <u>unambiguous</u> if and only if each canonical form, and therefore each sentence, has a unique canonical parse. It follows immediately that each canonical form of an unambiguous grammar has a unique canonical derivation.

2.2 Characteristic Strings

We would like to describe a particular canonical parser, but first we define some strings which together provide a useful characterization of the decisions which must be made while parsing.

<u>Definition 2.1.</u> Let G be a CF grammar with $s + 1$ productions. Let $\{ \#_0, \#_1, \ldots, \#_s \}$ be a set of special symbols not in V, called <u>#-symbols</u>, such that $\#_0$ is associated with production 0, $\#_1$ with 1, ..., and $\#_s$ with s. Let the p-th production be $A \to \omega$, and let $\alpha' = \rho A \beta$ and $\alpha = \rho \omega \beta$ be canonical forms such that there exists a canonical derivation $S \to_R^* \alpha' \to_R \alpha$. Then $\rho \omega \#_p$ is a <u>characteristic string</u> of α. We call $\rho \omega$ the <u>stack string</u> of $\rho \omega \#_p$ and <u>a</u> stack string of α, and we call β an <u>input string</u> of α.

A characteristic string of α is, in essence, a summary of information about α useful for canonical parsing. It indicates that there exists a canonical derivation of α in which it is immediately preceded by another form α' which can be formed as follows: remove from the end of the stack string $\rho \omega$ the substring ω which matches the right part of production p,

replace ω with the left part A, and concatenate the result with the input
string β. We describe this procedure as "making a reduction" via
application of the "applicable production" to the end of the stack string.
In concert with this terminology we often refer to productions as
reductions, visualizing them written $\omega \rightarrow A$.

As examples we give several canonical forms of grammar G_1
with corresponding characteristic strings:

$$\eta_1 = \vdash i \uparrow i + i \dashv \qquad\qquad \vdash i \#_5$$
$$\vdash P \uparrow i + i \dashv \qquad\qquad \vdash P \uparrow i \#_5$$
$$\vdash P \uparrow P + i \dashv \quad \cdot \qquad\qquad \vdash P \uparrow P \#_4$$

Theorem 2.1: A CF grammar G is unambiguous if and
only if each canonical form α of G, except S, has a
unique characteristic string.

Proof: We exclude $\alpha = S$ because we defined no
characteristic string for it. Clearly S has a unique
canonical derivation so the exclusion does not effect
the following.

if part: To prove G is unambiguous we must show that
every canonical form has a unique canonical parse. We
proceed by induction, letting P_n be the proposition that
every canonical form derived in n steps has a unique

canonical parse. P_1 is true, because there is only
one derivation consisting of one step, namely
$S \to \vdash S' \dashv$. For some $n > 0$ we assume that P_n
is true and prove P_{n-1}. Consider a form α derived
in n+1 steps and having a unique characteristic string.
Every canonical derivation of α must end in the same
step $\alpha' \to \alpha$, for some α' derivable in n steps, by
definition of characteristic strings. Thus, any canonical
parse of α must be the production applied in $\alpha' \to \alpha$,
followed by some canonical parse of α'. But α' has
only one such parse, by the inductive hypothesis, so
α has only one canonical parse. Thus, G is unambiguous
by definition.

only if part: If G is unambiguous, each such α has a
unique canonical parse and derivation. Therefore by
definition it can have only one characteristic string.   Q. E. D.

2.3   A Canonical Parser

Our canonical parser is described simply as follows. Commencing
with string η in L(G), iteratively (i) determine a characteristic string of
the current canonical form, (ii) output the production indicated by the last

symbol of that characteristic string, (iii) make the corresponding reduction, and (iv) stop when the new canonical form is $\alpha = S$.

Several comments are in order with regard to this algorithm. First, it is incomplete since we have not stated how to determine characteristic strings. We investigate this problem thoroughly in Chapters 3, 4, and 5, but we solve it there only for a restricted class of CF grammars which we are about to define. Second, since these special grammars are all unambiguous, we can change part (i) to read "determine the characteristic string . . . " Thus, the algorithm is well defined, and deterministic, for the grammars of interest. Third, since each iteration is the reverse of a step in a canonical derivation, it is clear that the process as a whole is just the reverse of a canonical derivation. Thus, the parser proceeds strictly from left to right, except perhaps for the computation required to determine characteristic strings. This is, of course, precisely why we are interested in this particular parser.

A determination of the canonical parse $(5, 5, 4, 3, 2, 5, 4, 1, 0)$ of the string η_1 derived above is exemplified below, where we underline the reducible substring in each canonical form and characteristic string.

Canonical Form	Characteristic String	Output
$\vdash \underline{i} \uparrow i + i \dashv$	$\vdash \underline{i} \quad \#_5$	5
$\vdash P \uparrow \underline{i} + i \dashv$	$\vdash P \uparrow \underline{i} \quad \#_5$	5
$\vdash P \uparrow \underline{P} + i \dashv$	$\vdash P \uparrow \underline{P} \quad \#_4$	4
$\vdash \underline{P \uparrow T} + i \dashv$	$\vdash \underline{P \uparrow T} \quad \#_3$	3
$\vdash \underline{T} + i \dashv$	$\vdash \underline{T} \quad \#_2$	2
$\vdash E + \underline{i} \dashv$	$\vdash E + \underline{i} \quad \#_5$	5
$\vdash E + \underline{P} \dashv$	$\vdash E + \underline{P} \quad \#_4$	4
$\vdash \underline{E + T} \dashv$	$\vdash \underline{E + T} \quad \#_1$	1
$\vdash E \dashv$	$\vdash E \dashv \quad \#_0$	0
S		

We now informally prove that our canonical parser operates as desired for the purposes of compiling; i.e., that when it is applied to a string η in L(G) it outputs the canonical parse of η and stops, and that when it is applied to a string η' not in L(G) it aborts somehow after a finite time. The former follows from the fact the parser executes the reverse of a canonical derivation. The latter depends on the fact no canonical derivation exists for any such string η', **and** on the following two assumptions. First, we assume that there is an auxiliary mechanism, a "loader" program, say, which checks all strings presented to the parser and ensures that the first and last symbols are $\vdash$ and $\dashv$, respectively. Second, we assume that whatever device is used to determine characteristic strings never looks to the left of $\vdash$ or to the right of $\dashv$. The way the parser must abort, then, is by determining that there is no characteristic string for the string from $\vdash$ to $\dashv$, inclusive. It is a finite

task to determine that η' of length n has no characteristic string because, if nothing else, we could simply generate all strings of length n using G and determine that η' is not one of these strings. Of course, the exact way in which our parser aborts will not become clear until we develop a device for determining characteristic strings.

2.4 LR(k) Grammars

In hopes of being able to develop practical parsers for them, we now restrict our attention to those CF grammars whose sentences can be parsed deterministically during a single scan from left to right.

Definition 2.2. Let k be a non-negative integer. A CF grammar G is LR(k) if and only if every canonical form $\alpha = \varphi\beta$ of G, except $\alpha = S$, has a unique characteristic string $\varphi\#_p$ which can be determined by investigating only ρ and $k:\beta$.

The original definition of LR(k) grammars appeared in (Knu 65). A definition very like our own can be found in (H&U 69).

Theorem 2.2. An LR(k) grammar is unambiguous.

Proof: The uniqueness of characteristic strings in conjunction with Theorem 2.1 proves this.   Q. E. D.

We have already seen that our canonical parser proceeds strictly from left to right as far as the making of reductions is concerned. The

implication of our definition is that for LR(k) grammars the process for determining characteristic strings need never get more than k symbols ahead of the reduction process. Further, if sufficient information can be remembered about the string already processed, no rescanning of that string is necessary and the parser as a whole may proceed from left to right, except when the process for determining characteristic strings peeks ahead as many as k symbols. We show in Chapters 3, 4, and 5, that sufficient information can be remembered via a finite number of machine states and a pushdown stack, and in fact, that our parser is equivalent to a DPDA.

We emphasize that the LR(k) definition allows parsing decisions to depend on arbitrarily large left context (φ) but only on finite right context $(k{:}\beta)$. Thus it defines the largest possible set of grammars consistent with our deterministic, left-to-right bent. This because no additional information about the parsing decisions which were made to reduce the left part of the original string to φ would be of any use in making new decisions, since we are concerned only with context-free grammars. In other words, none of the "substructure" associated with φ is relevant to any future parsing decisions.

As an example of an LR(0) grammar, consider G_0 whose productions follow.

(0) $S \to \vdash E \dashv$ (4) $E \to b\,B$

(1) $E \to a\,A$ (5) $B \to c\,B$

(2) $A \to c\,A$ (6) $B \to d$

(3) $A \to d$

Because G_0 is small and simple it is easy to confirm that it is, indeed, LR(0). The canonical forms of G_0 are indicated in the following two derivations, where $n \geq 0$:

$$S \to \vdash E \dashv \to \vdash a\,A \dashv \to \ldots \to \vdash a\,c^n A \dashv \to \vdash a\,c^n d \dashv$$
$$S \to \vdash E \dashv \to \vdash b\,B \dashv \to \ldots \to \vdash b\,c^n B \dashv \to \vdash b\,c^n d \dashv$$

Since these represent all possible derivations, it is easy to see from definition 2.1 that the corresponding characteristic strings are unique, and as follows:

$$\vdash E \dashv \#_0, \ \vdash a\,A\#_1, \ldots, \ \vdash a\,c^n A \#_2, \ \vdash a\,c^n d \#_3$$

$$\vdash E \dashv \#_0, \ \vdash b\,B\#_4, \ldots, \ \vdash b\,c^n B \#_5, \ \vdash a\,c^n d \#_6$$

Further, it can easily be determined by exhaustive testing that the characteristic string $\varphi \#_p$ of each canonical form $\alpha = \varphi\beta$ can be determined without regard to any right context (β). Thus, G_0 is LR(0). We shall prove this in a more satisfying way in Chapter 3.

Now, because G_0 _is'_ LR(0), we can generate a parser for it via the simplest version of our technique, as we shall see. However, of all the parser-generating techniques discussed in (F&G 68), Knuth's is the only one which covers G_0. This is because the most general of the other techniques covers only the "bounded right context" grammars (Flo 64); i e., grammars whose sentences can be parsed from left to right with no decisions depending on more than a bounded amount of left _or_ right context.

To see that G_0 is not bounded right context, consider the string $\eta_0 = \vdash a\, c^n d \dashv$. To parse η_0 the reduction which must be made first is $d \to A$. But that decision depends on the fact there is an "a" arbitrarily far to the left. Had the "a" been "b" instead, the applicable reduction would have been $d \to B$.

We illustrate that our previous example grammar G_1 is _not_ LR(0) by exhibiting two similar canonical forms of G_1 which have distinct characteristic strings:

Canonical Forms	Characteristic Strings	Reductions
$\vdash \underline{P} + i \dashv$	$\vdash \underline{P}\, \#_4$	(4) $P \to T$
$\vdash P \uparrow \underline{i} \dashv$	$\vdash P \uparrow \underline{i}\, \#_5$	(5) $i \to P$

Given the canonical form $\alpha = \vdash P + i \dashv$, we could not conclude on the basis

of the prefix "$\vdash P$" alone that the characteristic string of α is "$\vdash P \#_4$".

We should have to look one symbol ahead to be sure α is not "$\vdash P \uparrow i \dashv$"

or the like. Because elimination of such uncertainties as these can always

be effected by a look-ahead of one symbol, G_1 is an LR(1) grammar. We

prove this in Chapter 4.

Of course, our parser need not look ahead in unambiguous situations.

For instance, there is never any uncertainty about whether "i" should be

reduced to "P" for grammar G_1, no matter what the context. This fact

illustrates that the smallest k for which a grammar is LR(k) is limited by

the worst case of necessary look-ahead.

As examples of grammars which are not LR(k) for any $k \geq 0$, we

could choose any ambiguous grammar. The violation of Theorem 2.2 is

immediate. Neither is the mirror image of grammar G_0 LR(k). This is

because the "d" would now appear on the left end of each sentence, and we

would need arbitrary right context to choose between the reductions $d \rightarrow A$

and $d \rightarrow B$.

This latter case suggests the concept of RL(k) grammars, whose

sentences can be parsed deterministically from right to left. We do not

pursue this concept further since the generalization is obvious.

2.5   The Meaning of the LR(k) Condition

We emphasize the fact that the LR(k) condition is one on the grammar, not the language.   For instance, the grammar $S \rightarrow \vdash E \dashv$, $E \rightarrow aEa$ $E \rightarrow a$ is not LR(k) for any k, but the language it generates $\vdash a \{aa\}^* \dashv$ is regular, and therefore recognizable by an FSM.   The grammar not being LR(k) corresponds to the fact the strings cannot be parsed by a DPDA.   There does, however, exist an LR(k) grammar which generates the same language. In fact, Knuth has shown that there exists an LR(k) grammar for every deterministic language; i. e. , every language which can be recognized by a DPDA has a grammar such that the sentences can be parsed by a DPDA.

The latter fact is only of somewhat academic interest from our point of view because we are ultimately interested in using grammars to specify translations from strings into structures, so we are as interested in the structural properties of grammars as we are in the languages they generate. The case just given is one where no LR(k) grammar exists which has the symmetrical structural property of the original grammar.   This corresponds to the fact that no DPDA could determine the center of an arbitrarily long string without looking arbitrarily far ahead to find the end of the string.

It is also of some academic interest that any "LR(k) language", i. e. one generated by an LR(k) grammar, can also be generated by an LR(0) grammar.[†]

[†] In (Knu 65) the result is that there is an LR(1) grammar for each LR(k) language, but this is because Knuth does not assume the left and right "pad" symbols to be built into the grammar.   One-symbol look-ahead is therefore necessary to detect the end of the string.

This fact is another which is not very interesting from our viewpoint because it has not been shown, and indeed, we suspect that it is not true, that an LR(0) grammar exists which is "structurally equivalent" to the original grammar. (See (Che 67) for a precise definition of the "structural equivalence" of grammars.)

2.6   Terminology in Automata Theory
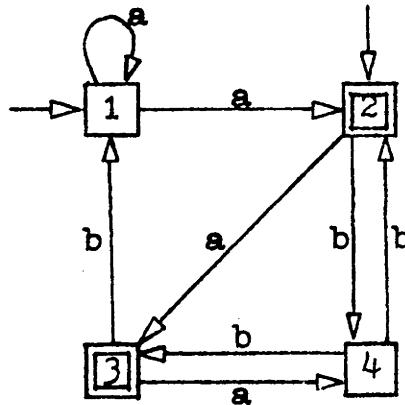
The following is intended only as a review of terminology, since we assume that the reader is already familiar with the concepts. However, the reader should pay special attention to the discussion of DPDAs, because our representations of them are unusual. We first discuss a link between formal grammars and automata theory.

A production is said to be right linear (Gin 66) if it is of the form $A \rightarrow \omega B$ or $A \rightarrow \omega$, where A and B are in V_N and ω is in V_T^*. A CF grammar is called right linear if all of its productions are right linear. A right linear grammar G_R is said to generate a regular language, and it is well known that the latter can be recognized by an FSM which can be derived from G_R (H&U 69).

FSMs. Formally, an FSM (Hen 68) is an abstract model consisting of a finite set of input symbols, a finite set of output symbols, a finite set of states, a next-state function, and an output function. For our purposes an FSM need only be a recognizer, so the output symbols need include only "1" and "0", or "yes" and "no". We consider an FSM to be synonymous

with one of its representations, namely a "transition graph", and we discuss FSMs in terms of the latter rather than in terms of the above five components.

A transition graph consists of a set of nodes with various arrows drawn between them. Each node represents a state and is indicated thus $\boxed{N}$, where N is the name of the state (we use integers for state-names). Each arrow is labeled with an input symbol s; it is said to be a transition under s, or simply an s-transition, and it represents an element of the next-state function. A starting state is indicated by a short incoming arrow which originates on no node of the graph. A terminal state is indicated thus $\boxed{\boxed{}}$. An example of an FSM (transition graph) is as follows.

A series of transitions leading through an FSM from state N_1, to state N_2... to state N_k is called a path from N_1 to N_k. Every such path spells out a unique string of input symbols (i. e., an input string) in the obvious way. An FSM accepts a given string η if and only if there exists at least one path that begins at a starting state, spells out η, and ends at a terminal state. The set of all strings accepted by an FSM is referred to as the set that is recognized by that FSM.

A state M is said to be <u>accessible</u> from state N if and only if there is a path from N to M; the input string spelled out by such a path is said to <u>access</u> M from N. When the initial state is not specified, it is understood to be a starting state.

If we associate the output symbol "1" with each terminal state and "0" with each of the others, each path also spells out a unique string of output symbols (i. e., an <u>output string</u>). States M and N are said to be <u>equivalent</u> if and only if for each input string η spelled out by some path from M (N), such that the path also spells out the output string η', there exists a path from N (M) which spells out the same two strings η and η', respectively.

An FSM is said to be <u>deterministic</u> if and only if it has a single starting state and from each state there is at most one transition under each distinct input symbol; otherwise, it is said to be <u>nondeterministic</u>. A deterministic FSM is said to be <u>reduced</u> if and only if every state is accessible from the starting state, some terminal state is accessible from every state, and no two states are equivalent. A reduced machine is unique within the names of its states, and, since it is a homomorphic image of other machines which recognize the same set, it can in a real sense be thought of as <u>minimal.</u>

We often think of a deterministic FSM as a physical machine, rather than as an abstract model, and this leads to the following terminology. To determine if a given FSM accepts a given string η, we say that we <u>initialize</u>

the machine (i.e., start it in its starting state), apply it to η, and determine if η takes the machine through a sequence of states to a terminal state. The machine is said to read the symbols in η from an input tape, to enter first one state and then the next, and to output symbols onto an output tape. If after reading the last symbol of η the machine outputs a "1", then it accepts η. However, if at that time it outputs a "0" or if it stops reading before it reaches the end of η, it does not accept η. The machine stops reading whenever it enters a state with no transition under the next symbol to be read.

DPDAs. Our treatment of DPDAs is less formal than that of FSMs. For our purposes a DPDA is a machine consisting of an input tape, an output tape, a finite control, and a pushdown stack.

The finite control can be thought of as a program consisting of instructions pertaining to the reading of symbols from the input tape and the outputting of symbols onto the output tape, the storage, interrogation, and removal of items on the stack, and jumps from one point in the program to another. The control can be represented by a transition graph whose nodes (we use circular nodes for DPDAs) are called states and whose labeled arrows are called transitions.

Each state represents a point in the program which can be jumped to, and it has a name which is given inside the node. There is a unique starting state, indicated thus ⬤ and a unique terminal state, indicated thus ◎ .

Each transition implies one of four kinds of <u>instructions</u>, the interpretations of which are indicated next. If the machine enters state N having a transition to state M, then, if the label of the transition is (1) a symbol s, the machine reads the next symbol and, if the symbol read is s, it then enters state M, (2) "push i", the machine pushes the item i on the stack and then enters state M, (3) "pop n, out p", the machine pops the top n items off the stack, outputs p, and then enters state M, or (4) "top i", the machine compares item i with the top item on the stack,[†] and, if they are the same, it then enters state M.

The following two conditions are sufficient to guarantee <u>determinism</u>: (1) any state having a transition under either "push i" or "pop n, out p" may have no other transitions, and (2) any other state must have either every transition under a symbol, or every one under "top i" for some item i.

The <u>initial configuration</u> of a DPDA is as follows. It is started in its starting state with the <u>input string</u> (the string to be parsed, in our case) on its input tape, with its <u>input head</u> (reading device) over the leftmost symbol (⊢) of the input string, and with its stack empty. The <u>final configuration</u>

[†] Our special application of DPDAs has prompted us to depart from the usual restrictions (D&D 69) of allowing "pops" of only one symbol at a time from the stack, and investigations of items on the stack only when popping them off. Also, outputs are usually associated with states, as in the case of FSMs. We believe it is obvious how to modify our DPDAs to abide by these restrictions. We have deviated from the norm for the sake of simplicity and practicality.

is: the input head one place to the right of the rightmost symbol ($\dashv$) of the input string, the stack empty, and the machine in its terminal state.

The similarity of DPDAs and FSMs is emphasized if we note that a DPDA which never uses its stack is equivalent to some FSM. This leads us to think of a DPDA, then, as being based on some FSM. We think of this FSM as reading symbols, as usual, but interspersed between some of the reads are some "bookkeeping" operations involving the stack, and these operations effect some of the state changes of the FSM. This viewpoint proves to be quite useful in Chapters 3, 4, and 5.

Chapter 3

PARSERS FOR LR(0) GRAMMARS

3.1   Perspective

Chapters 3, 4, and 5 are difficult ones to read because they contain

many detailed definitions, lemmas, theorems and corollaries, and intricate

proofs. But alas, the difficulties cannot be circumvented entirely because

the material covered is fundamental to the dissertation and must be precise

and proven, and because it is distinctly nontrivial. We can, however,

minimize problems by providing perspective via an informal preview of

the results to come.

The objective of the present chapter is merely to show how to construct

parsers for LR(0) grammars, but in the process we lay a foundation upon

which we ultimately build to cover all LR(k) grammars.

We begin by showing that the set of characteristic strings of a

given CF grammar G is a regular language. Thus, the set can be recognized

by an FSM. We next show that if G is LR(0) the reduced, deterministic FSM

which does this recognition is adequate, without modification, for use in

parsing. In particular, the FSM can be used to determine characteristic

strings of canonical forms, as is necessitated by our parsing algorithm.

It follows rather directly that the parsing algorithm as a whole can be

converted to a DPDA, the finite control of which can be derived directly

from the FSM.

In Chapter 4 we define the "Simple LR(k)" grammars; i. e., those grammars for which the special FSMs can be used to determine characteristic strings if they are extended by the addition of certain "look-ahead" information which can be computed in a simple way. The conversion of the modified FSM to a DPDA is straightforward, simply resulting in a DPDA with "look-ahead".

In Chapter 5 we address the problem of constructing parsers for general LR(k) grammars. We find that in some of these cases the modification needed for the FSM is the same as above, but that the "look-ahead" information is more difficult to compute than for the "Simple LR(k)" grammars. In the general LR(k) case, however, some of the states of the FSM must be split into several copies because of complex correspondences between left and right contexts. The state splitting process is explained simply as "building into the machine" the capability to remember more left context so that the corresponding right contexts can be checked to make parsing decisions. Thus, the construction of the parser in the general case can become computationally complex.

In conclusion, what we develop in the next three chapters is a method for constructing parsers which grows in complexity as it discovers the complexity of the grammar it is working on. That is, we first assume the grammar is LR(0) and set out to generate a parser for it. In the process of constructing the parser we are able to determine if the grammar

is, indeed, LR(0). If it is, we complete our construction and are finished. However, if the grammar is not LR(0), we assume it is "Simple LR(k)" and compute the "look-ahead" information in a simple way. If certain conditions do not hold regarding this "look-ahead", we use more complex methods and perhaps discover that some state splitting is necessary. Ultimately, we are able to determine if a given grammar is LR(k) for any finite value of k given a priori[†], and if it is, we can construct a parser for it.

3.2 Foundation

To complete the specification of our canonical parser we develop an automaton which is capable of determining characteristic strings. We first concentrate on LR(0) grammars and then gradually generalize to include all LR(k) grammars. The following theorem, regarding both amgibuous and unambiguous grammars, is fundamental to our development.

Theorem 3.1. The set of characteristic strings of a given CF grammar $G = (V_T, V_N, S, P)$ is a regular language.

Proof: Consider a canonical derivation of some canonical form α:

[†] Knuth (Knu 65) has shown that it is undecidable, in general, whether a grammar is LR(k) if k is not given a priori.

$$S \rightarrow$$
$$(S \rightarrow \omega_0 A_0 \omega_0')$$

$$\omega_0 A_0 \omega_0' \rightarrow$$

$$\quad \vdots$$
$$\quad (\omega_0' \rightarrow_R^* \omega_0'', \ \omega_0'' \in V_T^*)$$
$$\quad \vdots$$

$$\omega_0 A_0 \omega_0'' \rightarrow$$

$$(A_0 \rightarrow \omega_1 A_1 \omega_1')$$

$$\omega_0 \omega_1 A_1 \omega_1' \omega_0'' \rightarrow$$

$$\quad \vdots$$
$$\quad (\omega_1' \rightarrow_R^* \omega_1'', \omega_1'' \in V_T^*)$$
$$\quad \vdots$$

$$\omega_0 \omega_1 A_1 \omega_1'' \omega_0'' \rightarrow$$

$$\quad \vdots \qquad \vdots$$
$$\quad \vdots \qquad \vdots$$
$$\quad \vdots \qquad \vdots$$

$$\omega_0 \omega_1 \cdots \omega_m A_m \omega_m'' \cdots \omega_1'' \omega_0'' \rightarrow$$

$$((p) \ A_m \rightarrow \omega)$$

$$\omega_0 \omega_1 \cdots \omega_m \ \omega \ \omega_m'' \cdots \omega_1'' \omega_0'' \ = \alpha$$

where $m \geq 0$, $A_m \rightarrow \omega$ is the p-th production in P, and

for $0 \leq i < m$ each ω_i'' is in V_T^*, each ω_i and ω_i' is in V^*

(recall that $V = V_T \cup V_N$), and each $A_i \rightarrow \omega_{i+1} A_{i+1} \omega_{i+1}'$ is

a production in P. Then a characteristic string of α is

$$\omega_0 \omega_1 \cdots \omega_m \omega \#_p.$$

This string can be generated by a grammar containing

the right linear productions:

$$S' \rightarrow \omega_0 A_0'$$
$$A_0' \rightarrow \omega_1 A_1'$$
$$\vdots$$
$$A_m' \rightarrow \omega \#_p$$

where S' and the A_i' are the nonterminals in this grammar.

Generalizing, we see that the following right linear grammar generates all possible characteristic strings of G:

$$F' = (V_T', \ V_N', \ S', \ P')$$

where

$$V_T' = V \cup \{ \#_0, \#_1, \ldots, \#_s \} \text{ and}$$

$$V_N' = \{ A' \mid A \text{ is in } V_N \} \text{ and}$$

$$S' = S \text{ primed and}$$

$$P' = \{ A' \to \omega \#_p \mid A \to \omega \text{ is the p-th production in } P \}$$

$$\cup \{ A' \to \omega_1 B' \mid A \to \omega_1 B \omega_2 \text{ is in } P \text{ and } B \text{ is in } V_N \}$$

Further, because there are no useless productions in G there corresponds to each derivation of a string $\varphi \#_p$ using grammar F' derivations using grammar G of one or more canonical forms, each of which has $\varphi \#_p$ as a characteristic string. Thus, the grammar F' generates all and only the characteristic strings of G.

Finally, F' generates a regular language because it is a right linear grammar.   Q. E. D.

Definition 3.1. The grammar F' of the proof of Theorem

3.1 is called the <u>characteristic grammar</u> of G.

As an example we present the productions of the

characteristic grammar of our example grammar G_0 (see page 29):

(0) $S' \rightarrow \vdash E \dashv \#_0$

(1) $S' \rightarrow \vdash E'$

(2) $E' \rightarrow a \, A \, \#_1$

(3) $E' \rightarrow a \, A'$

(4) $A' \rightarrow c \, A \, \#_2$

(5) $A' \rightarrow c \, A'$

(6) $A' \rightarrow d \, \#_3$

(7) $E' \rightarrow b \, B \, \#_4$

(8) $E' \rightarrow b \, B'$

(9) $B' \rightarrow c \, B \, \#_5$

(10) $B' \rightarrow c \, B'$

(11) $B' \rightarrow d \, \#_6$

3.3   <u>CFSMs: Characteristic FSMs</u>

We now concentrate on a particular FSM which can be derived from

a characteristic grammar.

Definition 3.2.   A <u>CFSM</u> (<u>characteristic FSM</u>) of a CF

grammar G is a reduced, deterministic FSM which

recognizes the set of characteristic strings of G.

Since any such FSM is unique within the names of its states we refer to <u>the</u>

CFSM of G. The CFSM can be derived from the characteristic grammar

of G via well known techniques (see for example, (H&U 69) page 33) or it

can be derived directly grom G, as we discuss in detail in Section 7.1.

We illustrate in Figure 3.1 the CFSM of our LR(0) grammar G_0. It is the CFSM which is capable of determining the characteristic strings of canonical forms for an LR(0) grammar and an extension of it which is so capable for an LR(k) grammar. However, the proofs of these statements require several preliminary results.

In the sequel we use **#-transition** to mean a transition under a #-symbol.

> Lemma 3.2. Several properties of the CFSM of a CF grammar G are as follows: (1) it has a single starting state, (ii) every state is accessible from the starting state, (iii) every #-transition is to a unique terminal state T, such that there are none other than #-transitions to T and such that there are no transitions from T, and (iv) the terminal state is accessible from every other state.
>
> Proof: (i) the machine is deterministic, (ii) the machine is reduced, (iii) every string accepted by the machine has exactly one #-symbol and it is the last symbol in the string; thus, any terminal state must have none other than #-transitions to it, and it must not have any transitions from it; there is a **unique** terminal state because the machine is reduced, and (iv) the machine is reduced. Q. E. D.
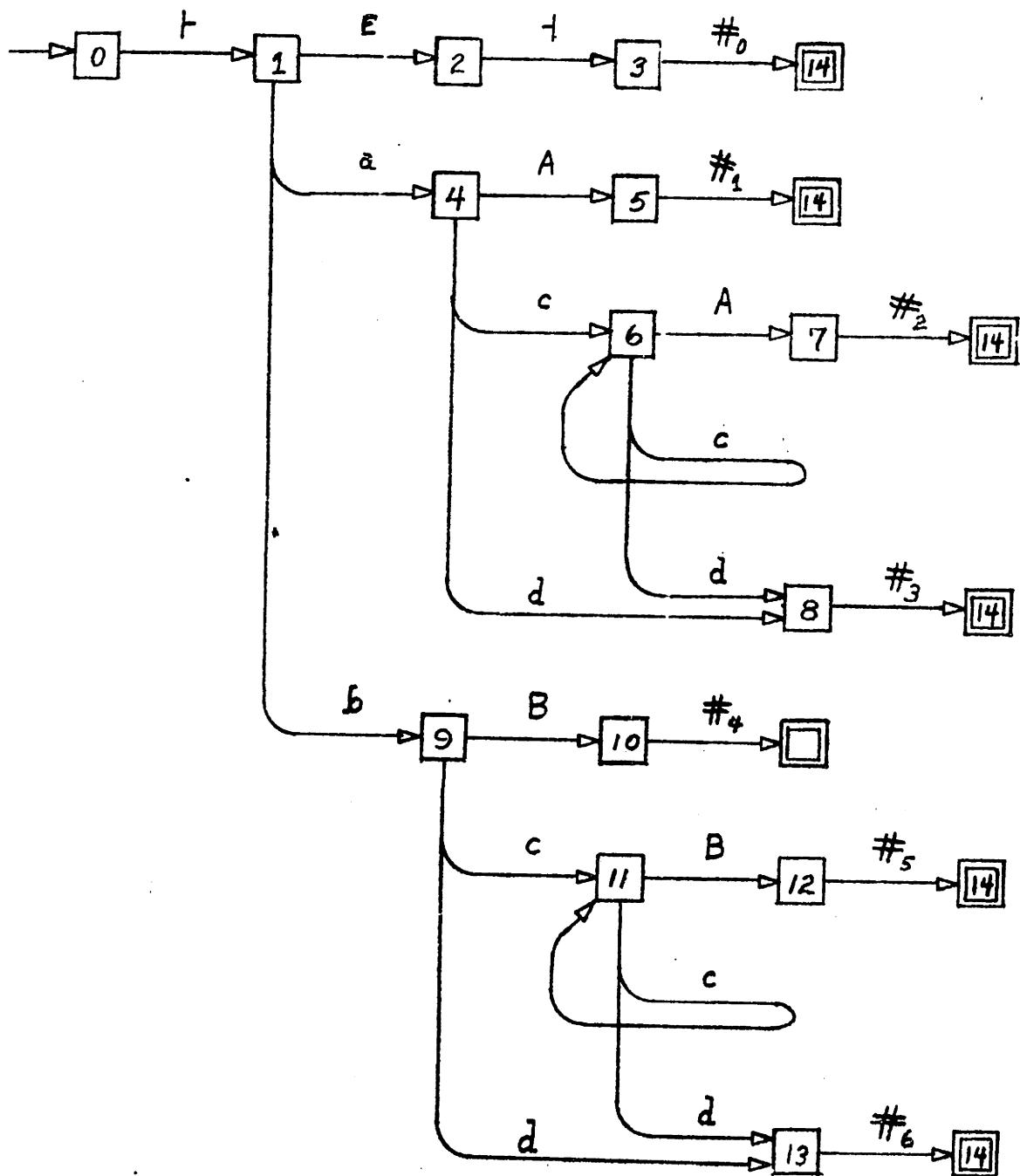
Figure 3.1. The characteristic FSM of our example grammar G_0: (0) $S \rightarrow \vdash E \dashv$, (1) $E \rightarrow a\ A$, (2) $A \rightarrow c\ A$, (3) $A \rightarrow d$, (4) $E \rightarrow b\ B$, (5) $B \rightarrow c\ B$, (6) $B \rightarrow d$. Although [14] appears at several locations above, it is to be taken as the unique terminal state.

Now we give convenient and, as we shall see presently, meaningful names to all the states of such an FSM, except for the terminal state.

Definition 3.3 Any state having no #-transitions is called a read state

Definition 3.4 Any state whose only transition is a #-transition is called a reduce state.

Definition 3.5 Any state having two or more transitions at least one of which is a #-transition, is called an inadequate state. In the case of a state with more than one #-transition, we sometimes refer to it as multiply inadequate.

The lattermost definition motivates the following one.

Definition 3.6 A CFSM with no inadequate states is said to be adequate, otherwise it is said to inadequate.

3.4 Parsers for LR(0) Grammars

Preliminaries. The following lemma is a concise and useful statement of the LR(k) condition specialized to the case k = 0. It provides a way to decide if a grammar G is LR(0) by checking properties of its characteristic strings, rather than of its canonical forms. This is a decided advantage. Informally, the lemma means that, if the stack string of one characteristic string is a prefix of another characteristic string, then G is not LR(0).

Lemma 3.3. Let G be a CF grammar. Let $\varphi_1 \#_p$ and $\varphi_2 \#_q$ be any two characteristic strings of G such that $\varphi_1 = \varphi_2 = \varphi$. Then G is LR(0) if and only if $\theta = \epsilon$ and q = p.

Proof: Our proof depends on the fact that by definition of

characteristic strings there correspond to $\varphi\#_p$ and

$\varphi\theta\#_q$ canonical forms $\alpha_1 = \varphi\beta_1$ and $\alpha_2 = \varphi\theta\beta_2$,

respectively, for some β_1 and β_2 in V_T^*.

if part: If $\theta = \epsilon$, and $q = p$, then α_1 and α_2 have the

same characteristic string $\varphi\#_p$. Consider the case

$\alpha_1 = \alpha_2 = \alpha$. This implies every canonical form α

has a unique characteristic string. Consider the case

$\alpha_1 \neq \alpha_2$. If we were given α alleged to be either α_1 or α_2,

we could determine the characteristic string $\varphi\#_p$ of α by

investigating only φ. Since α_1 and α_2 can be any canonical forms

as given above, we have shown that G is LR(0) by definition.

only if part: If G is LR(0) then, if $\alpha_1 = \alpha_2 = \alpha$, we

must have $\theta = \epsilon$ and $q = p$, since each canonical form

α has a unique characteristic string. If $\alpha_1 \neq \alpha_2$ and

if $\theta = \epsilon$ and/or $q \neq p$, then α_1 and α_2 have distinct

characteristic strings, and given α alleged to be

either α_1 or α_2, we could not determine the charac-

teristic string of α on the basis of φ alone. Since this

is a contradiction of the LR(k) definition for k = 0,

we again have $\theta = \epsilon$ and $q = p$.   Q. E. D.

We use this lemma immediately to verify another and still more useful method for deciding if a grammar is LR(0).

Theorem 3.4.   A CF grammar G is LR(0) if and only if its CFSM is adequate.

Proof: if part: If the CFSM is adequate and if it accepts the string $\varphi\#_p$, then it cannot accept the string $\varphi\theta\#_q$ for $\theta = \epsilon$ and/or $q = p$.   For if it did, the state accessed by φ would be inadequate, having distinct transitions under $\#_p$ and $1\!:\!\theta\#_q$.   G is therefore LR(0) by the "if part" of Lemma 3.3.

only if part: By Lemma 3.2, part (ii), each state N of G's CFSM is accessible by some string φ.   Assume that N has a $\#_p$-transition; i.e., that the CFSM accepts $\varphi\#_p$.   If N had another distinct transition, it would be either to the terminal state or to a state from which the terminal state is accessible, by Lemma 3.2, part (iv). Thus, the machine would also accept $\varphi\theta\#_q$ for some $\theta \neq \epsilon$ and/or $q \neq p$.   But by the "only if part" of Lemma 3.3, $\varphi\#_p$ and $\varphi\theta\#_q$ cannot both be characteristic strings, i.e.   the CFSM cannot accept both, unless $\theta = \epsilon$ and $q = p$.   Thus, any such N must have only the $\#_p$-transition, and the CFSM is adequate by definition.   Q. E. D.

Thus, we have proved that our example grammar G_0 is LR(0) by exhibiting its CFSM (Figure 3.1), which is adequate by inspection.

Parsers. We now prove that for the special case of an LR(0) grammar, the corresponding CFSM is capable of determining the characteristic strings of canonical forms.

Theorem 3.5. Let G be an LR(0) grammar and $\alpha = \varphi\beta$ be a canonical form of G with characteristic string $\varphi\#_p$.

The stack string φ accesses a reduced state of G's CFSM whose only transition is under $\#_p$.

Proof: The CFSM accepts the string $\varphi\#_p$. Thus, φ accesses a state N with a transition under $\#_p$.

But since G is LR(0), Theorem 3.4 implies N is not an adequate state. Therefore, N must be a reduce state whose only transition is under $\#_p$.   Q. E. D.

