LR(k) Sparse-Parsers and their Optimisation

by

J.M. Rushby

Department of Computer Science
University of Newcastle upon Tyne

Ph.D Thesis
September 1977

Acknowledgements

It is a pleasure to record my appreciation of the assistance and encouragement which I have received from my Parents, friends, and colleagues during the preparation of this thesis.

I am especially grateful to Dr. Tom Anderson and Dr. James Eve for their interest in this work, their advice, and their scrupulous and valuable criticism of preliminary drafts. I am also pleased to acknowledge the guidance of my supervisor, Mr. L.B. Wilson.

I am grateful to Mrs. L.C. Woodcock for her speedy and cheerful typing of a difficult manuscript and to Gill Nyfield for typing an earlier draft.

Financial support for the research described here was received from the Science Research Council.

ABSTRACT

A method of syntactic analysis is developed which is believed to surpass all known competitors in all major respects.

The method is based upon that associated with the LR(k) grammars but is faster because it bypasses all reduction steps concerned with 'chain' productions. These are freely selected productions which are considered semantically irrelevant and whose right parts consist of just a single symbol. The parses produced by the method are 'sparse' in that they contain no references to chain productions - they are termed 'chain-free' parses.

The CFLR(k) grammars are introduced as the largest class which can be Chain-Free parsed from Left to Right while looking k symbols ahead of the current point of the parse. The properties of these grammars are examined in detail and their relationship to the conventional LR(k) grammars is explored. Techniques are presented for testing grammars for the CFLR(k) property and for constructing chain-free parsers for those grammars possessing the property. Methods are also presented for converting ordinary LR(k) parsers into chain-free parsers.

CFLR(k) parsers are more widely applicable than their LR(k) counterparts, are faster and provide the same excellent detection of syntactic errors. Unfortunately they also tend to be rather larger. A simple optimization is presented which completely overcomes this single disadvantage without sacrificing any of the advantages of the method.

These theoretical techniques are adapted to provide truly practical chain-free parsers based on the conventional SLR and LALR parsing methods. Detailed consideration is given to use of 'default reductions' and related techniques for achieving compact representations of these parsers. The resulting chain-free parsers are not only faster than their ordinary counterparts, but probably smaller too. We believe their advantages are such that they should substantially replace other parsing methods currently used in programming language compilers.

Contents

CHAPTER 1.

INTRODUCTION

At the heart of every modern compiler there lies a parser. The performance and quality of the parser strongly influence those of the compiler as a whole. Therefore the parser needs to be good; in particular it should be small, fast, and able to detect syntactic errors as soon as possible. Nowadays, parsers are not constructed by ad-hoc manual methods, but are produced by (automated) parser construction algorithms. In the attempt to produce good parsers, these algorithms usually sacrifice generality and restrict the class of grammars to which they may be applied. When evaluating such algorithms it is necessary to consider the extent of their applicability as well as the quality of the parsers which they produce. Also important are the time and space consumed by the algorithm and by any ancillary algorithms which may be used to test whether a given grammar is acceptable to the main parser construction algorithm.

Prominent among parser construction algorithms is that associated with the LR(k) grammars of Knuth (1965). (The LR(k) grammars are those which can be parsed from Left to Right while looking k symbols ahead, where k is a natural number which parameterises the method). This method is of great theoretical interest because of its elegance and power but founders in practice because its parsers are too large. However, modifications of the

LR(k) method have been developed which mitigate this problem while retaining most of the advantages of the basic technique. The most important of these are the SLR and LALR methods of DeRemer (1969,1971) and Anderson (1972). They are among the best methods currently available for producing parsers for programming languages; while other methods can produce parsers of comparable speed and size, few can match their error detection or their generality, and no other method competes with their excellence on all four of these counts simultaneously. (See, for example, the theoretical and empirical comparisons by Anderson (1972) and the empirical study by Lalonde (1971).)

Not all the steps of a parse are significant to the process of translation; parsers would go faster if they could ignore parse steps associated with productions lacking such 'semantic' significance. Much attention has been focused on the problem of modifying LR(k)-type parsers so that they do just that in the important special case where the productions to be ignored are of the form $A \rightarrow X$ where X is a single symbol. Productions of this type are called 'single' or 'unit' or, as we shall prefer, 'chain' productions.

Among those who have proposed techniques for eliminating chain productions from LR(k)-type parsers are Anderson (1972), (see also Anderson et al, (1973)) Aho and Ullman (1973b), Pager (1974), Demers (1975), Backhouse (1976), Lalonde (1976) and Soisalon-Soininen (1977). The methods of these authors have limited aesthetic or theoretical appeal and suffer from a variety of difficulties in

practice. Their deficiencies are discussed in detail in Chapter 7. These methods are not without merit or utility however: measurements by Anderson (1972) and Joliat (1973) have shown that bypassing chain productions can double the speed of an SLR parser for a conventional programming language and increase the speed of its compiler as a whole by about 15%.

This thesis continues the investigation of the problem of eliminating chain productions from LR(k)-type parsers but unlike previous authors we do not take the basic LR(k) parsers as given, nor do we seek, at least initially, to modify them directly. Instead, we look at the problem afresh and consider the issue of producing parses from which all chain productions have been eliminated as an independent topic in its own right. These parses are a special case of the 'sparse parses' of Gray and Harrison (1972); we call them 'chain free' parses. By analogy with the LR(k) grammars, we introduce the CFLR(k) grammars as the largest class of grammars which can be Chain Free parsed from Left to Right while looking k symbols ahead. Techniques are presented for testing grammars for the CFLR(k) property and for constructing chain free parsers for those grammars possessing the property. The relationship between the LR(k) and CFLR(k) grammars is explored and methods are derived for converting LR(k) parsers into chain free parsers. An optimisation is introduced which substantially reduces the number of states in a CFLR(k) chain free parser. The effectiveness of this optimisation is such that our chain free parsers are not only much faster than their LR(k) counterparts, but usually smaller too.

These CFLR(k) techniques are then subjected to modifications in the spirit of the SLR and LALR methods, thereby producing techniques which we term the CFSLR and CFLALR methods. It is shown that standard methods for reducing the space required to represent SLR and LALR parsers can, with a little care, be applied successfully to CFSLR and CFLALR chain free parsers. In this way chain free parsers can be produced which are suitable for practical exploitation.

We claim several advantages for our material over previous work in this field. Our methods have the virtue of complete generality and are founded upon a sound and, we submit, elegant theoretical basis which others lack. At the same time they retain all the benefits of earlier methods and shirk none of the difficulties that may arise in practice.

In summary, while LR(k)-type methods may fairly be said to be among the very best for producing conventional parsers for programming languages, we believe that our CFLR(k) techniques offer worthwhile improvements and should substantially replace them and other methods used in current practice.

Briefly, the structure of this thesis is as follows. The rest of this first Chapter is concerned with basic definitions and an exposition of the ordinary bottom-up parsing strategy. This is followed, in Chapter 2, by a detailed account of the standard LR(k) theory. None of this material is original; it is included because no existing work structures the material in the manner we require to support our subsequent developments. Chapter 3

introduces the notion of chain free parsing and defines the CFLR(k) property. Theorems are presented which relate this property to the conventional LR(k) property. Methods are given for testing for the CFLR(k) property and for constructing chain free parsers for grammars possessing this property. The performance of these chain free parsers is examined theoretically. Chapter 4 is concerned with the problem of converting ordinary LR(k) parsers into chain free parsers. It is shown that the conversion process may generate chain free parsers which are different (and inferior) to those produced by the method of Chapter 3. In Chapter 5 the presence of redundancy within CFLR(k) chain free parsers is revealed and an optimisation is presented which exploits this redundancy in order to reduce the size of CFLR(k) chain free parsers. Chapter 6 extends our CFLR(k) techniques to the SLR and LALR methods and explores some issues of practical concern. Our conclusions and comparison with previous work are given in Chapter 7. Each chapter, except this and the final one, ends with a summary. The reader may find it helpful to examine these summaries, together with Chapter 7, before reading the thesis as a whole.

Finally, a word of encouragement to the reader: although this thesis contains a substantial amount of rather severe formalism, much of its length is due to the presence of examples and informal explanations which are intended to sweeten the bitter pill of an unrelieved technical development.

1.1. Sets, Relations, Functions and Sequences.

We assume familiarity with the conventional terminology and notation of elementary set theory, but in order to preclude misunderstanding we briefly review the form in which the notation will be employed here.

Braces ($\{$ and $\}$) are used exclusively for sets. Set membership is indicated by the symbol $\in$ and the empty set is denoted by $\emptyset$. The cardinality of a set A is written as $|A|$ while the powerset of A is written 2^A. When A and B are sets we denote their union, intersection, set difference, and cartesian product by $A \cup B$, $A \cap B$, $A \setminus B$ and $A \times B$ respectively. Set inclusion (of A within B) is written as $A \subseteq B$ or, when the inclusion is strict, as $A \subset B$.