Parsing algorithm. Thus for an LR(0) grammar G our parsing algorithm can be restated as follows. Commencing with $\alpha = \eta$, where η is a string in L(G), and with the CFSM of G:

(i) Initialize the CFSM and apply it to the current canonical form α. When the machine enters a reduce state R, it will have read the stack string φ of α and will have left to read the input string β of α.

(ii) The only transition from R must be under $\#_p$ for some production p, so output p.

(iii) Apply reduction p to the end of φ and concatenate the result and β to form the next canonical form α.

(v) If the new form is $\alpha = S$ then stop; otherwise start at step (i) again.

Note that in this algorithm characteristic strings are determined without checking the entire string. Thus, in general, when it is applied to a string η' not in L(G), it goes through several iterations, making reductions on the left part of η', but it ultimately aborts when the CFSM is applied to a string $\alpha' = \varphi'\beta'$ such that φ' accesses a state with no transition under $1:\beta'$; i.e., when the CFSM stops reading. This must be the case because there is no other way for the algorithm to fail, and because if it were successful, that would imply there exists a canonical parse of η'. (Recall the discussion at the end of Section 2.3.)

Obviously this parser is neither efficient nor strictly left-to-right since it starts back at the beginning of the stack string at each iteration. We now solve these two problems by converting our string-manipulation algorithm to a DPDA.

3.5 Conversion of the Parsers to DPDAs

Our conversion technique is most easily understood if it is presented in two steps. We first convert our parser to a "stack algorithm"; i.e., an algorithm incorporating a pushdown stack. The use of the stack eliminates the need for rescanning the stack string at each iteration. Then we give a

technique for converting the CFSM to the finite control of a DPDA, such that the DPDA simulates the stack algorithm.

Consider an iteration of our parsing algorithm. We begin with some canonical form $\alpha' = \rho\omega\beta$ whose characteristic string is $\rho\omega\#_p$. We apply the CFSM to α'. The string $\rho\omega$ is read, the CFSM enters a reduce state, and the characteristic string is determined. If production p is $A \rightarrow \omega$, we replace ω with A to form $\alpha = \rho A\beta$ and start anew.

Now, on the next iteration the first action of the CFSM is to read ρ again. But the CFSM is deterministic and will therefore go through the same sequence of states while reading ρ this time as it did on the previous step. Thus, had we remembered in the previous step the state N of the CFSM immediately after reading ρ, we could in this step merely start the CFSM in N and apply it to $A\beta$ to get the desired result.

The stack algorithm: To eliminate the rescanning of the stack string at each iteration we use a pushdown stack. As the CFSM reads a canonical form we push onto the stack the names of the states entered by the CFSM. Upon determining the characteristic string, say $\rho\omega\#_p$ where production p is $A \rightarrow \omega$, we pop the top $|\omega|$ state-names off the stack and output p. We then return the CFSM to the state whose name is at the top of the stack (determining the top name is called looking back) and continue the process by reading $A\beta$. The process ends when the string to be read is simply S.

It should be clear, in light of the two paragraphs preceding, the algorithm, that the stack algorithm is equivalent in effect to our previous algorithm. However, it is more efficient than the previous one.

We emphasize, for reasons which will become apparent shortly, that the sequence of state-names stored in the stack at a particular time T represents a path through the CFSM. The path is the one which would be taken by the CFSM were it to be applied to the prefix which is implicitly the left context at time T. This property is the basis of several observations which we make below.

Note that at this stage we have substantially departed from the string-manipulation notions with which we began. Our stack algorithm has no further interactions with symbols after it has read them. Instead, it interacts with the state-names of the CFSM. We now move another step away from our original parsing notions by coverting the stack algorithm plus CFSM to a DPDA.

The conversion technique. We consider the CFSM to be the basis from which we construct the finite control of our DPDA. Since both FSMs and finite controls can be represented by transition graphs, the technique can be described as a piecewise conversion of one graph into another.

We think of the CFSM-graph as a skeletal program which we must convert to a detailed program (finite control) by filling in more instructions. The basic structure and the read instructions are already in the program,

and we must add the stack-manipulation instructions. Our guide to this programming task is, of course, the stack algorithm.

For each state N of the CFSM there is a state named N in the DPDA, such that the actions of the DPDA immediately subsequent to entering state N are similar to the actions of the stack algorithm when the CFSM is in state N. The CFSM can be converted to the appropriate finite control by applying to it the three transformations indicated in Figure 3.2.

Figure 3.2a indicates a transformation for replacing #-transitions with "reduction procedures". Consider a reduce state R corresponding to production p, $A \rightarrow \omega$. We replace the $\#_p$-transition from R with a transition under "pop $|\omega|$, cut p" to a new look-back state R'. There is one transition from R' under "top N" to state M for each pair (N, M) in the set Q, where $Q = \{(N, M) \mid$ there exists an A-transition from N to M and a path from N to R which spells out $\omega\}$.

Note that there is an optimization implicit in this transformation. The reduction procedure executed by the stack algorithm can be described via the following sequence: "pop $|\omega|$, out p"; look back and see N; return to the CFSM to state N; read A (which causes the CFSM to enter state M). However, the reduction procedure for the DPDA is simply: "pop$|\omega|$, out p"; look back and see N; enter state M. That is, the DPDA does not manipulate the nonterminal A. The optimization might be described as precomputing part of the reduction procedure and "wiring the results into the machine".

(a)

(exception: if p=0 then $\boxed{R} \xrightarrow[\text{out } 0]{\text{pop } 4} \circledcirc$)

(b)

(c) $\xrightarrow{A} \quad \Rightarrow \quad$ poof!

(i.e., delete all transitions under nonterminals.)
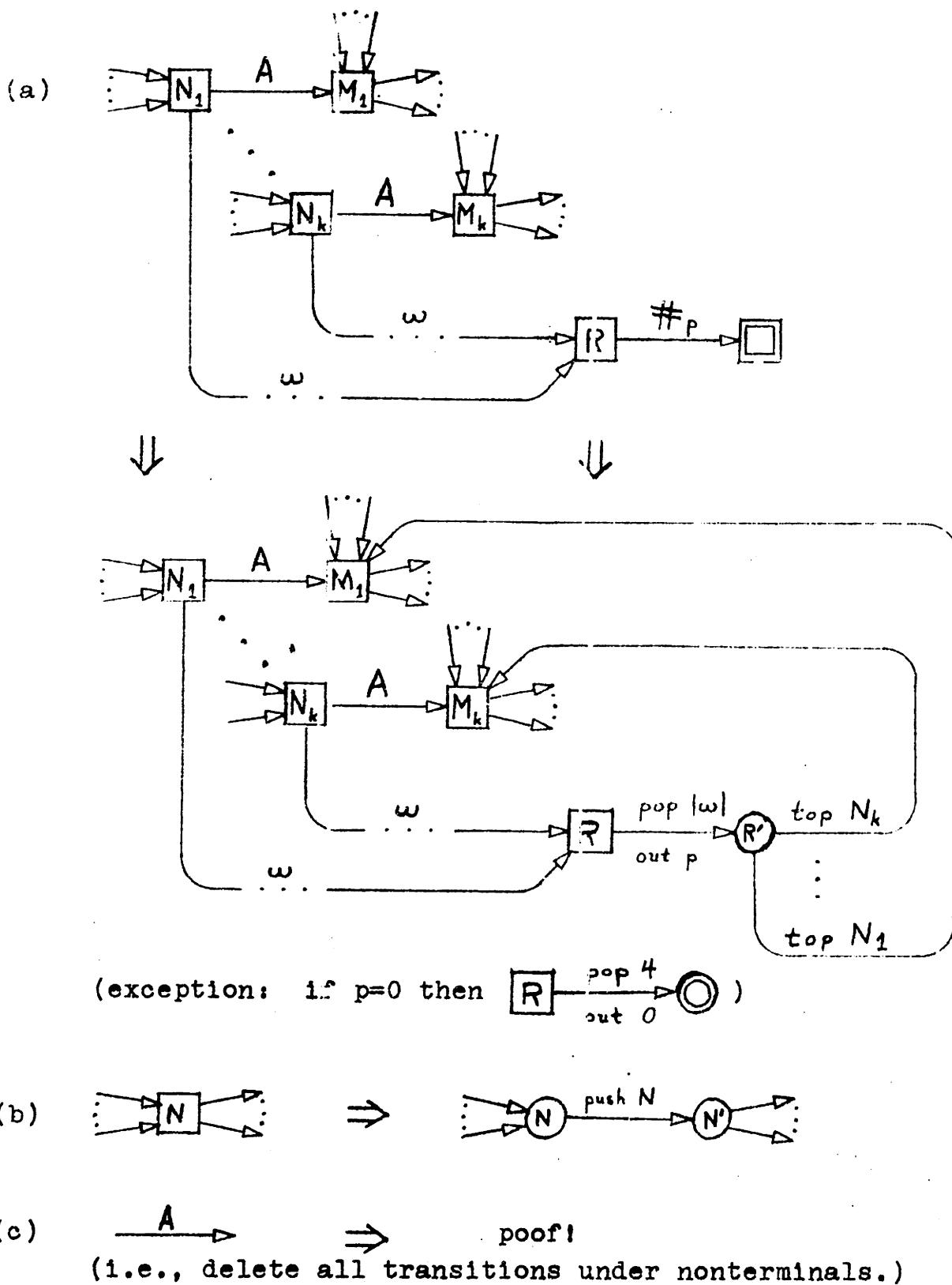
Figure 3.2. Transformations for converting the CFSM of an LR(0) grammar G to a DPDA-parser for G.

There is one exception to our first transformation. If $p = 0$ then we replace the $\#_0$-transition from R with one under "pop 4, out 0" to the terminal state. This follows because the associated production is known to be of the form $S \rightarrow \vdash S' \dashv$; i.e., because we know that R is associated with the final reduction. If we analyze first the parsing algorithm at the end of Section 3.4 and then the stack algorithm, we see that when our DPDA enters state R, the implicit left context must be $\vdash S' \dashv$, and therefore, that there must be four state-names in the stack. Thus, "pop 4" empties the stack so that the final configuration of the machine will be correct.

Figure 3.2b indicates a transformation which causes the DPDA to push the same state-names on its stack as the stack algorithm does on its, and at the same time. That is, when the DPDA enters state N, it first pushes the name N on its stack and then it enters a new state N' where it continues doing whatever the stack algorighm would do with the CFSM in state N.

Figure 3.2c indicates the deletion of all transitions under nonterminals. This is possible because of the optimization implicit in Figure 3.2a and because the DPDA is assumed to be parsing only terminal strings.[†]

In Figure 3.3 we present the result of applying the first and third of our transformations to the graph of Figure 3.1. We did not apply the second

---

[†] However, we believe that, if the transitions under nonterminals were retained, the DPDA could parse <u>any</u> sentential form.
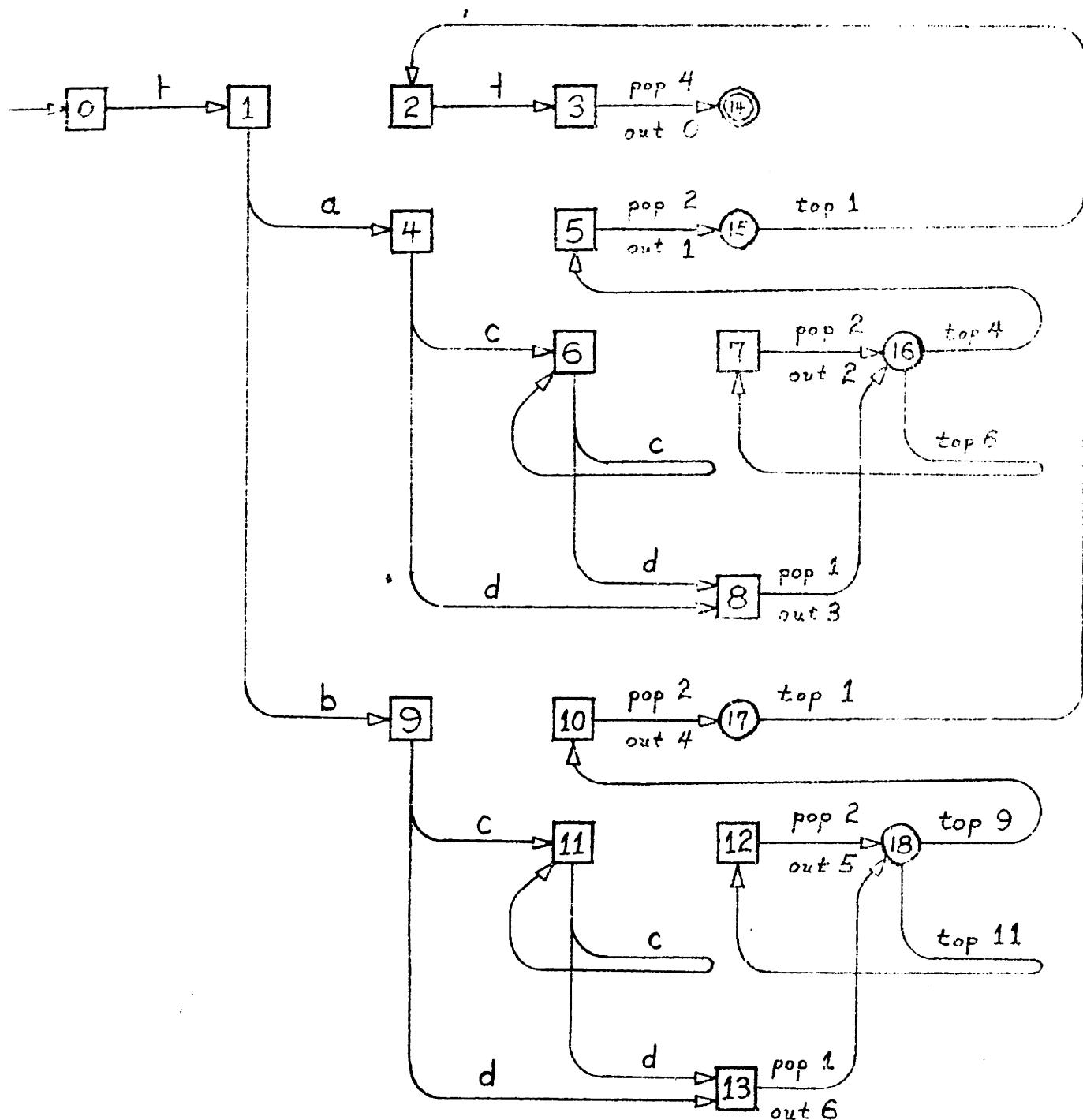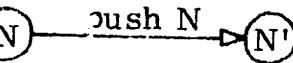
Figure 3.3. The finite control of the DPDA-parser for our
example grammar G_0: (0) $S \rightarrow \vdash E \dashv$, (1) $E \rightarrow a\,A$, (2) $A \rightarrow c\,A$,
(3) $A \rightarrow d$, (4) $E \rightarrow b\,B$, (5) $B \rightarrow c\,B$, (6) $B \rightarrow d$. This figure
was derived from Figure 3.1.

transformation for two reasons: (1) the figure would have gotten too large and unreadable, and (2) in our implementation in Chapter 7 we find it efficient to implement "states which push their names on the stack when they are entered"; i.e., we implement (N) —push N→ (N') as a single state. Thus, $\boxed{N}$ can be thought of as an abbreviation for such a state.

To illustrate the operation of our machines, we indicate in Table 3-1 the history which results when the DPDA implied by Figure 3.3 is applied to the string $\vdash$acd$\dashv$ in $L(G_0)$. Note that for perspecuity we indicate at each step the symbols of what is implicitly the left context. Of course, those symbols are not stored in the stack by the DPDA.

Comments: A read state of a DPDA is one all of whose transitions are under symbols. When a DPDA for an LR(0) grammar G is applied to a string η' not in L(G), it must abort in a way similar to the way the stack algorithm aborts. This follows because the DPDA simulates the stack algorithm. In particular, the machine will ultimately enter a read state N having no transition under the next symbol to be read. Further, the corresponding state N of the CFSM is the one in which the CFSM would abort if the stack algorithm were applied to η'.

The only other seemingly possible time that the DPDA could abort is when it is in a look-back state. But this possibility is ruled out, again because the DPDA simulates the stack algorithm. The stack algorithm looks back only to decide in which state to

Table 3-1. The history of grammar G_0's DPDA-parser applied to the string $\vdash$ acd $\dashv$ in $L(G_0)$.

State	Stack	Input String	Output
none		- a c d $\dashv$	
0	0	$\vdash$ a c d $\dashv$	
1	$^1\!1$ $\vdash$	a c d $\dashv$	
4	$^0{\vdash}\ ^1a\ ^4$	c d $\dashv$	
6	$^0{\vdash}\ ^1a\ ^4c\ ^6$	d $\dashv$	
8	$^0{\vdash}\ ^1a\ ^4c\ ^6d\ ^8$	$\dashv$	3
16	$^0{\vdash}\ ^1a\ ^4c\ ^6$	$\dashv$	
7	$^0{\vdash}\ ^1a\ ^4c\ ^6A^7$	$\dashv$	2
16	$^0{\vdash}\ ^1a\ ^4$	$\dashv$	
5	$^0{\vdash}\ ^1a\ ^4A^5$	$\dashv$	1
15	$^0{\vdash}\ ^1$	$\dashv$	
2	$^0{\vdash}\ ^1E^2$	$\dashv$	
3	$^0{\vdash}\ ^1E^2{\dashv}^3$		0
14			

restart the CFSM after a reduction. It does not look back to check the
validity of the information in the stack since, as noted above, that information
always represents a path through the CFSM. Thus, looks back cannot fail.

We do not formally prove that our DPDA for a given LR(0) grammar
G is a correct parser for the sentences of G. Instead, we informally argue
that the DPDA is equivalent in effect to the stack algorithm, which in turn
is equivalent to the algorithm at the end of Section 3.4, which in turn
is equivalent to our canonical parser that was informally proved to be
correct in Section 2.3. We implicitly rely on a similar line of reasoning
with respect to our parsers throughout the remainder of the dissertation.

3.6 Optimizing the DPDAs

As noted above our DPDAs have already been optimized with respect
to the stack algorithm. By precomputing part of the reduction procedures,
we increase both the time- and space-efficiency of our machines. Less
time is used because the reductions are executed with fewer machine
operations, and less space is used because transitions under nonterminals
are unnecessary. There are three more ways in which the DPDAs can be
optimized and all three are related to look-back in one respect or another.

(1) Two look-back states R_1' and R_2' are said to be __equivalent__ if and
only if for each transition from $R_1'(R_2')$ under "top N" to state M there is
a similar transition from $R_2'(R_1')$. Clearly, equivalent look-back states

may be eliminated in favor of a single state, in the obvious way. Note that the machine of Figure 3.3 has already been optimized in this way; e.g. 7 and 8 have transitions to the same look-back state. Clearly, the effect of this optimization is only to increase space-efficiency.

(2) Another optimization arises from the fact that we look-back only to determine which state to enter after a reduction. Thus, if all the transitions from a given lock-back state R' are to the same state M, then R' is unnecessary. States 15 and 17 of Figure 3.3 can be eliminated due to this property, increasing both the time- and space-efficiency of the DPDA. That is, the transitions from states 5 and 10 may by-pass states 15 and 17, respectively, and go directly to state 2.

(3) Finally, note that reduce states need not push their names on the stack since the names are immediately popped off again without ever being interrogated (via a "top R"). Thus, the node $\boxed{R}$ in the lower part of Figure 3.2a can be changed to $\textcircled{R}$, and "pop $|\omega|$" must then be changed to "pop $|\omega| - 1$".

In fact, in almost all cases only those states in the set $X = \{N \mid$ there is a transition under "top N" in the machine $\}$ need push their names on the stack; i.e., be represented by square nodes. Of course the "pop $|\omega|$" instructions must be changed accordingly, and thence arise the only exceptions to the previous statement. If we follow the path from N_1 to R in Figure 3.2a, starting with a counter set to zero as we leave N_1,

and increment the counter by one each time we encounter a state in the set X, we can reduce "pop $|\omega|$" to "pop n", where n is the value of the counter after reaching R. However, the same statement applies to the path from N_2 to R , ..., and N_k to R. Clearly, each path must imply the same n, or if this is not the case, some extra states not in X must push their names so that the paths are "balanced", in the obvious sense.

In the case of our DPDA of Figure 3.3, only states 1, 4, 6, 9, and 11 (the ones in the corresponding set X) need push their names. The effect of this optimization is, of course, to increase both time- and space-efficiency, but it also reduces the depth of the stack during execution.

Comments. To indicate the significance of these optimizations in a practical case, we give some statistics relating to our DPDA which is presented in Chapter 7. The DPDA corresponds to the grammar of a programming language which is quite practical, syntactically. The optimized machine has 172 states. The first optimization reduced the potential number of look-back states from 82 to 32. The second optimization further reduced the number to 22. The third optimization reduced the number of states pushing their names on the stack from 157 to 61 (again only those states in the corresponding set X); i.e., it reduced the depth of the stack during execution to about 3/8 of what it would otherwise have been.

We delay any specific estimates of the time-efficiencies of our machines until we have discussed parsers for "Simple LR(k)" grammars,

the subject of Chapter 4. The LR(0) grammars are not very interesting
for our purposes, so the efficiencies of their parsers are also uninteresting.
However, we find the "Simple LR($\underline{1}$)" grammars, and therefore their
parsers, quite interesting, as we shall see.

We delay discussion of specific space-efficiencies until Chapter 7,
where we are concerned with implementation issues. Space-efficiency
is most easily discussed in terms of an actual implementation.

Regarding implementation issues, the fact that look-back is not for
validation of information on the stack, also implies two possible optimizations
when implementing these parsers. (1) If the implementation is sequential
in nature (as is the one presented below), then, if in all but a few cases
the transitions from a look-back state R' go to a single state M, the "odd
balls" may be checked first and, if the top of the stack is not one of them,
a default transition to M may be made. (2) If the implementation is para-
llel in nature (e. g., array or matrix look-ups), then "compatible" look-
back states may profitably be merged into a single state. For instance,
in Figure 3.3 the four look-back states are "compatible" and can be merged
to form a single state having transitions under "top 1" to state 2, "top 4"
to 5, "top 6" to 7, "top 9" to 10, and "top 11" to 12. (The fact that the
first number in each case is one less than the second is a "red herring".)
We do not pursue the parallel possibilities in the present dissertation, even
though they have significant potential.

Finally, we emphasize that, since all of these optimizations concern look-back, they have no effect on error detection. That is, the optimized DPDA will detect that its input string is not in L(G), if indeed that is the case, at the same time relative to the reading of η' as would the unoptimized DPDA.

3.7 Conclusion

At this point it is advisable that the reader should reread Section 3.1 to place the foregoing results into perspective.

We have now developed much "machinery" for converting CFSMs to optimized DPDA parsers. Of course, our results thus far are useful only for LR(0) grammars, but we shall see in Chapters 4 and 5 that with the addition of one more transformation rule, namely one relating to "look-ahead", we shall have the "machinery" necessary for covering all LR(k) grammars. The problem of generating parsers for "Simple LR(k)" grammars, then, reduces to that of appropriately adding "look-ahead" information to CFSMs, and that for general LR(k) grammars reduces to appropriately splitting some states of the CFSMs and then adding "look-ahead" information.

Chapter 4

PARSERS FOR SIMPLE LR(k) GRAMMARS

We now investigate a class of grammars which is of substantial interest from the viewpoint of programming-language design and specification. The class is a subset of the LR(k) grammars for which parsers are only slightly more difficult to construct than are parsers for LR(0) grammars. The class includes the LR(0) grammars, and the accompanying parser-constructing technique is based on our LR(0) technique.

We begin by discussing the nature of the "inadequacy" of CFSMs for non-LR(0) grammars and a solution for that "inadequacy".

4.1 Inadequacy, Look-ahead

In the case of a grammar G which is not LR(0), Lemma 3.3 implies that G has at least one pair of characteristic strings of the form $\varphi \#_p$ and $\omega \theta \#_q$ such that $p \neq q$ and/or $\theta \neq \epsilon$. By definition of characteristic strings, then, there exist canonical forms $\alpha_1 = \varphi \beta_1$ and $\alpha_2 = \varphi \beta_2$ which have the characteristic strings $\varphi \#_p$ and $\varphi \theta \#_q$, respectively.

Assume that we attempt to use G's CFSM to determine the characteristic string of a form α alleged to be either α_1 or α_2. If we apply the CFSM to α, it reads φ and enters a state having distinct transitions under $\#_p$ and $1:\theta\#_q$ (recall the proof of Theorem 3.4); i.e., the machine enters an inadequate state. What do we do then?

If $\alpha = \alpha_1$ then we should stop and apply reduction p to the end of φ. However, if $\alpha = \alpha_2$ then, if $\theta \neq \epsilon$, we should allow the CFSM to continue reading, whereas if $\theta = \epsilon$, we should stop and apply reduction q to the end of φ. The problem is that there is not a <u>unique</u> parsing decision associated with an inadequate state, as is the case with a read or reduce state.

Stated another way, the state, and therefore the CFSM, are indeed "inadequate" for use in determining characteristic strings. However, the LR(k) definition itself hints at a solution to this inadequacy. By using the CFSM we have, in effect, investigated and remembered some pertinent features of the left context φ. However, we have not investigated the right context at all; i. e., we have not <u>looked ahead</u> of the decision point.

Let us consider an example. There follow the <u>productions</u>[†] of the characteristic grammar of our example grammar G_1 (page 19).

[†] Note that the production E' → E' makes the grammar "infinitely ambiguous"; i. e., each sentence has infinitely many canonical parses. This is of no concern to us here because we are not interested in the "structural properties" of the grammar. We are only interested in the strings which the grammar generates and the CFSM which accepts them.

(0) $S' \rightarrow \vdash E \dashv \#_0$

(1) $S' \rightarrow \vdash E'$

(2) $E' \rightarrow E + T \#_1$

(3) $E' \rightarrow E + T'$

(4) $E' \rightarrow E'$

(5) $E' \rightarrow T \#_2$

(6) $E' \rightarrow T'$

(7) $T' \rightarrow P \uparrow T \#_3$

(8) $T' \rightarrow P \uparrow T'$

(9) $T' \rightarrow P'$

(10) $T' \rightarrow P \#_4$

(11) $P' \rightarrow i \#_5$

(12) $P' \rightarrow (E) \#_6$

(13) $P' \rightarrow (E'$

The corresponding CFSM is illustrated in Figure 4.1. For our purposes here the only state of interest is the inadequate one, state 7.

Consider the two canonical forms of G_1 $\alpha_1 = \vdash P + i \dashv$ and $\alpha_2 = \vdash P \uparrow i \dashv$. The unique characteristic strings of α_1 and α_2 are $\vdash P \#_4$ and $\vdash P \uparrow i \#_5$, respectively, as the reader may easily confirm by canonically deriving the forms. Clearly, the prefix $\vdash P$, which is common to α_1 and α_2, accesses state 7 of G_1's CFSM.

Now, if we were given α alleged to be either α_1 or α_2, we could determine α's characteristic string as follows. First, we apply G_1's CFSM to α. Then, when the CFSM enters state 7, we look ahead at, but do not let the CFSM try to read, the next symbol to be read. If the symbol is +, then the characteristic string is the prefix read by the CFSM thus far ($\vdash P$) concatenated with $\#_4$. However, if the symbol is $\uparrow$, we must allow the CFSM to continue reading to determine the characteristic string. (In this
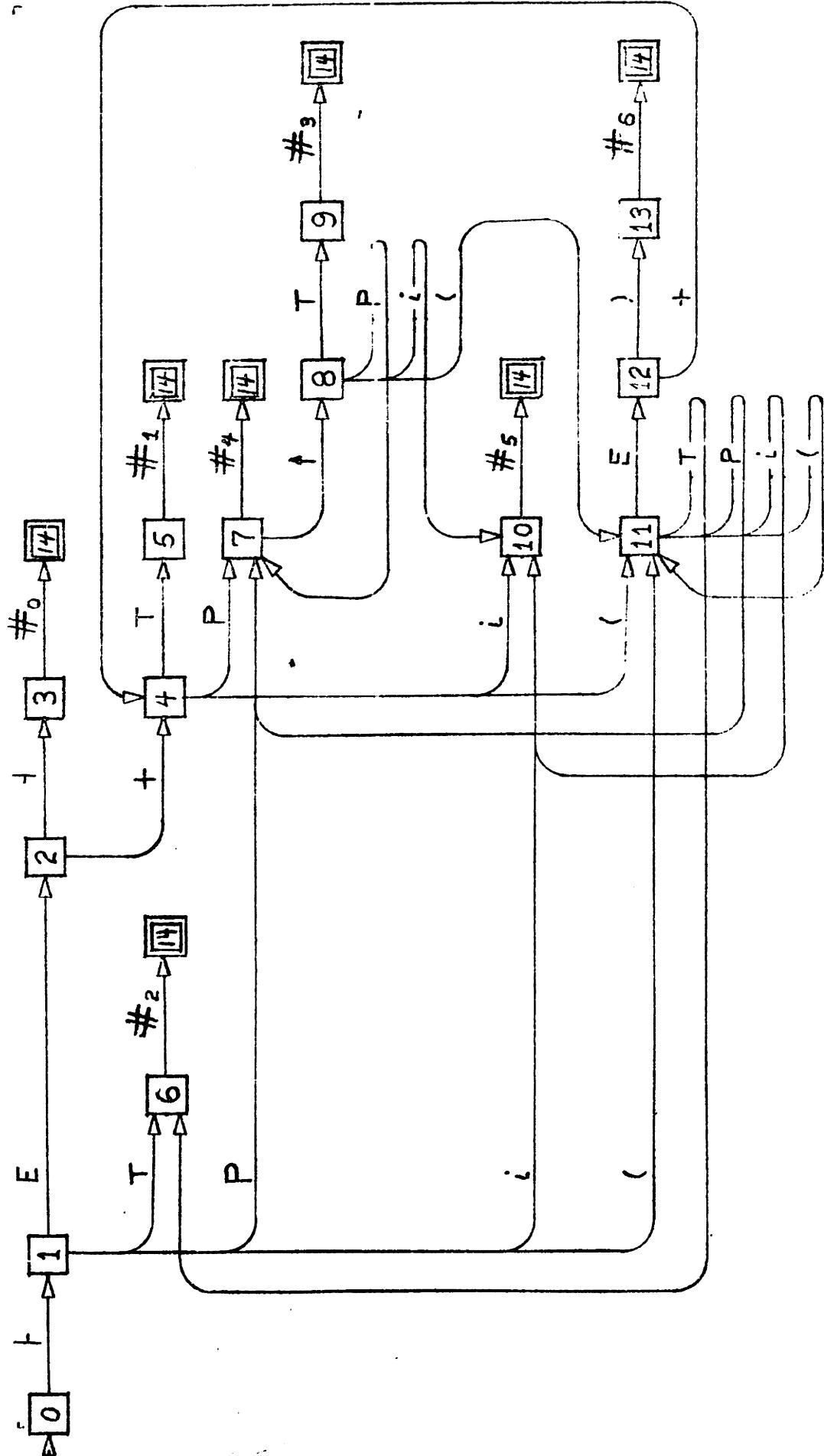
Figure 4.1. The CFSM of our example grammar G_1: (0) $S \to \vdash E \dashv$, (1) $E \to E + T$, (2) $E \to T$, (3) $T \to P \uparrow T$, (4) $T \to P$, (5) $P \to i$, (6) $P \to (E)$. Here as in

Figure 3.1 $\boxed{14}$ denotes a single state.

case the machine would read $\vdash$ i and enter state 10, thus determining that the characteristic string is $\vdash P \uparrow i \#_5$.)

In fact, we show below that no matter what canonical form α of G_1 we are given, if a prefix φ of α accesses state 7, then we can determine via one symbol look-ahead whether α's characteristic string is $\varphi\#_5$ or $\varphi \uparrow \ldots$ In particular, if we look one symbol ahead and see a symbol in the set $\{\dashv, +,)\}$, the characteristic string is $\varphi\#_4$, but if we see one in the set $\{\uparrow\}$, it is $\varphi\uparrow\ldots$

LALR(k) Grammars. The above discussion and example might lead one to think that perhaps every LR(k) grammar has the property that its sentences can be parsed, in a manner similar to that just illustrated, by using its CFSM and some look-ahead sets associated with the transitions from inadequate states. Unfortunately, this is not the case. However, for purposes of discussion let us informally define a CF grammar to be LALR(k) (for look-ahead LR(k)) if and only if it has the above stated property.

Clearly every LALR(k) grammar is LR(k), since the determination of characteristic strings for such a grammar is based on some knowledge of left context and at most k symbols of right context. In fact, the determination concerns only the equivalence class of the left context. Further, a minimum number of equivalence classes is involved, since we use an FSM with a minimum number of states to remember relevant information about left context.

We illustrate in Chapter 5 that the LALR(k) grammars are a subset of the LR(k) grammars by giving a grammar for which adding look-ahead alone is not a sufficient modification to the CFSM; it must have some of its states split to make it remember more about left context; i. e., to increase the number of equivalence classes of left context.

Unfortunately, again as we shall see in Chapter 5, even the LALR(k) grammars cannot be described as a "simple" subset, since the computation of the look-ahead sets for some of those grammars is distinctly nontrivial. Thus, if we are to have a parser-constructing technique which grows in complexity as it discovers the complexity of the grammar at hand, we should not jump from a procedure covering the LR(0) grammars to one covering the LALR(k) grammars.

Instead, we consider next a smaller subset of the LR(k) grammars which are distinguished both by the fact that adding look-ahead to the corresponding CFSMs is sufficient to render them useful for determining characteristic strings and that the computation of look-ahead sets is simple. It turns out, as we shall see in Section 4. 8, that even this smaller subset is a large and useful set of grammars.

4. 2   Simple LR(k) Grammars

Expediency dictates that we define this subset of the LR(k) grammars in terms of our parser-constructing technique, as we did in the case of

the LALR(k) grammars. This is not unreasonable since there seems to be no good, intuitive definition in terms of canonical forms and parsing decisions, anyway.

The "simple" function which is central to our definition and which is useful for computing look-ahead sets is as follows.

<u>Definition 4.1.</u> Let k be a positive integer and let $G = (V_T, V_N, S, P)$ be a CF grammar, one of whose nonterminals is A. Then

$$\underline{F}_T^k(\underline{A}) = \{(k{:}\beta) \in V_T^* \mid S \to^* \rho A \beta \text{ for some } \rho, \beta\}.^\dagger$$

Thus, $F_T^k(A)$ is the set of all terminal strings of length k which may follow A in a canonical form of G. We are interested in look-ahead sets containing only terminal strings because our ultimate DPDAs will operate in a strictly left-to-right manner and will be applied to nothing but sentences.

As an example we compute $F_T^1(P)$ for grammar G_1: P appears in the right parts of two productions. The production $T \to P \uparrow T$ implies that $\uparrow$ is in $F_T^1(P)$. The production $T \to P$ implies that all the strings in $F_T^1(T)$ are also in $F_T^1(P)$. $E \to E + T$ and $E \to T$ each imply that the members of $F_T^1(E)$ are also in $F_T^1(T)$. $S \to \vdash E \dashv$ implies that $\dashv$ is in $F_T^1(E)$; $E \to E + T$ adds $+$;

$\dagger$ Our set notation is an abbreviation of the usual mathematical notation: $\{\sigma \in V_T^* \mid S \to \rho A \beta \text{ for some } \rho, \beta \text{ and } \sigma = k{:}\beta\}$.

and $P \rightarrow (E)$ adds ")". Thus, we have determined that $F_T^1(P) = \{\dagger, \dashv, +,) \}$,

and in the process that $F_T^{1'}(T) = F_T^1(E) = \{ \dashv, +,) \}$.

Warshall (War 62) has a fast "bit-matrix technique" which can be used (Che 67) for computing these sets for $k = 1$. This is particularly important since we expect the large majority of the grammars of interest to be "Simple LR(1)", as we indicate in Section 4.8. Further, for those few grammars which are not "Simple LR(1)" we expect to have to resort to $k = 2$ or 3, say, with respect to only one or two inadequate states. Thus, we have a reasonable step up in complexity from the LR(0) grammars.

We now define the look-ahead sets in terms of which we later define the "Simple LR(k)" grammars.

Definition 4.2 (Recursive on the value of k.) Let G be a CF grammar and k be a positive integer. There is associated with each terminal- and #-transition of G's CFSM a simple k-look-ahead set which is as follows. For a $\#_p$-transition, where production p is $A \rightarrow \omega$, the set is $F_T^k(A)$. For a transition under the terminal t the set is $\{t\}$ if $k = 1$ and otherwise $\{t\beta' \in V_T^* \mid$ the t-transition is to a state N and β' is in the simple (k-1)-look-ahead set associated with some transition from N$\}$.

Comments: (1) We do not define look-ahead sets for transitions under nonterminals because our ultimate DPDAs will have no such transitions,

and (2) although for ease of definition sets are associated with every terminal- and #-transition of the CFSM, we are interested only in the sets for transitions from inadequate states.

For the value as an example we illustrate the computation of the simple 3-look-ahead set for the ↑-transition in Figure 4.1. The computation is actually unnecessary for grammar G_1, since G_1 is "Simple LR(1)".

First, we follow all paths leading from state 7, never taking transitions under nonterminals, until either a string of length three is spelled out or until the terminal state is reached. The strings spelled out by all such paths are ↑i#$_5$, ↑(i, and ↑((. Next, the desired set of strings can be derived from these strings as follows. First, each string which contains no #-symbol is in the desired set. Second, for each string of the form $\sigma\#_p$, where production p is $A \rightarrow \omega$ and $|\sigma| = n$, every string which can be formed by concatenating σ with a member of $F_T^{k-n}(A)$ is in the desired set. In our special case the latter means ↑i concatenated with the members of $F_T^1(P)$. Thus, the simple 3-look-ahead set for the ↑-transition is { ↑(i, ↑((, ↑i↑, ↑i⊣, ↑i +, ↑i) }.

Finally we come to our main definition.

Definition 4.3. Let k be a positive integer. A CF grammar G is Simple LR(k), abbreviated SLR(k) if and only if for each

inadequate state N (if any) of G's CFSM the simple k-look-ahead sets associated with the (terminal- and #-) transitions from N are mutually disjoint. G is $\underline{SLR(0)}$ if and only if it is LR(0).

Our example grammar G_1 is SLR(1). Proof: The simple 1-look-ahead set associated with the $\uparrow$-transition of its CFSM is $\{\uparrow\}$ and that of the $\#_4$-transition is $F'_T(T) = \{ \dashv, +,) \}$, as we have seen. Obviously, these sets are disjoint.

4.3   SLRkFSMs

We now turn to the question of how to explicitly encode look-ahead sets into CFSMs. We desire an explicit encoding for two reasons: (1) it facilitates proofs that "CFSMs-plus-look-ahead sets" can be used to determine characteristic strings, and (2) it facilitates our discussion of a technique for converting those machines to DPDAs.

The encoding is accomplished by adding to each CFSM transitions under "generalized symbols". If R is a look-ahead set associated with a given X-transition (X not a nonterminal) of the CFSM, then X^R is a $\underline{generalized\ symbol}$ associated with the X-transition and the set R.
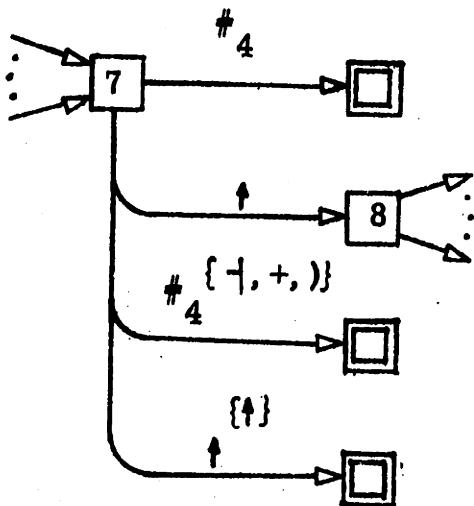
$\underline{Definition\ 4.4.}$ Let G be an SLR(k) grammar. We construct G's $\underline{SLRkFSM}$ from its CFSM as follows. For each inadequate state N (if any) of the CFSM and for each X-transition (X not a

nonterminal) from N having associated with it the simple k-look-ahead set R, we add a transition from N, under the generalized symbol X^R, to the terminal state.
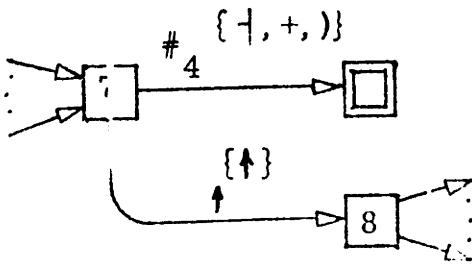
Clearly an SLRkFSM is a reduced, deterministic FSM. It accepts the characteristic strings of G plus the strings in the set $\{\varphi X^R \mid \varphi \text{ accesses}$ an inadequate state N of G's CFSM and N has an X-transition (X not a nonterminal) with which is associated the simple k-look-ahead set R}.

As in the case of CFSMs we use the terms "read", "reduce", and "inadequate" with regard to states of SLRkFSMs, in the obvious way. However, for emphasis we sometimes refer to the inadequate states as "modified-inadequate states".

In the case of grammar G_1, its SLR1FSM is the graph in Figure 4.1 with state 7 replaced by the following:

In the sequel we sometimes use an abbreviated notation for modified-inadequate states. For instance, the above can be abbreviated:

We emphasize that this is only an abbreviation. Our theorems below are easier to prove in terms of the former notation than the latter.

The modified stack-algorithm. In a manner similar to the way in which we developed DPDA-parsers for LR(0) grammars, we first state a stack-algorithm which uses an SLRkFSM to determine characteristic strings, and then we convert the SLRkFSM to a DPDA which simulates the stack-algorithm. Our stack-algorithm here is simply our previous one modified to "look ahead" at the appropriate times. We present the algorithm next and prove that it works correctly afterward.

Commencing with a string $\alpha = \eta$, where η is in L(G), with an empty stack, and with G's SLRkFSM in its starting state:

(i)   Apply the SLRkFSM to α; store on the stack the names of the states entered by the machine as it reads.

(ii)   If, after reading some prefix φ of α such that $\alpha = \varphi\beta$, the machine enters a reduce or inadequate state N, then

(a) if N is a reduce state whose only transition is under $\#_p$, where production p is $A \to \omega$, then output p, pop the top $|\omega|$ names off the stack, return the SLRkFSM to the state whose name is at the top of the stack, set $\alpha = A\beta$, and go to step (iii).

(b) if N is an inadequate state with (among others) transitions under the generalized symbols $X_1^{R_1}$, $X_2^{R_2}$, $\ldots$, $X_n^{R_n}$, compare the strings in the sets R_1, R_2, $\ldots$, R_n with $k{:}\beta$. Exactly one match will occur, say with a string in R_j.

    (1) If X_i is a $\#$-symbol, execute step (ii), part (a), as if N were a reduce state whose only transition is under X_i.

    (2) However if X_i is a terminal symbol, treat N as if it were a read state (i. e., as if it had only its transitions under symbols), continue the reading and name-storing processes, and return to step (ii).

(iii) If $\alpha = S$ then stop; otherwise, return to step (i).

<u>Proof</u> . Since the present stack-algorithm is like our previous one except for the addition of a procedure related to inadequate states, we need only prove that it operates correctly when the SLRkFSM enters such a state. Informally, we prove in Theorem 4.1 that, if the SLRkFSM reads to the end of a canonical form's stack string, the algorithm will correctly determine the characteristic string. Then, in Theorem 4.2 we prove that in reading the stack string the algorithm will not make an incorrect choice before reaching the end.

Theorem 4.1 Let G be an SLR(k) grammar and $\alpha = \varphi\beta$ be a canonical form of G with characteristic string $\varphi\#_p$. Then the stack string φ of α accesses a state of G's SLRkFSM which is either (1) a reduce state whose only transition is under $\#_p$, or (2) a (modified-inadequate) state with transitions under the generalized symbols, $X_1^{R_1}, X_2^{R_2}, \ldots, X_n^{R_n}$ for some $n \geq 2$, such that $k{:}\beta$ is in R_i but not in R_j for $1 \leq i \neq j \leq n$, and such that $X_i = \#_p$.

Proof: Our proof depends upon the similarity of the CFSM and the SLRkFSM of G. There are only two cases since φ must access either a reduce state or an inadequate state of the CFSM. (1) Clearly, from the trivial nature of the modification required to form the SLRkFSM from the CFSM, if φ accesses a reduce state of the CFSM, then it also accesses a reduce state of the SLRkFSM. (2) Similarly, if φ accesses an inadequate state N of the CFSM, it must access a state of the SLRkFSM which is similar to N except for the extra transitions under generalized symbols. Furthermore, $k{:}\beta$ may appear in at most one of the sets $R_1, R_2, \ldots, R_n$, since by definition of an SLR(k) grammar they are mutually disjoint sets. $k{:}\beta$ must appear in R_i such that $X_i = \#_p$, because, if production p is $A \to \omega$,

$R_i = F_T^k(A) = \{k{:}\beta \mid S \to^* \rho A\beta\}$ by definition, and the latter includes $\{k{:}\beta \mid S \to^* \rho\omega\beta\}$.

Q. E. D.

Theorem 4.2. Let G be an SLR(k) grammar and $\alpha = \varphi\theta\beta$ be a canonical form of G with characteristic string $\varphi\theta\#_p$ such that θ is in V_T^* but $\theta \neq \epsilon$. Then, if φ accesses an inadequate state N of G's SLRkFSM having transitions under the generalized symbols, $X_1^{R_1}, X_2^{R_2}, \ldots, X_n^{R_n}$ for some $n \geq 2$, the string $k:\theta\beta$ is in R_i but not in R_j for $1 \leq i \neq j \leq n$, such that $X_i = 1:\theta$.

Proof: $k:\theta\beta$ may appear in at most one of the sets $R_1, R_2, \ldots, R_n$, since the sets are mutually disjoint. $k:\theta\beta$ must appear in R_i such that $X_i = 1:\theta$ for the following reasons. Since both G's SLRkFSM and its CFSM accept $\varphi\theta\#_p$, there is a path leading from N (of both) which spells out $\theta\#_p$. It is easy to see from the definition of a simple k-look-ahead set R for a terminal transition (in particular, one under $1:\theta$) that if $|\theta| \geq k$ then $k:\theta$ is in R, whereas if $|\theta| = n < k$ then every string formed by concatenating θ with a member of $F_T^{k-n}(A)$ is in R, where production p is $A \to \omega$. The latter includes $k:\theta\beta$ by definition of $F_T^k(A)$.   Q. E. D.

4.4   Minimizing Look-ahead

We noted in Chapter 2 (page 31) that the smallest value of k for which a grammar is LR(k) is limited by the worst case of necessary look-ahead. A similar statement is true regarding the SLR(k) condition. In fact, we

could have defined SLR(k) grammars in the following alternate way. We could have first defined a grammar G to be "SLR(k) with respect to" a given inadequate state of its CFSM, in the obvious way. Then we could have defined G to be SLR(k) if and only if it is "SLR(k) with respect to" each of its CFSM's inadequate states.

This alternate definition emphasizes the fact that the look-ahead sets for the transitions from a given state N may be computed for the smallest value of k such that the sets are mutually disjoint. In effect, we recognized this fact to a limited extent in Theorem 4. 1; i. e. , we recognized that every grammar is "SLR(0) with respect to" each reduce state of its CFSM. Only notational and expositional difficulties prevented us from incorporating this fact into our definition of SLRkFSMs and Theorems 4. 1 and 4. 2, rather than belatedly bringing it up now.

Fine tuning. In some cases not only may the amount of look-ahead required be different for distinct states, but even a single state may have strings of various lengths in its look-ahead sets. Consider, for instance, a state N having only the two look-ahead sets, {ab, cd}, and {ae}. Clearly, if the SLRkFSM is in N and the next symbol to be read is c, we need not investigate the second symbol to make the associate parsing decision. That is, the first set above may be changed to {ab, c}.

In general, look-ahead sets may have the lengths of their strings minimized as follows. Consider a state N with look-ahead sets $R_1, R_2, \ldots R_n$

for some $n \geq 2$. We change each set R_i by removing from the right end of each string in R_i the maximum number of symbols such that the result is not a prefix of a string in R_j, for $1 \leq i \neq j \leq n$. Clearly, the sets remain mutually disjoint after these changes.

Note that this optimization is not applicable to simple 1-look-ahead sets, since ϵ is a prefix of every string.

4.5 The Conversion of SLRkFSMs to DPDAs

It should be clear from the modified stack-algorithm that the transformations implied by Figure 3.2 remain valid ones, as regards the read and reduce states of our SLRkFSMs. Furthermore, the computation of look-back states implied by Figure 3.2a is also valid for the #-transitions from inadequate states. Thus, all we need now is one more transformation rule; i.e., one for mapping modified-inadequate states, whose associated look-back states have already been computed, into look-ahead states[†] of a DPDA. The appropriate transformation is implied by Figure 4.2, and the conversion technique goes as follows.

First, we apply the transformation implied by Figure 3.2a to each reduce state of the SLRkFSM. Also, for each inadequate state I of the

---

[†] Again we are abusing strict automata theory by allowing our DPDAs to "look ahead". We do so for the sake of simplicity and practicality. It is well known (Knu 65) that DPDAs without "look ahead" can perform the same computations as ours.
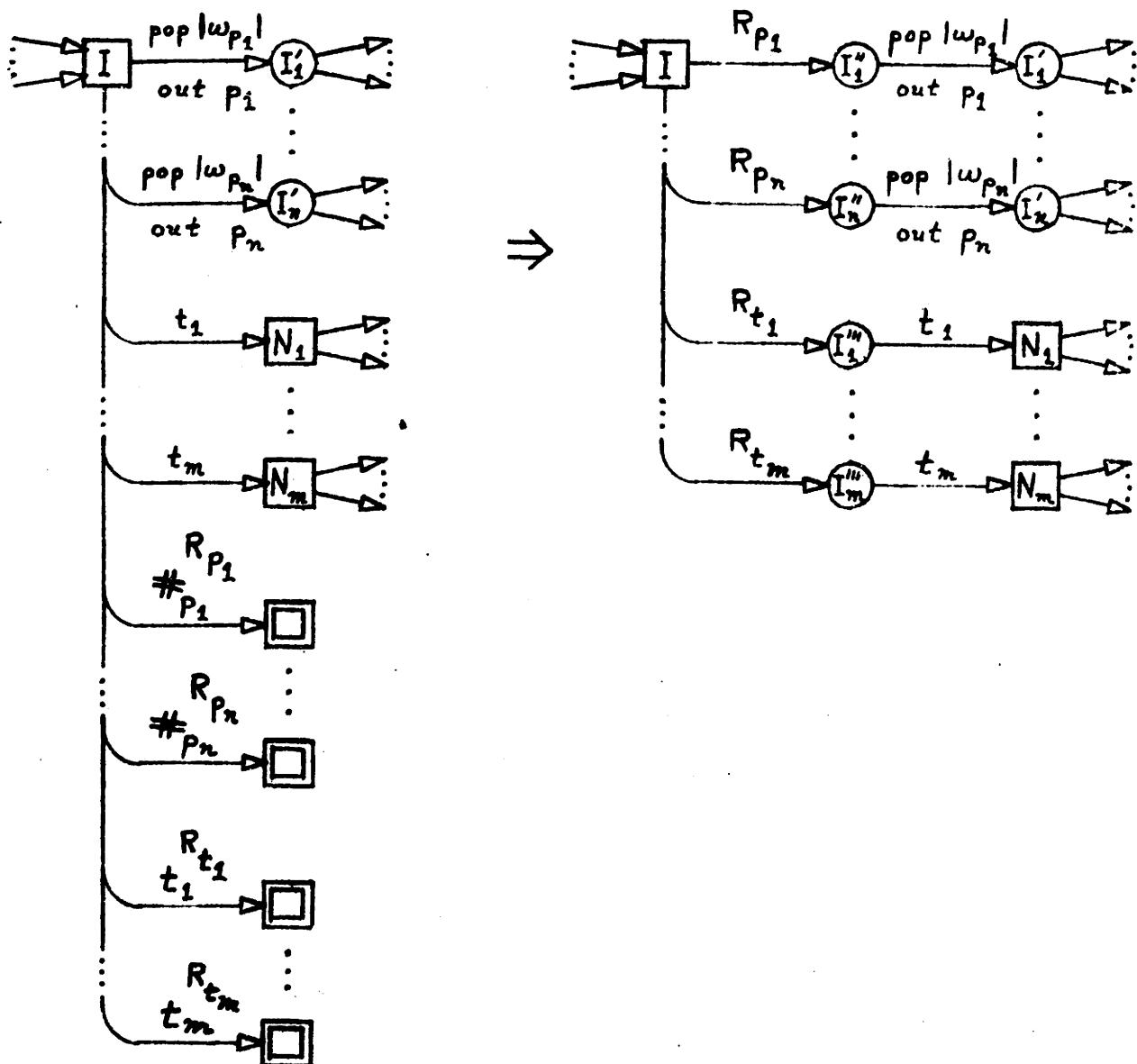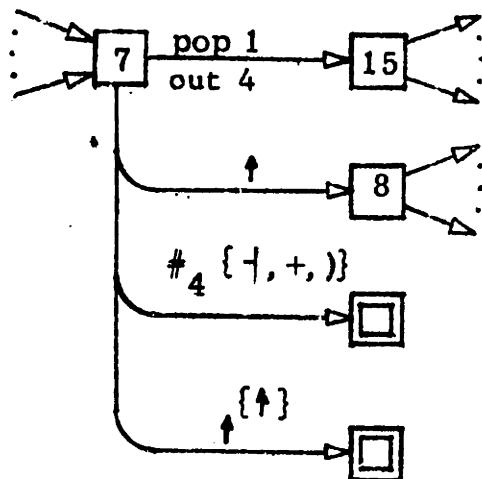
Figure 4.2. The transformation for converting modified-inadequate states to look-ahead states. This transformation plus those implied by Figure 3.2 are all that are needed for converting an appropriate FSM to a DPDA-parser for any LR(k) grammar.
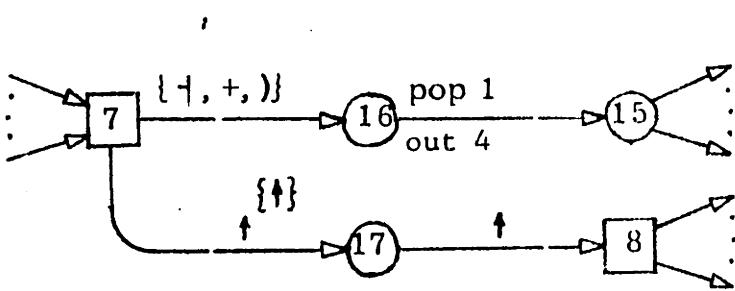
machine and for each #-transition T from I, we apply the former transformation to I, as if it were a reduce state whose only transition is T. The result after this first step is, of course, a machine with "inadequate states" of the form indicated in the left part of Figure 4.2, where if $n = 1$ then $m \geq 1$, but if $n > 1$ then $m \geq 0$.

In the case of the inadequate state 7 of G_1's SLRkFSM (illustrated in Section 4.3), the result is as follows:

Next, we apply to each inadequate state I resulting from the first step the transformation implicit in Figure 4.2. The latter indicates a conversion to a look-ahead state I of the DPDA. The intent, of course, is that when the DPDA is in state I it should simulate the modified stack-algorithm when the SLRkFSM is in state I (recall step (ii) of the algorithm).

The result of applying this second step to state 7 illustrated above is as follows.

{⊢, +,)}   pop 1   7 ──── 16 ──────── 15   out 4

{↑}   ↑   ↑   17 ──── 8

Finally, we apply the transformations implied by Figures 3.2b and 3.2c to the machine, and we have the desired "DPDA with look-ahead", except for optimizations.

Optimizations. Since the optimizations discussed in Section 3.6 applied only to look-back, which is independent of look-ahead, each of those optimizations is also applicable to "DPDAs with look-ahead". Only one more optimization presents itself, and it is applicable only to (the very important case of) 1-symbol look-ahead states. We illustrate this final optimization in conjunction with the presentation in Figure 4.3 of the fully optimized DPDA-parser for grammar G_1.

For present purposes consider only state 7. The intent is that, when the DPDA enters state 7, it should look-ahead as usual and, if the next symbol is ⊢, +, or), it should enter state 16 next, as usual; however if the symbol is ↑, it should move its read head to the right one place and then enter state 8. That is, the state is sort of a combination "look-ahead read-state", and it eliminates the inefficiency of investigating the ↑ twice. We allow such states because it is easy to implement them, as we show in Chapter 7.
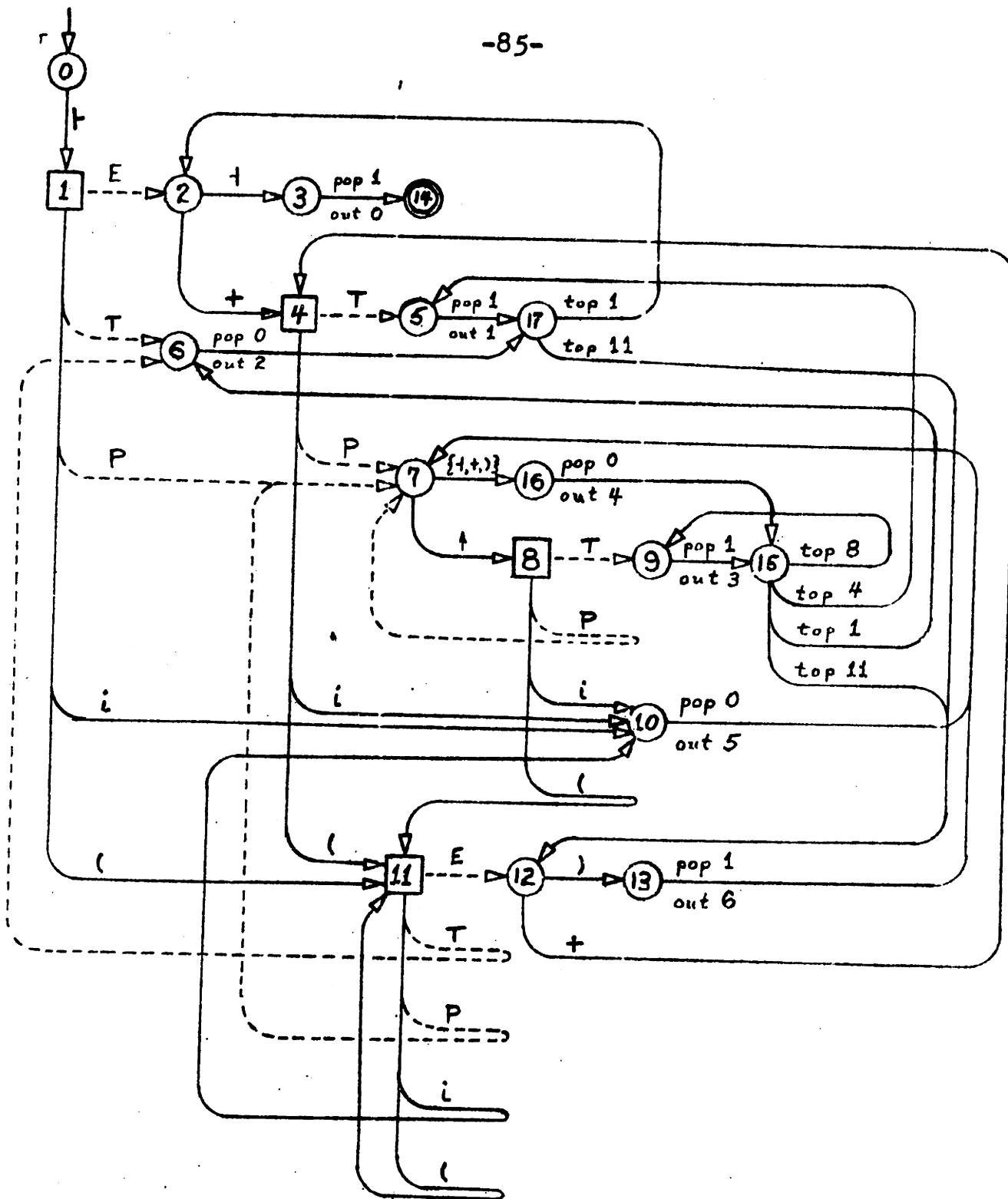
Figure 4.3. The fully optimized DPDA-parser for grammar G_1. This figure was derived from Figure 4.1. The dashed arrows are not intended as part of the machine. Recall that when the DPDA enters a state represented by a square, it pushes the name of that state on its stack.

The dashed arrows in Figure 4.3 indicate the transitions under nonterminals which were removed from G_1's S.R1 FSM in forming the DPDA. That is, they are not to be taken as part of the DPDA. We include them to facilitate future discussions and to aid the thoroughly interested reader in reviewing the transformation rules as they apply to this example.

Recall that when the DPDA enters a state represented by a square, it pushes the name of the state on the stack.

4.6 Time-Efficiency

From the automata-theoretic viewpoint a parser is simply a translator; it is a machine which translates strings into parses; i.e., strings of symbols into strings of production numbers. We adopt this viewpoint for the purpose of discussing the time-efficiency of our parsers.

We informally define time-efficiency in terms of an "ideal machine". The latter is assumed to be able to translate a string of n symbols into a string of m symbols with only 2(n+m) "machine operations" of approximately equal complexity (execution time); i.e., it takes n reads, m outputs, and n+m accompanying state-changes. By the "time-efficiency" of a DPDA, then, we mean the number of machine operations required by the ideal machine to perform a given translation divided by the number required by the DPDA to perform the same translation.

In Table 4-1 we illustrate the history which results when the DPDA of Figure 4.3 is applied to the string $\eta_1 = \vdash i \uparrow i + i \dashv$ in $L(G_1)$. Counting

Table 4-1. The history which results when the DPDA of Figure 4.3 is applied to the string $\eta_1 = {\vdash}i{\uparrow}i+i{\dashv}$.

State	Stack	Input String	Output	Machine Operations					
				Read	Push	Pop	Look-ahead	Look-back	Output
0		⊢i↑i+i⊣		x					
1	1	i↑i+i⊣		x	x				
10	1	↑i+i⊣	5						x
7	1	↑i+i⊣		x					
8	1 8	i+i⊣		x	x				
10	1 8	+i⊣	5						x
7	1 8	+i⊣						x	
16	1 8	+i⊣	4						x
15	1 8	+i⊣				x	x		
9	1	+i⊣	3						x
15	1	+i⊣						x	
6	1	+i⊣	2						x
17	1	+i⊣						x	
2	1	+i⊣		x					
4	1 4	i⊣		x	x				
10	1 4	⊣	5						x
7	1 4	⊣						x	
16	1 4	⊣	4						x
15	1 4	⊣				x			
5	1	⊣	1				x		x
17	1	⊣						x	
2	1	⊣		x					
3	1	⊣	0			x			x
14									

23 state changes   Totals   7 3 3 2 5 9

Time efficiency $= \dfrac{2(7+9)}{23+7+3+3+2+5+9} = \dfrac{32}{52} \simeq 62\%$

all the pushes, pops, reads, outputs, and state-changes executed by the machine, it requires 52 machine operations to map η_1 into its canonical parse. Since $|\eta_1| = 7$ and since there are nine symbols (production numbers) in the parse, the ideal machine could have performed the translation in $2(7+9) = 32$ machine operation. Thus, the time-efficiency of the DPDA is $32/52$ or about 62% for η_1.

If a similar table is constructed for the unoptimized version of G_1's DPDA parser, we find that it takes 79 machine operations; i.e., for η_1 its time-efficiency is $32/79$ or about 41%. Thus, the optimized DPDA is 1.5 times as fast as the unoptimized one.

A general case. Let us consider the time-efficiency for a more general case. In particular, let us compute the worst-case time-efficiency for the DPDA-parser of some SLR(1) grammar, when it is applied to a string of n symbols having a canonical parse of m symbols. We merely analyze the behavior of the DPDA (assumed to be similar to the one of Figure 4.3) and determine the maximum number of machine operations which can be associated with each of the $n+m$ symbols.

At worst we may need a push, a read, and a state-change for each of the n input symbols, since we may need to push the name of each read and look-ahead state. For each of the m output symbols (i.e., for each reduction), we may need a push, a look-ahead and a state-change, then a pop, an output and a state-change, and finally a look-back and a state-change.

Thus, the DPDA could take as many as $3n + 7m$ machine operations to perform the translation. The time-efficiency in the worst case, therefore, is $2(n+m)/(3n+7m)$, which is a minimum of 29% when $m \gg n$.

4.7 Error Detection

In the present section we have three points to make regarding the actions of a DPDA-parser for an SLR(k) grammar G when the DPDA is applied to a string η' not in L(G):

(1) The machine must ultimately detect the "error".

(2) It may detect the error either while reading or while looking ahead.

(3) It may not detect the error as soon as it would have had its look-ahead sets been computed by using functions complex enough to cover the LALR(k) or general LR(k) grammars.

(1) The first point follows from the facts that the DPDA ultimately simulates our canonical parser of Chapter 2 and that there exists no canonical parse for η'. (Recall the argument at the end of Section 3.6.)

(2) Our DPDAs without look-ahead had only one way in which to abort, namely by entering a read state with no transition under the next symbol to be read. Clearly, by adding look-ahead states we add another possibility. The machine may enter a look-ahead state N such that none of the strings in the look-ahead sets of the transitions from N match the beginning of the string remaining to be read.

(3) We illustrate our third point by example. Consider an SLRkFSM with two inadequate states N_1 and N_2, each having a $\#_p$-transition, where production p is $A \to \omega$. For i = 1, 2 let $RC_i = \{\beta \in V_T^* \mid \varphi\beta$ is a canonical form with characteristic string $\varphi\#_p$ such that φ accesses state $N_i\}$. Assume that RC_1 and RC_2 are disjoint sets. Then if φ_1 accesses N_1 and β_2 is in RC_2, $\varphi_1\beta_2$ is not a canonical form. And yet, if our DPDA is in state N_1 with implicit left context φ_1 and right context β_2, it will not detect the error immediately via look-ahead. This follows because the simple k-look-ahead set corresponding to the $\#_p$-transition contains $k{:}\beta_2$, by definition.

Clearly, if the look-ahead sets of the $\#_p$-transitions from N_1 and N_2 are reduced to $R_i = \{k{:}\beta \mid \beta \in RC_i\}$ for i = 1, 2, respectively, then the DPDA continues to correctly parse sentences in L(G). However, after this change, it will detect the above error via look-ahead when it is in state N_1, since $k{:}\beta_2$ is not in R_1.

What we have dis-covered is that, if the look-ahead sets for a state N are computed independently of the left contexts which access N, as is the case when we use F_T^k, the sets sometimes contain strings which cannot begin a legitimate right context when the machine is in state N. Thus, in a sense, F_T^k is not always "restrictive" enough. Note, however, that this situation may obtain only if there is more than one transition in the machine

under some #-symbol. (In our practical example in Chapter 7 only 2 of 82 productions have more than one corresponding #-transition in the CFSM.)

Our example also illuminates the difference between LALR(k) grammars and SLR(k) grammars. If a grammar G is LALR(k) but not SLR(k) for a particular value of k, then some state of G's CFSM must have overlapping simple k-look-ahead sets. And yet, if those sets are reduced by considering corresponding left contexts, they become mutually disjoint. In Chapter 5 our first example illustrates such a grammar, and we find that in general the functions necessary for computing look-ahead sets for LALR(k) grammars are the same complex functions which are necessary for general LR(k) grammars.

4.8 On the Extent of the SLR(k) Grammars

We should like to give the reader some intuitive feel for the usefulness and the extent of the SLR(k) grammars; that is, a feel for which grammars are SLR(k) and which are not. But alas, given our conceptual framework there seems to be no good intuitive explanation, so we resort to discussing some inclusion relations between SLR(k) and other well-known grammars.

In the Appendix we show that the "weak precedence" grammars of Ichbiah and Morse (I&M 69) are included in the SLR(1) grammars. Since those authors have shown that the "simple precedence" grammars of Wirth and Weber (W&W 66) are a subset of the "weak precedence" ones, it follows that the "simple precedence" grammars are SLR(1). Further,

it is easy to see from the proofs in the Appendix that, if the "precedence relations" were extended to include k symbols of right context, the resulting "right-extended weak-precedence" grammars would also be SLR(k). This leads us to suspect that the "(1, k) precedence" grammars of Wirth and Weber (W&W 66), the "(0, k) bounded context[†]" grammars of (F&G 68), and the "ICOR (0, k)" grammars of Lynch (Lyn 68) are all SLR(k).

But these inclusions really undersell the SLR(k) grammars, for the latter include many grammars which are in none of the above classes or their generalizations. They include <u>all</u> LR(0) grammars and many other LR(k) grammars for which arbitrary left context is necessary to make parsing decisions. Our example grammar G_0 is a case in point, as we noted in Section 2.4.

The ability of the CFSM for a given grammar to remember some left context which may be arbitrarily far to the left seems to arise because the confusion between contexts, which may obtain when two productions may be applicable to the same part of a string, is minimized in the CFSM in the following sense. If there exists an inadequate state N in the CFSM, then no matter how much left context we investigate we will not be able to make the parsing decision associated with N. The former statement

----

[†] These grammars should really have been called "(0, k) bounded right context" (Flo 64).

is implied by Lemma 3.3: if φ accesses N, then there exist characteristic strings $\varphi\#_p$ and $\varphi\theta\#_q$ and corresponding canonical forms.

Chapter 5

PARSERS FOR GENERAL LR(k) GRAMMARS

5.1   Objective

In the present chapter we continue the development of our parser-constructing technique.   However, before we proceed we (1) place the foregoing results into perspective by reviewing them from the viewpoint of a TWS attempting to construct a parser for a given grammar, (2) preview the results of the present chapter, and (3) disclaim any interest in these results from the practical viewpoint.

Review. Assume that we are given a CF grammar G and that we are to construct a parser for it.   We first assume that G is LR(0) and construct its CFSM. If the CFSM is adequate, G is LR(0) so we convert its CFSM to a DPDA and are finished. If, however, the CFSM is inadequate, we determine if G is SLR(1) by computing the simple 1-look-ahead sets for the transitions from the inadequate states.   If the sets for each inadequate state are mutually disjoint, G is SLR(1) so we convert the CFSM to an SLR1 FSM and then convert the latter to a DPDA with one-symbol look-ahead.   As noted above, we expect none of the grammars of interest to be LR(0), but most of them to be SLR(1).

Of course, it may be that there are one or more inadequate states which have overlapping, simple 1-look-ahead sets, in which case our work is not done. For the transitions from each such state we compute the simple

Chapter 5

PARSERS FOR GENERAL LR(k) GRAMMARS

5.1   Objective

In the present chapter we continue the development of our parser-
constructing technique. However, before we proceed we (1) place the
foregoing results into perspective by reviewing them from the viewpoint
of a TWS attempting to construct a parser for a given grammar, (2) preview
the results of the present chapter, and (3) disclaim any interest in these
results from the practical viewpoint.

Review. Assume that we are given a CF grammar G and that we are
to construct a parser for it. We first assume that G is LR(0) and construct
its CFSM. If the CFSM is adequate, G is LR(0) so we convert its CFSM to
a DPDA and are finished. If, however, the CFSM is inadequate, we deter-
mine if G is SLR(1) by computing the simple 1-look-ahead sets for the
transitions from the inadequate states. If the sets for each inadequate state
are mutually disjoint, G is SLR(1) so we convert the CFSM to an SLR1 FSM
and then convert the latter to a DPDA with one-symbol look-ahead. As noted
above, we expect none of the grammars of interest to be LR(0), but most of
them to be SLR(1).

Of course, it may be that there are one or more inadequate states
which have overlapping, simple 1-look-ahead sets, in which case our work
is not done. For the transitions from each such state we compute the simple

k-look-ahead sets for some values of k > 1. Since the time-efficiency of our ultimate parser will go down as k goes up (because look-ahead means multiple interrogation of some symbols), we shall undoubtedly be interested in only a restricted range of values of k, probably k ≤ 3 or so. If it turns out that the simple k-look-ahead sets are mutually disjoint, i. e. that G is SLR(k) for an acceptable value of k, then we can construct a DPDA-parser which has perhaps some one-symbol, and one or more k-symbol, look-ahead states.

In some cases, of course, we shall find that G is not SLR(k) for an acceptable k. However, there remains the possibility that G is LR(k) for such a k. For instance, our first example below is a grammar which is not SLR(k) for any k, but which is LR(1). In such a case we need more complex methods, first for determining if a grammar is LR(k) for a given k and second for constructing a corresponding parser if the former is the case.

Preview. These more complex methods are the subjects of the present chapter. In some cases (more of the LALR(k) grammars) we find that our modification of the CFSM is the same as for an SLR(k) grammar, but that the look-ahead sets are more difficult to compute then for the latter. In other cases, however, we find that some states must be split into several copies so the CFSM will remember more left context and so we can check corresponding right contexts to determine characteristic

strings. The determination of the appropriate state-splitting and corresponding look-ahead requires techniques which are substantially more complex, computationally, than our previous methods.

We introduce these notions by defining a set of grammars via some "sets of bounded-context pairs" and by showing how to extend our techniques to cover those of the latter grammars which are not SLR(k). The reasons for state-splitting come out rather naturally in the discussion, which leads eventually to a method for covering all LR(k) grammars.

Impracticality. The reader should keep in mind throughout this chapter that we expect to have to resort to these techniques only rarely, if at all. This expectation stems primarily from two sources. First, the grammars which were shown in Section 4.8 to be included in the SLR(k) grammars have been found to be quite useful for describing much of the syntax of many programming languages (F&G 62). The prime example is, of course EULER (W&W 66). Second, our own experience with languages, particularly with the language whose grammar and translator are presented in Chapter 7, has been especially encouraging in this respect. The latter grammar generates an extremely powerful, useful, and readable language with many constructs in common with languages like FORTRAN, ALGOL, EULER, PL/I, etc. The grammar was designed to be unambiguous, small, concise, and useful as a syntactical reference for programmers (i.e., for

the determination of operator precedences, associativities, etc.), but it was _not_ designed with our parser-constructing techniques in mind. Indeed, the techniques did not exist when the grammar was designed. And yet, the grammar turns out to be SLR(1).

Thus, the material in this chapter is here more because of a desire for completeness and for a fuller understanding of the LR(k) grammars than for its expected usefulness in practice. Consequently, we do not in this chapter concern ourselves particularly with the efficiencies of the techniques discussed. We are primarily interested in getting across the ideas.

5.2   "Bounded-Context" Examples

In this section we analyze two grammars which are not SLR(k). The first is an LALR(k) grammar for which the look-ahead sets can be determined by using a function which computes "bounded-context pairs". The second grammar is not LALR(k), but it is LR(k); i.e., its CFSM needs both state-splitting and look-ahead. The above mentioned function is found to be useful in the second case, also.
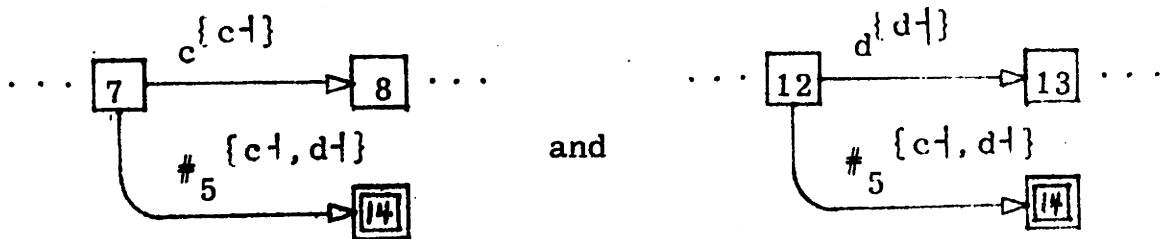
The two examples motivate the definition of a set grammars which we call "L(m)R(k)", and a parser-constructing technique to cover them. These grammars include, and their definition has similarities with the definition of, the "bounded right context" grammars (Flo 64); i.e., those grammars whose sentences can be parsed during a deterministic, left-to-

right scan with each parsing decision being made on the basis of the knowledge of a bounded amount of context surrounding the decision point.