Any subset of $A \times B$ is called a relation between A and B; when θ is such a relation we usually prefer to write $a\theta b$ instead of $(a,b) \in \theta$. The inverse of θ is denoted by θ^{-1} and is defined as the relation between B and A given by

$$\theta^{-1} = \{(b,a) \mid (a,b) \in \theta\} .$$

When $\theta \subseteq A \times B$ and $\psi \subseteq B \times C$ are relations, their composition is written $\theta\psi$ and is defined to be the relation between A and C given by

$$\theta\psi = \{(a,c) \mid a\theta b \text{ and } b\psi c \text{ for some } b \in B\}.$$

When $\theta \subseteq A \times A$ we say that θ is a relation on A and the operations of composition and union are used to define further relations on A as follows :

(a) $\theta^0 = \{(a,a) \mid a \in A\}$,

(b) $\theta^{n+1} = \theta^n\theta$ for each natural number n,

(c) $\theta^+ = \bigcup_{n>0} \theta^n$, and

(d) $\theta^* = \bigcup_{n\geq 0} \theta^n$.

The relation θ^+ is called the transitive closure of θ while θ^* is called the reflexive transitive closure of θ. Note that θ^0 is equal to the identity relation on A while θ^1 is equal to θ itself.

We assume familiarity with certain elementary properties of relations and in particular with the way in which an equivalence relation imposes a partition on its domain. We also make use of directed graphs as a means of representing relations.

Functions are considered as single-valued relations; mappings are functions which are total. When f is a function from A to B we express the fact by writing $f:A \to B$. When f is a partial function and $a \in A$ is not in the domain of f, it will usually be convenient to suppose that $f(a)$ has the special value ϕ (read as 'undefined').

We also need some notation for sequences, which are defined as ordered lists of objects taken from a set. We write sequences in angle brackets thus: $\langle a_1, a_2, \ldots, a_m \rangle$. The same sequence may also be written more concisely as $\langle a_i \rangle_{i=1}^{m}$. We sometimes need sequences in which the subscripts are arranged in descending order: we abbreviate the sequence $\langle a_m, a_{m-1}, \ldots, a_1 \rangle$ by writing $\langle a_i \rangle_{i=m}^{1}$. Concatenation of sequences is indicated by the operator $\circledast$. Thus $\langle a_1, a_2, a_3 \rangle \circledast \langle a_4, a_5 \rangle$ denotes the sequence $\langle a_1, a_2, a_3, a_4, a_5 \rangle$.

The *length* of a sequence is simply the number of objects it contains; the sequence of zero length is called the *null* sequence.

1.2. Alphabets, Strings and Languages.

An alphabet is a finite, non-empty set of objects called symbols. A string over an alphabet A is a finite list of zero or more symbols taken from A (written without any intervening punctuation marks) where each symbol is permitted to occur many times. The string consisting of zero symbols is called the empty string and is always denoted by Λ . The length of a string α is denoted by len (α) and is defined as the number of symbols in α , where each symbol is counted as many times as it occurs. For example, Λ , a, ab, and aba are all strings over the alphabet $\{a,b\}$ and we have len(Λ) = 0, len (a) = 1, len (ab) = 2 and len (aba) = 3.

When α and β are strings, their concatenation, written as $\alpha\beta$, is the string composed of the symbols of α followed by the symbols of β . For example, if α = aba and β = ab then $\alpha\beta$ = abaab.We say that a string α is a substring of another string β if there exist two further strings γ and δ such that $\beta = \gamma\alpha\delta$. If $\gamma = \Lambda$ then α is called a prefix of β , while if $\delta = \Lambda$ we say that α is a suffix of β . When α is a string and n is a natural number we use the following notation to permit the convenient naming of certain frequently used substrings of α :

(i) $n{:}\alpha$ denotes that prefix of α with length $\min(n, \text{len}(\alpha))$,

(ii) $\alpha{:}n$ denotes that suffix of α with length $\min(n, \text{len}(\alpha))$, and

(iii) n/α denotes that suffix of α with length $\max(0, \text{len}.(\alpha)-n)$.

Informally, $n{:}\alpha$ and $\alpha{:}n$ respectively denote the first n symbols and the last n symbols of α while n/α is the string that remains when the first n symbols of α are deleted.

The set of all strings over an alphabet A is denoted by A^*. Subsets of A^* are called languages over A. Languages which do not have Λ as a member are said to be Λ - free. The Λ - free language A^+ is defined by :

$$A^+ = A^* \setminus \{\Lambda\}$$

When n is a natural number, two frequently used languages over A are defined by :

(i) $A^n = \{ \alpha \in A^* \mid \text{len}(\alpha) = n \}$ and

(ii) $A^{*n} = \{ \alpha \in A^* \mid \text{len}(\alpha) \leq n \}$.

That is, A^n contains all strings over A which have length n, while A^{*n} contains all strings with length at most n. Care is sometimes needed in order to distinguish languages such as A^0 which consist of just the empty string from the empty language $\emptyset$.

Since languages are sets, the set operations of union, intersection and so on may be applied to languages. The operation of concatenation can be applied to languages as well as to strings: if L_1 and L_2 are languages then their concatenation, denoted by $L_1 L_2$, is the language defined by $L_1L_2 = \{\alpha\beta \mid \alpha \in L_1 , \beta \in L_2 \}$.

The positive closure L^+ and the simple closure L^* of a language L are defined as follows :

(i) $L^0 = \{\Lambda\}$,

(ii) $L^{n+1} = L^n L$ for each natural number n,

(iii) $L^+ = \bigcup_{n>0} L^n$, and

(iv) $L^* = \bigcup_{n\geq 0} L^n$

That is, L^+ is the language composed of the concatenation of arbitrary members of L , while $L^* = L^+ \cup \{\Lambda\}$. Note that the alphabet A and the language A^1 denote the same set. Thus each alphabet is also a language over itself. Our definitions ensure that the interpretations of A^*, A^+ and A^n are consistent, independently of whether A is regarded as an alphabet or as a language. For this reason it is unnecessary to distinguish between the alphabet A and the language A^1. Similarly, we do not usually distinguish between the symbol a and the alphabet $\{a\}$. Thus we may speak, for example, of the language a^*b^+ - this is understood to denote the language $\{a\}^*\{b\}^+$.

We sometimes need to refer to the language formed by taking all prefixes of length n from the strings of some other language. We provide for this by extending our existing notation as follows : if L is a language and n is a natural number, then $n{:}L$ is the language defined by

$$n{:}L = \{n{:}\alpha \mid \alpha \in L\}.$$

1.3. Grammars.

Since languages may be infinite, we are interested in finite techniques for specifying them. For our purposes the notion of a grammar, and in particular of a context free grammar is especially important in this regard. A context free grammar is a 4 - tuple : $G = (V_N, V_T, P, S)$ where V_N and V_T are disjoint alphabets and S is a distinguished member of V_N. Symbols in V_N and V_T are called nonterminals and terminals respectively while S is called the goal symbol. The union $V_N \cup V_T$ is called the vocabulary of G and is conventionally denoted by V. P is a finite relation between V_N and V^* and members of P are called the productions of G. When (A,θ) is a production, we call A its left part and θ its right part. The degree of a production q is denoted deg (q) and is defined as the length of the right part of q. Productions of degree zero are called Λ-rules; grammars containing no Λ-rules are said to be Λ-free.

In future, the simple term grammar should always be understood to mean a context free grammar and if we say only that G is a grammar, without specifying it further, then it is to be understood that G has the form $G = (V_N, V_T, P, S)$ and that V will be used to denote $V_N \cup V_T$. Furthermore, in order to avoid excessive qualification we adopt a strict convention regarding the naming of strings and symbols related to such a grammar. Our convention is the following :

(i) $A,B,C,\ldots$ denote members of V_N,

(ii) $a,b,c,\ldots$ denote members of V_T,

(iii) $Z,Y,X,\ldots$ denote members of V,

(iv) $\alpha,\beta,\gamma,\ldots$ denote members of V^*, and

(v) $z,y,x,\ldots$ denote members of V_T^*.

Also, p and q will usually denote members of P while k,m and n will denote natural numbers. These conventions should always be assumed to hold except where it is explicitly stated otherwise.

Before describing how a grammar is used to define a language we need one more definition. When G is a grammar and $\alpha, \beta \in V^*$, we say that α directly derives β (with respect to G) and write $\alpha \xrightarrow{G} \beta$ if and only if there exists a production $(A,\theta) \in P$ and a pair of strings $\gamma, \delta \in V^*$ such that

$$\alpha = \gamma A \delta \quad \text{and} \quad \beta = \gamma \theta \delta.$$

We write $\rightarrow$ rather than $\xrightarrow{G}$ when the identity of G is clear. The interpretation of $\alpha \rightarrow \beta$ is that β can be constructed from α by replacing in α an occurrence of the left part of some production by its corresponding right part. Clearly $\rightarrow$ is a relation on V^* and we note in passing that $P \subseteq \rightarrow$, so that a production (A,θ) may also be written as $A \rightarrow \theta$. This latter form will be preferred in future. The closures $\rightarrow^+$ and $\rightarrow^*$ of $\rightarrow$ are pronounced " strictly derives" and "derives" respectively.

We are now able to define L(G), the language generated by the grammar G as : $L(G) = \{ x \in V_T^* \mid S \rightarrow^* x \}$. Members of L (G) are called the sentences of G. Languages which can be generated by context free grammars are called context free languages. Not all languages are context free.

When G is a grammar, it is usually convenient to require that no members of V, nor of P, are redundant for the purpose of generating sentences. Grammars which satisfy this requirement are said to be reduced. Formally, a grammar is reduced if and only if for each $X \in V$ there exist strings $\alpha, \beta \in V^*$ and $x \in V_T^*$ such that $S \rightarrow^* \alpha X \beta$ and $X \rightarrow^* x$. There is a straightforward algorithm (see Hopcroft and Ullman (1969) Theorems 4.2 and 4.3) to determine whether a given grammar is reduced or not. Furthermore, a grammar which is not reduced can be easily modified so that it becomes so, without changing the language which it generates (provided the language is not empty).

When we wish to specify a particular grammar for the purpose of illustration, we will do so by listing just its set of productions. The nonterminal and terminal vocabularies of a grammar specified in this way are implicit in the list of productions. By convention, the goal symbol of the grammar is assumed to be the left part of the first production appearing in the list. We abbreviate sets of productions which share the same left part by use of the metasymbol "|" (read as "or"). For example, $A \rightarrow x \mid y \mid z$ is a shorthand for the three productions $A \rightarrow x$, $A \rightarrow y$ and $A \rightarrow z$.

We will use the following grammar for demonstration purposes throughout the rest of this chapter :

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow Aa \mid \Lambda \\
B &\rightarrow Bb \mid b
\end{aligned}
$$

It can be seen that this grammar generates the language $a^{*}b^{+}$.

1.4. Derivations.

When two strings $\alpha, \beta \in V^*$ are related by $\alpha \rightarrow^* \beta$, we can always find (not necessarily uniquely) a sequence $\langle \psi_i \rangle_{i=0}^{n}$ of strings in V^* such that

$$\alpha = \psi_0 \rightarrow \psi_1 \rightarrow \psi_2 \rightarrow \cdots \rightarrow \psi_{n-1} \rightarrow \psi_n = \beta.$$

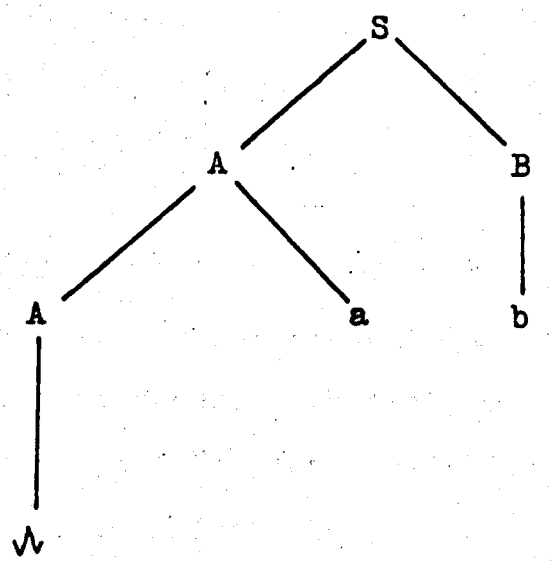
Such a sequence is called a derivation of β from α. If mention of α is omitted, so that we speak of simply "a derivation of β", then a derivation of β from S, the goal symbol of the grammar, is to be understood. In many applications the sentences of a grammar are considered to convey some "meaning" and our interest in derivations is due to the fact that the meaning of a sentence is usually defined as a function of its derivation(s) from the goal symbol. In general a sentence will possess more than one derivation and this can complicate the determination of its meaning. In our example grammar, for instance, the sentence ab has the following three distinct derivations :

(i) $\langle$ S, AB, Ab, Aab, ab $\rangle$,

(ii) $\langle$ S, AB, AaB, Aab, ab $\rangle$, and

(iii) $\langle$ S, AB, AaB, aB, ab $\rangle$.

However, all three of these derivations correspond to the following "parse tree" (a concept which we do not define formally).



In order to avoid difficulty caused by the existence of several derivations which all correspond to the same parse tree (and which are therefore considered to differ from one another only trivally) it is usual to introduce a canonical form for derivations. We shall be concerned with "right-canonical" derivations which, along with two other restricted types of derivation, we now proceed to define.

Recall that when $\alpha, \beta \in V^*$ satisfy $\alpha \rightarrow \beta$ then, by definition, there exist strings $\gamma, \delta \in V^*$ and a production $A \rightarrow \theta$ in P such that $\alpha = \gamma A \delta$ and $\beta = \gamma\theta\delta$. We say that α directly <u>right-canonically</u> derives β and write $\alpha \underset{R}{\rightarrow} \beta$ in the special case that $\delta \in V_T^*$; and we say that α directly <u>empty-free-first</u> derives β and write $\alpha \underset{EFF}{\rightarrow} \beta$ in just the case that $\gamma\theta \neq \Lambda$. Thus a right-canonical derivation step differs from an ordinary one in that it must be the <u>right-most</u> nonterminal in α that is replaced to form β; an empty-free-first derivation step is one which precludes the application of an Λ-rule to the leading symbol of α.

In the case that both of these conditions obtain simultaneously, we say that α directly right- canonically and empty-free-first derives β and we write $\alpha \underset{REFF}{\rightarrow} \beta$. Thus $\underset{REFF}{\rightarrow} = \underset{R}{\rightarrow} \cap \underset{EFF}{\rightarrow}$.

For convenience we use the hyphenated prefix " r- " to stand for "right canonically" or "right canonical" as the context demands. Thus the closures $\underset{R}{\rightarrow}^{+}$ and $\underset{R}{\rightarrow}^{*}$ are pronounced "strictly r-derives" and "r-derives" respectively. Similarly we use the prefixes "eff-" and "reff-" to stand for "empty-free-first" and "right-canonically and empty-free-first".

We illustrate these relations using our example grammar. Given below (Figure 1.1) are four pairs of strings and we indicate the relations which hold between each pair by a tick (relation does hold) or a cross (relation does not hold).

			$\rightarrow$	$\underset{R}{\rightarrow}$	$\underset{EFF}{\rightarrow}$	$\underset{REFF}{\rightarrow}$
(i)	ABA,	BA	✓	×	×	×
(ii)	Ab,	b	✓	✓	×	×
(iii)	ABA,	AbA	✓	×	✓	×
(iv)	ABA,	AB	✓	✓	✓	✓

Figure 1.1. Some Pairs of Strings from the Example Grammar and the Relations which hold between them.

When $\alpha, \beta \in V^{*}$ are related by $\alpha \underset{R}{\rightarrow}^{*} \beta$ there must be some sequence of strings $\langle \psi_i \rangle_{i=0}^{n}$ (again, not necessarily unique) such that

$$\alpha = \psi_0 \underset{R}{\rightarrow} \psi_1 \underset{R}{\rightarrow} \psi_2 \underset{R}{\rightarrow} \cdots \underset{R}{\rightarrow} \psi_{n-1} \underset{R}{\rightarrow} \psi_n = \beta .$$

Such a sequence is called an r-derivation of β from α. It is easy to prove that if $x \in V_T^*$ and $\alpha \in V^*$ satisfy $\alpha \rightarrow^* x$, then they also satisfy $\alpha \rightarrow_R^* x$. Therefore every sentence of G possesses an r-derivation and in general each r-derivation will correspond to several ordinary derivations. For instance, we earlier exhibited three derivations of the string ab with respect to the example grammar. Only one of these derivations, namely $\langle S, AB, Ab, Aab, ab \rangle$ is an r-derivation. We may define eff-derivations and reff-derivations in a similar manner but note that not all sentences of a grammar need possess eff or reff-derivations. (For instance, the sentence ab has no eff-derivation - and therefore no reff-derivation, with respect to our example grammar).

We close this section with the definition of two functions which will be needed subsequently. When G is a grammar, $\alpha \in V^*$ and k is a natural number, we define :

$$FIRST_k^G(\alpha) = \{k:x \mid x \in V_T^* \text{ and } \alpha \rightarrow^* x\}$$

and

$$EFF_k^G(\alpha) = \{k:x \mid x \in V_T^* \text{ and } \alpha \rightarrow_{EFF}^* x\}$$

We will omit the superscript G and/or the subscript k from the names of these functions when their identities are clear. It can be seen that $FIRST_k^G(\alpha)$ contains the prefixes of length k to all terminal strings which can be derived from α, while $EFF_k^G(\alpha)$ captures all those members of $FIRST_k^G(\alpha)$ having derivations which do not involve applying an Λ-rule to the leading symbol of a string.