<u>Example 1.</u> Consider the CFSM shown in Figure 5.1. It corresponds to grammar G_2 which contains the following productions

(0) $S \to \vdash E \dashv$ (3) $E \to b \, A \, c$

(1) $E \to a \, A \, d$ (4) $E \to b \, e \, d$

(2) $E \to a \, e \, c$ (5) $A \to e$

There are two inadequate states in the CFSM, states 7 and 12, both involving production 5 whose left part is A. Since G_2 generates only four strings, namely $\vdash aed \dashv$, $\vdash aec \dashv$, $\vdash bec \dashv$, and $\vdash bed \dashv$, it is trivial to compute the appropriate simple k-look-ahead sets. In particular, for any $k > 1$, $F^k(A) = \{c\dashv, d\dashv\}$, is the set for the $\#_5$-transitions; that for the c-transition from state 7 is $\{c\dashv\}$; i.e., c followed by the only member of $F^{k-1}(E) = \{\dashv\}$; and that for the d-transition from state 12 is $\{d\dashv\}$; i.e., d followed by the only member of $F^{k-1}(E)$. We represent this information, as we did in Chapter 4, using generalized symbols:

and
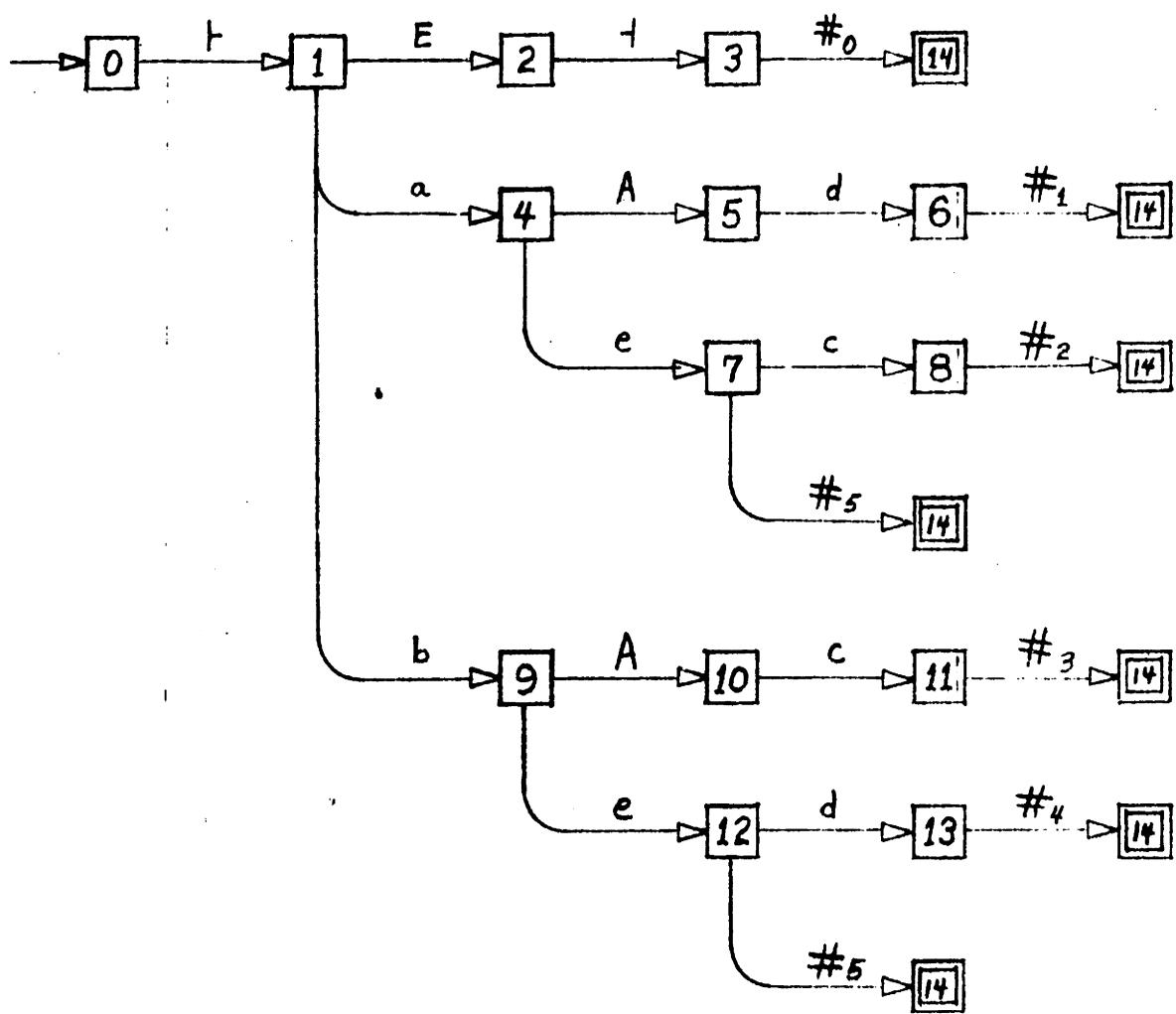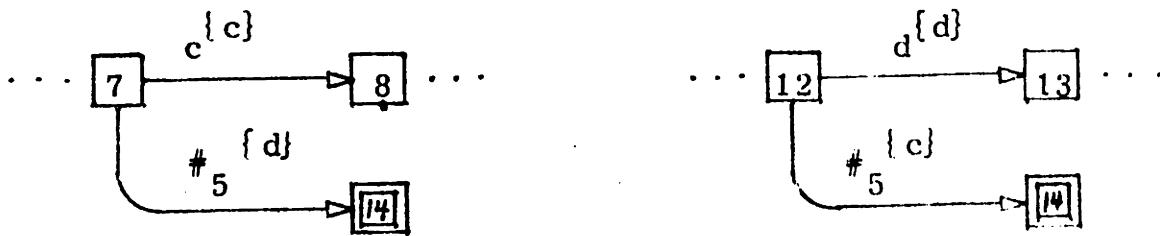
Figure 5.1. The CFSM for grammar G_2.

Clearly, G_2 is not SLR(k) for any k since the simple k-look-ahead sets have strings in common for both the inadequate states regardless of the value of k.

However, because the grammar generates only four strings, we can easily determine by exhaustive tests that the look-ahead sets can be reduced to those indicated as follows; i.e., that G_2 is LALR(1).

Clearly, a parser constructed using these look-ahead sets is a correct one for this grammar. But how do we compute these look-ahead sets in general?

For G_2 and many other grammars we can use the function $^m C^k$ which is defined below and whose value is a set of ordered pairs of left and right contexts. The definition requires the following two preliminary definitions: (1) if φ is a string, then $\varphi{:}m$ denotes the last m symbols of φ if $|\varphi| > m$ and φ otherwise, and (2) $\{(V^*, V_T^*)\}$ denotes the set of pairs whose first components are in V^* and whose seconds are in V_T^*.

<u>Definition 5.1</u>   Let $G = (V_T, V_N, S, P)$ be a CF

grammar and m and k be positive integers.   Then

$$\underline{{}^m C^k (\#_p)} = \{(\rho\omega{:}m,k{:}\beta) \in \{(V^*, V_T^*)\} \mid S \to^*_R \rho A\beta \text{ and}$$

production p is $A \to \omega\}$.

Each pair in this set consists of the last m symbols of a stack string

ω and the first k symbols of a corresponding input string β, respectively,

such that the canonical form $\alpha = \varphi\beta$ has a characteristic string $\varphi\#_p$.   In

other words, we have the ordered pairs of left and right contexts which

may surround a point in a canonical form where, during a deterministic,

left-ro-right parsing, we should decide to make a reduction using production

p.

The ${}^m C^k(\#_p)$ sets play a part in the definition of "L(m)R(k)"

grammars similar to that played by the $F^k(A)$ sets in the definition of

SLR(k).   The former sets can be computed in a way resembling the manner

in which the $F^k(A)$ sets are computed (recall the example on page 71), except

that, of course, <u>corresponding</u> left and right contexts must be tallied.

The former sets are certainly more difficult to compute than the latter,

but their computation is a reasonable next-step in our parser-generating

procedure.

In the case of grammar G_2 we have

$$^2 C^1 (\#_5) = \{(ae, d), (bc, c)\}$$

and we can observe that any string ending in ae will not access state 12, therefore the look-ahead set for the $\#_5$-transition need not contain the string d⊣. Similarly, the look-ahead set for the $\#_5$-transition from state 7 need not contain c⊣.[†] If we minimize the lengths of the strings in the look-ahead sets which result after these deletions, we arrive at the same sets deduced above.

Example 2. Our second grammar G_3 is rather similar to G_2. G_3's productions follow, and its CFSM is illustrated in Figure 5.2.

(0) $S \rightarrow \vdash E \dashv$ (4) $E \rightarrow b\,B\,d$

(1) $E \rightarrow a\,A\,d$ (5) $A \rightarrow e$

(2) $E \rightarrow a\,B\,c$ (6) $B \rightarrow e$

(3) $E \rightarrow b\,A\,c$

Again we have a grammar which is not SLR(k), since $F^k(A) = \{c\dashv, d\dashv\} = F^k(B)$ for any $k > 1$. In this case, however, the conflict is not as easily removed as was that of the previous case. If we compute the context pairs, we get

$$^2C^1(\#_5) = \{(ae, d), (be, c)\} \text{ and}$$

$$^2C^1(\#_6) = \{(ae, d), (be, d)\}.$$

[†] This example illustrates that the simple k-look-ahead sets may contain some strings which cannot appear as the prefix of the input string β of a canonical form $\alpha = \varphi\beta$ such that φ accesses the state in question; i. e., that the set $F^k(A)$ is not sufficiently restrictive. In the current case this "causes" the grammar not to be SLR(k). In other cases it may only cause the parser to be slower (because it checks too many possibilities for look-ahead) and to detect some errors somewhat later than it otherwise would. Recall the discussion in Section 4.7.
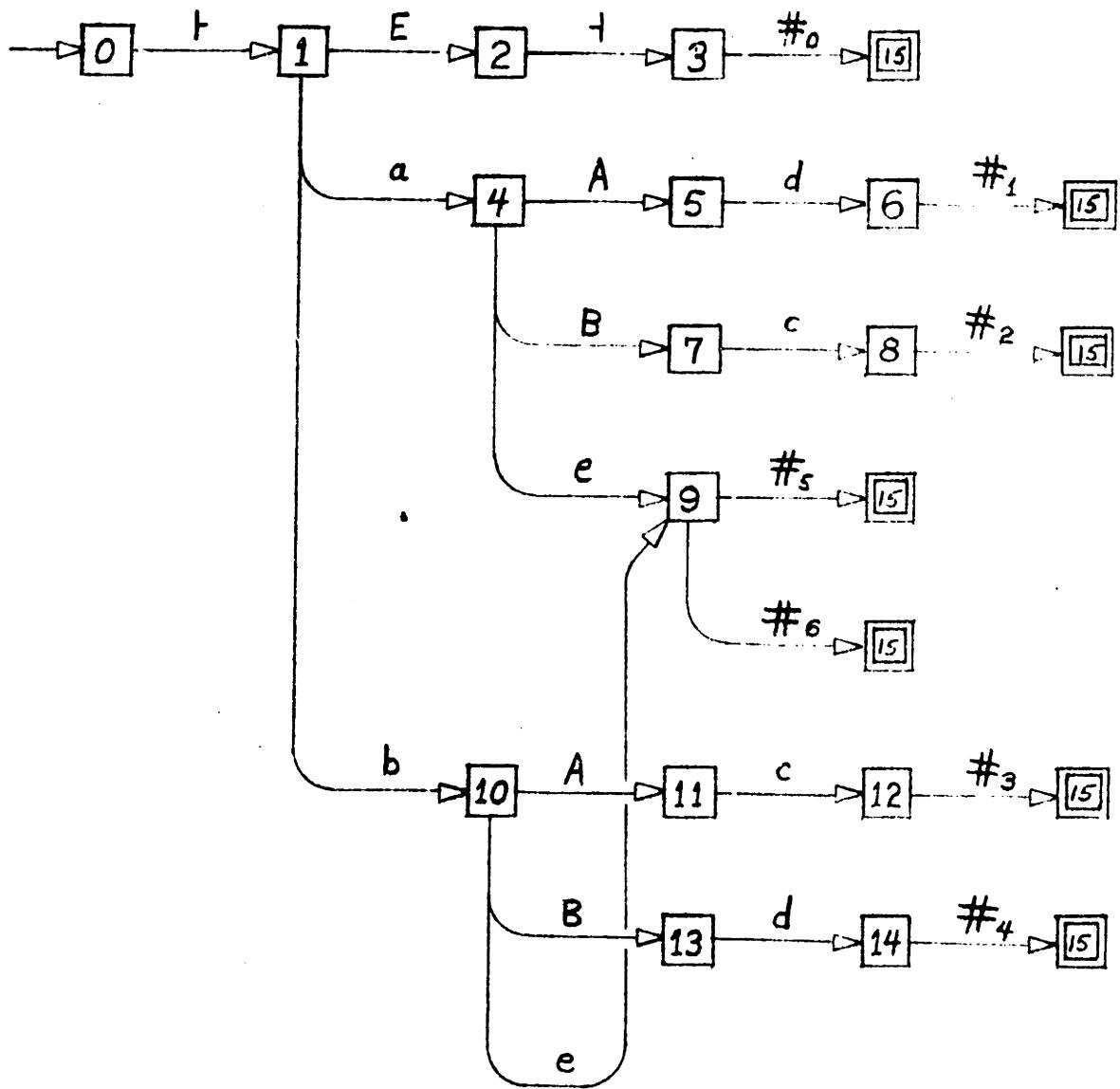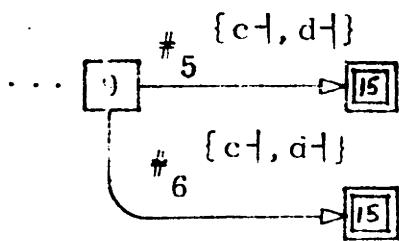
Figure 5.2. The CFSM for grammar G_3.

Analyzing this case as before we see that these context pairs imply no restrictions on the look-ahead sets, so we are left with the overlap:

There is, however, a simple solution in this case, too. Note that we could make the parsing decision associated with state 9 by looking at **both** our left and right contexts after arriving there. If we look to our left and see "ae" then, if we look to our right and see d, $\#_5$ is the correct transition, but if we see c, $\#_6$ is correct. On the other hand, if we see "be" to our left then the correspondences are d with $\#_6$ and c with $\#_5$.

Although we could build a parser for G_3 which decides whether to reduce using production 5 or 6 by looking at both left and right context, we prefer to eliminate the special look-to-the-left for two reasons: (1) it would be less time-efficient and also possibly less space-efficient than an alternate approach which we give below, and (2) we can easily generalize our other approach to cover all LR(k) grammars, but we cannot easily generalize this one.

What we chose to do is to "build into the machine" some extra memory for the extra left context. Note that in the case of grammar G_2 the machine

implicitly remembers the appropriate left context; i. e. , we know that if the machine is in state 7, the two-symbol, left context is "ae", whereas if the machine is in state 12, the context is "be". Unfortunately the CFSM of G_3 forgets this context; i. e. , when the machine is in state 9 the left context may be either "ae" or "be".

We solve the problem for G_3 by splitting state 9 into two copies, 9^1 and 9^2, as shown in Figure 5.3. Note that the look-ahead sets are indicated and that there is no overlap. The sets may be determined (in this case) just as they were for the CFSM of G_2, after the state splitting has been performed.

5.3   L(m)R(k) Grammars

The preceding examples motivate the definition of a set of grammars which can be described informally as those whose sentences can be parsed by using (1) corresponding CFSMs to determine potential characteristic strings and (2) sets of context pairs computed using $^mC^k$ to make parsing decisions associated with inadequate states. Our method of defining these grammars is similar to our method of defining the SLR(k) grammars, and we point out the similarities as we proceed.

We first need two preliminary definitions.

Definition 5.2 . Let G be a CF grammar, m be a positive integer, and N be a state of G's CFSM. Then the set $^mL(N)$
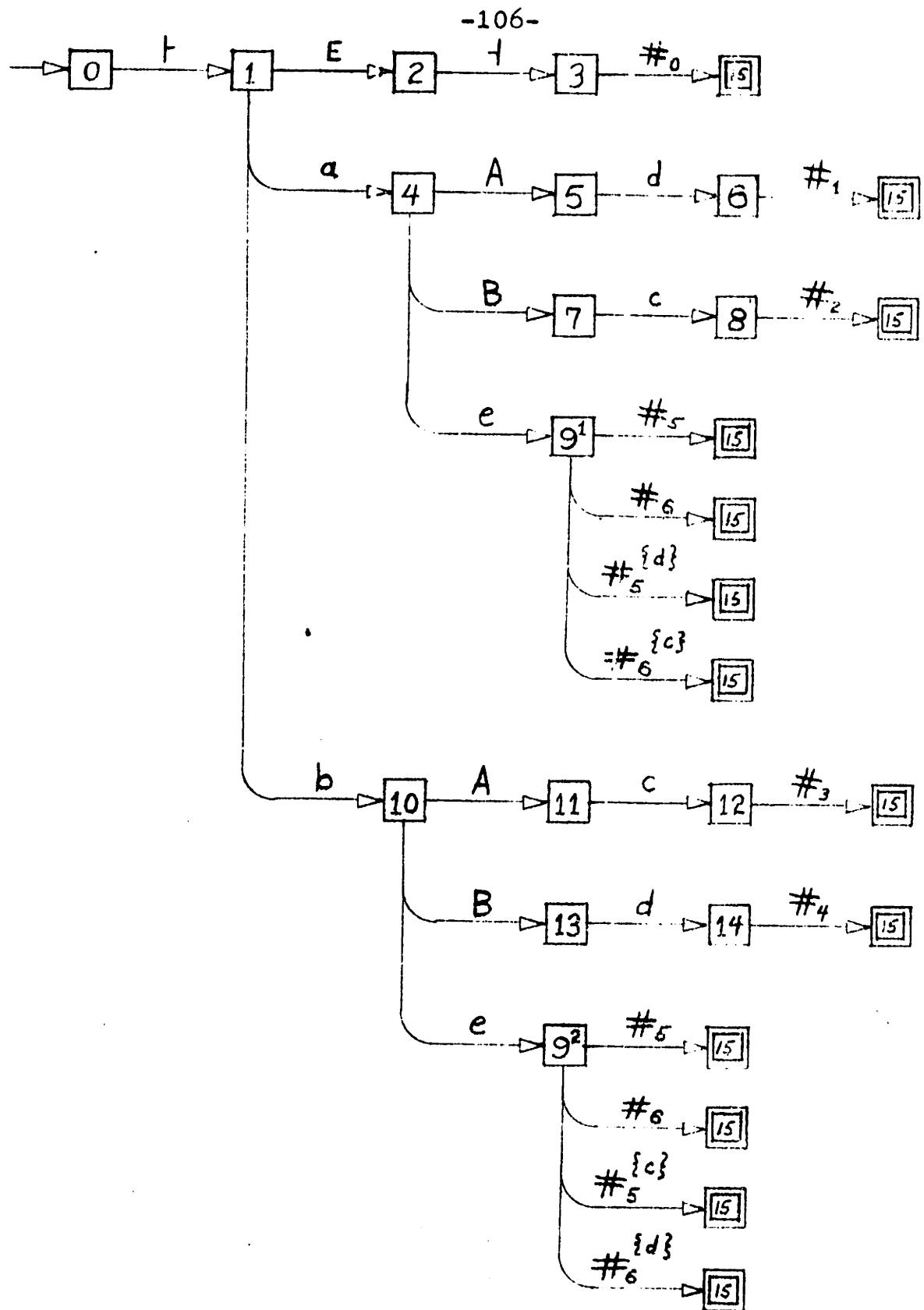
Figure 5.3. The CFSM of grammar G_3 after state-splitting and with look-ahead sets indicated via generalized symbols. The machine is later called the L2R1FSM of G_3.

is the set of left contexts of length m which end strings which access N; i.e.,

$$^{m}L(N) = \{(\varphi:m) \in V^{*} \mid \varphi \text{ accesses } N\}.$$

This set can be computed by following all possible pathes backwards through the CFSM, from N, for m steps or until the starting state is reached. Since the connectivity of the CFSM (graph) can be represented by a bit-matrix, the computation involves some fast bit-matrix manipulations (Pro 59).

Now we define some "sets of bounded-context pairs" associated with the transitions of CFSMs. The definition is to our "$L(m)R(k)$" definition what the definition (4.1) of simple k-look-ahead sets is to the SLR(k) definition.

Definition 5.3. (Recursive on the value of k.)

Let G be a CF grammar and m and k be positive

integers. There is associated with each transition T

of G's CFSM a set of (m, k)-bounded-context pairs,

$^{m}BC^{k}(T)$, as follows:

If T is a $\#_{p}$-transition from state N then

$$^{m}BC^{k}(T) = \{(\sigma, \mu) \in \, ^{m}C^{k}(\#_{p}) \mid \sigma \in \, ^{m}L(N)\}.$$

Or if T is a transition under the terminal t from state N

to state M then

$$^m BC^k(T) =$$

if k = 1 then $\{(\sigma, t) \in \{(V^*, V_T^*)\} \mid \sigma \in \ ^m L(N)\}$

otherwise $\{(\sigma, t\mu') \in \{(V^*, V_T^*)\} \mid \sigma \in \ ^m L(N)$ and

$(\sigma t, \mu') \in \ ^{m+1} BC^{k-1}(\text{some transition from M})\}$.

As in the case of simple k-look-ahead sets:
(1) we do not define these sets of pairs for transitions under nonterminals because our ultimate DPDAs will have no such transitions, and (2) although for the ease of definition sets are associated with every terminal-and #-transition of the CFSM, we are interested only in the sets for transitions from inadequate states.

The computation of these sets of pairs for a $\#_p$-transition primarily consists of computing $^m C^k(\#_p)$ and $^m L(N)$, as can be seen from the definition. For a transition under a terminal and for k > 1, the computation proceeds in a manner similar to that illustrated above (page 73) for the computation of a simple k-look-ahead set, except that, of course, corresponding left and right contexts must be tallied.

In the case of G_3's CFSM and for m = 2 and k = 1, we have for inadequate state 9:

$$^2 BC^1(\text{the } \#_5\text{-transition}) = \ ^2 C^1(\#_5) = \{(ae, d), (be, c)\} \text{ and}$$

$$^2 BC^1(\text{the } \#_6\text{-transition}) = \ ^2 C^1(\#_6) = \{(ae, c), (be, d)\} .$$

Of course, this agrees with our results above.

We now come to the main definition of this section.

<u>Definition 5.4</u>. Let G be a CF grammar and m and k
be positive integers. Let N be an inadequate state
(if any) of the CFSM of G. Then G is <u>L(m)R(k)</u> if
and only if the sets of (m, k)-bounded-context pairs
associated with the transitions from N are mutually
disjoint. Also, G is <u>L(0)R(k)</u>, <u>L(m)R(0)</u>, and <u>L(0)R(0)</u>,
if and only if it is SLR(k), LR(0), and LR(0), respectively.

We include the three special cases solely for completeness; we do not discuss
them further.

Note that grammar G_3 is L(2)R(1) by definition, as can be seen
from the disjoint sets $^2C^1(\#_5)$ and $^2C^1(\#_6)$ above.

5.4   <u>LmRkFSMs</u>

We now define an FSM which can be used by our modified stack-
algorithm of Section 4.3 to determine characteristic strings for an L(m)R(k)
grammar. This new machine is the CFSM modified to accept some extra
strings in which correspondence between (bounded) left and right contexts
is explicit.

<u>Definition 5.5</u>. Let G be an L(m)R(k) grammar. We construct
G's <u>LmRkFSM</u> from its CFSM as follows. For each inadequate
state N (if any) of the CFSM and for each string σ in $^mL(N)$, we
follow each path backward through the CFSM under the reverse

of σ, say to state M; from M we add a new path (of

new transitions and new states) under σ to a new

state N'; from N' there is a transition to the

accepting state under the generalized symbol X^{R_σ}

for each X-transition (X not a nonterminal) from N

such that $R_\sigma = \{\mu \in V_T^* | (\sigma, \mu)$ is in the set of (m, k)-

bounded-context pairs associated with the X-transition $\}$.

This results in a non-deterministic FSM. We change

the latter to an equivalent, deterministic FSM (via

well known techniques (H&U 69) and reduce the

result to form the LmRkFSM.

In the case of our example grammar G_3 the nondeterministic

FSM is shown in Figure 5.4. The reduced, deterministic version, i. e.

the L2R1FSM, is exactly the machine shown in Figure 5.3. Thus, the

state splitting and look-ahead sets which we deduced were necessary

above have "fallen out" of our procedure.

Proof. We need the following preliminary result to prove that the

LmRkFSM can, in fact, be used to determine characteristic strings.

Lemma 5.1. Let G be an L(m)R(k) grammar and N be

an inadequate state (if any) of G's CFSM. Every string

φ which accesses N also accesses a state N' of G's LmRkFSM

such that for every X-transition from N there is an X-transition

and, if X is not a nonterminal, an X^R-transition from N' such
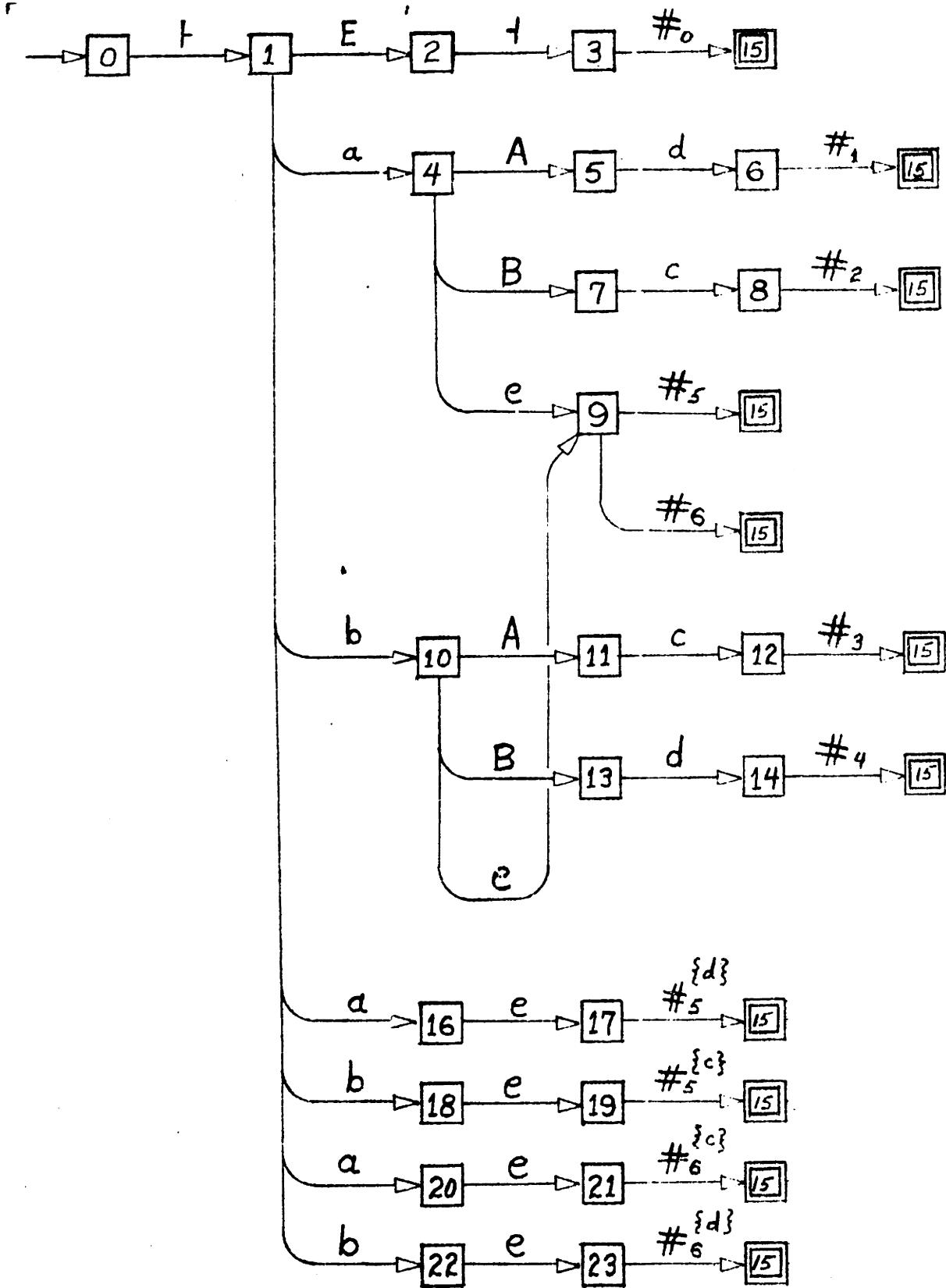
Figure 5.4. The nondeterministic FSM which is an intermediate result in the process of computing the L2R1FSM for grammar G_3.

that $R = R_{\varphi:m} = \{\mu \in V_T^* | (\varphi:m, \mu)$ is in the set of (m, k)-bounded-context pairs associated with N's X-transition$\}$.

Furthermore, there are no other transitions from N'.

Proof: By construction the LmRkFSM is a reduced, deterministic FSM which recognizes the characteristic strings of G plus some extra strings for each inadequate state N of G's CFSM. The extra strings are as follows. If the string φ accesses N, the LmRkFSM accepts the string φX^R where X and R are as given above. Now, because the machine is deterministic, any string, in particular φ, must access a unique state, say N', of the LmRkFSM; because both the CFSM and the LmRkFSM accept the characteristic strings, in particular those with prefix φ, there must be an X-transition from N' for each such transition from N; and because the LmRkFSM accepts the extra strings with prefix φ, state N' must have the extra transitions given above. Furthermore, we have accounted for all strings with prefix φ which are accepted by the <u>reduced</u> machine, so there can be no other transitions from N'.

Q. E. D.

The following two theorems serve the same purpose with respect to an LmRkFSM as do Theorems 4. 1 and 4. 2 with respect to an SLRkFSM.

<u>Theorem 5. 2</u>. Let G be an $L(m)R(k)$ grammar and $\alpha = \varphi\beta$ be a canonical form of G with characteristic string $\varphi\#_p$. Then the stack string φ of o accesses a state of G's LmRkFSM which is either (1) a reduce state whose only transition is under $\#_p$ or (2) a state (like N' of Lemma 5. 1) with transitions under the generalized symbols,

$$X_1^{R_1}, \ X_2^{R_2}, \ldots, X_n^{R_n}, \text{ for some } n \geq 2, \text{ such that}$$

$k{:}\beta$ is in R_i but not in R_j for $1 \leq i \neq j \leq n$, and such that $X_i = \#_p$.

Proof: Our proof depends upon the similarity of the CFSM and the LmRkFSM of G.   There are only two cases since φ must access either a reduce state or an inadequate state of the CFSM.   (1) If it accesses a reduce state of the CFSM, it must also access a reduce state of the LmRkFSM, because they both are deterministic and, although the LmRkFSM accepts more strings than does the CFSM, the extra ones are formed by adding symbols to the end of prefixes which access inadequate states but not reduce states of the CFSM.   Further, the only transition from the reduce state accessed by φ must be under $\#_p$, since the machine

accepts $\varphi\#_p$. (2) If φ accesses an inadequate state N' of the CFSM, it must access a state N' of the LmRkFSM with transitions under generalized symbols, by Lemma 5.1. Consider the sets $R_1, R_2, \ldots, R_n$ which are associated with the generalized symbols labeling transitions from N'. These sets must be mutually disjoint because they were derived from the mutually disjoint sets of context pairs associated with the transitions from N as follows: each set is the set of right contexts which are paired with a common left context, in particular $\varphi{:}m$, in the set of context pairs associated with some transition from N. Thus, $k{:}\beta$ can be in at most one of the sets. Furthermore, by Lemma 5.1 one of the generalized symbols $X_i^{R_i}$ has $X_i = \#_p$, and R_i must contain $k{:}\beta$ because it is computed from ${}^m C^k(\#_p)$ which by definition (5.1) contains $(\varphi{:}m,\ k{:}\beta)$.

Q. E. D.

<u>Theorem 5.3</u>. Let G be an L(m)R(k) grammar and $\alpha = \varphi\theta\beta$ be a canonical form of G with characteristic string $\varphi\theta\#_p$ such that θ is in V_T^* but $\theta \neq \epsilon$. Then, if φ accesses a state (like N' of Lemma 5.1) of G's LmRkFSM having transitions under the generalized

symbols, $X_1^{R_1}$, $X_2^{R_2}$, ..., $X_n^{R_n}$ for some $n \geq 2$, the string $k{:}\theta\beta$ is in R_i but not in R_j for $1 \leq i \neq j \leq n$, such that $X_i = 1{:}\theta$.

Proof: $k{:}\theta\beta$ may appear in at most one of the sets $R_1, R_2, \ldots, R_n$, since the sets are mutually disjoint, as was shown in the previous proof. $k{:}\theta\beta$ must appear in R_i such that $X_i = 1{:}\theta$ for the following reasons. G's CFSM accepts $\varphi\theta\#_p$; thus, if φ accesses state N of the CFSM, then there is a path leading from N which spells out $\theta\#_p$. It is easy to see from the definition of the set $^mBC^k$ of (m, k)-bounded-context pairs for a terminal-transition (in particular, one under $1{:}\theta$) that if $|\theta| \geq k$ then $(\varphi{:}m, k{:}\theta)$ is in $^mBC^k$, whereas if $|\theta| = n < k$ then every pair $(\varphi{:}m, \theta\mu')$ is in $^mBC^k$ such that μ' is in the set $\{\mu' \in V_T^* | \varphi\theta$ accesses state M of the CFSM and $(\varphi\theta{:}(m+n), \mu')$ is in the set of $(m+n, k-n)$-bounded-context pairs of some transition from M$\}$. Furthermore, $(\varphi\theta{:}(m+n), (k-n){:}\beta)$ must be in the latter set of pairs because there must be a $\#_p$-transition from M and $^{m+n}C^{k-n}(\#_p)$ includes the former pair by definition. Finally, since we have shown that the set of bounded-context pairs associated with the $(1{:}\theta)$-transition from N of the CFSM

contains $(\varphi:m, k:\theta\beta)$, Lemma 5.1 implies that the $(1:0)^{R_i}$ -

transition from N' of the LmRkFSM is such that R_i

contains $k:\theta\beta$.   Q. E. D.

Summary. In review, our technique for constructing an LmRkFSM

for a CF grammar G which is L(m)R(k) is as follows.   Compute the

context pairs associated with the transitions from the inadequate states

of G's CFSM.   Form a nondeterministic FSM by adding to the CFSM certain

new transitions and states.   The result is a nondeterministic machine which
recognizes some extra strings in which correspondences between left and

right contexts are explicit.   Change the machine to an equivalent, deter-

ministic FSM and and reduce it.   Violà!   Of course, we can minimize the

lengths of the strings in the look-ahead sets here just as we did for SLRkFSMS.

It should be clear from Theorems 5.2 and 5.3 that LmRkFSMs can

be used by our modified stack-algorithm just as are SLRkFSMs.   It

therefore follows that we can replace "SLRkFSM" with "LmRkFSM"

throughout the description of our technique for converting SLRkFSMs to

DPDAs to get the appropriate procedure for LmRkFSMs.

It should also be clear that for a given L(m)R(k) grammar, we

need to resort to the L(m)R(k) techniques only for inadequate states with

overlapping simple k-look-ahead sets.   To formalize this we would have

to prove theorems similar to Theorems 5.2 and 5.3 stated for a machine

having reduce states, inadequate states with simple k-look-ahead sets,

and states like N' of Lemma 5.1. That is, the new theorems would be a combination of Theorems 4.1 and 4.2, and 5.2 and 5.3, respectively. We do not state and prove these theorems since the notation would get out of hand and since the exercise would be of little intellectual value.

5.5 Parsers for General LR(k) Grammars

We now turn to the problem of constructing a parser for a general LR(k) grammar. That is, we want a method for covering grammars which are LR(k) but which are not SLR(k) or even L(m)R(k). Again we choose to illustrate the solution first by example and then to give the general solution. We do not formalize the results of this section because they are similar to those of the previous section, however, we do include an informal proof regarding the only significantly different feature.

Example. Consider the grammar G_4 (also similar to G_2) whose productions follow.

(0) $S \rightarrow \vdash E \dashv$ (5) $A \rightarrow e A$

(1) $E \rightarrow a A d$ (6) $A \rightarrow e$

(2) $E \rightarrow a B c$ (7) $B \rightarrow e B$

(3) $E \rightarrow b A c$ (8) $B \rightarrow e$

(4) $E \rightarrow b B d$

The corresponding CFSM is shown in Figure 5. 5.   It has one inadequate
state, state 9.