1.5. Ambiguity.

We have seen that the number of distinct derivations possessed by a sentence may be reduced by considering only those which are r-derivations. Even so, certain grammars possess sentences with more than one r-derivation. Such grammars are usually considered unsuitable for the purpose of specifying the syntax of programming languages (although we will weaken this assertion later) and are said to be ambiguous. A grammar is unambiguous if it is not ambiguous; that is if each of its sentences has exactly one r-derivation. Note that if a grammar is both unambiguous and reduced, then it is not just its sentences which have unique r-derivations; the r-derivation of β from α will be unique for any $\alpha, \beta \in V^*$ such that $\alpha \rightarrow_R^* \beta$. It should also be noted that some (and for our purposes, pathological) languages can be generated by ambiguous grammars but not by unambiguous ones. These languages are called inherently ambiguous and we shall not consider them further.

1.6. Further Notation Concerning Derivations.

When two strings are related by $\alpha \rightarrow \beta$ we often wish to be able to indicate explicitly the production which is involved in the transformation of α into β and also the position at which it is applied. We provide for this as follows. Suppose that $\alpha = \gamma A \delta$ and $\beta = \gamma\theta\delta$ and that $A \rightarrow \theta \in P$. Let the production $A \rightarrow \theta$ be called q and let m = len ($\gamma\theta$). Then we say that " α directly derives β <u>by applying production q at position m</u>" and we write $\alpha \multimap(q,m)\rightarrow \beta$. Similarly, we may write

(i) $\alpha \multimap(q,m)\rightarrow_R \beta$ if $m/\beta \in V_T^*$,

(ii) $\alpha \multimap(q,m)\rightarrow_{EFF} \beta$ if $m > 0$, and

(iii) $\alpha \multimap(q,m)\rightarrow_{REFF} \beta$ if both $m/\beta \in V_T^*$ and $m > 0$.

Using our example grammar we have, for instance,

$$ABA \multimap(A\rightarrow\Lambda,\ 0)\rightarrow BA\ ,$$
$$Ab \multimap(A\rightarrow\Lambda,\ 0)\rightarrow_R b\ ,$$
$$ABA \multimap(B\rightarrow b,\ 2)\rightarrow_{EFF} AbA\ ,\ \text{and}$$
$$ABA \multimap(A\rightarrow\Lambda,\ 2)\rightarrow_{REFF} AB\ .$$

When $q \in P$ and m is a natural number we call the pair (q,m) a <u>derivation step</u> (or more usually, simply a <u>step</u>) in G. It may be seen that for each step (q,m) in G, $\multimap(q,m)\rightarrow$ is a relation on V^* and in fact

$$\rightarrow \ = \bigcup_{\substack{(q,m) \text{ is a} \\ \text{step in G}}} \multimap(q,m)\rightarrow$$

Analogous results hold for the relations $\rightarrow_R$, $\rightarrow_{EFF}$, and $\rightarrow_{REFF}$.

If we have a derivation $Q = \langle \psi_i \rangle_{i=0}^{n}$ in G there must be a sequence of derivation steps $R = \langle (q_i, m_i) \rangle_{i=1}^{n}$ such that

$$\psi_0 \text{—}(q_1, m_1)\rightarrow \psi_1 \text{—}(q_2, m_2)\rightarrow \psi_2 \cdots \psi_{n-1} \text{—}(q_n, m_n)\rightarrow \psi_n ;$$

We say that the sequence R is an explicit derivation of ψ_n from ψ_0 and, in order to distinguish it from R, we will henceforth call Q an implicit derivation. Explicit and implicit r- , eff - and reff-derivations are defined in an exactly similar manner. Note that in general the correspondence between explicit and implicit derivations is many to one. However, in the case of explicit and implicit r-derivations the correspondence is one to one. Furthermore, when $\alpha \text{—}(q,m)\rightarrow_R \beta$, the values of m and β can be deduced uniquely given only the values of q and α . This is because in r-derivations there is only one place at which a production may be applied. It follows that the component steps of an explicit r-derivation $R = \langle (q_i, m_i) \rangle_{i=1}^{n}$ of ψ_n from ψ_0 can be deduced uniquely given only the identity of the string ψ_0 and the sequence of productions $R' = \langle q_i \rangle_{i=1}^{n}$. This sequence R' is called a parse of ψ_n from ψ_0.

We have now seen that the explicit and implicit r-derivations and also the parses in G are in one to one correspondence; they are really just alternative notations for the same concept. None is redundant though; each has its particular uses. When Q is a parse (or one of the equivalent notions) of β from α , we may indicate this fact by writing $\alpha \text{—}[Q]\rightarrow_R \beta$.

To illustrate these ideas we use our example grammar and the sentence aabb . This sentence has the following implicit r-derivation.

$$\langle S,\ AB,\ ABb,\ Abb,\ Aabb,\ Aaabb,\ aabb \rangle,$$

the following explicit r-derivation

$$\langle (S \to AB, 2),\ (B \to Bb, 3),\ (B \to b,\ 2),\ (A \to Aa, 2),\ (A \to Aa, 2),\ (A \to \Lambda,\ 0) \rangle,$$

and the following parse.

$$\langle S \to AB,\ B \to Bb,\ B \to b,\ A \to Aa,\ A \to Aa,\ A \to \Lambda \rangle.$$

Given any one of these sequences we can reconstruct the other two.

1.7. Parsing.

Given a grammar G and a string $x \in V_T^*$, the problem of deciding whether x is a sentence of G is called a recognition problem for G. An algorithm which solves all recognition problems for G is called a recognizer for G. The problem of deciding not only whether x is a sentence of G but also (in the case that it is indeed a sentence) of specifying each of its parses is called a parsing problem for G. An algorithm which solves all parsing problems for G is called a parser for G.

Parsers and methods for constructing them are interesting objects of study in their own right and are of practical concern because of their application in the construction of compilers for programming languages. Most modern compilers are of the "syntax directed" variety. This means that the process of compilation (or more generally, of translation) is largely controlled by the structure (that is to say, the parse) imposed upon the source program by the grammar which defines the syntax of the language concerned. Typically, each production of the grammar has certain "semantic actions" associated with it and the total compilation process is effected by composing these actions in a manner determined by the parse of program being compiled. A parser therefore lies at the heart of every modern compiler and the properties and quality of this parser will strongly influence those of the complete compiler. For this reason, algorithms for constructing parsers have received considerable attention and many such algorithms have been proposed.

From among the many questions which may be asked about a parser construction algorithm we select the following as being particularly important :

(i) To what class of grammars can the algorithm be applied?

(ii) To what class of languages can it be applied?

(iii) How long does it take to test whether a grammar is in the required class?

(iv) How long does it take to generate a parser?

(v) How complicated is the algorithm - is it feasible to construct the parsers by hand?

(vi) How fast are the parsers produced by the algorithm?

(vii) How big are they?

(viii) What quality of error detection do they provide?

Of course the importance attached to each question will depend upon the intended application. We shall be concerned with the "LR(k) parsing algorithm" which is a parsing method applicable to the class of grammars possessing the so-called LR(k) property. The LR(k) parsers perform so excellently in certain respects (notably in their generality, their speed, and the quality of their error detection) that they serve as a yardstick by which other methods may be judged.

Precise discussion of the LR(k) parsers and grammars necessitates the use of considerable formalism. While this provides for exactitude it also tends to obscure the basic ideas and motivations. For this reason we now introduce the fundamental ideas behind the LR(k) parsing algorithm in a different and somewhat more leisurely fashion to that which is usual. It is hoped that this extended discussion will enable the acquisition of sufficient

intuitive insight to support the rather severe formalism of the later chapters.

The LR(k) parsing algorithm is a particular example of a general class of algorithms known collectively as "bottom up" parsing methods. The concept of a "handle" is fundamental to algorithms of this type. A derivation step (q,m) is said to be a <u>handle</u> of a string β whenever there exists a second string α such that $\alpha \overset{(q,m)}{\underset{R}{\longrightarrow}} \beta$ Next we define an <u>r-sentential</u> form (abbreviated <u>rsf</u>) of a grammar to be a member of the set $\{\alpha \in V^* \mid S \overset{*}{\underset{R}{\longrightarrow}} \alpha\}$. The ambiguity of a grammar is related to the uniqueness of the handles of its rsf's as the following theorem shows.

<u>THEOREM 1.1</u>

Let $G = (V_N, V_T, P, S)$ be a reduced grammar in which $S \rightarrow^+ S$ does not occur. Then G is unambiguous if and only if each rsf of G has but a single handle, except S which has none. □

This result may be proved by elementary means and allows us to speak of <u>the</u> handle of each rsf of an unambiguous grammar. We now introduce some additional terminology concerned with handles.

If (q,m) is the handle (which we suppose to be unique) of β, then q is called the <u>handle production</u>, m is called the <u>handle position</u>, and the strings $m{:}\beta$ and m/β are respectively called the left and right <u>contexts</u> of the handle. If production q is $A \rightarrow \Theta$ then β can be written in the form $\beta = \gamma\Theta x$ where $\text{len}(\gamma\Theta)=m$.

This occurrence of Θ within β is called the

handle phrase of β. The act of replacing the handle phrase of a string by the left part of its handle production is known as reducing the string (by its handle). Reducing a string β by its handle (q,m) will yield the unique string α which satisfies $\alpha \dashv(q,m)\rightarrow_R \beta$.

If, with respect to some unambiguous reduced grammar we have :

$$S = \psi_0 \dashv(q_1,m_1)\rightarrow_R \psi_1 \dashv(q_2,m_2)\rightarrow_R \psi_2 \cdots \psi_{r-1}\dashv(q_r,m_r)\rightarrow_R \psi_r = x$$

then (q_r, m_r) is clearly the handle of the sentence x. Knowing this fact we can reduce x and thereby obtain the string ψ_{r-1}. The handle of ψ_{r-1} is (q_{r-1},m_{r-1}) and so reducing ψ_{r-1} in turn yields the string ψ_{r-2}. If we continue in this way we will eventually arrive at the goal symbol S and the sequence of handles found during this process will be $\langle (q_i, m_i) \rangle_{i=r}^{1}$ - which is just the explicit r-derivation of x in reverse order. This is the basic strategy underlying the bottom up parsing method. The following algorithm describes this method more exactly.

ALGORITHM 1.2

Bottom up parsing algorithm

Input : An unambiguous grammar G and a sentence $x \in L(G)$ which is to be parsed.

Output : The explicit r-derivation (in reverse order) of x with respect to G.

Method :

1. Set $\beta = x$.
2. Repeat steps 3 and 4 until $\beta = S$.
3. Determine the handle of β and output it.
4. Reduce β by its handle and let the result replace β. □

Notice that we give no indication of how the handle might actually be determined in step 3 of this algorithm - this is because we are presently concerned only with the overall form of the strategy employed. Notice also that the algorithm is not a true parser since it assumes that its input x is known beforehand to be a sentence of the grammar concerned. These issues will be resolved later. Observe that the algorithm must terminate after executing a finite number of steps since each execution of step 3 outputs a different step of the r-derivation of x, and every r-derivation in an unambiguous grammar is finite. (A sentence can possess an r-derivation of infinite length only if it possesses an infinity of r-derivations.)

If we imagine a parse tree for the input sentence drawn in the conventional manner with the goal symbol at the top, then Algorithm 1.2 essentially enumerates the nodes of this tree form the bottom to the top. This is the origin of the name "bottom up" used to describe

the algorithm.

As it stands Algorithm 1.2 is not well suited to computer implementation. Even if some method were prescribed for finding the handle in step 3, the manipulation required in Step 4 would remain decidedly inconvenient. It is possible, however, to modify the basic bottom up approach so that the reduction of strings may be performed more economically. In this modified form the method is known as the "shift-reduce" bottom up parsing algorithm. The idea behind this new method is to work through the input sentence, one symbol at a time, removing symbols from the input and placing them onto a stack (usually called the "parse stack"). This process continues until the handle phrase lies wholly on top of the stack. At this point the input string must be reduced and this is easily accomplished by first "popping" the handle phrase off the stack and then "pushing" the left part of the handle production onto it. It is an elementary, though to this algorithm crucial,property of r-derivationsthat after this has been done the string formed by the concatenation of the new stack contents and the so far unconsumed input is such that its handle position is either at, or to the right of, the point of concatenation. This means that the entire process can be performed repeatedly until the parse is complete. It can be seen that this method is composed of two primitive operations; at each step we either move a symbol from the input string to the stack (we say that a symbol is "shifted" onto the stack and so this operation is called a <u>shift move</u>) or we replace a handle phrase lying on top of the stack by the left part of the handle production. This latter operation

is called a reduce move. It is from these two types of "move" that the shift-reduce bottom up parsers get their name.

The decision as to which is the appropriate type of move at each step is determined by a parsing action function. If α denotes the contents of the stack and z the unconsumed input at some particular point, then the parsing action function $f(\alpha, z)$ yields the value "SHIFT" if a shift move is correct or the value "REDUCE q" if a reduce move involving production q is required. This last means that deg (q) symbols are to be discarded from the top of the stack and the left part of production q is then to be pushed onto it, We now give a precise description of this algorithm.

ALGORITHM 1.3

Shift - reduce bottom up parsing algorithm.

Input : An unambiguous grammar G and a sentence x e L(G) which is to be parsed.

Output : The parse of x with respect to G (in reverse order).

Method : We use z to represent the unconsumed input and α to represent the parse stack. The top of the stack is assumed to be to the right.

1. (Initialise) Set $\alpha = \Lambda$ and z = x.

2. Repeat step 3 until $\alpha z = S$.

3. Evaluate $f(\alpha, z)$ and execute whichever of sub-steps (a) or (b) is appropriate.

   (a) If $f(\alpha, z)$ = SHIFT then remove the first symbol from z and push it onto the parse stack.

   (b) If $f(\alpha, z)$ = REDUCE q then output q, pop deg (q) symbols off the parse stack and then push the left part of production q onto it. □

The precise behaviour of Algorithm 1.3 clearly depends upon that of its action function and in order to establish the correctness of the algorithm it is therefore necessary to specify this function more completely. Now we have seen from the discussion preceding its introduction that Algorithm 1.3 is intended simply as a more convenient formulation of the basic bottom up parsing method given in Algorithm 1.2. We can indicate this correspondence more exactly as follows. Let β_i denote the contents of the variable β in Algorithm 1.2 immeidately before the i'th execution of step 3; also let α_i and z_i denote the contents of the variables α and

z in Algorithm 1.3 immediately before the i'th execution of sub-step 3 (b) in that algorithm. Then the intention is to maintain β_i equal to the concatenation $\alpha_i z_i$ and this implies that the action function of Algorithm 1.3 must satisfy the following condition. Whenever αz is an rsf of G whose handle (q,m) satisfies $m \geq \text{len}(\alpha)$ we must have :

$f(\alpha, z)$ = REDUCE q if $m = \text{len}(\alpha)$, and

$f(\alpha, z)$ = SHIFT if $m > \text{len}(\alpha)$.

Given that this condition is satisfied, it is easy to see that the correctness and finite termination of Algorithm 1.3 follow directly from those of Algorithm 1.2.

Notice that we are only concerned about the value of the action function in the case $m \geq \text{len}(\alpha)$. This is because Algorithm 1.3 always maintains its parse stack α so that its contents are a prefix to the left context of the handle of αz. We say that the parse stack always contains a "viable prefix" of the grammar. More precisely a string α is a <u>viable prefix</u> of G if and only if there is some string β such that $\alpha\beta$ is an rsf of G with a handle (q,m) satisfying $m \geq \text{len}(\alpha)$. The set of all viable prefixes of G is denoted VP^G. Plainly the domain of the action function in Algorithm 1.3 is contained within the cartesian product $VP^G \times V_T^*$.

If we wish to construct practical parsing methods based upon Algorithm 1.3 then we must propose methods for determining the correct values of the action function. One possible approach is to relinquish the generality of this parsing method and to seek special classes of grammars whose action functions have a particularly simple form. This

might be done by requiring, for example, that the value of $f(\alpha, z)$ be uniquely determined by just the last few symbols of α and the first few symbols of z. Several practical parsing methods are essentially of this type. A generalisation of this idea is to partition the domains of each of the arguments to the action function into a finite number of equivalence classes and to require that all members of each equivalence class yield the same function value. The previously mentioned technique of considering just a few symbols from each of the arguments is merely a particularly simple way of imposing these partitions.

No matter how this partitioning is performed, it effectively reduces the domain of the action function to the cartesian product of the two finite sets of equivalence classes. If we redefine the action function so that it operates on the equivalence classes of strings, rather than on the strings themselves, then the domain of the action function becomes finite and so its values may, in principle, be tabulated. By doing so we can reduce the determination of the correct move at each step of Algorithm 1.3 to a straightforward (and potentially very fast) table look-up operation. Observe that in order to look up the value of the function $f(\alpha, z)$ it is first necessary to determine the equivalence classes to which the arguments α and z belong. When the equivalence classes are constructed on the basis of considering only a few symbols from each argument this determination is, of course, trivial. In fact, virtually all parsing methods of this general type do treat the second argument, that is the unconsumed input, in this simple fashion. Typically

they will consider just its first k symbols, where k is a fixed natural number which parameterises the method. These methods are said to employ a k symbol "lookahead". However, certain methods (and the LR(k) parsers are among them) while employing a k symbol lookahead in order to partition the second argument to the action function, do not partition the first argument (that is the set VP^G) on the same simple basis but are rather more subtle. With these methods it becomes non-trivial to discover the equivalence class to which a given viable prefix belongs.

This difficulty can be circumvented by imposing additional constraints upon the way in which VP^G is partitioned. Let Q be the set of equivalence classes into which VP^G is partitioned (these classes are called states) and let EQUIV : $VP^G \rightarrow Q$ be the function which maps viable prefixes into their corresponding equivalence classes. If α and β are viable prefixes of G such that EQUIV (α) = EQUIV (β) and if $X \in V$ is such that both αX and βX are also viable prefixes, then we shall require that EQUIV (αX) = EQUIV (βX). Not all partitions will satisfy both this constraint and that attendant upon the correct behaviour of the action function; some grammars may fail to possess any satisfactory partitions at all. However, when the condition above is satisfied we may construct a parsing goto function $g : Q \times V \rightarrow Q$ as follows. Whenever $\alpha \in V^*$ and $X \in V$ are such that both α and αX are viable prefixes, we define

$$g(\text{EQUIV}(\alpha), X) = \text{EQUIV}(\alpha X).$$

The condition given above is simply that necessary to ensure that g is truly a function (that is it is single-valued). Note that g is a partial function, when α is a viable prefix but αX is not, the value given to g(EQUIV(α), X) is unimportant. Since its domain is finite, the values of the goto function may be tabulated, just like those of the action function.