The grammar is not SLR(k) because $F^k(\#_6) = F^k(\#_8) \supseteq \{c\dashv, d\dashv\}$
for any $k > 1$. Since these sets overlap for all k, we need not bother
to compute the simple k-look-ahead set L_e^k for the e-transition; however,
we do so for the value as an example:

$$L_e^k = \{e\beta \in V_T^* \mid \quad \beta \text{ is in a simple (k-1)-look-ahead set} \atop \text{Associated with a transition from state 9}\}$$

$$= \{e\beta \mid \beta \text{ is in } \{c\dashv, d\dashv\} \cup L_e^{k-1}\}$$

$$= \{ec\dashv, ed\dashv\} \cup eL_e^{k-1}$$

$$= \{ec\dashv, ed\dashv, eec\dashv, eed\dashv, \ldots, e^{k-2}c\dashv, e^{k-2}d\dashv,$$
$$e^{k-1}c, e^{k-1}d, e^k\}$$

for $k > 2$. Obviously, this adds no new overlaps. Thus, the parsing
decision associated with state 9 about whether to read or reduce can be
made on the basis of one-symbol look-ahead ($\{e\}$ and $\{c, d\}$ are the
respective look-ahead sets); but the decision as to which reduction to make
cannot be determined via look-ahead alone, even if we look all the way to
the end of the string. Having discovered this, we need not discuss the
e-transition further below, although we do so, again for exemplary value.
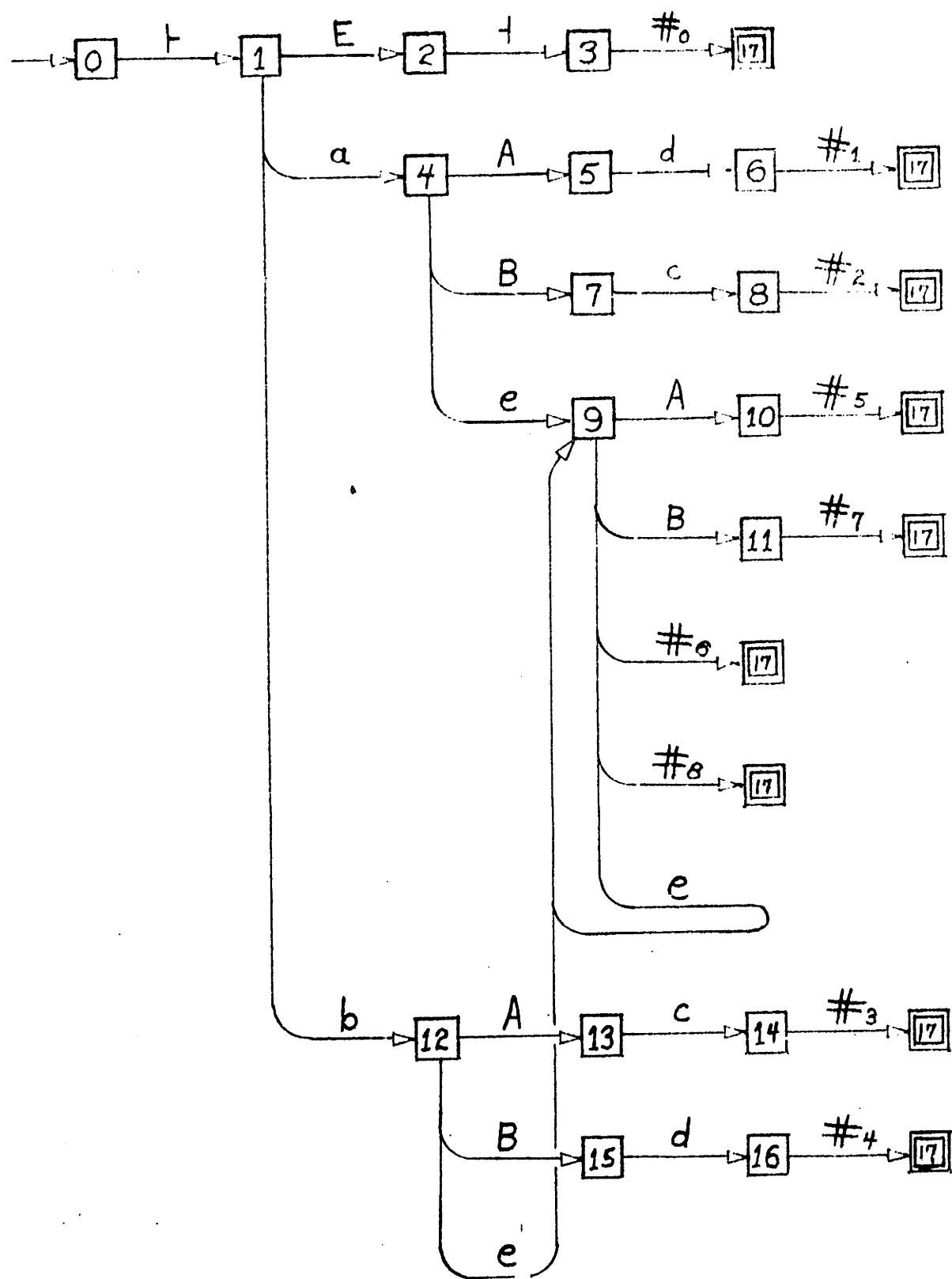
Figure 5.5. The CFSM for grammar G_4.

Neither is the grammar $L(m)R(k)$. Because it is small it is easy to compute by hand the context pairs for the transitions for state 9 which are as follows for $m > 2$ and $k > 2$:

$$\#_6\text{-transition}\_\{ \quad (\vdash ae, d\dashv), \quad (\vdash be, c\dashv),$$
$$(\vdash aee, d\dashv), \quad (\vdash bee, c\dashv),$$
$$\vdots$$
$$(\vdash ae^{m-2}, d\dashv), \; (\vdash be^{m-2}, c\dashv),$$
$$(ae^{m-1}, d\dashv), \; (\vdash be^{m-1}, c\dashv),$$
$$(e^m, d\dashv), \quad (e^m, c\dashv), \}$$

$$\#_8\text{-transition}\_\{ \quad (\vdash ae, c\dashv), \quad (\vdash be, d\dashv),$$
$$(\vdash aee, c\dashv), \quad (\vdash bee, d\dashv),$$
$$\vdots$$
$$(\vdash ae^{m-2}, c\dashv), \; (\vdash be^{m-2}, d\dashv),$$
$$(ae^{m-1}, c\dashv), \quad (be^{m-1}, d\dashv),$$
$$(e^m, c\dashv), \quad (e^m, d\dashv) \}$$

e-transition__

$$\left\{\left(\left\{\begin{array}{l}\vdash ae, \\ \vdash aee, \\ \vdots \\ \vdash ae^{m-2}, \\ ae^{m-1} \\ e^m, \end{array}\right\}, \left(\left\{\begin{array}{l} ed\dashv, \\ eed\dashv, \\ \vdots \\ e^{k-2}d\dashv, \\ e^{k-1}d, \\ e^k \end{array}\right\}\right)\right)\right\} \cup \left\{\left(\left\{\begin{array}{l}\vdash be, \\ \vdash bee, \\ \vdots \\ \vdash be^{m-2}, \\ be^{m-1} \\ e^m, \end{array}\right\}, \left(\left\{\begin{array}{l} ec\dashv, \\ eec\dashv \\ \vdots \\ e^{k-2}d\dashv, \\ e^{k-1}d, \\ e^k \end{array}\right\}\right)\right)\right\}$$

where the notation $\{(\{\ \}), (\{\ \})\}$ is to be understood as was $\{(V^*, V_T^*)\}$ above. Because the context pairs $(e^m, c\dashv)$ and $(e^m, d\dashv)$ appear in both the

sets associated with the $\#_6$ and $\#_8$-transitions, the grammar is not L(m)R(k), and our informal solution of looking at a finite amount of both left and right context to make the parsing decision associated with state 9 will not work here. The problem is, of course, that the left context in which we are interested.(the a or b) may be arbitrarily far to our left[†].

The essential reason that we shall be able to solve this problem is that, although the context of interest can appear arbitrarily far to the left at the time we need it, the states and transitions of the CFSM which are involved in reading that context are only a finite distance from the inadequate state (since the CFSM is a finite machine!). Our solution again involves state-splitting, but this time to get the machine to remember extra context which may be arbitrarily far to the left.

For instance, the CFSM of Figure 5.6 must have state 9 split into two copies so it will remember whether an a or b is to its left. The appropriate FSM is shown in Figure 5.6. Note that because of space limitations we have drawn the FSM in the abbreviated form. Because grammar C_4 is small the reader should easily be able to convince himself that this is

_____

[†] In the case of Grammar G_0 the CFSM is obliging enough to remember the a or b for us. The difference seems to be that for G_0 the a or b has no implication about the symbols in the right context, but only about how they should be parsed, whereas with G_4 there is a correspondence between left and right symbols. We see no general way of discovering such complexities in a grammar except by trying to generate a parser for it.
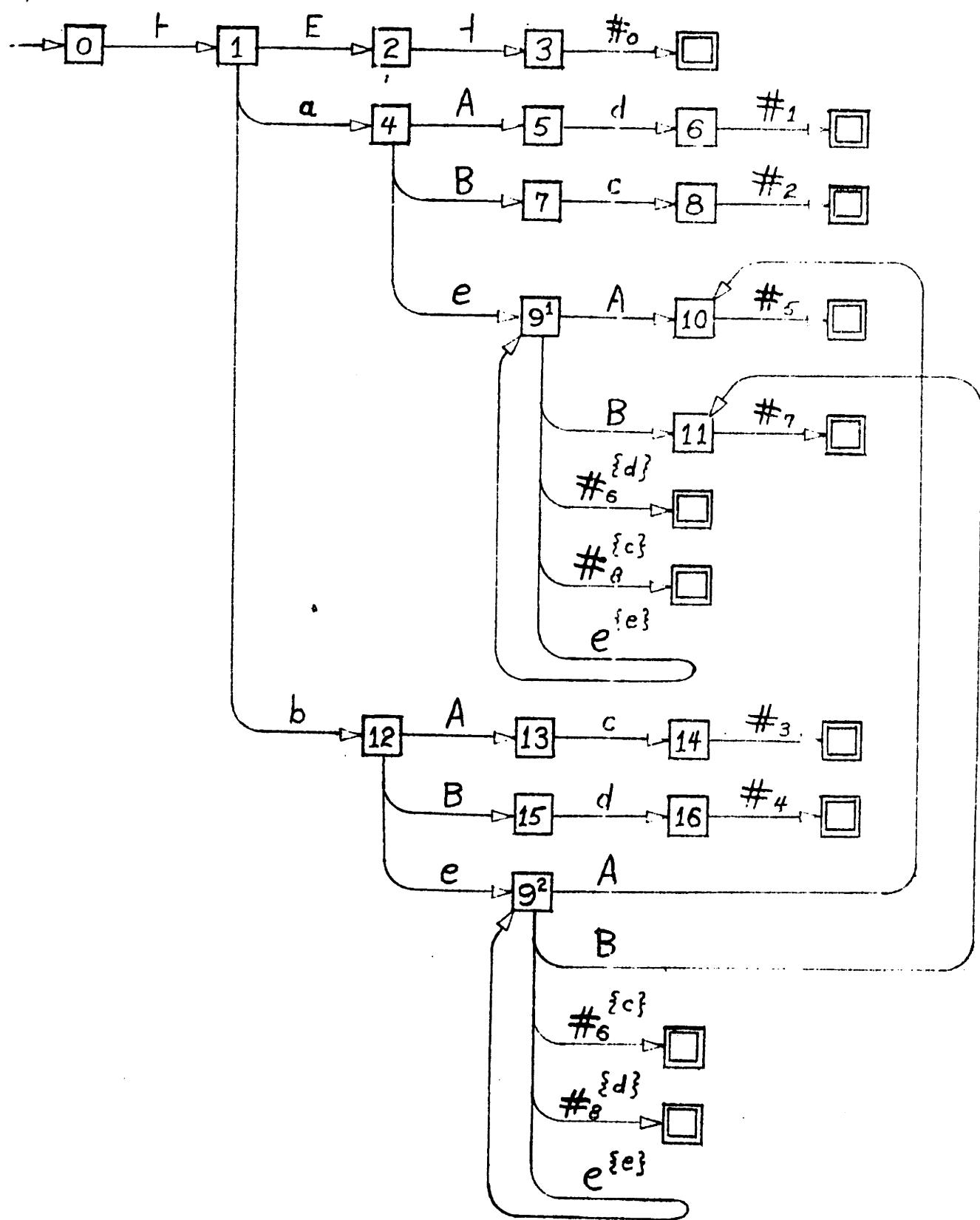
Figure 5.6. The CFSM of grammar G_4 after state-splitting and with look-ahead sets indicated via generalized symbols; i.e., the (abbreviated) LR1FSM for G_4.

the appropriate FSM. Note that no more than one-symbol look-ahead is necessary, therefore the grammar is LR(1).

LRkFSMs: Now, for the general case we have two questions confronting us: (1) how do we compute the necessary state-splitting, and (2) how do we compute the look-ahead sets? The answers to these two questions are rather similar to those for L(m)R(k) grammars. We answer these questions next, continuing to use G_4 as an example, and we justify our answers afterwards.

In the general case the left context which must be remembered may be anywhere to our left, thus we must search for it all the way back to the beginning of the string. In terms of the CFSM this means all the way back to the starting state. The procedure for a general LR(k) grammar G whose CFSM has an inadequate state N goes as follows.

We first find the set of strings $^kLL(N) = \{\varphi \in V^* | \varphi$ accesses N via a path through the CFSM which contains no more than k instances of any given cycle of states}. Because our CFSM can be represented via a directed graph some of the results of graph theory are appropriate for use in computing such paths and the corresponding strings. In fact, well known, even fast, techniques exist for doing just that (Pro 59) (War 62).

In the case of our LR(1) grammar G_4 the strings are $\vdash$ae, $\vdash$aee, $\vdash$be, and $\vdash$bee, and they correspond to paths which can be

represented by the sequences of state names: 0, 1, 4, 9 (no cycles); 0, 1, 4, 9, 9, (one cycle, 9 to 9); 0, 1, 12, 9; and 0, 1, 12, 9, 9, respectively, none of which contain more than $\kappa = 1$ instance of the only cycle in G_4's CFSM.

Next we compute the set $^k LC(N) = \{\varphi X^{R_{\varphi, X}} | \varphi$ is in $^k LL(N)$ and $X^{R_{\varphi, X}}$ is a generalized symbol such that there is an X-transition (X not a nonterminal) from N and $R_{\varphi, X} = \{(k:\theta\beta) \in V_T^* | \varphi\theta\beta$ is a canonical form with characteristic string $\varphi 0\#_p$, and $X = (1:\theta\#_p)\}$.

Each generalized symbol $X^{R_{\varphi, X}}$ represents the set of terminal strings of length k which may follow φ in a canonical form $\alpha = \varphi\beta$ such that the characteristic string of α accesses state N and then takes the X-transition; i.e., it is the look-ahead set corresponding to φ and the parsing decision associated with the X-transition. We reference a method for computing these sets below.

For G_4 the set of such $\varphi X^{R_{\varphi, X}}$ for k = 1 is:

$$\{ \quad \vdash ae\#_6^{\{d\}}, \quad \vdash aee\#_6^{\{d\}}, \quad \vdash be\#_6^{\{c\}}, \quad \vdash bee\#_6^{\{c\}},$$
$$\vdash ae\#_8^{\{c\}}, \quad \vdash aee\#_8^{\{c\}}, \quad \vdash be\#_8^{\{d\}}, \quad \vdash bee\#_8^{\{d\}},$$
$$\vdash aee^{\{e\}}, \quad \vdash aeee^{\{e\}}, \quad \vdash bee^{\{e\}}, \quad \vdash beee^{\{e\}} \quad \}$$

as the reader may compute for himself.

We now form a nondeterministic FSM in a manner similar to that for an L(m)R(k) grammar. For each string $\varphi X^{R_{\varphi, X}}$ in $^k LC(N)$ we add to the starting state of the CFSM a new path (of new transitions and new states) under $\varphi X^{R_{\varphi, X}}$ leading to the accepting state. We convert the mahine to a deterministic device, reduce it, minimize the strings in the look-ahead sets, and presto --- the appropriate FSM with look-ahead sets built in; i.e., the "LRkFSM".

To specify the procedure fully we must provide two things: (1) a procedure for computing the look-ahead sets implicit in the generalized symbols $X^{R_{\varphi, X}}$, and (2) the reason why we need to consider only such left contexts (the φ's) as take paths through the CFSM which contain no more than k occurrences of a given cycle.

Regarding the first point, we use the simple expedient of a reference. Knuth (Knu 65, see especially page 617) has already solved this problem. His parsing algorithm in a sense computes the states of our CFSM dynamically, as it is parsing a string. However, it also computes much more information, all of which is bundled neatly into what are called "state sets". If we simply apply his algorithm to each string φ, we can deduce the look-ahead sets from the "state set" computed just after the algorithm has read the last symbol of φ, as he describes in "Step 2" of the algorithm. (His set "Z" is the look-ahead set for all transitions under terminals, and the set "Z_p" is the look-ahead set for a $\#_p$-transition.)

Regarding the second point, we provide an informal proof. Recall the canonical derivation of a form α of CF grammar G illustrated on page 42 . Let $\alpha = \varphi\beta$ where $\varphi = \omega_0\omega_1\ldots\omega_m\gamma$ and $\beta = \gamma''\omega_m''\ldots\omega_1''\omega_0''$ for some γ and γ'' such that $\gamma\gamma' = \omega$ and $\gamma' \to^* \gamma'' \in V_T^*$. Note the correspondence between left and right contexts: each ω_i (or γ) has a matching ω_i'' (or γ''). In the next two paragraphs we investigate the implications of this correspondence with regard to the computation of look-ahead strings corresponding to a particular φ which accesses an inadequate state N of G's CFSM.

We consider first a string φ spelled out by a path through the CFSM which accesses a given cycle only once. That is, φ first accesses a state in the cycle, then goes around the cycle several, say r, times, and then accesses state N. In this case φ can be written

$$\omega_0\omega_1\ldots\omega_i(\omega_{i+1}\ldots\omega_{i+n})^r\omega_{i+nr+1}\ldots\omega_m\gamma.$$

The subexpression $(\ldots)^r$ cannot include only a part of an ω_i. Since r can have any nonnegative integral value and since there are only a finite number of productions, the canonical derivation must also have a cycle in it; i.e., we can write the numbers of the productions used in the derivation

$$p_1p_2\ldots p_i(p_{i+1}\ldots p_{i+n})^r p_{i+n\,r+1}\ldots p_n\,p.$$ But each application of a production in this sequence adds a whole ω_i to the left context, never a part of one. Thus, the first k symbols of the right context can be written

$$k{:}\beta = k{:}\gamma''\omega_m''\ldots\omega_{i+n\,r+1}''(\omega_{i+n}''\ldots\omega_{i+1}'')^r\omega_i''\ldots\omega_1''\,\omega_0''.$$

But if $r \geq k$, this is equivalent to $k: \ldots (\ldots)^k \ldots$, since in the worst case

where $|\gamma'' \omega''_m \ldots \omega''_{i+n \; r+1}| = 0$ and $|\omega''_{i+n} \ldots \omega''_{i+1}| = 1$ we have $k:\beta =$

$(\omega''_{i+n} \ldots \omega''_{i+1})^k$. The point is that the look-ahead strings for any φ

which accesses a cycle r times in succession, where $r \geq k$, are exactly

the same as those for a similar φ which accesses the cycle only k times

in succession.

We must now consider the case where φ accesses a cycle once,

goes around it several, say r_1, times, wanders around the machine

elsewhere, returns, goes around the cycle several more, say r_2, times,

etc. Because the notation gets out of hand otherwise, we shall argue the

case for only two separate accesses of the cycle and let the reader

generalize for himself. In this case β can be written

$$\gamma'' \omega''_m \ldots \omega''_{i_2+n_2 r_2+1} (\omega''_{i_2+n_2} \ldots \omega''_{i_2+1})^{r_2} \omega''_{i_2} \ldots \omega''_{i_1+n_1 r_1+1}$$

$$(\omega''_{i_1+n_1} \ldots \omega''_{i_1+1})^{r_1} \omega''_{i_1} \ldots \omega''_1 \omega''_0 \text{ for } i_2 \geq i_1 + n_1 r_1. \text{ In the worse}$$

case where $|\gamma'' \omega''_m \ldots \omega''_{i_2+n_2 r_2+1}| = 0$ and $|\omega''_{i_2} \ldots \omega''_{i_1+n_1 r_1+1}| = 0$ and

$|\omega''_{i_1+n_1} \ldots \omega''_{i_1+1}| = 1$ and $|\omega''_{i_2+n_2} \ldots \omega''_{i_2+1}| = 1$,

we see that if $r_1 \geq k$ then $k:\beta = (\omega''_{i_2+n_2} \ldots \omega''_{i_2+1})^{r_2} (\omega''_{i_1+n_1} \ldots \omega''_{i_1+1})^{r'}$
where $r' =$

maximum of $k - r_2$ and zero. Thus, if $r_2 \geq k$ the look-ahead strings for

φ are the same as for a similar φ but with $r_2 = k$ and $r_1 = 0$. However,

if $0 \leq r_2 < k$, they are the same as the ones for a similar φ but with $r_1 = k - r_2$.

In conclusion (and generalizing), the look-ahead strings for a given φ whose path through the CFSM goes around a given cycle a total of $r \geq k$ times are the same as those for a similar φ with only k such loops[†]. Therefore, our procedure above which computes look-ahead sets by considering only the φ's with no more than k such loops computes all possible look-ahead strings.

Conclusion. Here, as above, it seems clear that these more complex techniques need be applied only to inadequate states for which our simpler techniques will not work. It is also clear that the procedure for converting an LRkFSM to a DPDA is the same as that for an LmRkFSM.

What we have not provided thus far, however, is a method which is convenient to use in the above procedure for deciding if a given grammar is LR(k). It should be clear from our informal proof above and the definition (2.2) of LR(k) grammars that a CF grammar G is LR(k) if and only if, for each inadequate state N of G's CFSM and each string φ in $^k LL(N)$, the set $\{R_{\varphi, X} |$ where is an X-transition (X not a nonterminal) from N$\}$ is a set of mutually disjoint sets. This, of course, means that the look-ahead sets

---

[†] Actually we could do better than this. If all the cycles were "separate" from each other in the CFSM, we could consider only φ's with a total of k loops around any cycle. Unfortunately our proof would get excessively complicated to cover the case where one cycle is a part of another. We are satisfied with the above simple, sufficient condition because our purpose here is to show that the task of computing the look-ahead sets is a finite one, not to develop a method for computing the sets which requires a minimum of time.

for each inadequate state of the LR*FSM are mutually disjoint.

5.6   <u>Comments</u>

We noted above that Knuth's LR(k) parsing algorithm in a sense computes the states of our CFSM dynamically, as it parses a string. Actually, we believe this to be accurate only in the case $k = 0$. If $k \geq 1$, Knuth's algorithm computes the states of a machine much larger than our CFSM. In effect, the processes of splitting states and computing look-ahead sets are bound together in his algorithm. Consequently, for $k = 1$ the number of states computed is the number of states of the CFSM times some number having to do with the number of symbols which appear in the look-ahead sets. In practical cases this multiplicative factor is impractically large (Kor 69). Further, the size of the machine increases rapidly with increasing k.

Korenjak (Kor 69) noticed that the multiplicative factor depends upon the size of the look-ahead sets, and he proposes a parser-construction technique to reduce the effect. He proposes that the grammar be partitioned into several sub-grammars, that a sub-parser be generated for each sub-grammar by using Knuth's algorithm for each, and that the desired parser be constructed by combining the sub-parsers appropriately. Since the look-ahead sets for each sub-grammar are much smaller than those for the entire grammar, the multiplicative factor for each sub-parser is much smaller than that for a parser constructed directly for the entire

grammar. Further, a relatively small number of extra states are required to combine the sub-parsers.

In a sense, we have taken Korenjak's approach to the extreme by analyzing the grammar production-by-production; or more precisely, we analyze the CFSM inadequate-state-by-inadequate-state. Our method seems to cause nearly a minimum of state-splitting and look-ahead. We leave as questions for future research, however, whether or not it does cause such minimums, and if not, how it could be modified to do so.

Chapter 6

TRANSLATORS

6.1   Philosophy

Thus far we have followed the lead of Knuth, concerning ourselves solely with grammatical analysis. However, our interest is ultimately in translators rather than parsers. We have addressed the parsing problem first because it gives us a convenient basis from which to address translation, a fact which will become abundantly clear below when we see that our method of specifying translations is based directly on CF grammars. It will follow that our translators can be based directly on our parsers.

We now, therefore, abandon the grammatical analysis approach and adopt the philosophy of Lewis and Stearns (L&S 68), namely that

"Implementing a translation should be regarded as an

automata theory problem of machine capability and

efficiency rather than as a problem of grammatical

analysis. "

We deal only with the capabilities of DPDAs here, so our main concern is in improving efficiency by making transformations on our machines which preserve their input/output relations. Of course our yen to perform transformations must be tempered by the implications of our desire to implement the translators ultimately on a modern digital computer.

Actually, we have already been abiding by part of this philosophy.
in preparation for the material in this chapter. In effect, we have
regarded parsers as translators which translate sentences into parses,
i.e., into strings of productions or production numbers. Although we
found it convenient to discuss grammatical analysis at first from the
string-manipulation viewpoint, we certainly made it a point to convert
to the automata-theory viewpoint when we converted our string-manipula-
tion parsers to DPDAs.

6.2   Objective

It is the objective of this chapter to show how our results are
relevant to (a) the specification of translations of programming languages,
and (b) the construction of compilers from those specifications.

In Section 6.3 we motivate an interest in string-to-string translators
similar to our DPDA-parsers. We do by discussing some well-known
approaches to compiler construction.

In Section 6.4 we show why we are not interested in parses, per
se. We motivate an interest in string-to-tree translators, each of which
can be regarded as a concatenation of two subtranslators: the first being
a string-to-string translator which maps input strings into strings (sequences)
of tree-building directives, and the second being a string-to-tree translator
which maps strings of directives into trees (by obeying the directives).

In Section 6.5 we present a formalism based on CF grammars for specifying string-to-string translations, and in 6.6 we show how to convert our DPDA-parsers to corresponding translators. The latter feat is trivial, but some nontrivial optimizations ensue. We formalize only string-to-string translators because our (linear) automata-theoretic approach seems inappropriate when discussing trees.

Finally, in Section 6.7 we present a compiler model,and in 6.8 we show the relevance of our results to the specification of languages, translations, and compilers; i.e., to TWSs.

We emphasize that the only formal results in the present chapter are those of Sections 6.5 and 6.6. The remainder of the chapter is intended as motivation for those two sections and discussion of their relevance to TWSs.

6.3   Syntax Directed Compilers

Many compilers in existence today, whether written by hand or partially or wholly written by a TWS, are termed "syntax directed"[†] compilers. The approach of Cheatham (Che 67) is fairly representative for our purposes here. He advocates the use of "augments" to productions to enhance the descriptive power of CF grammars so they can be used to specify programming languages fully[††]. These "augments" are in the

---

[†] In the sense we have in mind here the term should perhaps be "syntax-analyzer directed".

[††] What amounts to a generalization and a formalization of this approach can be found in (Knu 66).

form of "actions", "conditions", and "interpretations" associated with

the productions. He envisions a parser as an "engine" operating in an

"environment". As the parser parses a string it drives other mechanisms

which (a) execute "actions", thus causing the 'environment" to change, (b)

check "conditions" in the "environment", thus providing context-sensitivity,

and (c) compute "interpretations" ("values", "meanings", or "semantics").

The auxiliary mechanisms are activated each time the parser makes a

reduction, and they then compute the "augments" associated with the

corresponding production. When the parser has finished parsing the input

string, intermediate object code has been output via "actions" and any

relevant tables are available via "interpretations" associated with the

entire program.

A basically similar approach is one due to Feldman (Fel 64) in

which "EXEC n" routines are associated with "Floyd-Evans productions"

(Eva 65) comprising a parsing program. Roughly speaking, the "EXEC

n" routines are the analogues to Cheatham's "augments".

An approach similar to one or the other of these two, or similar

to our own approach (described below) where an "abstract syntax tree"

or "parse tree" is built, is used in every compiler or TWS effort

described in (F&G 68). Implicit in and fundamental to the compilers of

all these schemes is a string-to-string translation: a translation from

the input string to a string (sequence) of commands to mechanisms to

compute "augments", or of calls to "EXEC n" routines, or "semantic routines", or "generators", etc. Thus, if the reader is partial to one of these schemes in particular, he may think of the "output symbols" below as the appropriate commands or calls to routines, and he may think of our "CFS's" as the corresponding string-to-string translators. For our purposes we think of the "output symbols" as tree-building directives, as we discuss next.
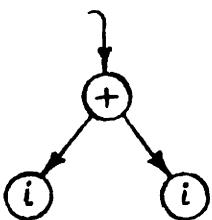
6.4   Abstract Syntax Trees

In previous chapters we devoted much time to the development of DPDA-parsers; i. e., string-to-string translators which map input strings into parses.   In the present section we discuss the reasons why parses, as such, are not as appropriate for purposes of compiling as are the strings of tree-building directives referred to above.

Inefficient coding. There are two problems with parses, per se: (1) they contain some information in which we are not interested, and (2) the information which we do desire is not explicit.

For instance, for grammar G_1 the string $\vdash i + i \dashv$ can be reduced to $\vdash E + T \dashv \rightarrow \vdash E \dashv \rightarrow S$. But for purposes of compiling we do not care that the reductions for the first i were $i \rightarrow P \rightarrow T \rightarrow E$ and the second were $i \rightarrow P \rightarrow T$, nor do we care which reductions were made first or which particular nonterminals were used. The only information which is both implicit in the parse and of interest to us is that one i is the left operand

of the operator + and that the other is the right operand. If we were

mathematically inclined, we might represent this information via a

functional form; e. g. , + (i, i) or PLUS (i, i). However, for purposes of

discussing compiling activities and for an explicit representation of the

"structure" which is implicit in parses, we find it more convenient to

represent the above information via the following graph (tree):

Such a graph, representing the "structural" information or relationships

which are implicit in a parse, has been called by some an "abstract

syntax tree" (W&E 69, Lan 66, McC 66). We elaborate on the reasons

for this name in Section 6. 7.

Now, (a) if we are not interested in all the information implicit

in a parse, it would be inefficient for our compiler to generate it. Further,

(b) if an abstract syntax tree represents all and only the information

implicit in the parse which is of interest for further compiling activities

and (c) if the tree can be represented in some convenient and useful way

in a computer, then our results would be more useful if we could show

(1) how to specify a translation from strings to trees in a manner based

on CF grammars and (2) how to convert our parsers to efficient translators

which affect the corresponding, string-to-tree translations.

Conceptual modularity. Although if-clauses (a) and (b) of the preceding paragraph probably represent good assumptions, (c) is subject to some question, partly because it is not clear that the abstract syntax tree, per se, ever needs to be built during the compiling process. But we do not let this stop us for the following reason: even if our compiler does not actually construct an abstract syntax tree, we can regard the process conceptually as building it.

We argue that even the string-to-string translator which we develop below can be regarded as the concatenation of two subtranslators, the first being a parser and the second affecting a translation from parsers to the desired strings. However, after we have thoroughly investigated the two subtranslators, we see that they can easily be combined so as to save us actually having to generate the parse.

Similarly, we can regard preliminary compiling activities as performing a translation from input string to abstract syntax tree and subsequent activities as performing a translation, again conceptually composed of several subtranslations, from abstract syntax tree to object code[†]. The advantage of this approach relative to a less modular one is, of course, that the otherwise complex task of compiling is broken into several relatively simple subtranslations. Hopefully, when we are finished analyzing

---

[†] This approach was largely inspired by (W&E 69) which in turn was based on (Lan 66). See Section 6.7.

the subtranslators separately, we will be able to see how to put them together in such a way as to minimize redundancies. This may mean that the abstract syntax tree, per se, need never actually be constructed.

Example. As an example of what we mean by "tree-building directives", consider the following. If our string-to-string translator maps the example string $\vdash i + i \dashv$ of above into the string i i ┬, we can regard the latter as the following sequence of directives: build a terminal node with name i; build another terminal node with name i; build a non-terminal node with name +, with right (or second) son the last node built, and with left (or first) son the next-to-the-last node built.

In general, if our tree builder is always to construct nonterminal nodes whose sons are the last few nodes constructed, and in the same order, the sequence of directives must be a linear representation of the tree which is commonly called a "suffix form". (See (Che 67) for a thorough discussion of the correspondences between trees and their linear representations.) Further, the device can keep track of the nodes it has built by maintaining a push-down stack of pointers to them, and the pushing and popping of this stack will occur in a sequence closely corresponding to that of the stack of our DPDA-translator which issues the directives. Our compiler model and another example below should shed more light on this subject.

6.5 Transduction Grammars, Translations

We now get down to business. As our method of specifying string-to-string translations we choose a technique which is based on CF grammars and which fits naturally and conveniently with our notions about both grammars and automata. The first and fourth paragraphs below are taken almost directly from (L&S 68).

A transduction grammar G_t based on a CF grammar G is a triple (G, V'_T, g), where V'_T is a set of output terminals and g is a mapping defined on G which associates a string ω' in $(V'_T \cup V_N)^*$ with each production $A \to \omega$ in G and which specifies a one-to-one correspondence that pairs each instance of a nonterminal in ω with an instance of the same nonterminal in ω'. We refer to the string ω' as the transduction element[†] for production $A \to \omega$.

We are interested, for the present at least, only in simple suffix transduction grammars (SSTGs), since they are trivially adaptable to our results thus far. "Simple" means the corresponding nonterminals are in the same order in ω and ω'. "Suffix"[††] implies the additional stipulation

---

[†] A similar definition in which "translation rules" were associated with the alternatives of Backus Naur Form definitions appeared in (Eva 65).

[††] We use "suffix" where "Polish" was used in (L&S 68) because it is more specific. Also, for those readers who like to reference "semantic routines" via output symbols in the middle of the right parts of productions, it is shown in (L&S 68) that for many simple transduction grammars based on LR(k) grammars there are "structurally equivalent" SSTGs which define the same translation and which are based on LR(k') grammars for some finite $k' \geq k$.

that the nonterminals in ω' must all be to the left of any output terminals.

An example SSTG G_{t1} based on our example grammar G_1 is as follows, where the transduction element for each production is in brackets to the right of the production.

(0) $S \rightarrow \vdash E \dashv$ $\{E\}$ (4) $T \rightarrow P$ $\{P\}$

(1) $E \rightarrow E + T$ $\{E\ T\ +\}$ (5) $P \rightarrow i$ $\{i\}$

(2) $E \rightarrow T$ $\{E\}$ (6) $P \rightarrow (E)$ $\{E\}$

(3) $T \rightarrow P \uparrow T$ $\{P\ T\ \uparrow\}$

The transduction elements may be thought of as defining an output grammar G', where production $A \rightarrow \omega'$ is in G' if and only if ω' is the transduction element for production $A \rightarrow \omega$ in G. Each derivation from S using G has a corresponding derivation using G' which is obtained by applying corresponding productions to corresponding nonterminals. Thus, for each derivation of a sentence η in L(G) there is a corresponding derivation leading to a string η' in $(V_T')^*$. The string η' is called a translation of η induced by G_t.

Our example SSTG G_{t1} above induces translations of strings in $L(G_1)$ which are commonly called "suffix forms" (Che 67). For example, the translation of $\eta_1 = \vdash i \uparrow i + i \dashv$ induced by G_{t1} is $\eta_i = ii \uparrow i +$.

6.6   Translators

We now show that the translations induced by an SSTG $G_t = (G, V_T', g)$ are in one-to-one correspondence to the parses of the sentences in $L(G)$.

Consider a canonical derivation $S = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_n = \eta$ of a sentence η using grammar G. If step i $(1 \leq i \leq n)$ is the application of production p_i, whose transaction element is $\omega_{p_i}' = \gamma_{p_i} \delta_{p_i}$ where γ_{p_i} is in V_N^* and δ_{p_i} is in $(V_T')^*$, then the translation of η induced by G_t is $\eta' = \delta_{p_n} \delta_{p_{n-1}} \cdots \delta_{p_1}$.

Thus, if we were given the reverse of the sequence of productions used in a canonical derivation of η, i.e., if we were given the canonical parse of η, we could generate its translation η' directly in a left-to-right manner by outputting first δ_{p_n}, then $\delta_{p_{n-1}}, \ldots$, then δ_{p_1}. That is, we can generate the translation η' of the sentence η simultaneously with the parsing of η.

A machine is called a <u>translator</u>[†] for a transduction grammar $G_t = (G, V_T', g)$ if and only if (1) it is a recognizer for $L(G)$ and (2) it maps each string in $L(G)$ into its translation induced by G_t. Clearly, our DPDA parser for G becomes a translator for an SSTG G_t based on G if

---

[†] This, of course, is our formal, automata-theoretic definition of a translator. Below we distinguish these from other translators (in the informal sense) by calling them "context-free syntactical translators (CFSTs)".

for each production p with transduction element $\omega' \to \gamma\delta$, where γ is in V_N^* and δ is in $(V_T')^*$, each "out p" is changed to "out δ".

Optimizations. The translator which results from this trivial transformation may have points where optimizations are applicable We consider first a "local" optimization and then a "global" one.

Consider the conversion of the parser of Figure 4.3 (page 85) to a translator for the SSTG G_{t1} above. The transitions from states 6 and 16 become under "pop 0, out ϵ", or equivalently, under "do nothing". Thus, the two states and the transitions are unnecessary, and they may be eliminated as follows. In the case of state 16 the look-ahead transition from state 7 may be redirected to go directly to state 15. In the case of state 6 the transition under "top 1" from state 15 may be redirected to go directly to state 17. But the latter results in a look-back transition to a state which is itself a look-back state. Clearly, if the "top 1" transition from 15 to 17 is taken, then the "top 1" transition from 17 to 2 will also be taken. Thus, we may redirect the "top 1" transition from 15 again, this time to go directly to state 2. The result of applying these changes to the DPDA of Figure 4.3 is depicted in Figure 6.1.