The goto function is employed in a modified version of Algorithm 1.3 in the following manner. Another stack, called the state stack is maintained in parallel with the parse stack. Each position in this second stack records the state (that is to say, the equivalence class) to which belongs the string lying below the corresponding position in the parse stack. Whenever symbols are popped off the parse stack the same number of states are discarded from the top of the state stack. When a new symbol, say X, is to be pushed onto the parse stack, the top element of the state stack is first inspected. This state, say s, will be the equivalence class to which the current parse stack contents, say α, belong. (That is to say, s = EQUIV (α).) The goto function is then applied to s and X in order to yield g(s,X) - which is the equivalence class to which the new parse stack contents αX will belong. The symbol X is then pushed onto the parse stack and g(s, X) is pushed onto the state stack, thereby maintaining the required correspondence between the contents of the two stacks.

In this way we arrive at a "table-driven shift-reduce bottom up parsing method using k symbol lookahead" (we will say simply a "table driven parser" in future). The

precise behaviour of a parser of this type is determined by the value of k (that is the amount of lookahead) employed, the way in which the viable prefixes of the grammar are partitioned into states, and the values which are given to parsing action and goto functions. In order to start the parser off we will also need to know the state to which the initial parse stack contents (that is the empty string, Λ) belong. We may collect all this information together and say that it comprises the "tables" which "drive" the parser.

Specifically, we will define a set of <u>parsing tables</u> (using k symbol lookahead) as a 4-tuple $T=(Q,s_0,g,f)$ where :

(i) Q is a finite non-empty set of parsing states (these are the equivalence classes into which VP^G is partitioned),

(ii) s_0 is a distinguished initial state,

(iii) $g : Q \times V \rightarrow Q$ is the parsing goto function, and

(iv) $f : Q \times V_T^{*k} \longrightarrow ACTIONS^G$ is the parsing action function where $ACTIONS^G$, which is the set of all possible parsing actions for the grammar G, is defined by :
$ACTIONS^G = \{ERROR, SHIFT\} \cup \{REDUCE\ q \mid q \in P\}$.

Notice that, in order to provide for error detection, we have now enlarged the range of the action function to include an ERROR action. However, this provision may be insufficient to ensure that all errors are detected (it depends upon the particular tables and value of k employed). For this reason the detailed specification of the table driven parsing method given below includes

additional error detection facilities. Full discussion of the detection of invalid inputs is postponed until later.

ALGORITHM 1.4

Table driven parsing algorithm using k symbol lookahead.

Input : A set $T = (Q, s_0, g, f)$ of parsing tables for G and the string $x \in V_T^*$ which is to be parsed.

Output : If $x \in L(G)$ then the parse of x (in reverse order), otherwise an error indication.

Method :

1. (Initialise) Empty the parse and state stacks and then push s_0 onto the state stack. Set $z = x$.

2. Repeat step 3 until either acceptance or rejection occurs. (In the latter case the algorithm halts with an error indication).

3. Determine the current lookahead string $u = k{:}z$ and set s equal to the state currently on top of the state stack. Look up $f(s, u)$ and execute whichever of sub-steps (a), (b) or (c) is appropriate.

   (a) If $f(s,u)$ = SHIFT do the following :
   (i) if $z = \Lambda$ then reject the input x and halt, otherwise

   (ii) remove the first symbol, say a, from z ;

   (iii) look up $g(s, a)$;

   (iv) If $g(s, a)$ is undefined then reject the input x and halt, otherwise

   (v) push a onto the parse stack and $g(s,a)$ onto the state stack.

(b) If f (s, u) = REDUCE q then output q and do the following :

(i) If the parse stack contains fewer than deg (q) symbols then reject the input x and halt, otherwise

(ii) remove deg (q) symbols from the parse stack and the same number of states from the state stack (note that the state stack is always one longer than the parse stack) ;

(iii) set A equal to the left part of production q;

(iv) if the state stack now contains just the single state s_o and A = S and $z = \Lambda$ then accept the input x and halt, otherwise :

(v) set r equal to the state currently on top of the state stack ;

(vi) look up g(r,A);

(vii) if g (r, A) is undefined then reject the input x and halt, otherwise

(viii) push A onto the parse stack and g (r, A) onto the state stack.

(c) If f(s, u) = ERROR then reject the input x and halt. □

Observe that in order to execute parts (i), (ii) and (iii) of step 3 (b), Algorithm 1.4 requires to know the degree and left part of each production. In practice this information is provided by a pair of additional tables which augment the parsing tables proper.

Notice also that the contents of the parse stack are never consulted by Algorithm 1.4; only the state stack is really necessary. However, for both theoretical and practical reasons it is often useful to retain the parse stack.

On its own, Algorithm 1.4 is not a complete parsing algorithm since its precise behaviour is determined by the particular set of parsing tables used to drive it.For this reason we do not usually consider the properties of Algorithm 1.4 independently of a particular method for constructing its tables. We will examine the conditions necessary to ensure the correctness of the algorithm shortly, but first we will consider an example of a parsing method of this type.

We will use the familiar example grammar which has been used throughout this chapter. For use within parsing tables it is convenient to give each production of the grammar a number. We reproduce the example grammar below with numbers written alongside the productions to which they refer.

1. S → AB
2. A → Aa |
3. Λ
4. B → Bb |
5. b

A set of parsing tables for this grammar using 1 symbol lookahead are shown in Figure 1.2. We will not indicate how these tables might have been constructed; this topic is considered in Chapter 2.

STATES	ACTION FUNCTION			GOTO FUNCTION				
	Λ	a	b	a	b	A	B	S
1.		3	3			2		
2.		sh	sh	3	4		5	
3.		2	2					
4.	5		5					
5.	1		sh		6			
6.	4		4					

Figure 1.2. Parsing Tables for the Example Grammar

In Figure 1.2. the parsing states are represented by integers. By convention, the initial state is always supposed to be state 1. The entries in the action function portion of the table are to be interpreted as follows : sh means SHIFT, a number means REDUCE q where q is the production with that number (e.g. 4 means REDUCE B → Bb) and a blank means ERROR. In the goto function portion of the table a blank means "undefined". Now consider the behaviour of Algorithm 1.4 when driven by the tables of Figure 1.2 and presented with the input string abb. Initially the parse stack will be empty and the state stack will contain just the initial state 1. The first lookahead string to be determined is a (remember we are using 1 symbol lookahead) and so f(1,a) is inspected. This yields the value 3 which means REDUCE $A \rightarrow \Lambda$. Since the degree of this production is zero, no symbols are popped off the stacks and so 1 remains the current state. The left part of the production, that is A, now

needs to be pushed onto the parse stack and so the value of g(1,A) is inspected in order to yield the identity of the corresponding state. We find that g(1,A) = 2 and so 2 is pushed onto the state stack while A is pushed onto the parse stack. The parser then inspects f(2,a) and obtains the value SHIFT. This means that the symbol a is removed from the input string and pushed onto the parse stack while 3 (the value of g(2,a)) is pushed onto the state stack. Continuing in this fashion, the parser will trace out the sequence of moves summarised in Figure 1.3.

SYMBOL STACK CONTENTS	STATE STACK CONTENTS	UNCONSUMED INPUT	ACTION
Λ	1	abb	REDUCE A → Λ
A	1,2	abb	SHIFT
Aa	1,2,3	bb	REDUCE A → Aa
A	1,2	bb	SHIFT
Ab	1,2,4	b	REDUCE B → b
AB	1,2,5	b	SHIFT
ABb	1,2,5,6	Λ	REDUCE B → Bb
AB	1,2,5	Λ	REDUCE S → AB and ACCEPT

Figure 1.3. The behaviour of Algorithm 1.4. while parsing the string abb using the tables of Figure 1.2.

It can be seen from figure 1.3. that the string abb is accepted by the parser and that the sequence of productions output during the reduce moves is $\langle A \rightarrow \Lambda$, A → Aa, B → b, B → Bb, S → AB$\rangle$ which is indeed the correct parse (in reverse order) for the given input.

We will now examine the conditions which the parsing tables $T = (Q, s_0, g, f)$ must satisfy if they are to drive Algorithm 1.4 correctly. We have already mentioned some of these conditions but we collect them all together here for convenience. First recall that the states in Q are intended to be the equivalence classes into which the viable prefixes of G are partitioned. In order that every viable prefix may

belong to some state and that no state be redundant we must require that the function EQUIV: $VP^G \longrightarrow Q$ which takes viable prefixes into their corresponding states be a surjective mapping. In order that the goto function fulfills its role correctly it must satisfy $g(\text{EQUIV}(\alpha), X) = \text{EQUIV}(\alpha X)$ whenever $\alpha \in VP^G$ and $X \in V$ are such that $\alpha X \in VP^G$. And in order to ensure that the goto function is truly a function we must require that whenever $\alpha, \beta \in VP^G$ and $X \in V$ are such that

(i) $\text{EQUIV}(\alpha) = \text{EQUIV}(\beta)$ and

(ii) $\alpha X, \beta X \in VP^G$

then $\text{EQUIV}(\alpha X) = \text{EQUIV}(\beta X)$.

These conditions are all straightforward consequences of the purposes for which the goto function is used. Notice that the function EQUIV which features prominently in the statements of these conditions is not itself included in the parsing table. This is because, in practice, it is never constructed explicitly; it is merely a convenient theoretical device.

Finally we need to examine the conditions which the action function must satisfy. Observe that the basic moves of Algorithm 1.4 are essentially the same as those of Algorithm 1.3 and that the present action function differs from the earlier one only in that its arguments are equivalence classes of strings rather than the strings themselves. Consequently the conditions which ensure the correctness of Algorithm 1.3 immediately yield the following conditions for the action function of Algorithm 1.4. Whenever αz is in rsf of G whose handle (q,m) satisfies

$m \geq len(\alpha)$ we must have :

$f(EQUIV(\alpha),k:z) = REDUCE\ q$ if $m = len(\alpha)$, and
$f(EQUIV(\alpha),k:z) = SHIFT$ if $m > len(\alpha)$.

The conditions above are those which are necessary and sufficient to cause the parsing tables $T = (Q, s_0, g, f)$ to drive Algorithm 1.4 correctly when its input string is a valid sentence of the grammar. But if the algorithm is to be a true parser, it must not only parse sentences correctly, it must also reject all inputs which are not valid sentences. However, we will not provide conditions upon the parsing tables which guarantee this detection of invalid inputs, since conditions of this sort depend crucially upon how early during the processing of invalid inputs we wish rejection to occur. Instead, whenever a method for constructing parsing tables is proposed, we will expect an argument to be provided which demonstrates that all invalid inputs will be rejected.

We have now reached the end of this rather long section and have barely mentioned the supposed subject matter of this thesis - the LR(k) grammars and parsers. These are discussed at length in Chapter 2 but their importance will become apparent when we state that the LR(k) grammars are the largest class of grammars which can be parsed by the table-driven method of Algorithm 1.4 using k symbol lookahead. Furthermore, a set of parsing tables of the type we have described can be mechanically constructed for any LR(k) grammar.

1.8 Classes of Languages and their Recognizers.

In the previous section we indicated that in the interests of efficiency and simplicity we are willing to consider parsing methods which are applicable to only a subset of the context free grammars. It is natural to suppose that such restricted classes of grammars may be capable of generating only a subset of the context free languages. Later on we shall be concerned to characterize these classes of languages and will seek to do so in terms of the classes of "deterministic" and "strict deterministic" languages which we shall define shortly. We shall also need to use the properties of "regular" languages and of "finite automata". Our treatment of these topics will be rather terse. For a full discussion of most of this material see the book by Hopcroft and Ullman (1969).

When A is an alphabet, the regular languages (also called the regular sets or regular events) over A are defined recursively as follows :

(i) $\emptyset$ is a regular language over A,

(ii) $\{\Lambda\}$ is a regular language over A,

(iii) $\{a\}$ is a regular language over A, for all $a \in A$, and

(iv) if P and Q are regular languages over A, then so are
   (a) $P \cup Q$,
   (b) PQ, and
   (c) P^*.

(v) Nothing is a regular language over A unless it is so by virtue of (i) to (iv) above.

It can be shown that the regular languages are precisely those which can be generated by context free grammars when certain restrictions are placed on the form of productions which may be used. Thus the regular languages are a subset of the context free languages and can, in fact, be shown to be a proper subset.

The regular languages may also be characterised as the class of languages which can be recognized by finite automata. Formally, a finite automaton is a aystem $M = (Q, q_0, F, I, \delta)$ where Q is a finite non-empty set of states, $q_0 \in Q$ is a distinguished initial state, $F \subseteq Q$ is a set of final states, I is the input alphabet and $\delta : Q \times I \rightarrow Q$ is the transition function. A finite automaton may be pictured as a control unit equipped with a reading head which can read symbols from a linear input tape in a sequential, left to right manner. The symbols on the input tape are chosen from the alphabet I. At any instant the control unit may assume one of the states of Q; initially it is in state q_0 and the read head is positioned over the left most symbol on the input tape. The interpretation of $\delta(q, a) = p$ for $p, q \in Q$ and $a \in I$ is that M, currently in state q and scanning the symbol a, moves its read head one symbol to the right and goes into state p.

The type of automata defined above are actually known as the deterministic finite automata (DFA for short) A related class of automata are known as the nondeterministic finite automata (NFA for short) and these are distinguished from the deterministic variety by their

ability to assume several states simultaneously. The formal definition of an NFA differs from that of a DFA only in that the value of $\delta(q,a)$ is allowed to be a (possibly empty) set of states rather than just a single state. This means that in an NFA, $\delta : Q \times I \to 2^Q$. We shall in fact be mainly concerned with <u>extended</u> NFA's (ENFA for short) which are NFA's augmented with the additional capability to change states without consuming input. In an ENFA, $\delta : Q \times (I \cup \{\Lambda\}) \to 2^Q$. Since DFA's and NFA's are special cases of ENFA's, we shall now concentrate on ENFA's.

In order to define the behaviour of ENFA's we introduce the notion of an <u>instantaneous description</u> (abbreviated to ID). An ID is a pair (q,α) where q is (one of) the current states of the control unit and $\alpha \in I^*$ is a string of symbols written on the input tape starting in the position currently under the read head and extending to the right. A <u>move</u> of an ENFA is denoted by the relation $\vdash$ on ID's. For $p,q,r \in Q$, $a \in I$ and $\alpha,\beta \in I^*$ we define $(q,a\alpha) \vdash (p,\alpha)$ if $p \in \delta(q,a)$, and $(q,\beta) \vdash (r,\beta)$ if $r \in \delta(q,\Lambda)$. We are principally interested in the reflexive transitive closure $\vdash^*$ of $\vdash$. The interpretation of $(q,\alpha) \vdash^* (p,\Lambda)$ is that, starting from state q with input α, the ENFA will be in state p (among possibly several others) after reading all the symbols of α. We use $\vdash^*$ to extend the domain of δ to $Q \times I^*$ by the definition :

$$\delta(q, \alpha) = \{p \in Q \mid (q, \alpha) \vdash^* (p, \Lambda)\}.$$

Thus $\delta(q, \alpha)$ is the set of all states which the ENFA can reach by starting from state q and reading the string α. A string $\alpha \in I^*$ is accepted by the ENFA if $\delta(q_0, \alpha) \cap F \neq \emptyset$, that is if the automaton can reach a final state when started from the initial state with input α. The language recognized by an ENFA M is denoted by T (M) and is defined as the set of all strings accepted by M.

Despite seeming to be more powerful devices, ENFA's and NFA's recognize exactly the same class of languages as their deterministic counterparts, that is the regular sets. The concept of nondeterminism is an important one, however, and we shall make use of ENFA's is some of our later constructions. We will often represent particular ENFA's pictorially by means of "transition diagrams". These are simply directed graphs in which nodes represent states and an arc labelled a is drawn from node q to node p if $p \in \delta(q,a)$.

Earlier we defined the context free languages as those which can be generated by context free grammars. They may also be characterized as the class of languages which can be recognized by nondeterministic pushdown automata (NPDA for short). An NPDA is basically an NFA augmented by a second tape which can be both read and written by the device and which is used as a pushdown store or stack. Unlike the case with finite automata, the deterministic variety of push down automata are less powerful than their nondeterministic counterparts. The

class of languages which can be recognized by a deterministic pushdown automaton (a DPDA for short) is called the deterministic languages. These are a proper subset of the context free languages but are probably more interesting and important theoretically and better suited as models of programming languages than the larger class. One reason for this is that the deterministic languages can all be recognized in linear time by conventional (i.e. deterministic) models of computation. Although it is not known for certain that the recognition problem for general context free languages is of non-linear complexity, the best algorithm known so far runs in deterministic time proportional to $n^{\log_2 7}$ where n is the length of the input string. (See Valiant (1975)).

The final class of languages which we need to introduce are the strict deterministic languages. These are the languages which can be recognized by "empty store" on a DPDA. They may also be defined as the class of prefix-free deterministic languages. (A language L is prefix-free if both α and $\alpha\beta$ in L implies $\beta = \Lambda$.) This class of languages has been extensively studied by Harrison and Havel (1973,1974) and we shall find it useful for the purpose of characterizing further classes of languages.

CHAPTER 2.

THE LR(k) PROPERTY

In this chapter we introduce the LR(k) grammars and languages and discuss their properties together with those of the associated parsing algorithm. Our main concern is to present a thorough yet straightforward development of these topics, thereby laying a secure foundation upon which to base the extensions and modifications which are the subject matter of later chapters.