We do not give an exhaustive list of all possible types of "local" optimizations which may be applicable after a parser is changed to a translator. Suffice it to say that (1) all such optimizations arise when a
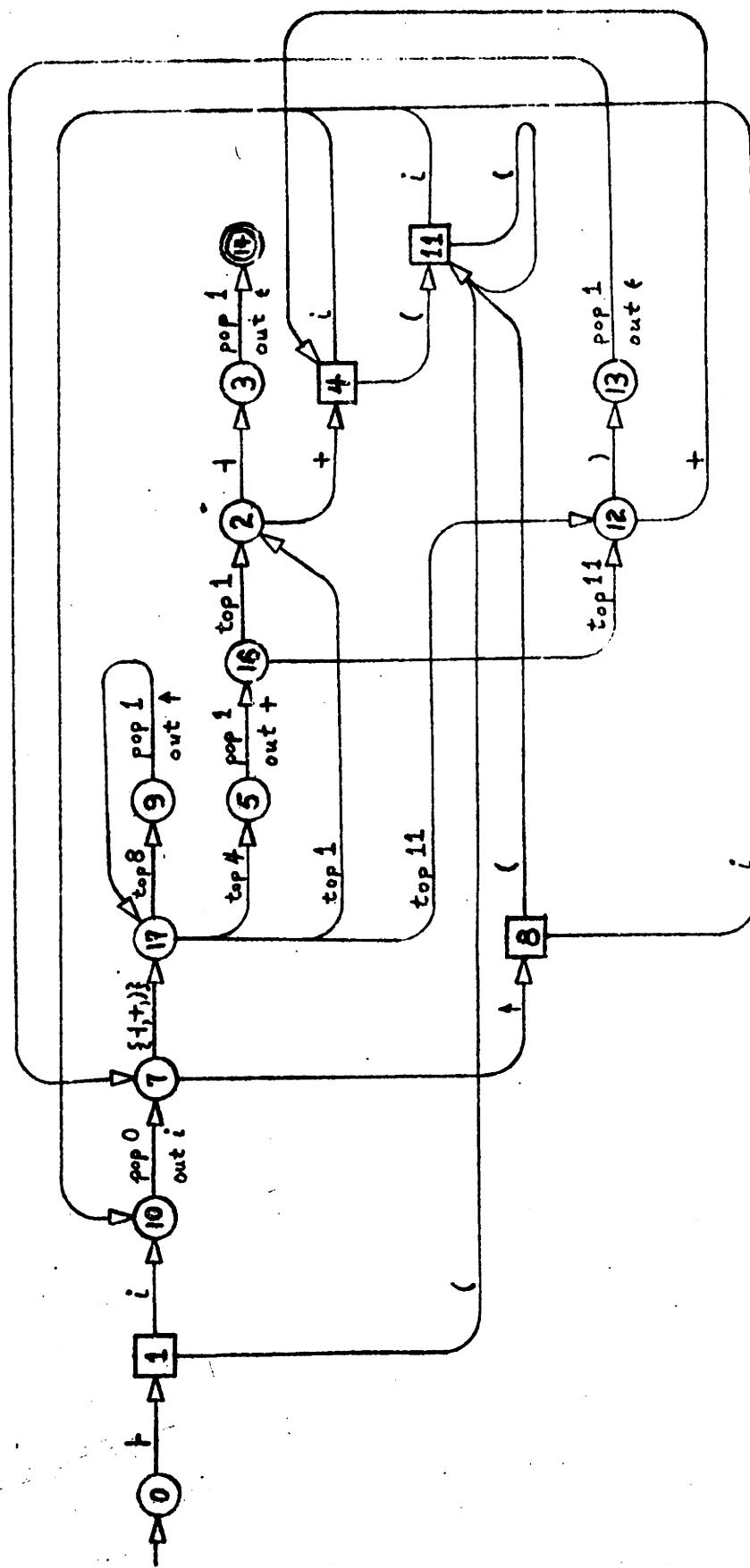
Figure 6.1. The optimized DPDA-translator (CFST) for the SSTG

G_{t1} based on the CF grammar G_1.

transition is found to be unnecessary due to its being under "pop 0, out ϵ", and (2) whenever a transition is redirected to a new state, an analysis of the device is in order to detect any redundancies, as in the case above, in its actions immediately after taking that transition.

Unfortunately, the efficiency of our DPDA as a translator is likely to be lower than it was as a parser, notwithstanding the above "local" optimizations. The problem is that our DPDA still goes through the motions of parsing but does not output anything along with many of its actions. This is not immediately obvious from our running example, but by analyzing it somewhat and generalizing we can illuminate the problem.

Consider the actions of the translator of Figure 6.1 associated with states 7 and 15. The decisions which are made there can be described in terms of operator precedences and associativities as follows.

Encoded in state 7 is the information that ↑ is a right associative operator and it has more binding power than any other operator of the subexpression which is implicitly stored in the stack when the machine is in state 7. Thus, when the machine is in state 7, if an ↑ is the next symbol in the input string, it should be read. The look-ahead set $\{ \dashv, +,) \}$ is just the set of other operators which may be the next symbol and which have less binding power than ↑. In case the next symbol is one of these, the device should not read but enter state 15, where it makes decisions regarding the past rather than the future. For instance, if it

has recently read ... i ↑ i, it makes a reduction and outputs ↑, again because ↑ is more binding than the operators in the look-ahead set. Similarly, if it has recently read ... i ⊢ i, it makes a reduction and outputs ⊢, because + is left associative and more binding than ⊣ or).

Now, if in our programming language there are many operators and many levels of precedence, it will happen that our translator, in translating a simple string like ⊢ i ⊣, will have to proceed through a cascade of pairs of states like 7 and 15. In effect, each pair of states will be associated with a precedence level. The first state will look ahead to check the precedence of the next operator to be read, and the second will look back to see if it should make a reduction and output. Of course, the decisions will be made relative to the precedence level associated with the pair.

The point of our generalization is that for a simple string like ⊢ i ⊣, many state transitions, look-aheads, and look-backs may have to be performed before reaching the accepting state, all for an output of the single symbol i. Of course, the problem can be equally bad with parenthetical expressions such as ... (i) .... and the inefficiency also creeps into a lesser extent with all subexpressions; e. g. , once a subexpression with one operator has been translated, the translator will have to proceed through the cascade from the level of precedence of that operator to the top.

To eliminate such inefficiencies we could, as in previous cases, precompute all possible results and "wire them into the machine". In this case this would mean modifying each look-ahead and look-back state so that the machine would, in effect, jump as far up any such cascade as it should given the next symbol(s) in the input string and the top state-name on the stack: i. e., given the relevant information about left and right context. Unfortunately, if we do this for a grammar of practical size and usefulness, the state diagram representation of our translator is likely to get disturbingly large. We suspect, however, that some clever coding tricks can be employed to implement these "jumps over cascades" in a reasonable amount of space. We do not pursue the subject here, since our objective is not to develop a "fine-tuned" implementation technique. Rather, we leave the problem as one for future development.

The reader should notice that in the case of our example grammar G_1 this "global" optimization amounts to noticing that the string $\vdash i \dashv$ can be reduced directly to $\vdash E \dashv$ without going through $\vdash P \dashv$ and $\vdash T \dashv$. However, he should also notice that this depends on the fact there are no output terminals in the transduction elements of the two productions $T \to P$ and $E \to T$. Since in general such productions could have output terminals, i. e., since transduction grammars give us that flexibility,

it is clear that we must wait until the translator is constructed, or at least until the SSTG is investigated, before attempting to make such an optimization.

6.7   A Compiler Model

Our compiler model is an incomplete one. Indeed, we detail only the "front end"; i. e. , the first three subtranslators and their interconnections and interactions. The model is similar to Cheatham's (Che 67), but much of our viewpoint and terminology are inspired by the approach of Landin (Lan 66) to programming language design. Landin's method goes something as follows.

A programming language is first designed on an abstract level. That is, the designer first decides what are to be the primitives of the language, what abstract objects are to be in the universe of discourse of the language, how things are to be defined in terms of other things, i. e. , what sort of definitional facilities are to be available, what sort of "structure of expressions" or "linguistic constructs" are to be available and how they are to be interconnected for the manipulation of abstract objects, etc. At this "abstract syntax" level programs in the language are represented by abstract syntax trees. Then the designer provides two functions: (a) one to define the mapping or "flattening" of abstract syntax trees into a convenient representation for use by programmers, i. e. , "source code", and (b) the other to define the flattening of the trees

into representations convenient for use by a computer, i.e., "object code".

Of course, we do not believe that any language has ever been designed in a single iteration of the above procedure, but the procedure seems to us a good model of the process which designers go through repeatedly before finally settling on a particular design. At the least, it provides a model of how the language might ideally have been designed and it suggests an intuitively reasonable method of formalizing programming language specifications (W&E 69).

In view of the above procedure, then, compiling can be regarded as first performing the reverse of mapping (a) above and then performing mapping (b). The two tasks correspond exactly to the "front end" and the "rear end" of our compiler model, respectively.

Landin subdivides the first of these mappings into two mappings, and we further subdivide one of them into two, so that the "front end" of our compiler consists of three subtranslators. We illustrate the corresponding mappings with the aid of Figure 6.2, in which are presented four representations of a program in a programming language based on grammar G_1. From the viewpoint of compiling, the mappings are as follows.

The first is from what Landin calls the "physical" level to what he calls the "logical" level. The "physical" level is the level at which

(a) ⊢ Abc ⨪ 123 + 4.5 ⊣

(b) ⊢ ↓ ⨪ ↓ + ↓ ⊣

       (name: Abc)

       (integer: 123)

       (real: 4.5)

(c) ↓ ↓ ⨪ ↓ +

       (name: Abc)

       (integer: 123)

       (real: 4.5)

(d)

TREE STORAGE TABLE (TST)

```
 0   identifier
 1      name
 2      reference to an entry in
 3   identifier               NAMELIST
 4      integer
 5      123
 6      ↑
 7      0
 8      3
 9   identifier
10      real
11      4.5
top node →  12  +
13      6
14      9
```

pointers to other nodes
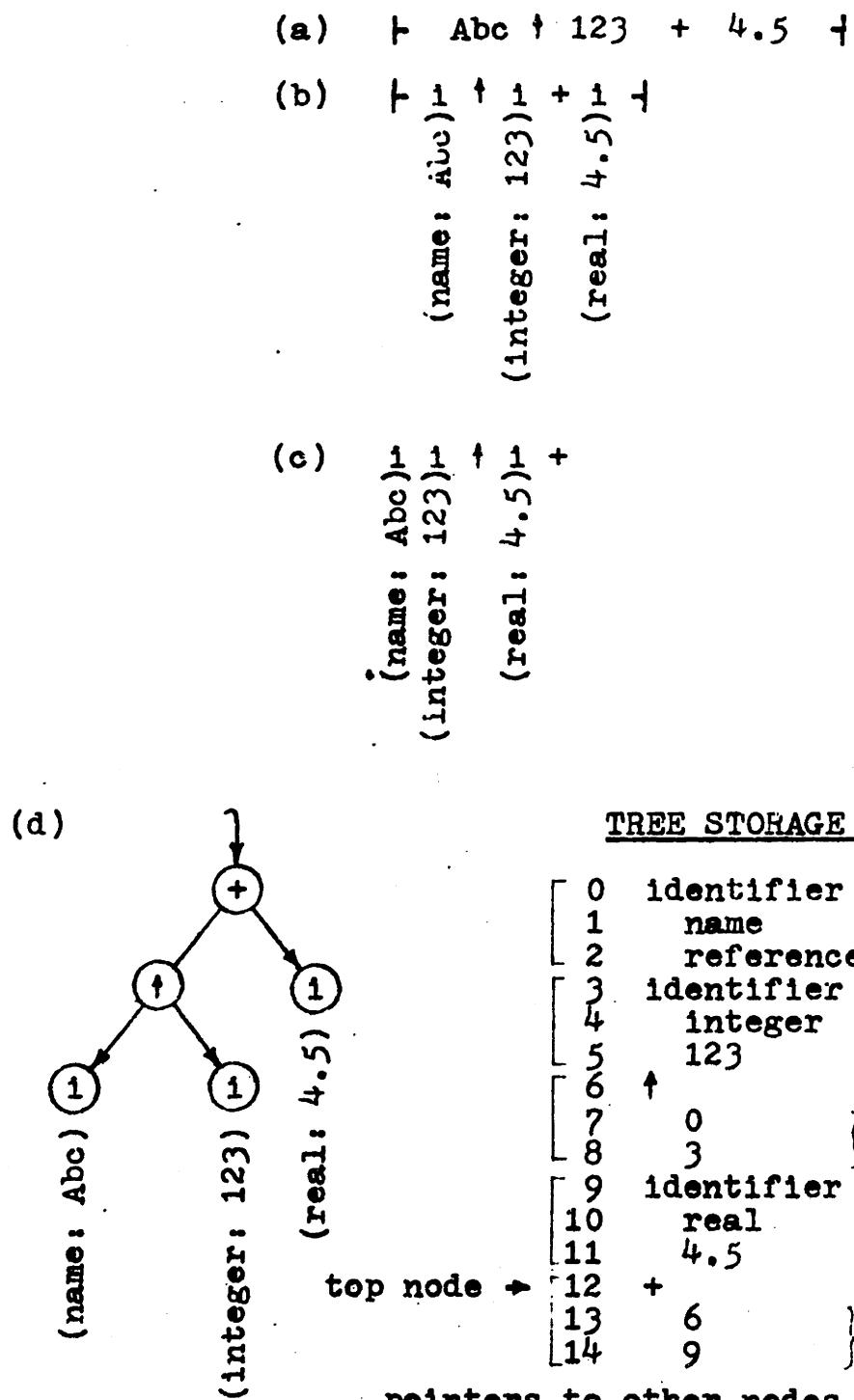
Figure 6.2. A program at four different levels: (a) the "physical" level, (b) the "logical" level, (c) the "tree-building directive" level, and (d) a graphical then a tabular representation at the "abstract syntax" level.

the programmer uses the language (Figure 6. 2a). The "logical" level is
the level at which certain strings of characters have been recognized
as"textual elements" denoting single entities. The strings might denote
constants, names, operators, key words, or the like (Figure 6. 2b). This
mapping is often called "lexical analysis". We call the corresponding
translator a <u>lexical</u> <u>translator</u>. It maps strings of characters, provided
by a programmer via some input device, say, into strings of <u>lexical</u>
<u>tokens.</u> The latter are the terminal symbols of a corresponding CF
grammar, some with certain "semantics" (values, types, etc.) associated
with them.

The second mapping is from the "logical" level to what we call
the "tree-building directive" level. This mapping is performed by our
translator of section 6.6, which we call here a <u>context-free syntactical</u>
<u>translator</u> (CFST) to distinguish it from other translators (in the informal
sense). The mapping results in a string of tree-building directives, some
of which have "semantics" associated with them as do some of the terminal
symbols (Figure 6. 2c).

The third mapping is from the "tree-building directive" level
to the "abstract syntax" level. It is performed by an <u>abstract-syntax</u>
<u>tree builder</u> (ASTB) and it results in an abstract syntax tree having
"semantics" associated with some of its nodes (Figure 6. 2d). (For present
purposes ignore the tabular representation of the tree; we discuss it below.)

Our compiler model, with emphasis on the "front end", is illustrated in Figure 6.3. Note that the "rear end" consists of the several subtranslators which are in the box labeled EVERYTHING ELSE and which affect the mapping from abstract syntax tree to "object code". The box labeled ERROR is intended to be a general error recovery device; it is called when any other device in the compiler discovers that the program being compiled is not in the given language.

The boxes labeled LEX and DICTIONARY and the two queues together form our lexical translator. LEX is basically an FSM which can be automatically constructed via the technique of Johnson, et al (Joh 68), also see (L&P 68) if the method of specifying the "lexicon" of the language is based on regular expressions. When LEX is activated it reads from the source code the next string of characters which represents a single entity, i.e., the next textual element, and it outputs one or two things: (1) to our CFST, via the "syntactic queue" Q1 which is necessary for look-ahead, it sends the terminal symbol t which is the "name" of the element just found, e.g., i for the identifier Abc or 123, (2) if the string must have some "semantic" information derived from it, LEX sends both the "name" t and the string of characters to DICTIONARY. The latter then derives the appropriate information from the string, stores the information in the TREE STORAGE TABLE (TST) as a terminal node, e.g., lines 0, 1, and 2 of the TST of Figure 6.2d, and sends a reference
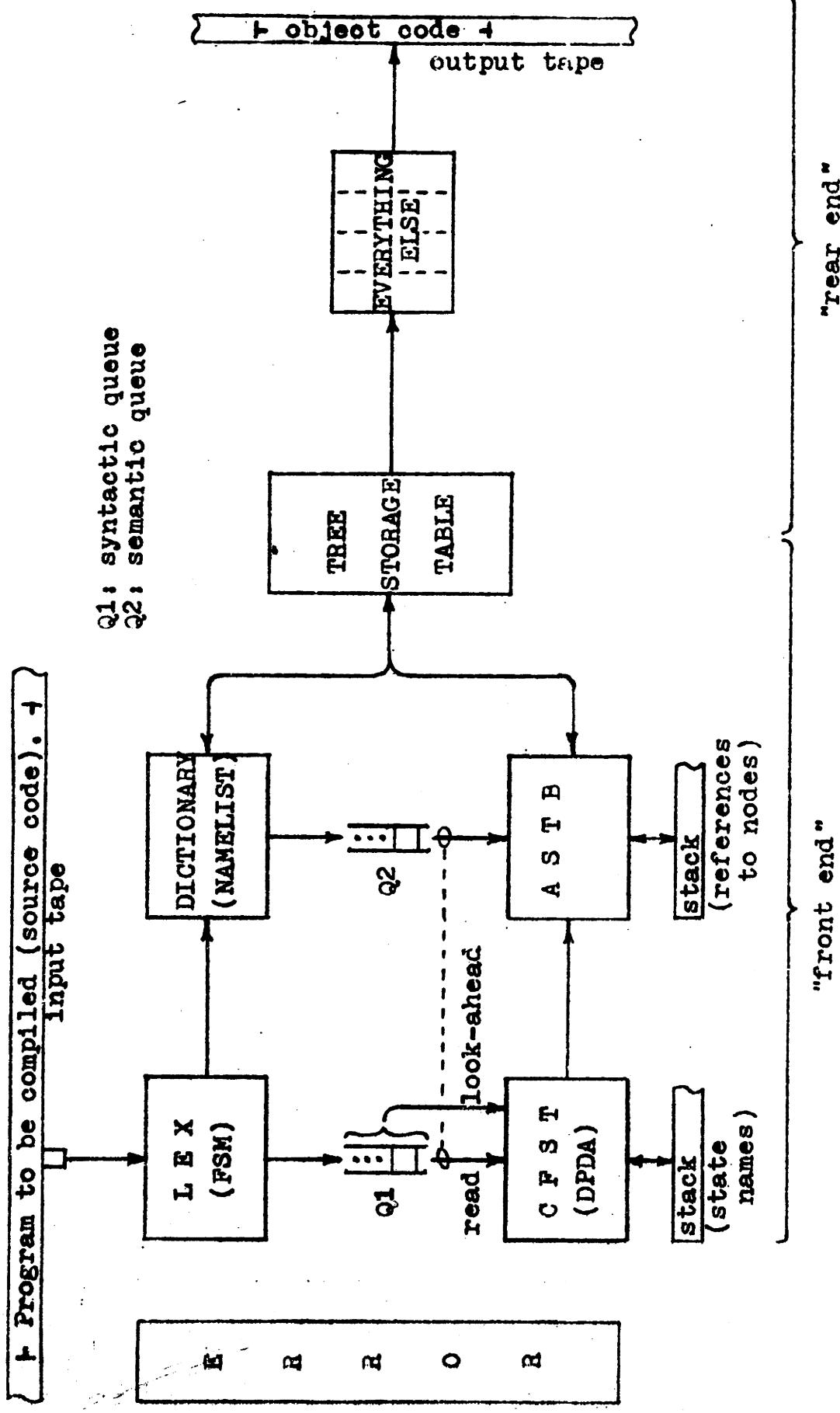
Figure 6.3. A compiler model with emphasis on the "front end".
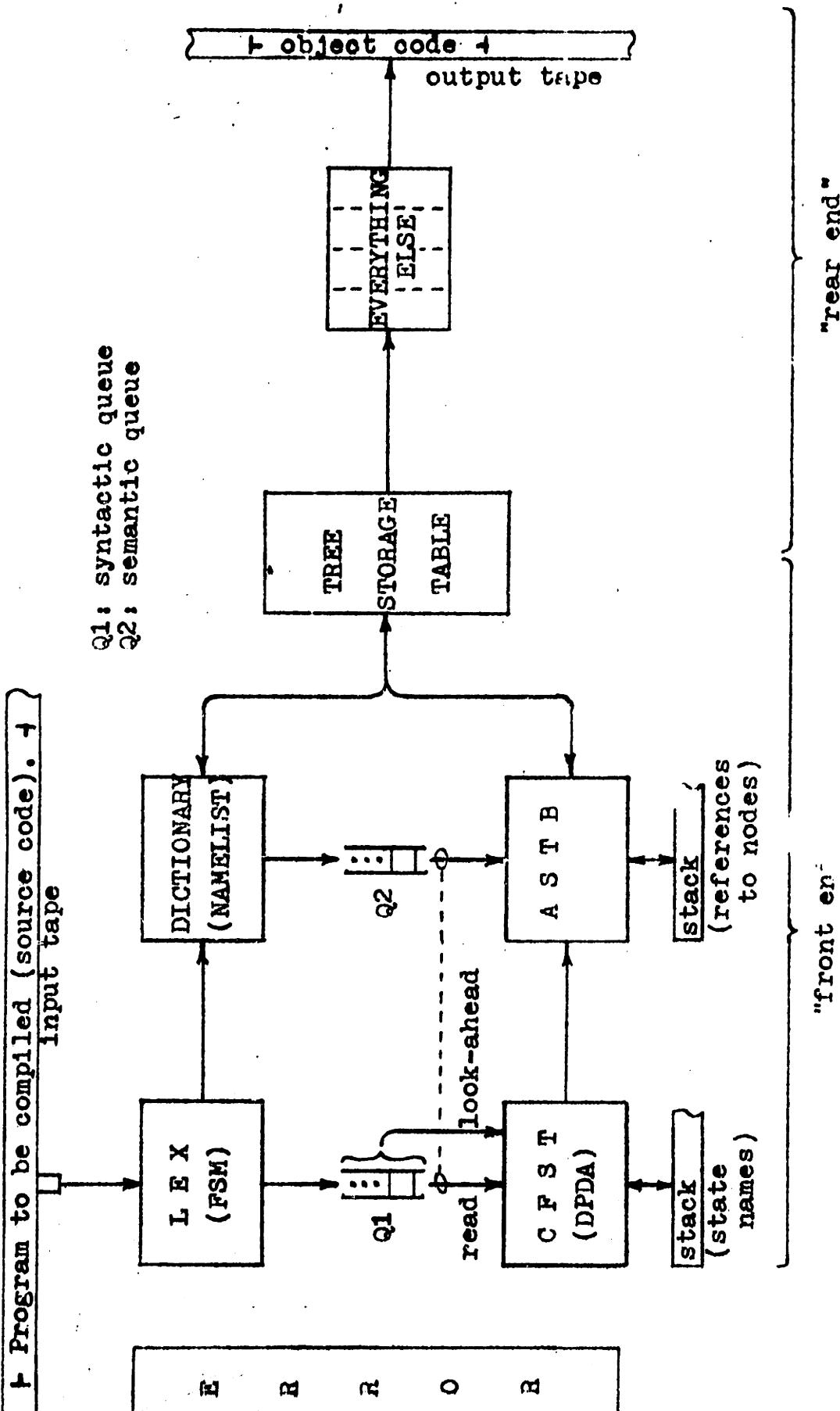
Figure 6.3. A compiler model with emphasis on the "front end".

to the node (TST line number) to the "semantic queue" Q2. Thus, it is actually DICTIONARY rather than the ASTB which constructs terminal nodes with associated "semantics".

Within DICTIONARY there is a NAMELIST in which names, e. g., Abc, are stored, and references to the appropriate entries in NAMELIST are stored in the TST rather than the names themselves. This is for the sake of fast name comparisons and other reasons regarding "attributes" of names which are irrelevant for our purposes.

Our CFST uses Q1 as its input tape, and LEX is activated to refill Q1 whenever it has insufficient symbols for a read or look-ahead by the CFST. That is, in effect, when the CFST desires to read or look ahead it makes the appropriate request of Q1. If Q1 has insufficient symbols to fill the request, it in turn requests the number it needs from LEX. As indicated in the figure, LEX deposits symbols into the top of Q1 and they are removed from the bottom via reads by the CFST. As noted in Section 2.3 we assume that the program which loads the source code onto the input tape assures that the last symbol is a $\dashv$, so that the compiler will not read past the end of the source code, and therefore, will stop after some finite time.

The dashed line in Figure 6.3 indicates that the two queues are "ganged", in an important sense. We have already seen that, when

LEX processes a textual element with "semantics", both Q1 and Q2 receive a new item. Likewise, as we shall see in the next paragraph, these pairs are removed from the queues simultaneously also. Thus, although at a given time there may not be as many references in Q2 as there are symbols in Q1 (because some symbols have no "semantics"), the order of the references in Q2 is the same as the order of their corresponding symbols in Q1. This correspondence is seen to be important below.

Let us refer to terminal symbols with associated "semantics" as "pseudo-terminals". We require that pseudo-terminals be distinguishable from terminals without "semantics", a not unreasonable restriction for our purposes. Whenever a pseudo-terminal is read from the bottom of Q1, the latter sends a signal to the ASTB which causes it to remove the bottom reference from Q2 and to push that reference on its stack, the "node-reference stack". It is this stack which the ASTB uses to hold references to the top nodes of pieces of a partially constructed abstract syntax tree. Thus, immediately after the CFST reads a pseudo-terminal, the top reference on the ASTB's stack is to a terminal node which corresponds to that pseudo-terminal.

Summary. In summary, the lexical translator (LEX plus DICTIONARY, Q1, and Q2) reads the "source code" and translates it into a string of symbols, some of which have associated "semantics".

Each time the CFST reads a pseudo-terminal a reference to a corresponding terminal node with "semantics" is pushed on the ASTB's stack. Each time the CFST outputs a symbol (i.e., a directive to the ASTB to build a nonterminal node), the ASTB pops the appropriate number of node-references off its stack, builds the appropriate nonterminal node whose sons are the nodes whose references were just popped, and pushed a reference to the new node on its stack.

In a sense, then, the language designer's problem is, in part (1) to design a transduction grammar such that the corresponding CFST issues the appropriate directives at the appropriate times, and (2) to specify an ASTB which constructs the appropriate trees, given that as the CFST reads pseudo-terminals the ASTB will be directed to build corresponding terminal nodes with "semantics". Of course, stated that way the design problem sounds like a fairly "low level" task. Our next order of business, then, is to transliterate this task of specifying CFSTs and ASTBs into one which can be performed at a "high level". This requires that we return to our approach to language design and work from there down to the level of tree-building directives.

6.8 Specifying Languages, Translations, Compilers

We have chosen to employ CF grammars as aids to that part of language specifications which we describe, after Landin, as specifying the mapping of abstract syntax trees into strings of lexical tokens. In more common parlance: we use a CF grammar both to define a set of
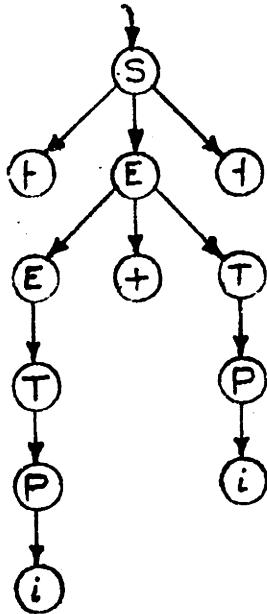
potentially legal programs, some of which may be screened out by context sensitive checks, and to define certain operator precedences, associativities, etc., by building into the grammar certain "structural properties". Unfortunately, due to the nature of CF grammars we usually get too much "structure" (at least from the viewpoint explained below). We therefore propose the use of something very like an SSTG for specifying only the amount of "structure" we desire. We elaborate on this subject by first considering just what "structural" information is implicit in a parse.

Consider a variation of our compiler model. Let us assume for the moment that _every_ textual element is sent to the DICTIONARY to have a corresponding terminal node built from it. If the element has no "semantics", then the node will just be a simple terminal node with no "semantics" and with the same name as that of the element. Further, let us assume that _every_ read by the CFST causes a corresponding terminal node reference to be pushed on the ASTB's stack. Finally, let us assume that the CFST is replaced by the parser for the grammar at hand, and that the ASTB is simply a collection of subroutines associated with the productions such that when the parser outputs production p, $A \rightarrow \omega$, a corresponding subroutine is activated which pops $|\omega|$ references from the node reference stack, builds a nonterminal node named A with $|\omega|$ sons which are the nodes corresponding to the references just

popped, the first popped being the $|\omega|$-th son, and then pushes a reference

to the new node on the node reference stack.

If this device is applied to some legal program, then after the

parser has made the final reduction, namely to S, there will be a single

reference on the ASTB's stack and it will be to the top node of what is

commonly called a "parse tree". The parse tree contains the same

information as the parse but the "structural properties" are explicit rather

than implicit. As an example the parse tree corresponding to our string

$\vdash i + i \dashv$ generated by G_1 is as follows.

Now, if our language designer has been careful to design into his

CF grammar all the "structural properties" he desires, then the abstract

syntax trees can be derived from the parse trees by removing any spurious

structure which may have crept in, and perhaps also "recoding" the informa-

tion slightly, e. g., by renaming nodes. This follows by definition of what

we mean by the above phrase "design into ... desires." That is, we view

this design problem as one of constructing a grammar which generates strings having parse trees from which the desired abstract syntax trees can be easily derived as just described.

Thus we must provide the language designer with a way to specify what information to keep, what to discard, and how to "recode" any of the information in the parse trees. One way he could do this would be on a node-by-node basis with respect to parse trees, and therefore, on a production-by-production basis with respect to his CF grammar. In effect, he could specify replacements for the subroutines which comprise the ASTB so nodes would be constructed differently. For instance, for a production like $E \rightarrow T$ he might replace the corresponding subroutine with one which does nothing, so that a node named E with only one son named T would never appear in the resulting tree. Similarly, he might change the subroutine for $E \rightarrow E + T$ to one which creates a node named $+$ with two sons.

We place only two restrictions on the designer with respect to his new subroutines. The first is really just a matter of the efficiency of our compiler. It is inefficient for us to build terminal nodes for textual elements with no "semantics" and to carry references to them on the node reference stack, because the designer may have no need for them in his tree, and even if he does, he can easily build them himself. Thus, he should be aware that only references to nodes corresponding to
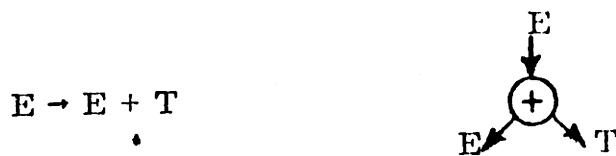
pseudo-terminals and nonterminals in the right part of a given production will be at the top of the stack when the corresponding subroutine is called. The second restriction is more severe than is necessary, but it is simple and it still allows adequate power for the purpose at hand. To be sure that references in the ASTB's stack are always kept in the appropriate correspondence with pseudo-terminals and nonterminals in the right parts of productions, we require that any new subroutine have the same effect relative to the node reference stack as does the one it replaces; i. e., if the original subroutine, or really the original modified to abide by the first restriction, pops n references and pushes one, then the new sub-routine must pop n references and push one[†], unless n = 1, in which case it may do nothing. Again we have a not unreasonable restriction, given the application.

A proposal. Now, we hope the reader has not taken the above discussion too literally. It was intended to illuminate the specification problem associated with our CFST and ASTB. We do not, however, propose that the designer should actually think of himself as modifying our compiler, or necessarily, writing any subroutines, per se. Having gone through this discussion though, it should be easy to see that the following proposal

_____

[†] We might have allowed simply pop n - 1, but pop n - 1 implies that some information is being discarded. We assume that, if n > 1 references are popped, then a reference to a new node will be pushed such that the new node has at least the n corresponding nodes as sons.

will serve as the desired "high level" specification of CFSTs and

ASTBs.

We propose that the language designer specify a correspondence

between strings generated by his CF grammar and abstract syntax

trees merely by associating tree nodes with his productions. For

example, for the + operator we might have

$$E \rightarrow E + T$$

and for a production with no corresponding node we might have

$$E \rightarrow T$$

Our second restriction above merely implies that for each instance of

a nonterminal or pseudo-terminal in the production there must be a

corresponding instance in the corresponding node. Thus, we have a method

of specification rather similar to a transduction grammar. In fact, if we

settle on some conventions about diagrams like the above, i.e., if we

develop a graphical language[†] for this purpose, a set of node building

_____

† To specify the language BASEL Jorrand (Jor 69) uses the AMBIT/G
graphical language (Chr 67) to specify the "augments" to productions. His
approach is an adaptation of Cheatham's and is similar to but more extensive
than our proposal.

subroutines and a corresponding SSTG can be derived from a set of such "production-node pairs", such that the corresponding CFST and ASTB are the appropriate ones for a corresponding compiler. For instance, corresponding to the above two examples would be the following components of an SSTG:

$$E \rightarrow E + T \quad \{E \ T \ + \}$$
$$E \rightarrow T \quad \{T\}$$

and a subroutine called PLUS, say, which would be activated when the CFST outputs + to the ASTB. PLUS would pop two references off the node-reference stack, use them to build a node named + with two sons, and push a reference to that node back on the stack. Of course, we have, in effect, made an optimization with regard to the second production: rather than have our CFST output a call to a nugatory subroutine, we have it not output anything when the reduction $T \rightarrow E$ is applied.

TWSs. Ideally, then, the portion of our TWS which builds the "front ends" of compilers would consist primarily of (1) a device which translates a specification based on regular expressions into a LEX and a DICTIONARY, (2) a compiler which translates a set of production-node pairs into a set of node-building subroutines, i. e., the ASTB, and an SSTG, and (3) a manifestation of our procedure (summarized in Chapter 7) for constructing a CFST from an SSTG.

Of course the latter component is useful only if language designers find it possible, natural, and convenient, to specify a significant portion of the translations of their languages via techniques similar to those we have proposed. More specifically, the value of our results depends on designers being able, once they have a set of abstract syntax trees in mind, to construct an LR(k) grammar which implies parse trees from which the abstract syntax trees can be easily derived. Of course, it would be even better if the grammar were SLR(1).

Unfortunately, we know of no significant formal results in this area. Currently, designers seem to build operator precedences, etc., into grammars purely on the basis of past experience and trial-and-error methods. We have pursued the research, then, only because of empirical evidence that some related results may be forthcoming. We hope because so many authors (F&G 68) have found LR(k) grammars useful in this way that there are some underlying principles which will some day come to the fore.

Conclusion. We conclude by further illustrating the similarity of our model to those of other authors. To do so we consider the absorption by the ASTB of some of the tasks conceptually performed by the "rear end" of our model.

As we have already seen, the ASTB can be regarded, even implemented as a collection of subroutines. Consider for example our subroutine PLUS

of above. It could be a much more sophisticated routine than we have

indicated thus far. For instance, it might check the two sons of the

node it would build to determine if they are both constants, or of one is

zero, and if so, perform the addition, i. e. , prune the tree; it might

reorder the sons in some way so that ultimately more efficient "object

code" would be generated; it might do "type-checking" ;... ; it might

even be able to perform the entire function of the "rear end" with

respect to the node in question and actually output object code.

It should be clear, then, how similar our approach is, basically,

to the approaches of Cheatham and Feldman

Chapter 7

IMPLEMENTATION ISSUES

We seek in this chapter to illustrate the practicability of our scheme. To do so we choose a particular method of implementing our translators and present the results when the method is applied to a particular, practical transduction grammar. Our implementation should be regarded only as a first approximation to an optimal one. We have not labored at getting an optimal solution, but only at getting one which would illustrate the potential of our methods. Undoubtedly, some empirical results would be invaluable aids in "tuning up" our implementation.

Before presenting our practical example we discuss further the construction of CFSMs and then we summarize our translator constructing technique as a whole.
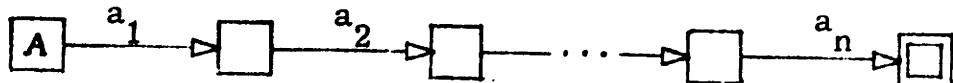
7.1 Constructing CFSMs

The CFSM of a CF grammar G can be constructed from the productions of G in a manner similar to the well known technique for constructing an FSM from the productions of a right linear grammar. (See for example (D&D 69) for a thorough discussion of the latter technique.) We review the technique here because our technique is derived from it.

The productions of a right linear grammar G_R are either of the form $A \rightarrow a_1 a_2 \ldots a_n$ or $A \rightarrow a_1 a_2 \ldots a_m B$ where n and m are ≥ 0, the a_i

are terminals, and A and B are nonterminals. We construct an FSM which recognizes the strings generated by G_R by forming a small piece of the machine for each production and then putting the pieces together. For the production $A \to a_1 a_2 \dots a_n$ the corresponding piece is

that is, a path which spell out $a_1 a_2 \dots a_n$ and leads from a state named A, the left part of the production, to the terminal state. For a production of the form $A \to a_1 a_2 \dots a_m B$ the corresponding piece is
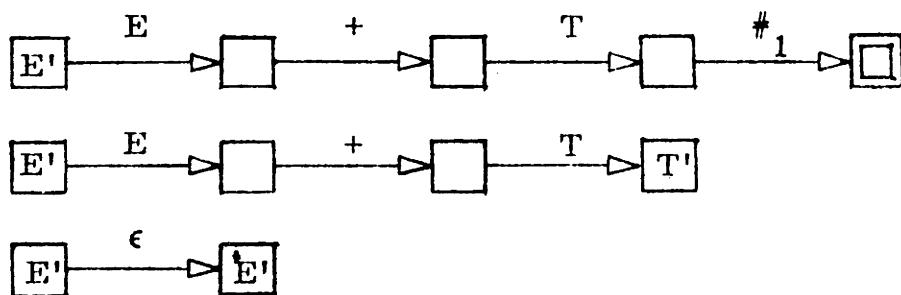
that is, a path which spells out the string of terminals in the right part and leads from a state named A to a state named B. If we simply put all the pieces together by identifying all states with the same name as the same state, we get the desired FSM, although it may be nondeterministic.
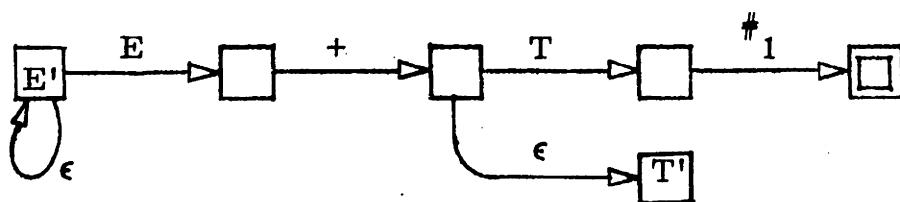
Now, to build our CFSM we could just apply the above procedure to G's characteristic grammar. However, since that grammar is so closely related to G, we can transliterate the procedure to one which will

work directly on G. We illustate the procedure using our example G_1.

Consider the production (1) E → E + T. There are three corresponding

productions in G_1's characteristic grammar, E' → E + T $\#_1$, E' → E + T'.

and E' → E'. The corresponding FSM pieces are as follows.

In the latter case we visualize the production written E' → ϵ E' so it fits

the second rule above. If we now combine all the pieces corresponding

to the single production of G_1, just as we would do if they were all of

the pieces, and change the result to a deterministic (piece of) FSM, we

get the following.

It is easy to see that, in general, the piece corresponding to a production

(p) A → ω consists of a path which spells out $\omega \#_p$ and leads from a state

named A to the terminal state, such that from each state in the path having a transition under a nonterminal B there is also an ϵ-transition to a state named B'. If all the pieces corresponding to the productions of G are put together by identifying all states with the same name as the same state, an FSM with ϵ-transitions results which recognizes the set of characteristic strings. The ϵ-transitions can be removed by well-known techniques (again see (D&D 69)) and the machine can be made deterministic and reduced. The result is the desired CFSM.

7.2   An Efficient Translator Constructing Procedure

We now review our procedure for the construction of a translator from an SSTG G_t based on a CF grammar G. The review is rather terse, being presented as an imperative "English program" with simple, forward jumps. Our purpose is to summarize the procedure as a whole and to point out the general order in which things might be done in a TWS. The order suggested here is largely a result of our experience with our single example presented below and should therefore be to some extent "taken with a grain of salt". Also, since the most useful TWS is undoubtedly an interactive one, some of the decisions built-in below should probably be made variable. Certainly, more empirical results are necessary for the development of an optimum strategy.

The translator constructing procedure is as follows. Note that we have referenced pertinent definitions, theorems, sections, and page numbers.

START:

Generate G's CFSM (Section 7. 1). In the process do the following for future purposes: (1) for each non-terminal A record in a "nonterminal-transition table" all pairs of states such that there is an A-transition from the first to the second, (2) note whether there are any inadequate states and if so which, and (3) associate with each production p a "set of p-states", those states which have $\#_p$-transitions.

LR(0):

If the CFSM has no inadequate states then G is LR(0) (Theorem 3. 4), so go to COMPUTE LOOK-BACK (below).

SLR(1):

For each inadequate state N compute the simple 1-look-ahead sets (Definition 4. 2) for the transitions from N. If these sets are mutually disjoint for each such state, then G is SLR(1) (Definition 4. 3) so convert the CFSM to the SLR1FSM (Definition 4. 4) and go to COMPUTE LOOK-BACK.

SLR(k):

For each inadequate state N with overlapping simple 1-look-ahead sets compute the simple k-look-ahead sets (Definition 4. 2) for the transitions from N for the

largest value of k for which we are willing to implement a translator with k-symbol look-ahead. (This value of k is probably dependent upon the number of such states, the implementation, and perhaps the language designer, if the TWS is interactive. Empirical results are needed here.)

If these sets are mutually disjoint, G is SLR(k) (Definition 4.3) so minimize look-ahead (Section 4.4), convert the CFSM to the SLRkFSM (Definition 4.4) and go to COMPUTE LOOK-BACK.

¬SLR(k):   Report to the language designer that his grammar is not SLR(k) for an acceptable k. Provide him with some information regarding what kinds of strings need more than k-symbol look-ahead and/or state-splitting to determine their characteristic strings. (Empirical results are needed regarding what information is useful to the designer.) Then, if the designer so desires, continue with the more complex techniques which follow.

LmRk: For each inadequate state N which has overlapping simple k-look-ahead sets, choose the above value of k and a similarly maximal value of m and compute the sets of (m, k)-bounded-context pairs (Definition 5.3) for the transitions from N.

If these sets are mutually disjoint, G is L(m)R(k)
(Definition 5.4) so convert the CFSM to the LmRkFSM
(Definition 5.5) with minimum-look-ahead (section 4.4),
change the "nonterminal-transition table" and the "sets
of p-states" (see START above) appropriately so that
they reflect the new states and transitions, and go to
COMPUTE LOOK-BACK.

LR(k): For each inadequate state N with overlapping context
pairs and for k as above, compute the strings φ (page 123)
which access N via paths with no more than k instances
of a given cycle, then compute the look-ahead sets
corresponding to each such φ (page 124) and each
transition from N.

If these look-ahead sets are mutually disjoint for each
such N, G is LR(k) (page 128) so convert the CFSM to
the LRkFSM (page 125) with minimal look-ahead
(section 4.4), change the "nonterminal-transition table"
and the "sets of p-states" appropriately, and go to
COMPUTE LOOK-BACK.

¬LR(k): Otherwise, G is not LR(k) for an acceptable k so reject
G and provide the language designer with some information
regarding what kinds of strings need more than k symbols of
look-ahead to determine their characteristic strings.

COMPUTE LOOK-BACK:	Associate with each #-transition in the FSM a "look-back set" of state pairs (the set Q on page 54), for the computation of look-back transitions below. For a $\#_p$-transition from state R, where production p is $A \rightarrow \omega$, the set is as follows. If there is but one $\#_p$-transition in the machine, the set is the set of pairs associated with A in the "nonterminal transition table". Otherwise, the set is the subset Q of A's set such that for each pair (N, M) in Q there is a path from N to R which spells out ω.
XLATOR:	If production p has transduction element ω' such that $\omega' = \gamma\delta$ and $\gamma = V^*$ and $\delta = (V_T^!)*$, replace the $\#_p$-transition with one under "pop $|\omega|$, output δ" (page 142) to a new state R'.
ADD LOOK-BACK:	R' (page 54) has a transition under "top N" to state M for each pair (N, M) in the "look-back set" associated above with the $\#_p$-transition. Eliminate equivalent look-back states (page 60).
ADD LOOK-AHEAD:	Convert each inadequate state (if any) to a look-ahead state (Figure 4. 2).
OPT:	Optimize the DPDA by (a) deleting transitions under nonterminals (page 56) and "pop 0, out ϵ" (page 142)

(b) eliminating redundancies via precomputation

(page 144), by minimizing look-back, pushing,

and popping, (page 61), and (c) precomputing jumps

over cascades of look-ahead and look-back states

(page 146).

END: All done.

We emphasize that we expect most of the grammars of interest

to be SLR(1), the remainder to be SLR(k) for k = 2 or 3 (caused by only

one or two inadequate states, at that), and none to require the more

complex L(m)R(k) or general LR(k) techniques. Thus, the poor state of

our strategy regarding those complex techniques is not likely to be a problem,

at least with respect to programming languages. However, if our TWS is

to be employed in some other application where more complex grammars are

to be expected, that strategy will require development. Otherwise, a con-

siderable amount of computation time is likely to be expended in deciding

whether a grammar is, indeed, LR(k) for an acceptable k.

7.3   Tabular Translators, an Interpreter

In this section we present a method of representing our translators

by means of tables, and we present via a flowchart an interpreter for those

tables. We first illustrate our storage method by using our trivial SSTG G_{t1}:

then we present the interpreter. (However, the reader may find it helpful

to reference the interpreter (Figure 7.2 below) as he follows the description of the storage method.) This implementation works only for LR(1) grammars whose CFSMs have no multiply inadequate states. Nonetheless, it covers our practical example which is presented in Section 7.4. We discuss in Section 7.6 the modifications necessary to cover the general case.

Shown in Figure 7.1 is a tabular representation of the translator of Figure 6.1 (page 143). Note that we have stored the information regarding states, transitions, and look-ahead sets in a STATE TABLE (ST), a TRANSITION TABLE (TT), and a LOOK-AHEAD TABLE (LAT), respectively. Each entry in the ST corresponds to a state and it has three components. The first, TYPE, indicates the type of state and it can have one of the seven values: READ, LA (look-ahead), POP (pop and output), LB (look-back), EXIT (the terminal state), ↓READ, and ↓LA, the last two of which indicate states which push (↓) their names (ST line numbers) on the stack. This covers all types of states which can appear in our translators. In the case of a POP, state, the second component, NUM, is the number of state names to pop from the stack. However, in all other cases NUM is the number of transitions from the state. The transitions are represented by contiguous entries in the TT and the third component, TTREF, is a reference to the topmost of these entries; i.e., it is a TT line number.

Each entry in the TT consists of two components, SYM and STATE. In the case of the entries for a READ or ↓READ state and all but the last

STATE	TABLE	(ST)
TYPE	NUM	TTREF
0 READ	1	0
1 ↓READ	2	1
2 READ	2	3
3 POP	1	6
4 ↓READ	2	1
5 POP	1	7
6 ---	-	-
7 LA	2	8
8 ↓READ	2	1
9 POP	1	10
10 POP	0	11
11 ↓READ	2	1
12 READ	2	4
13 POP	1	12
14 EXIT	-	-
15 LB	4	13
16 ---	-	-
17 LB	2	13

TRANSITION	TABLE (TT)
SYM	STATE
0 ⊢	1
1 i	10
2 (	11
3 ⊣	3
4 +	4
5)	13
6 ∈	14
7 +	17
8 ↑	8
9 1	15
10 ↑	15
11 1	7
12 ∈	7
13 1	2
14 11	12
15 8	9
16 4	5

LOOK-AHEAD TABLE (LAT)

	⊢	⊣	+	↑	1	(	)
1		1	1		1		
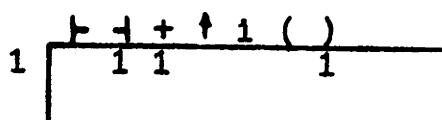
Figure 7.1. The DPDA-translator for the example SSTG G_{t1} represented by tables; i.e., a tabular version of Figure 6.1.

In such cases our interpreter will actually output something, namely
ε, therefore our ASTB will have to have a nugatory subroutine which
will be called when this happens. Our practical translator below
has so few such POP states that we thought it not worth the cost
of eliminating the inefficiency.

(3) From line seven of the ST we see that state 7 is a look-ahead state
with two transitions. One is actually a read transition under ↑
to state 8 and the other is a look-ahead transition to state 15. The
SYM component of line nine of the TT indicates that the look-ahead
set { ⊣, +,)} is implied by line one of the LAT. Note that the LAT
is included purely for the sake of earlier error detection since, if
only strings in $L(G_1)$ were being translated, we could be sure upon
arriving in state 7 that the next symbol would be ↑, ⊣, +, or).
However, since our string may not be in $L(G_1)$, we take the
attitude that once a symbol has affected any decision it must be
validated.

After two more comments we present the interpreter. First, the
"holes" in the ST, lines 6 and 16, could obviously have been filled by
renumbering the states; however, we choose not to so that the one-to-one
correspondence with Figure 6.1 would be preserved. Second, it should be
noted that some of the lines of the TT are referenced by more than one
state. For example lines one and two are referenced by states 1, 4, 8,

and 11. This is an important optimization of the use of space in the TT which we use extensively in our practical example. We have computed the optimization by hand here; however, there exists a graph-theoretic method for doing it automatically (I&M 69).

The interpreter. Since the reader presumably already knows what the actions of our DPDA are supposed to be and what the meanings of the tables are, we will not elaborate extensively on the operation of the interpreter. However several comments are in order. (1) The interpreter is presented via a flowchart in Figure 7.2 and it is described as if it were part of our compiler model of Chapter 6. (2) The variable, Stack, denotes a large vector which we use as our pushdown stack. The variable, S, is used as the stack index. The top name on the stack is always Stack (S-1). (3) We do not have to initialize any input string or pointer to one, since that initialization is affected when the lexical translator is initialized, before the interpreter is activated. Input and look-ahead symbols are acquired from the syntactic queue Q1 as described in Chapter 6. When Q1 is called with argument LA, the symbol in the queue is returned as the value, but the symbol is not removed from the queue. When Q1 is called with argument READ it both returns the symbol as its value and removes the symbol from the queue. (4) The variables, READ, LA, POP, LB, EXIT, ↓READ, and ↓LA, may be thought of as denoting some distinct constant values. (5) The variables, ST, TT, and LAT, denote two dimensional
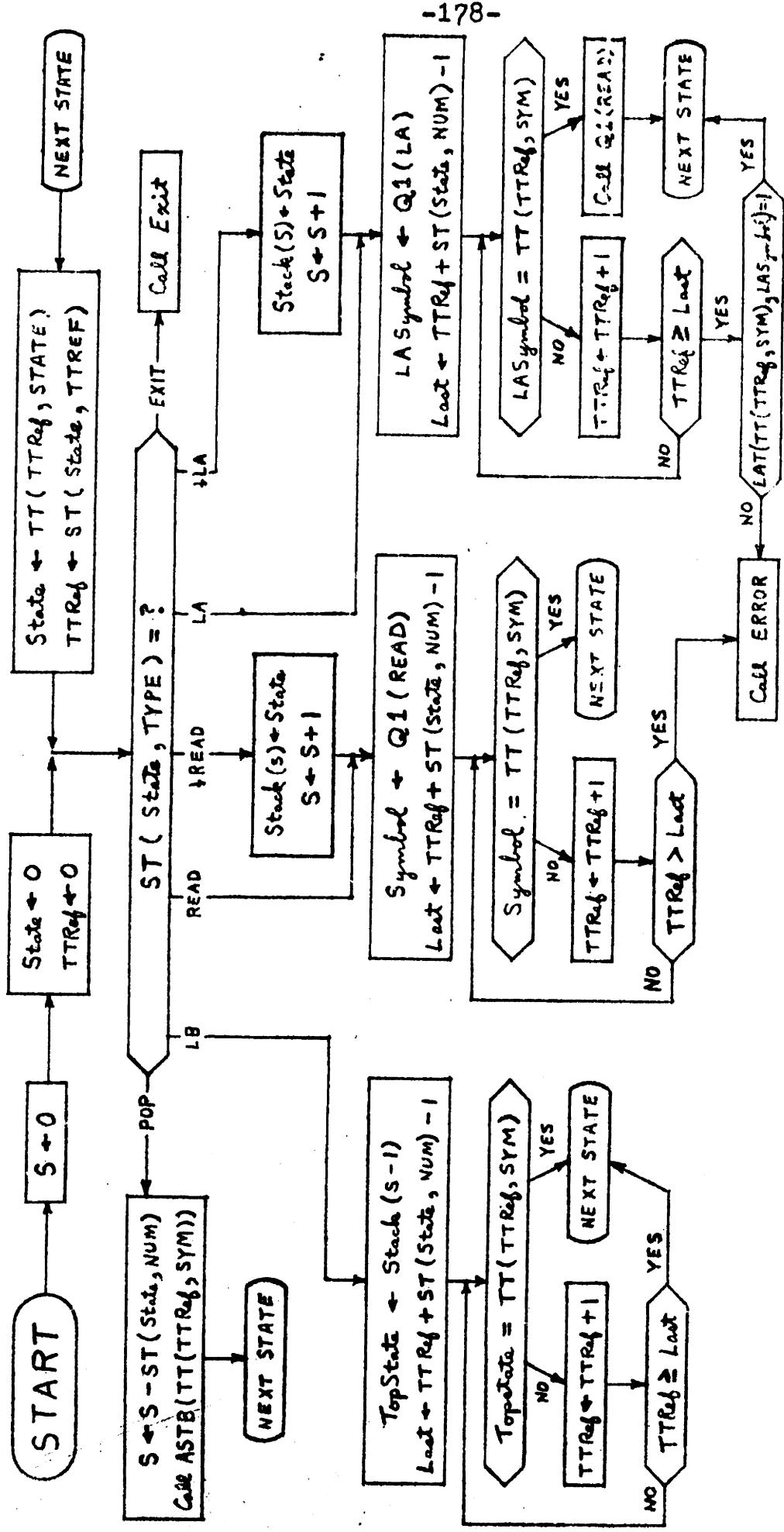
Figure 7.2. The interpreter for our tabular translators.

arrays which represent tables such as the ones in Figure 7.1. The variable, State, denotes the current state, which is represented by an ST line number. The current reference is kept in TTRef. We can view TYPE = 1, NUM = 2, TTREF = 3, SYM = 1, and STATE = 2, so that, e.g., if State = 10, ST(State, NUM) has the value stored in the tenth row, second column of the ST. (6) ASTB is, of course, the abstract-syntax-tree builder of Chapter 6.

7.4   A Practical Example

The programming language PAL (Pedagogic Algorithmic Language) (Eva 68, Eva 69, W&E 69) is used as a vehicle to teach some of the fundamentals of programming linguistics to undergraduates interested in computer science at the Massachusetts Institute of Technology. It is one of the more progressive languages in existence today, being a decendent of ISWIM (Lan 66). In a sense PAL is a generalization of ALGOL 60 (Nau 63); it has the general functional capabilities of LISP (McC 65), generalized structures, and generalized jumps.

PAL's Grammar. Of course most of this is irrelevant for our purposes here. It is the syntax of PAL in which we are interested. Since the formal definition of PAL specifies the set of legal programs as a CF language, we do not have to remove any "context-sensitive features" from the syntax. The syntax is similar to that of ALGOL 60, but it is considerably

"cleaner" and it is unambiguous. It is specified via modified Backus Naur Form (BNF) which, for our purposes, is just a shorthand way of writing CF productions.

As we noted above, the PAL grammar was designed, for the sake of pedagogy, to be unambiguous, small, concise, and useful as a syntactical reference. Except for the fact it was designed to be unambiguous, it can truly be said that the grammar was not designed to be within the domain of our parser constructing technique. And yet, the grammar turns out to be SLR(1).

A slightly modified version of the PAL grammar is presented in Table 7-1 where nonterminals are denoted by one or two capital letters, pseudo-terminals by three or more capitals, and other terminals by strings of small letters and/or special characters. The grammar differs from real PAL in several respects, which, for our purposes, are minor: (1) it includes new constructs which the author has proposed be added to PAL, (2) the original uses "regular expressions" in some alternatives to indicate nonassociative operators, e.g., DA ::= DR $\{$ and DR$\}_0^\infty$, and we have changed these in an obvious way to get a strict CF grammar which generates the same strings, (3) the original grammar has the definitions of CONST and RLN built-in, whereas we have moved them into the lexical domain, and (4) the operator $ here has different precedence relative to other operators than it has in real PAL.

```
(0)   S   ::=  ⊢ P ⊣

(1)   P   ::=  PL  |  E
(3)   PL  ::=  def D PL  |  def D

(5)   E   ::=  let D in E  |  fn VB . E  |  EW
(8)   EW  ::=  EV where DR  |  EV
(10)  EV  ::=  valof C  |  C

(12)  C   ::=  CL ; C  |  CL
(14)  CL  ::=  NAME : CL  |  CC
(16)  CC  ::=  test B ifso CL ifnot CL  |  test B ifso CL ifnot CL
            |  if B do CL  |  unless B do CL  |  while B do CL
            |  until B do CL  |  CB
(23)  CB  ::=  T := T  |  goto R  |  res T  |  T

(27)  T   ::=  TA , T  |  TA
(29)  TA  ::=  TA aug TC  |  TC              ⎧ comment: "bar" really
(31)  TC  ::=  B -> TC bar TC  |  TE         ⎪ should be "|" but if it
(33)  TE  ::=  $ R  |  B                     ⎨ were, the BNF would
                                            ⎩ read incorrectly.

(35)  B   ::=  B or BT  |  BT
(37)  BT  ::=  BT & BS  |  BS
(39)  BS  ::=  not BP  |  BP
(41)  BP  ::=  A RLN A  |  A

(43)  A   ::=  A + AT  |  A - AT  |  + AT  |  - AT  |  AT
(48)  AT  ::=  AT * AF  |  AT / AF  |  AF
(51)  AF  ::=  AP ** AF  |  AP
(53)  AP  ::=  AP % NAME R  |  R

(55)  R   ::=  R RN  |  RN
(57)  RN  ::=  NAME  |  CONST  |  ( E )  |  [ E ]

(61)  D   ::=  DI within D  |  DI
(63)  DI  ::=  DI inwhich DA  |  DA
(65)  DA  ::=  DR and DA  |  DR
(67)  DR  ::=  rec DB  |  DB
(69)  DB  ::=  VL = E  |  NAME V = E  |  ( D )  |  [ D ]

(73)  V   ::=  VB V  |  VB
(75)  VB  ::=  NAME  |  ( VL )  |  ( )
(78)  VL  ::=  NAME , VL  |  NAME
```

Table 7-1. The PAL grammar. It has 48 terminals, 3 of which are pseudo-terminals (NAME, CONST, and RLN), 32 nonterminals, and 80 productions. The grammar is SLR(1).

Some statistics pertinent to the PAL grammar are as follows. It has 48 terminals, 3 of which are pseudo-terminals, 32 nonterminals, and 80 productions. The corresponding CFSM has 157 states, 26 of which are inadequate, but none of which are multiply inadequate, and 61 of which must push their names on the stack during parsing.

Since our interest here is primarily in PAL's CFST, we concentrate on its transduction grammar rather than the production-node pairs. The SSTG is implied by PAL's output grammar which is presented in Table 7-2. The output grammar is our own concoction; heretofore, the correspondence between PAL programs and abstract syntax trees has been specified informally by PAL designers.

When the SSTG is viewed as a specification of the CFST, the pseudo-terminals should be regarded as nonterminals; however, if the outputs from the CFST and the lexical translator together, as seen by the ASTB, are being specified, the pseudo-terminals should be regarded as terminals. (Recall the restriction discussed on page 160 and the summary of the interactions of the components of our compiler model on page 154).

In most cases the abstract-syntax-tree node corresponding to a given production can be determined from its transduction element ω' as follows: if ω' consists only of a nonterminal, there is no node, or if only a pseudo-terminal, then a terminal node with "semantics", of if $\omega' = \gamma\delta$ where γ is a string of nonterminals and pseudo-terminals and δ is a

```
(0)   S    ::=   P

(1)   P    ::=   PL  |  E
(3)   PL   ::=   D PL def  |  D lastdef

(5)   E    ::=   D E let  |  VB E λ  |  EW
(8)   EW   ::=   EV DR where  |  EV
(10)  EV   ::=   C valof  |  C

(12)  C    ::=   CL C ;  |  CL
(14)  CL   ::=   NAME CL :  |  CC
(16)  CC   ::=   B CL CL test-t  |  B CL CL test-f  |  B CL if
                 |  B CL unless  |  B CL while  |  B CL until  |  CB
(23)  CB   ::=   T T :=  |  R goto  |  T res  |  T

(27)  T    ::=   TA T ,  |  TA
(29)  TA   ::=   TA TC aug  |  TC
(31)  TC   ::=   B TC TC test-t  |  TE
(33)  TE   ::=   R $  |  E

(35)  B    ::=   B BT or  |  BT
(37)  BT   ::=   BT BS &  |  BS
(39)  BS   ::=   BS not  |  BP
(41)  BP   ::=   A RLN A rln  |  A

(43)  A    ::=   A AT +  |  A AT -  |  AT pos  |  AT neg  |  AT
(48)  AT   ::=   AT AF *  |  AT AF /  |  AF
(51)  AF   ::=   AP AF **  |  AP
(53)  AP   ::=   AP NAME R %  |  R

(55)  R    ::=   R RN γ  |  RN
(57)  RN   ::=   NAME  |  CONST  |  E  |  E

(61)  D    ::=   DI D within  |  DI
(63)  DI   ::=   DI DA inwhich  |  DA
(65)  DA   ::=   DR DA and  |  DR
(67)  DR   ::=   DB rec  |  DB
(69)  DB   ::=   VL E =  |  NAME V E ff  |  D  |  D

(73)  V    ::=   VB V bv  |  VB
(75)  VB   ::=   NAME  |  VL  |  ()
(78)  VL   ::=   NAME VL vl  |  NAME
```

Table 7-2. The output grammar for PAL.

terminal, the node has name δ and $|\omega'|-1$ sons which are the nodes corresponding to the symbols in γ and in the same order. Exceptional cases are trivially different and of no importance here, since they concern the node-building subroutines of PAL's ASTB, but not its CFST.

PAL's CFST is presented in Figure 7.3 where the ST spans the first two pages, the TT spans the first three, and the LAT is shown in the fourth page. In the latter case only the numbered lines are to be considered in the LAT; we have included the extra lines to indicate for each nonterminal A the set $F_T^1(A)$ for the thoroughly interested reader.

Space-efficiency. For lack of a better choice, we define the space-efficiency of a translator T corresponding to an SSTG G_t to be the ratio of the space necessary for storing G_t to that for storing T.

Let us compute a rough estimate of the space-efficiency of PAL's CFST. The ST contains 172 entries. There are seven possible values for the TYPE component, requiring three bits, values as high as 18 in the NUM component, requiring five bits, and values as high as 254 in the TTREF component, requiring eight bits. Thus, the ST requires $172*(3+5+8) = 2752$ bits. The TT contains 255 entries. The largest values in the SYM and STATE components are 154 and 171, respectively, requiring eight bits each. Thus, the TT requires $255*(8+8) = 4080$ bits. The LAT requires 16 rows, each with 48 binary entries, or 768 bits. In total the translator requires 7600 bits, or 238 words at 32 bits per word, of memory space.

STATE	TABLE		TRANSITION TABLE			STATE	TABLE		TRANSITION TABLE	
	TYPE	NUM	TTREF	SYM	STATE		TYPE	NUM	TTREF	SYM	STATE
0	READ	1	0	⊢	1	44	POP	1	93	115	149
1	↓READ	19	1	def	5	45	↓LA	2	94	116	150
2	READ	1	22	let	6	46	LA	3	96	117	151
3	POP	1	254	fn	7	47	POP	1	248	118	152
4	LB	2	252	valof	10	48	LA	2	99	156	159
5	↓READ	4	23	NAME	13	49	↓READ	3	24	157	160
6	↓READ	4	23	test	15	50	LB	2	101	---	12
7	READ	2	27	if	16	51	READ	1	103	:=	69
8	LB	7	29	unless	17	52	↓LA	4	104	3	14
9	LA	2	35	while	18	53	↓READ	4	23	,	72
10	↓READ	15	5	until	19	54	↓READ	4	23	aug	73
11	POP	1	251	goto	22	55	READ	1	108	4	162
12	LA	2	37	res	23	56	READ	1	109	->	74
13	LA	2	39	$	28	57	LB	3	245	or	75
14	LB	10	41	not	31	58	↓READ	2	110	5	27
15	↓READ	7	14	+	34	59	↓READ	4	23	73	121
16	↓READ	7	14	-	35	60	POP	1	112	74	122
17	↓READ	7	14	CONST	41	61	↓READ	15	5	153	158
18	↓READ	7	14	(	42	62	↓READ	15	5	---	24
19	↓READ	7	14	[	43	63	↓READ	3	113	&	77
20	POP	1	250	NAME	41	64	LB	2	243	6	163
21	LA	2	51	10	167	65	READ	2	115	75	114
22	↓READ	4	17	⊣	44	66	READ	2	117	---	29
23	↓READ	8	13	rec	49	67	READ	2	118	77	123
24	LA	3	53	NAME	52	68	READ	2	120	---	30
25	POP	1	249	(	53	69	↓READ	8	13	RLN	79
26	LA	3	56	[	54	70	↓LA	5	122	+	80
27	LB	4	59	NAME	99	71	POP	1	127	-	81
28	↓READ	4	17	(	58	72	↓READ	8	13	7	164
29	LA	2	63	1	4	73	↓READ	8	13	*	84
30	LB	2	65	42	89	74	↓READ	8	13	/	85
31	↓READ	6	15	96	136	75	↓READ	7	14	8	165
32	LB	2	67	104	142	76	↓LA	5	128	34	82
33	LA	4	69	105	143	77	↓READ	7	14	35	83
34	↓READ	4	17	138	155	78	POP	1	133	80	125
35	↓READ	4	17	where	59	79	↓READ	6	15	81	126
36	LA	3	73	1	8	80	↓READ	4	17	---	36
37	LB	5	76	;	61	81	↓READ	4	17	**	86
38	LA	3	81	2	161	82	LA	3	134	%	87
39	↓LA	5	17	:	62	83	LA	3	137	9	166
40	LB	4	84	11	41	84	↓READ	4	17	22	70
41	LB	5	88	62	111	85	↓READ	4	17	28	76
42	↓READ	18	2	112	146	86	↓READ	4	17	130	154
43	↓READ	18	2	113	147	87	READ	1	140	---	39

Figure 7.3. PAL's CFST (through page 188).

STATE TABLE (left)

	TYPE	NUM	TTREF
88	POP	1	141
89	READ	1	142
90	READ	1	143
91	POP	2	144
92	↓READ	4	23
93	↓READ	4	23
94	↓READ	4	23
95	POP	1	145
96	↓READ	18	2
97	↓READ	2	110
98	READ	1	146
99	↓LA	3	147
100	LB	2	241
101	POP	1	240
102	READ	1	150
103	READ	1	151
104	↓READ	18	2
105	↓READ	18	2
106	READ	1	152
107	LA	2	153
108	POP	1	155
109	POP	1	156
110	POP	1	157
111	POP	1	158
112	↓READ	15	5
113	↓READ	15	5
114	LA	2	159
115	↓READ	15	5
116	↓READ	15	5
117	↓READ	15	5
118	↓READ	15	5
119	POP	1	161
120	POP	1	162
121	POP	1	163
122	READ	1	164
123	POP	1	165
124	LA	3	166
125	LA	3	169
126	LA	3	172
127	POP	1	175
128	POP	1	176
129	POP	1	177
130	↓READ	4	17
131	POP	1	178
132	POP	1	239
133	POP	1	179

TRANSITION TABLE (left)

SYM	STATE
39	88
70	88
76	88
154	88
---	40
€	171
def	5
0	168
within	92
inwhich	93
12	169
and	94
13	170
59	109
---	48
=	96
,	97
NAME	99
(	58
15	148
in	104
.	105
NAME	107
)	108
valof	9
ifso	112
ifnot	113
or	75
do	115
do	116
or	75
do	117
do	118
or	75
CONST	41
(	42
[	43
NAME	41
3	145
res	114
CONST	41
(	42
[	43
NAME	41
5	141
not	30

STATE TABLE (right)

	TYPE	NUM	TTREF
134	POP	1	180
135	POP	1	181
136	POP	1	182
137	POP	2	183
138	↓READ	18	2
139	POP	1	184
140	POP	1	185
141	POP	1	238
142	POP	2	186
143	POP	2	187
144	POP	1	188
145	POP	1	237
146	READ	1	189
147	READ	1	190
148	LB	3	234
149	POP	2	191
150	POP	2	192
151	POP	2	193
152	POP	2	194
153	↓READ	7	13
154	LA	5	195
155	POP	2	200
156	↓READ	15	5
157	↓READ	15	5
158	POP	2	201
159	POP	4	202
160	POP	4	203
161	LB	3	204
162	LB	3	207
163	LB	6	210
164	LB	2	216
165	LB	2	218
166	LB	4	220
167	POP	1	224
168	POP	2	225
169	LB	5	226
170	LB	3	231
171	EXIT	-	-

TRANSITION TABLE (right)

SYM	STATE
*	84
/	85
8	132
*	84
/	85
8	101
NAME	130
γ	40
)	131
]	131
def	100
rec	50
=	138
NAME	99
(	58
14	64
)	140
]	140
)	144
,	97
15	57
()	99
where	8
;	9
:	12
&	77
6	47
:=	14
,	21
aug	24
bar	153
&	29
+	80
-	81
7	25
*	84
/	85
8	20
*	84
/	85
8	11
*	36
/	36
**	37
€	41
within	46

Figure 7.3. Continued.

	TRANSITION TABLE				TRANSITION TABLE	
	SYM	STATE			SYM	STATE
180	inwhich	46		218	79	124
181	and	170		219	---	33
182	=	4		220	84	127
183	vl	148		221	85	128
184	bv	57		222	86	129
185	ε	4		223	---	37
186	let	8		224	ε	38
187	λ	8		225	lastdef	100
188	ε	99		226	5	45
189	ifnot	156		227	6	55
190	ifso	157		228	53	102
191	if	14		229	54	103
192	unless	14		230	92	133
193	while	14		231	93	134
194	until	14		232	94	135
195	CONST	41		233	---	46
196	(	42		234	58	106
197	[	43		235	97	137
198	NAME	41		236	---	51
199	10	3		237	goto	14
200	ff	4		238	$	27
201	testT	24		239	ε	33
202	testT	14		240	neg	33
203	testF	14		241	1	2
204	10	60		242	45	91
205	61	110		243	52	98
206	---	9		244	99	139
207	23	71		245	58	106
208	69	119		246	97	137
209	72	120		247	---	51
210	15	63		248	or	26
211	16	65		249	rl	32
212	17	66		250	+	33
213	18	67		251	-	33
214	19	68		252	49	95
215	---	26		253	---	50
216	31	78		254	%	38
217	---	32				

Figure 7.3. Continued.

LOOK-AHEAD TABLE

```
         ↑ let    where    test  ifnot do    while   :=    res  aug  bar  or   not  +    *    **   NAME         (         [       within    and  =
         def in   .    valof  ;   ifso  if   unless until   goto  .    !    $    &    RLN  !    /    %    CONST    )         ]       inwhich   rec

      ┌─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
      │ 1                                                                                                                        P
 0    │ 1                                                                                                                        PL
      │ 11  1                                                                                             1   1111               E
 1    │ 11  1                                                                                             1   1111               EW
      │ 11  1   1                                                                                         1   1111               EV
 2    │ 11  1   1                                                                                         1   1111               C
      │ 11  1   1   1    11                                                                               1   1111               CL
      │ 11  1   1   1    11                                                                               1   1111               CC
 3    │ 11  1   1   1    11                                                                               1   1111               CB
 4    │ 11  1   1   1    11         1                                                                     1   1111               T
      │ 11  1   1   1    11         1        11                                                           1   1111               TA
      │ 11  1   1   1    11         1        11   1                                                       1   1111               TC
 5    │ 11  1   1   1    11         1        11   1                                                       1   1111               TE
 6    │ 11  1   1   1    11   1     1        1111 1                                                       1   1111               B
      │ 11  1   1   1    11   1     1        1111 11                                                      1   1111               BT
      │ 11  1   1   1    11   1     1        1111 11                                                      1   1111               BS
 7    │ 11  1   1   1    11   1     1        1111 11                                                      1   1111               BP
 8    │ 11  1   1   1    11   1     1        1111 11   111                                                1   1111               A
      │ 11  1   1   1    11   1     1        1111 11   11111                                              1   1111               AT
 9    │ 11  1   1   1    11   1     1        1111 11   11111                                              1   1111               AF
10    │ 11  1   1   1    11   1     1        1111 11   1111111                                            1   1111               AP
      │ 11  1   1   1    11   1     1        1111 11   1111111111111111                                       1111               R
11    │ 11  1   1   1    11   1     1        1111 11   1111111111111111                                       1111               RN
12    │ 11  1                                                                                             1   1                  D
      │ 11  1                                                                                             1   111                DI
13    │ 11  1                                                                                             1   111                DA
      │ 11  1                                                                                             1   1111               DR
      │ 11  1                                                                                             1   1111               DB
14    │                                                                                                                    1     V
      │         1                                                                                         1   1              1    VB
15    │                                                                                                       1                   VL
```

Figure 7.3. Continued.

Now there are 80 distinct symbols necessary to store the SSTG, thus requiring seven bits each. If for each production p, $A \to \omega$, we assume we need store only $|\omega|+2$ symbols, one for the left part, $|\omega|$ for the right part, and one for an output terminal, then it takes $342*7 = 2394$ bits = 75 words to store PAL's SSTG. Thus the space-efficiency of the PAL translator is a respectable $75/238 = 31\%$. It seems clear that this figure could be increased somewhat by bringing to bear some coding tricks; however, it is not our purpose here to develop an optimal implementation as regards either space or time. In fact, our scheme is already competitive with existing schemes, as we show next by comparing it with one which is well known to be fast and efficient (F&G 68).

7.5 Comparison with a Precedence Scheme

For the sake of simplicity we compare parsers rather than translators, and we use the PAL grammar when we need pertinent statistics. In Figure 7.4 we present a flowchart describing a variation of a "Simple Precedence" parser (W&W 66) which is compatible with our terminology and compiler model. We do not detail the actions of the parser but only note that it makes read-reduce decisions and locates reducible substrings by looking up "precedence relations" in a precedence matrix (PM), and it determines which production, with left part A and output symbol OutSym, is applicable by searching (via Search) the set of productions to find one with a right part that matches the reducible substring.
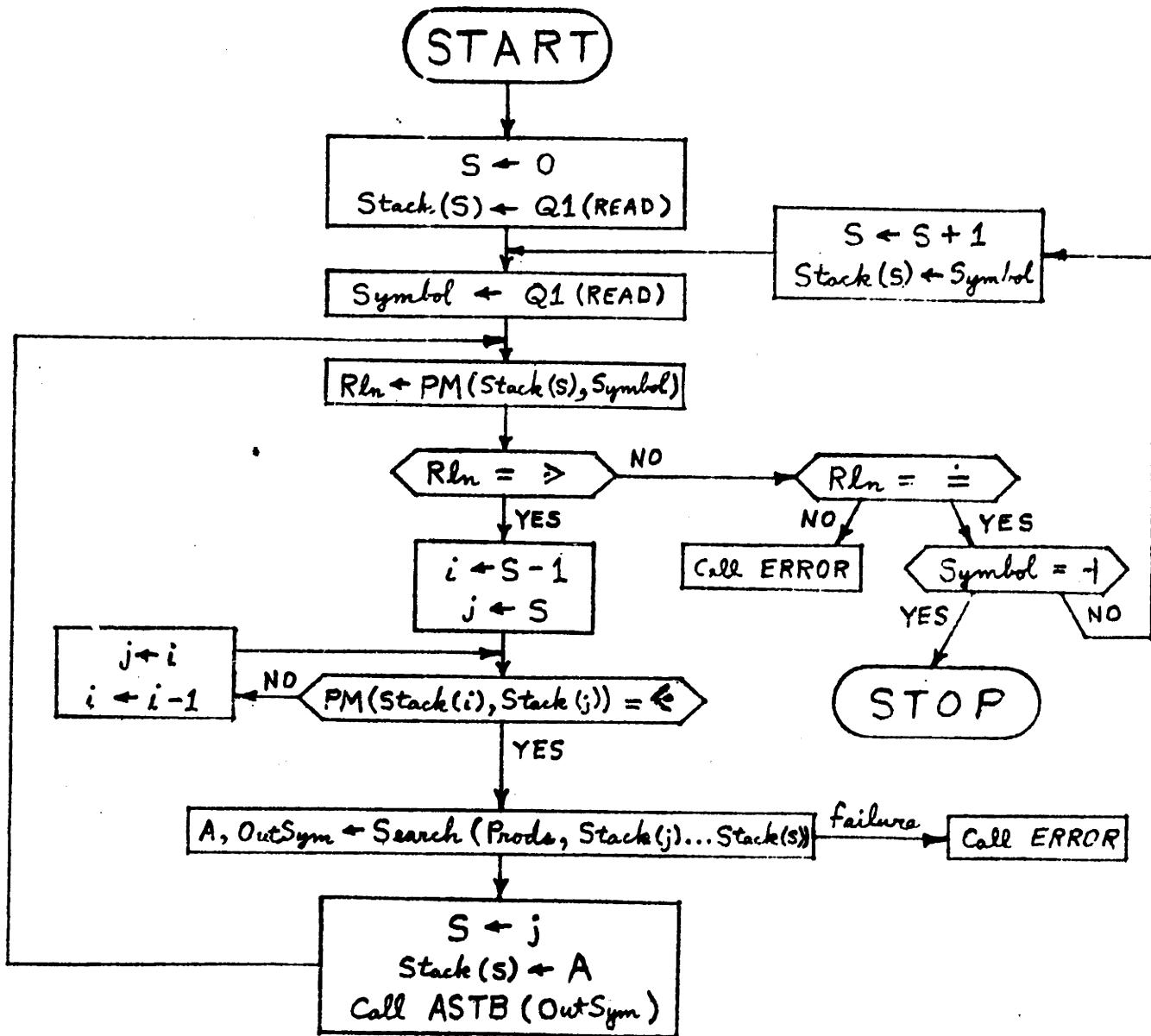
Figure 7.4. The interpreter for a "Simple Precedence" parser.

Space. Each entry in the PM can have one of four values, ∸ , ∸ ,
∸>, or "none", so two bits are required for each. The rows and columns

of the PM correspond to the symbols in the grammar. For PAL there

are 80 symbols so the size of the corresponding PM would be

80*80*2 = 12, 800 bits = 400 words. The size of the production table with

output symbols would probably be greater than what we calculated above:

75 words. Thus, the whole parser would require greater than 475 words,

or twice as much as our translator above. Of course our parser might

be somewhat larger than the CFST above because of the extra output

symbols, but probably no more than 15% larger. A more significant

difference would be that our interpreter would be larger than that for the

precedence scheme, perhaps by an extra 50 words or so (an "educated

guess"). On the other hand, the amount of space necessary for the stack

during execution for our scheme would be less than that for the PM scheme

(see page 62). In conclusion, then, the two schemes are roughly com-

parable in space usage.

Time. Let us now compare the speeds of the schemes. Following

this paragraph are four lists of statements which must be executed in the

performance of reads and reductions by the two schemes. To the left of

the statements we indicate very rough estimates of the time required to

execute each statement individually (generally one time unit per statement)

and each group of statements. The groups comprise statements which

are executed variable numbers of times depending on the production or state involved. We weight these groups according to statistics derived from PAL's grammar, CFSM, or translator, as appropriate, and we indicate the pertinent statistics to the right of the statements.

Precedence Matrix (PM) read:

```
1   Symbol ← Q1(READ)      we count only the store; no lexical
2   Rln ← PM(Stack(S),Symbol)                          analysis
1   Rln = > ?
1   Rln = ≐ ?
1   Symbol = ⊣ ?
1   S ← S + 1
1   Stack(S) ← Symbol
```
───────────────────
8   time units total

CFST read:

```
1   ST(State,TYPE) = (↓) AD or (↓)LA ?
      { 1   Stack(S) ← State }      61  ↓READ and ↓LA states
1.5 { 1   S ← S + 1       } *    ─────────────────────────
      {                    }        82  (↓)READ and (↓)LA states
1   Symbol ← Q1(READ)
      ⎧ 1   Last ← TTRef + ST(State,NUM) - 1 ⎫
      ⎪ 1   Symbol = TT(TTRef,SYM) ?          ⎬ *1.7
6.8 ⎨ 1   TTRef ← TTRef + 1                   ⎪
      ⎪ 1   TTRef > Last ?                     ⎭
      ⎪ (½ (linear search) * avg. no. of read transitions
      ⎩  from (↓)LA and (↓)READ states)
1   State ← TT(TTRef,STATE)
1   TTRef ← ST(State,TTRef)
```
─────────────────────────
12.3   time units total --- about 1.5 times as long
                                as a PM read

PM reduce:

```
0   Symbol ← Q1(READ)    happens infrequently
2   Rln ← PM(Stack(S),Symbol)
1   Rln = > ?
1   i ← S - 1
1   j ← S
```

$$9.2 \begin{Bmatrix} 2 & PM(Stack(i),Stack(j)) = < ? \\ 1 & j ← 1 \\ 1 & i ← i - 1 \end{Bmatrix} * 2.3 \frac{symbols}{right\ part}$$

```
6.6  A, OutSym ← Search(Prods,Stack(j)...Stack(i))
     (1 time unit per store + 2 per each symbol in right part)
1   S ← j
1   Stack(S) ← A
1   ASTB(OutSym)
```
23.8 time units total

CFST reduce:

$$3.6 \begin{Bmatrix} 1 & ST(State,TYPE) = (↓)LA ? \\ \begin{Bmatrix} 1 & Stack(S) ← State \\ 1 & S ← S + 1 \end{Bmatrix} * \frac{6\ ↓LA\ states}{26\ (↓)LA\ "} \\ 1 & LASymbol ← Q1(LA) \\ 1 & Last ← TTRef + ST(State,NUM) - 1 \\ \begin{Bmatrix} 1 & LASymbol = TT(TTRef,SYM) ? \\ 1 & TTRef ← TTRef + 1 \\ 1 & TTRef ≥ Last ? \end{Bmatrix} * \frac{53}{26} \\ (avg.\ no.\ read\ transitions \\ from\ (↓)LA\ states\ ) \\ 2 & LAT(TT(TTRef,SYM),LASymbol) = 1 ? \end{Bmatrix} * \frac{26\ (↓)LA\ states}{80\ productions}$$

$$0.6 \{ 1 \quad ST(State,TYPE) = POP ? \} * 48/80 \ POP \ states/productions$$

```
1   S ← S - ST(State,NUM)
1   Call ASTB(TT(TTRef,SYM))
```

$$1.8 \begin{Bmatrix} 1 & ST(State,TYPE) = LB ? \\ 1 & TopState ← Stack(S-1) \\ 1 & Last ← TTRef + ST(State,NUM) - 1 \\ \begin{Bmatrix} 1 & TopState = TT(TTRef,SYM) ? \\ 1 & TTRef ← TTRef + 1 \\ 1 & TTRef ≥ Last \end{Bmatrix} * \frac{80}{22} \\ (avg.\ no.\ transitions\ from\ LB \\ states * \frac{1}{2}\ (linear\ search)) \end{Bmatrix} * \frac{22\ LB\ states}{80\ productions}$$

8.0 time units total --- about 1/3 as long as a PM reduce

In conclusion, we see from the above that on the average the PM scheme reads symbols about 1.5 times as fast as does ours, but our scheme makes reductions about three times as fast as the PM scheme.

Of course, our estimates are <u>very</u> rough, but we believe it is clear from
this that the two schemes are also roughly comparable as far as speed
goes.
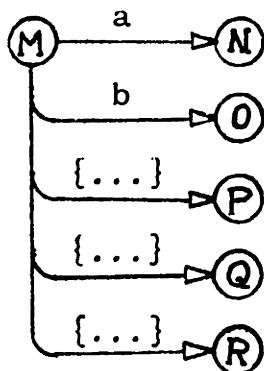
7.6   Variations, Extensions

There are, of course, many ways in which our scheme could be
speeded up.   We mention two here because they seem particularly
appropriate.   First, the read states with many transitions could be
implemented as "transition matrix (TM)" look-ups; i. e., such that, if
the next symbol is Symbol and the current "TMREAD" state is State,
then TM(State, Symbol) would be the next state.   This would substantially
increase the average read speed at some storage cost.   Since for PAL
there are 18 read states with 10 or more transitions, the cost would be
about 18*48* = 6912 bits = 216 words extra to implement those 18 as
"TMREAD" states. Second, whether or not the first method is used, the
ST and TT could be compiled rather than interpreted.   In the nature of these
things we might expect a factor of ten increase in speed for a factor of
four increase in space, say.   Since this would still leave us with a
reasonable amount of space usage, it would represent a reasonable space-
time trade-off for our purposes.   The main point here is that our implement-
ation method is flexible.

<u>Extentions.</u>   We next discuss the modifications to our implementation
methods necessary to cover multiply inadequate states and k-symbol look-ahead.

Our intent is merely to indicate the ease with our method can be extended to cover these "exceptional cases."
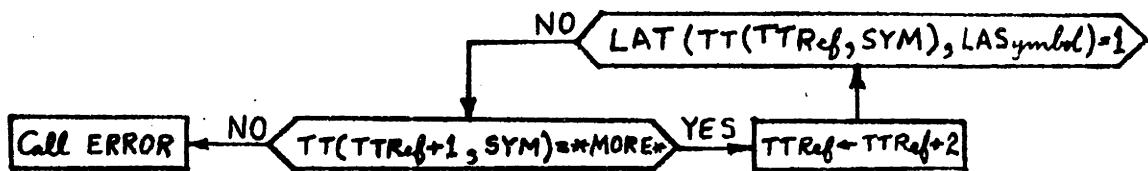
Multiply inadequate states. In general, the multiply inadequate state may have several read transitions and several look-ahead transitions. For example, we might have the following:
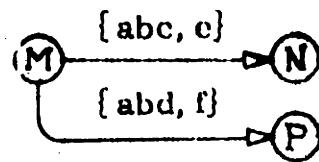

One way to implement such a state would be first to implement it as we did above but with only one of the look-ahead transitions and then to store the extra look-ahead transitions in the TT immediately below the other transitions as follows. For each transition add two entries to the TT: (1) the first having an irrelevant STATE component and a special symbol, *MORE*, in the SYM component which has a representation distinct from all other items which can appear in the SYM component, and (2) the second having a regular (SYM, STATE) pair corresponding to the transition in question. For the above state the table entries would be as follows.

| STATE TABLE				TRANSITION TABLE	
TYPE	NUM	TTREF		SYM	STATE
M (↓)LA		3		a	N
				b	O
				r1	P
				MORE	---
				r2	Q
r1, r2, and r3				*MORE*	---
are references				r3	R
to the LAT					
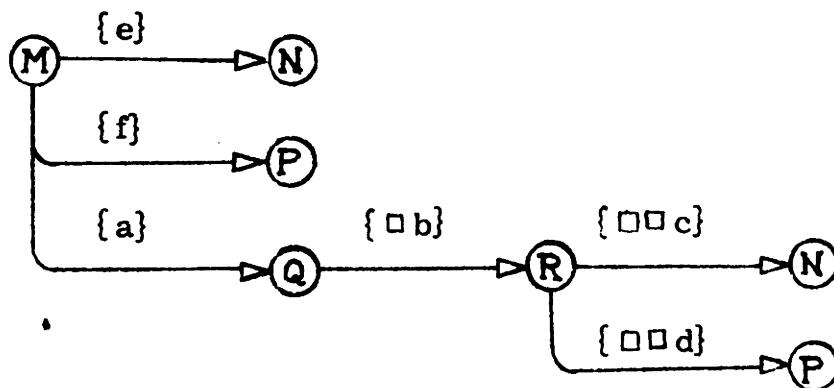
It should be clear that we can implement any multiply inadequate state in

this way. Of course our interpreter will have to be modified to be ready

for such states. The modification is trivial. It concerns only the bottom-

most decision box in Figure 7. 2. The NO exit must be changed as indicated

next.

Look-ahead for k > 1. To cover look-ahead of more than one

symbol, we could add a new type of state, namely LAk for "look-ahead

at the k-th symbol. " This would require an additional exit point from the

topmost decision box in Figure 7. 2. We illustrate our proposed modification

via example. Suppose we want to implement the following state.

What we propose can be represented diagrammatically as follows.

We intend to imply by this diagram that M is a look-ahead state with three look-ahead transitions of the normal type, that Q has one look-ahead transition but it indicates a comparison with the second symbol ahead rather than the first, and that R has two look-ahead transitions which investigate the third symbol ahead. State M could be implemented as we have just discussed. States Q and R would be LAk states with tabular representations as follows.
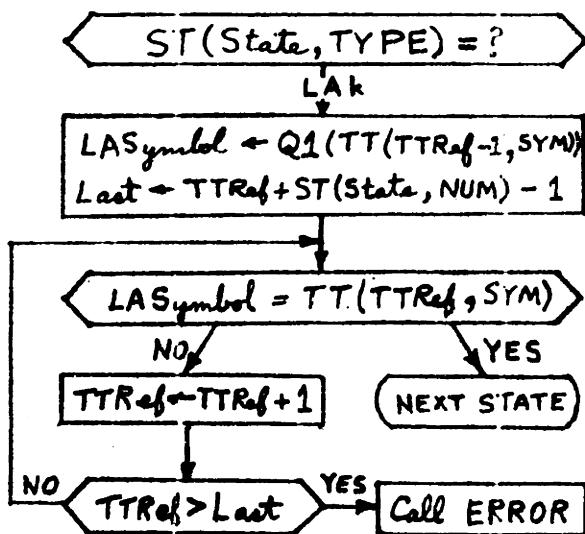
STATE	TABLE				TRANSITION	TABLE
	TYPE	NUM	TTREF		SYM	STATE
					LA2	---
Q	LAk	1			b	R
					LA3	---
R	LAk	2			c	N
					d	P

The corresponding section of the interpreter would be as follows, where we assume that, when Q1 is called with argument LAn for some specific n = 2, 3, ..., it returns as value the n-th symbol in the queue, counting from the bottom, but it does not change the queue.

Note that if we use the variable Symbol rather than LASymbol above, the flowchart from the line beginning Last... down is exactly the same as the counterpart in the READ section. Thus, the modification requires only one new exit from the TYPE-test box, one extra statement, and a transfer into the READ portion of the interpreter. The only question remaining, then, is how do we deduce the new states and their interconnections? We believe the answer to this question is obvious so we do not treat it here.

Thus, we have shown that "exceptions" like multiply inadequate states and k-symbol look-ahead states can be implemented with little change to our

interpreter and with changes which do not affect the speed of the interpreter for "normal" states. We have, then, a very flexible method.

Chapter 8

CONCLUSIONS

8.1   Future Development

Before our results will be ready for actual incorporation in a

TWS several variations should be investigated as possible improvements.

These variations concern computational methods, strategy, diagnostics,

and translator-implementation methods.

Computational Methods. We believe, for instance, that Knuth's

parser-generating technique (Knu 65) could be adapted for use in the

generation of CFSMs. Specifically, we believe that the set of all possible

"state sets" generated by his algorithm for grammar G and $k = 0$ is

isomorphic to the set of states of G's CFSM. Furthermore, if the latter

is true, the "bit matrix" techniques of Lynch (Lyn 68) can probably be

used for the very fast generation of CFSMs. We suspect that the resulting

method would be faster than our piecemeal method in Section 7.1.

Another possible area of improvement regards the computation of

look-ahead sets and context pairs and the attendant strategy. We do not

think this will be a critical issue for programming languages because we

expect most of the related grammars to be either SLR(1) or very nearly

SLR(1). However, for the sake of generality, exceptional cases, and

the possible use of our TWS to build systems for more general "syntax-

directed" computations, it would be reasonable to research further in this

area.

Strategy: Of course, the obvious thing to do in complex cases
is to make the TWS interactive so the language designer, who presumably
knows the grammar best, can assist in determining strategy. It may
be reasonable, however, to provide more (or other) than the three
methods for computing look-ahead sets that were provided above. Con-
veniently, our technique as a whole is amenable to other such methods;
i. e., we do not care how look-ahead sets and state splitting are computed
as long as they result in a correct parser.

Computation of look-ahead, state-splitting. With regard to this
area of possible improvement we briefly list three methods which should
be investigated.

(1) Especially if Knuth's algorithm is adapted for the generation of CFSMs,
it should also be investigated for possible adaptation for computing simple
1-look-ahead sets. This would require the separation in his technique of
the computations of look-ahead sets and state-splitting, which we believe
to be easy to do. Actually, we believe the resulting technique would cover
slightly more than the SLR(1) grammars, perhaps with little or no more
complexity (computation time) than our SLR(1) technique. We do not
believe, however, that the technique would be nearly as fast as our
SLR(k) technique for $k > 1$ because we see no simple way of using it to
compute look-ahead for a single state.

(2) Lynch (Lyn 68) has a fast technique for computing left and right context in which each symbol is computed independently of all others. This technique should be investigated as a possible prelude to or replacement for the computation of <u>corresponding</u> context pairs.

(3) Finally, for the general LR(k) case the look-ahead sets might be most easily computed by simulation of the parser in a nondeterministic manner; i. e., for each left context φ see if $\varphi\beta\gamma$ is a canonical form, where $\beta\gamma \in V_T^*$ and $|\beta| \leq k$, by determining if there is any sequence of actions by the stack algorithm which will cause $\varphi\beta$ to be read. Of course it must be proven that this method would result in the appropriate look-ahead sets.

<u>Diagnostics</u>. A related area which needs investigation concerns diagnoistic messages to the language designer. What information would be useful to the designer when his grammar is found not to be SLR(k) or LR(k)? Presumably, in such cases the designer has inadvertently submitted an ambiguous grammar, since we expect all of his unambiguous grammars to be SLR(k) or at least LR(k). The diagnostics should, of course, lead the designer to find the reason why the grammar is ambiguous.

<u>Implementation methods</u>. Finally, there are several possible ways in which our translator implementation could be improved. First, a way of implementing in a reasonable amount of space states which jump over long cascades of look-ahead and look-back states is desirable. We suspect that these can be implemented by using bit matrices in a manner similar to

precedence techniques. Second, similar bit-matrix techniques may also be useful for speeding up read states with many transitions, rather than using transition matrices. Third, we believe that the obvious way (for most applications) to implement the state and transition tables which remain after the above modifications is to compile them into machine code.

8.2   Conclusions

We believe that we have demonstrated the validity of our thesis. We have a practical translator-constructing technique which grows in complexity as it discovers the complexity of the grammar at hand, and it generates practical translators for SSTGs which are based on LR(k) grammars and which partically specify useful, readable programming languages. Thus we have a basis for a TWS in which the key feature is flexibility.

First and foremost, we have given the language designer flexibility in the design of his grammar. From the beginning it has been our desire to get a method which would accept a CF grammar as it was designed as a syntactical reference for a language, with no modifications. That is, we wanted a method that would accept a "humanized" version of the syntax. To the extent that unambiguity is considered a desirable trait of such a reference, we believe we have such a method. This belief is founded on the intuitive grounds that, when a designer sets out to define part of the

syntax of a programming language via a CF grammar, he will just naturally come up with an LR(k) grammar, and in fact, probably an SLR(1) grammar.

Second, we have given the implementor of the TWS flexibility. He has the flexibility to build-in whatever strategy is appropriate for the purposes at hand for deciding whether grammars are LR(k), and he can leave some of that strategy to be decided by the language designer. He also has the flexibility not only to implement each translator as a whole in a variety of ways, but also to implement particular states in special ways. In fact, see (DeR 68) for a proposal concerning the use of different kinds of parsing techniques on different parts of a grammar.

8.3 Future Research, Extensions

The area of future research most important to our results is that of language design and specification itself. The value of our results is somewhat limited until there is developed a useful, unified methodology for specifying programming languages fully, which incorporates something similar to SSTGs and/or production-node pairs. We have proceeded on the assumption that such a methodology is forthcoming, and we have faith that one is (see for example (Knu 66) and (Tho 69)).
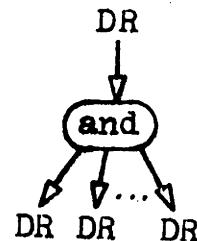
A more specific design problem, which is part of the above area and important to our results, is the one discussed in detail in Section 6.8. Once the designer has in mind a set of abstract syntax trees, operator

precedences and associativities, scopes of variables, etc., how can he algorithmically generate an appropriate CF grammar which has the corresponding "structural properties" and which is guaranteed to be LR(k), or even better SLR(1)? Currently, the generation of such grammars is definitely an art, being performed on the basis of past experience and trial-and-error methods.

Another related problem is that of extending the usefulness of CF grammars, and therefore, BNF. There are three ways in particular in which we would like to see their powers extended. It goes without saying that we would also like to see our techniques extended to cover these extensions.
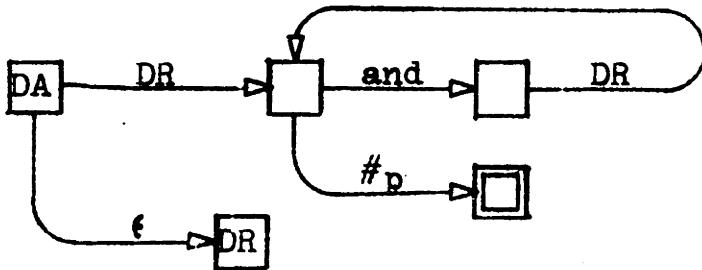
(1) We often like to indicate via regular expressions in right parts of productions that certain operators are nonassociative. For instance, for PAL the following production-node pair specifies the correspondence between an abstract-syntax-tree node and strings involving the nonassociative (syntactic) operator "and":

(p)   DA ::= DR { and DR }*

There seems to be no natural way of indicating this correspondence using pure BNF. Can we construct a CFSM from a grammar including the above

production in a manner similar to that given in Section 7.1? Presumably,

the piece of FSM corresponding to that production is as follows:

But what should be the "reduction procedure" executed by the corresponding

DPDA for the $\#_p$-transition? How should that procedure interact with the

ASTB?

(2) One can often indicate a reduction in the need for parentheses in certain

special contexts via special context-sensitive productions. For instance,

to use a trivial example, the meaning of the following subexpression seems

clear:

$$...(1 + \underline{if}\ B\ \underline{then}\ 2\ \underline{else}\ 3)...$$

And yet the ALGOL 60 syntax disallows it, requiring the programmer to

write:

$$...(1 + (\underline{if}\ B\ \underline{then}\ 2\ \underline{else}\ 3))...$$

Often the set of legal programs can be extended to include subexpressions

such as the former one above by adding either a large number of CF pro-

ductions or only one or two context-sensitive productions to the grammar.

If in the latter case the resulting language is CF, our results should, in theory, still be applicable. Can we get sufficient conditions on the allowable contexts such that we still have a CF language? Can we modify our DPDA in a simple way so that it checks these contexts at the appropriate times and therefore recognizes the intended language?

(3) Finally, since we are really interested in translations rather than parses, can we change our notions of unambiguity to correspond to the former rather than the latter in such a way that we can extend our techniques to cover all "unambiguous" SSTG's? See (Eva 65) for some results in this area.

More in regard to compilers, we note that it is probably true that if we retain transitions under nonterminals, we would have an "incremental compiler"; i.e., one which would accept a string which is already partially parsed. (A proof is needed.) If these transitions were stored in some special place, rather than directly in the CFST, and if the reads and look-aheads concerning nonterminals were treated as special cases, our compiling speed for terminal strings would not be reduced. Perhaps a compiler would be constructed using this technique which would have good recompilation characteristics, and therefore, good overall "efficiency".

Finally, our automata-theoretic tendencies lead us to ask if we are on the verge of a result regarding the minimality of DPDAs, at least with respect to parsers for CF grammars. Our DPDA-parsers are based on

CFSMs which are reduced, and therefore minimal. Is the DPDA which we get by starting with a minimal FSM in some meaningful sense a minimal version of any other DPDA which affects the same parsings? We know of no existing results in this area.

APPENDIX

WEAK PRECEDENCE GRAMMARS

A CF grammar is called ϵ-free if and only if it has no productions with empty right parts.

An ϵ-free CF grammar is called weak precedence (14M 69) if and only if (1) no two productions have identical right parts, (2) at most one of the following relations hold between any two of the symbols in V:

(a) $X_1 < X_2$ if $A \to \sigma_1 X_1 X_2 \sigma_2$ is a production, or $A \to \sigma_1 X_1 A_2 \sigma_2$
is a production and $A_2 \to^* X_2 \sigma_3$

(b) $X_1 > X_2$ if $A \to \sigma_1 A_1 X_2 \sigma_2$ (or $A \to \sigma_1 A_1 A_2 \sigma_2$) is a production
and $A_1 \to^* \sigma_3 X_1$ (and $A_2 \to^* X_2 \sigma_4$),

and (3) neither of these relations hold between X_1 and A_2 if there exist productions $A_1 \to \sigma_1 X_1 X_2 \sigma_2$ and $A_2 \to X_2 \sigma_2$.

The sequence of theorems below proves that any weak precedence grammar is SLR(1). The inverse is not true: grammar G_0 (page 29) is LR(0) (and therefore SLR(1)), as was shown in Chapter 3, but it is not weak precedence since productions 3 anu   identical right parts.

Lemma A. 1. Let G be a CF grammar and N be a state of G's CFSM having a $\#_p$-transition, whose production p is $A \to \omega$.

Then any string φ which accesses N must end in

ω; i.e., $\varphi = \rho\omega$ for some $\rho \in V^*$.

Proof: The CFSM accepts the characteristic string

$\varphi\#_p$, therefore there exists a canonical form $\varphi\beta = \rho\omega\beta$

which reduces to $\rho A\beta$, by definition of characteristic

strings.   Q. E. D.

Theorem A. 2. When the CFSM of a weak precedence

grammar G_{wp} enters an inadequate state, the last

symbol of the left context is implicitly known.

Proof: The fact that G_{wp} is ϵ-free in conjunction with

Lemma A. 1 proves this.   Q. E. D.

Lemma A. 3. Let G be a CF grammar with characteristic

string $\rho\sigma_1 X_1 X_2 \gamma\#_p$. Then $X_1 < X_2$.

Proof: By definition of characteristic strings

$\rho\sigma_1 X_1 X_2 \gamma\beta$ is a canonical form, for some β in V_T^*. Thus,

either

$$S \to^* \rho A\beta \to \rho\sigma_1 X_1 X_2 \sigma_2 \beta \to^* \rho\sigma_1 X_1 X_2 \gamma\beta$$

$$\text{where } \sigma_2 \to^* \gamma, \text{ or}$$

$$S \to^* \rho A\beta \to \rho\sigma_1 X_1 A_2 \sigma_2 \beta \to^* \rho\sigma_1 X_1 X_2 \sigma_3 \sigma_2' \beta = \rho\sigma_1 X_1 X_2 \gamma\beta$$

$$\text{where } A_2 \to^* X_2 \sigma_3, \; \sigma_2 \to^* \sigma_2' \in V_T^*, \text{ and } \gamma = \sigma_3 \sigma_2'.$$

These are the only two possibilities and each implies that

$$X_1 < X_2.$$

Theorem A. 4. The CFSM of a weak precedence grammar

has no multiply inadequate states.

Proof: Lemma A. 1 implies that any φ which accesses a

state N with transitions under distinct $\#_p$ and $\#_q$ must

end in both ω_p and ω_q, where productions p and q are

$A_p \to \omega_p$ and $A_q \to \omega_q$, respectively. But we cannot have

$\omega_p = \omega_q$ for distinct p and q, because that would violate

condition (1) in the definition of weak precedence.

Furthermore, if $|\omega_p| > |\omega_q|$ then we have $\omega_p = \sigma_1 X_1 X_2 \sigma_2$

and $\omega_q = X_2 \sigma_2$. But this implies that $\varphi\#_q = \rho\sigma_1 X_1 X_2 \sigma_2 \#_q$

is a characteristic string (by Lemma A. 1), and therefore,

that $\rho\sigma_1 X_1 A_2 \beta$ is a canonical form, for some β in V_T^*,

whose characteristic string is $\rho\sigma_1 X_1 A_2 \theta\#_r$ for some

θ in V_T^* and some production r. Thus, Lemma A. 2 implies

that $X_1 < A_2$. But that violates condition (3) of the definition,

so no such state N can exist.   Q. E. D.

Theorem A. 5. Let G_{wp} be a weak precedence grammar.

Then G_{wp} is SLR(1).

Proof: Because of Theorem A. 4 and the SLR(k) definition (page 73) we need only prove, for any inadequate state N of G_{wp}'s CFSM with (among others) transitions under some terminal t and $\#_p$ for some production p which is $A \to \omega$, that t is not in the set $F_T^1(A)$. Consider the following.

$$F_T^1(A) = \{(1;\beta) \in V_T^* \mid S \to \rho A \beta\}$$

$$= \{X_2 \in V_T \mid S \to^* \rho A \beta \to \rho \sigma_1 A_1 X_2 \sigma_2 \beta \text{ or}$$

$$\rho A \beta \to \rho \sigma_1 A_1 A_2 \sigma_2 \beta \text{ and } A_2 \to^* X_2 \sigma_3 \text{ or}$$

$$\rho A \beta \to \rho \sigma_1 A_1 X_2 \sigma_2 \beta \text{ and } A_1 \to^* \sigma_3 X_1 \text{ or}$$

$$\rho A \beta \to \rho \sigma_1 A_1 A_2 \sigma_2 \beta \text{ and } A_1 \to^* \sigma_3 X_1$$

$$\text{and } A_2 \to^* X_2 \sigma_4 \}$$

Thus, the relation $>$ holds between the last symbol of the left context implicit when the CFSM s in N (i. e., ω:1 by Lemma A. 1 and Theorem A. 2) and every symbol in $F_T^1(A)$. But from Lemma A. 1 all of the characteristic strings which correspond to the t-transition are of the form $\varphi t \theta \#_q = \rho \omega t \theta \#_q$, so Lemma A. 3 implies that $(\omega:1) < t$. Since condition (2) of the definition of weak precedence states that both the relations $<$ and $>$ cannot hold between $(\omega:1)$ and t, we see that t is not in $F_T^1(A)$.

Q. E. D.

REFERENCES

Che 67 Cheatham, T. E., Jr. The Theory and Construction of
of Compilers. Massachusetts Computer Associates, Inc.,
Wakefield, Mass., 1967.

Chr 67 Christensen, C. An example of the manipulation of
graphs using the AMBIT/G programming language.
Proc. Symp. Interative Systems in Experimental
Appl. Math., Washington, D.C., 1967.

D&D 69 Dennis, J.B., and Denning, P.J., Machines, Languages
and Computation, Dept. of Electrical Engineering,
Mass. Inst. of Tech., Cambridge, Mass., 1969.

DeR 68 DeRemer, F. L. On the automatic generation of
a compiler for an extensible language. Report no.
KC-T-055, NASA Electronics Research Center,
Cambridge, Mass., Aug., 1968.

Eva 65 Evans, A., Jr. Syntax analysis by a production
language. Ph.D. thesis Carnegie Inst. of Tech.,
Pittsburgh, Pa., 1965.

Eva 68 Evans, A., Jr. PAL - a language designed for teaching
programming linguistics, Proc. 23rd Nat. Conf. ACM,
1968, pp. 395-403.

Eva 69 Evans, A., Jr. PAL - Pedagogic Algorithmic Language -
A Reference Manual and Primer, Dept. of Elec. Eng.,
Mass. Inst. of Tech., Cambridge, Mass., June 1969.

Fel 64 Feldman, J.A. A formal semantics for computer oriented
languages. Ph.D. thesis, Carenegie, Inst. of Tech.,
Pitsburgh, Pa., 1964.

F&G 68 Feldman, J.A., and Gries, D. Translator writing systems.
CACM 11 (February 1968) pp. 77-113.

Flo 64 Floyd, R. W. Bounded context syntactic analysis. CACM 7
(February 1964) pp. 62-67.

Gin 66   Ginsburg, S.   <u>The Mathematical Theory of Context-free</u> <u>Languages</u>, McGraw-Hill, Inc., New York, 1966.

Han 68   Hennie, F. C. <u>Finite-State Models for Logical Machines</u>, John Wiley and Sons, Inc., New York, 1968.

H&U 69   Hopcroft, J. E., and Ullman, J. D.   <u>Formal Languages</u> <u>and their Relation to Automata</u>, Addison-Wesley, Inc., Reading, Mass., 1969.

I&M 69   Ichbiah, J. D., and Morse, S. P.   Optimal generation of Floyd-Evans productions for precedence grammars. Report No. DR.S. 69. 65. ND., Compagnie Internationale pour L'informatique, Les Clayes-Sous-Bois, France, April 1969.

Joh 68   Johnson, W., et al.   Automatic generation of efficient lexical processors using finite-state techniques. CACM 11 (December 1968) pp. 805-813.

Jor 69   Jorrand, P., and Hammer, M.   The formal definition of BASEL, parts I (Introduction), II (Compiler), and III Interpreter), report nos. CA-690-1511, 1512, 1513. Massachusetts Computer Associates, Inc., Wakefield, Mass., August 1969.

Knu 65   Knuth, D. E.   On the translation of languages from left to right. Inf. Contr. 8 (October 1965) pp. 607-639.

Knu 66   Knuth, D. E.   Semantics of context-free languages.   Math. Sys. Th. 2 (February 1966), pp. 127-145.

Kor 69   Korenjak, A.   Efficient LR(1) processor construction. Conf. Rec. ACM Symp.Th. of Computing, Marina Del Rey, Calif., May 1969, pp. 191-200.

Lan 66   Landin, P. J.   The Next 700 Programming Languages. CACM 9 (March 1966) pp. 157-166.

Lyn 68   Lynch, W. C.   A high-speed parsing algorithm for ICOR grammars. Andrew R. Jennings Comp. Ctr. report no. 1097, Case West. Res. Univ., Spring 1968.

L&P 68   Lynch, W.C., and Pierson, H.L. A finite-state transducer model for compiler lexical scanners. Andrew R. Jennings Comp. Ctr., report no. 1098, Case West. Res. Univ., Spring 1968.

L&S   Lewis, P.M., and Stearns, R.E. Syntax-directed transduction. JACM 15 (July 1968) pp. 465-493.

McC 65   McCarthy, J., et al. LISP 1.5 Programmar's Manual, M.I.T. Press, Mass. Inst. of Tech., Cambirdge, Mass., 1965.

McC 66   McCarthy, J. A formal description of a subset of ALGOL, Formal Language Description Languages, (Ed Steele, T.B.) North Holland Pub. Co., Amsterdam, Holland, 1966.

Nau 63   Naur, P. (Ed.) Revised report on the algorithmic language ALGOL 60. CACM 6 (January 1963) pp. 1-17.

Pro 59   Prosser, R.T. Applications of Boolean matrices to the analysis of flow diagrams. Proc. Easter Joint Comp. Conf., No. 16 (1969)p. 133.

Tho 69   Thomas, R.H. A conceptual basis for language extension. Sc.D. thesis proposal, Mass. Inst. of Tech., Cambridge, Mass., September 1969.

War 62   Warshall, S. A theorem on Boolean matrices, JACM 9 (January 1962), pp. 11-12.

W&E 69   Wozencraft, J.M., and Evans, A., Jr. Notes on Programming Linguistics, Dept. of Elec. Eng., Mass. Inst. of Tech., Cambridge, Mass., July 1969.

W&W 66   Wirth, and Weber, H. EULER - a generalization of ALGOL, and its formal definition: parts I, II. CACM 9 (Jan., Feb., 1966) pp. 13-25, 89-99.

BIOGRAPHICAL NOTE

Franklin Lewis DeRemer was born on December 3, 1942, in Baltimore, Maryland. Shortly thereafter his parents moved to Dallas, Texas and took the unknowing tike along with them. It was in Dallas that he grew up, attended public schools, and graduated from South Oak Cliff High School in 1961.

Ultimately, this <u>Texan</u> received a Eugene McDermott Scholarship to attend M. I. T. as an undergraduate. At M. I. T. he majored in Electrical Science and Engineering, and he was elected to membership in Eta Kappa Nu, Tau Beta Pi, and Sigma Xi. In his sophomore year he jointed the "cooperative program" in Electrical Engineering and went to work for the RCA Laboratories in Princeton, New Jersey. He received his B. S. in 1965 and, after writing his master's thesis at RCA, received his M. S. in 1966.

As a graduate student at M. I. T., Mr. DeRemer was a teaching assistant for six semesters and a research assistant for three.

On September 7, 1968, he married the former Miss Sherry Lynn Gulmon, an M. I. T. graduate of the Department of Biology.

Mr. DeRemer is the author of a paper entitled "Generating Parsers from BNF Grammars" which he presented at the 1969 Spring Joint Conference Computer Conference. He is also the author of several internal publications at the RCA Laboratories, the Department of Computer Science at the University of Illinois, and NASA's Electronics Research Center in Cambridge, Massachusetts.

Currently, he is an assistant professor of Information and Computer Science and the assistant director of the computation center at the University of California at Santa Cruz.