In the next chapter we will construct a theory which is a true generalization of the one to be discussed here. Much of the development in that chapter will parallel that in this one - in this way we hope to make clear the relationship which exists between our new theory and the established theory from which it is derived. The rather considerable length of the present chapter is due to the need to introduce and state carefully all results which have counterparts in the generalized theory. This is necessary not only for the purposes of comparison and analogy, but also because many proofs in the next chapter depend upon these results from the standard theory.

Our treatment of the standard LR(k) theory follows approximately that to be found in the original paper by Knuth (1965) and the standard work of Aho and Ullman (1972a) although slight changes in both treatment and notation are necessary in order to support our subsequent extensions. We also include material from other sources, notably Geller and Harrison (1973), Harrison and Havel (1973,1974) and Hunt,Szymanski and Ullman (1974,1975).

Because we are primarily interested here in the general flow of the development, rather than its details, most results are stated without proof but are provided with explicit references to sources in the literature where proofs may be found. However,some results are proved here in full even though similar proofs are available elsewhere. The reason for this is that modified versions of these proofs will be used to establish more general results in the next chapter. The proofs given in this present chapter are structured so that these later modifications are supported most naturally and easily.

We begin by considering a definition of the LR(k) grammars and briefly discuss its relationship to alternative definitions.

2.1. The LR(k) Grammars and Languages.

Knuth (1965) introduced the LR(k) property in order to characterize those languages which are "translatable from Left to Right with bound k". That is the class of languages with the property that "if we read the characters of a string from left to right, and look a given finite number (i.e. k) of characters ahead, we are able to parse the given string without ever backing up to reconsider a previous decision" (op.cit., page 607). In other words, the LR(k) property attempts to characterize the languages which can be parsed deterministically using k symbol lookahead. Of course, a given language may be generated by several different grammars, and the difficulty of parsing its strings will usually depend critically upon the particular grammar with respect to which the parse is required. For this reason, the LR(k) property is defined as a property of grammars, and not of languages directly.

As a first attempt at the formalisation of this notion, we may take the following definition which is paraphrased from Knuth (op.cit., page 610) : 'a grammar is LR(k) if and only if any handle is always uniquely determined by its left context and the first k symbols of its right context'. Now when α is an rsf with a handle (p,m), the string which comprises 'its left context and the first k symbols of its right context' is given by $(m+k):\alpha$. Using this fact, the previous definition may be re-expressed more formally as follows : a grammar is LR(k) if and only if whenever α and β are rsf's with handles (p,m) and (q,n) respectively such

that $(m+k):\alpha = (m+k):\beta$, then $(p,m) = (q,n)$. This is the basis of the following precise formal definition, due to Harrison and Havel (1974).

DEFINITION 2.1

Let $G = (V_N, V_T, P, S)$ be a grammar and k a natural number. Then G is LR(k) if and only if the following conditions are satisfied.

(i) G is reduced and $S \Rightarrow^+ S$ does not occur in G, and

(ii) whenever α and β are rsf's of G having handles (p,m) and (q,n) respectively, such that $m/\beta \in V_T^*$ and $(m+k):\alpha = (m+k):\beta$, then necessarily $(p,m)=(q,n)$

A language is said to be LR(k) if it is generated by some LR(k) grammar. □

Before discussing the details and implications of this definition we will examine two illustrative examples.

Consider first the grammar whose productions are :

$$\begin{aligned} S &\longrightarrow aAc \\ A &\longrightarrow bAb \mid b \end{aligned} \qquad \text{(Grammar G1)}$$

This grammar generates the language $\{ab^{2n+1}c \mid n \geqslant 0\}$ and is not LR(k) for any k. This is because, for each $k \geqslant 0$, we can construct the pair of derivations :

$$S \Rightarrow_R^* ab^kAb^kc \xrightarrow{(A \to b,\ k+2)}_R ab^{2k+1}c ,$$

$$S \Rightarrow_R^* ab^{k+1}Ab^{k+1}c \xrightarrow{(A \to b,\ k+3)}_R ab^{2k+3}c .$$

It is easily seen that the handles of the rsf's $ab^{2k+1}c$ and $ab^{2k+3}c$ violate condition (ii) of Definition 2.1. From the point of view of parsing, the problem with this grammar is that having read the partial string ab^m,

no definite information about replacing the last b is provided by the next k symbols; we must wait until the symbol c is encountered. On the other hand, the grammar :

$$\begin{array}{lll} S & \rightarrow & aAc \\ A & \rightarrow & Abb \mid b \end{array} \qquad \text{(Grammar G2)}$$

which generates exactly the same language as G1, is LR(0). These examples clearly show that LR(k) is a property of the grammar, not of the language alone.

Three points concerning Definition 2.1. deserve mention. Firstly, the requirement that the grammar be reduced is made simply for mathematical convenience. This restriction should cause no practical difficulties since any grammar may easily be amended so that it becomes reduced without changing its essential structure, nor the language which it generates. Secondly, exclusion of derivations of the form $S \rightarrow^{+} S$ is necessary because certain ambiguous grammars would otherwise be admitted - the grammar

$$S \rightarrow S \mid a$$

would be an ambiguous LR(0) grammar for example. Geller and Harrison (1973) report that Salomaa (1973) first noted that the production $S \rightarrow S$ should not be allowed in LR(k) grammars, while they attribute the exclusion of all derivations of the form $S \rightarrow^{+} S$ to S. Graham.

Our definition of the LR(k) property admits no ambiguous grammars. This fact is established in the following theorem.

THEOREM 2.2

If G is an LR(k) grammar, then it is unambiguous.

PROOF. Because we stipulate that $S \rightarrow^+ S$ cannot occur in any LR(k) grammar, S can have no handle. Obviously every other rsf of G possesses at least one handle, but because G is LR(k), it is immediate from part (ii) of Definition 2.1. that no rsf can have more than one handle. Thus every rsf of G has exactly one handle, except S, which has none. The conclusion of the theorem then follows directly from Theorem 1.1. □

The third point we wish to make about our definition of the LR(k) property concerns the condition $m/\beta \in V_T^*$ which appears in part (ii) of Definition 2.1. As noted by Harrison and Havel (1974), this condition is absent from certain rival definitions (for example that of Salomaa (1973)) and can be shown to make no difference to the class of grammars defined in the case $k > 0$. When $k = 0$, however, omitting this condition causes the unnecessary exclusion of all grammars containing Λ-rules. For example, contrary to our natural intuition, the grammar

$$S \rightarrow Sa \mid \Lambda$$

would fail to be LR(0) even though it can be parsed without lookahead. As well as reducing the class of grammars defined in the case $k = 0$, omitting the condition $m/\beta \in V_T^*$ also causes mathematical difficulties when $k = 0$.

Since we have argued for the retention of the condition $m/\beta \in V_T^*$ in Definition 2.1, it may be wondered why we do not also include the similar condition

$n/\alpha \in V_T^*$. The reason is that this new condition is not independent of the others and may, in fact, be deduced from them. For technical purposes this result is useful in its own right and provides the following lemma.

LEMMA 2.3

Let G be a grammar and let α and β be rsf's of G with handles (p,m) and (q,n) respectively such that $m:\alpha = m:\beta$. Then $n/\alpha \in V_T^*$.

PROOF. Because (p,m) is a handle for α, we have $m/\alpha \in V_T^*$. Similarly, the fact that (q,n) is a handle for β provides $n/\beta \in V_T^*$. Clearly, if $n \geq m$ then the conclusion of the lemma is satisfied immediately. Suppose, on the other hand, that $n < m$. Certainly α contains no nonterminals to the right of the m'th position. But also, because $m:\alpha = m:\beta$ and $n/\beta \in V_T^*$, neither can it contain any nonterminals between the n+1'st and m'th positions. Thus $n/\alpha \in V_T^*$ and the lemma is proved. □

Although the LR(k) grammars have been extensively studied, several different definitions have been employed. Not all of these definitions are equivalent to ours (we have already seen that this is so in the case of Salomaa's definition) and so we briefly mention two of the major alternatives. Definition 2.1. differs from the formal definition used by Knuth (1965,page 610) in that it does not use endmarkers. Essentially, Knuth's definition requires that the grammar be augmented by the addition of a production $S' \rightarrow S \perp$ where S' is a new goal symbol and $\perp$ is a special endmarker symbol. Geller and Harrison (1973) show that this definition is equivalent to ours when $k > 0$ but that it reduces the class of grammars (though not the class of languages) considered in the case $k=0$. Aho and Ullman (1972a) define the LR(k) grammars differently again. They augment the grammar with a production $S' \rightarrow S$ where S' is again a new goal symbol. Geller and Harrison (1973) also show that this definition too is equivalent to ours when $k > 0$ but that it reduces not only the class of grammars but also the class of languages considered in the case $k=0$.

In summary, from among the several rival definitions of the LR(k) property, we have chosen in Definition 2.1. the one which yields the largest class of unambiguous grammars.

Now that we have agreed upon a definition for the LR(k) grammars, we may proceed to explore their properties. First we will examine the generative power of these grammars. That is to say, given a value for k, we will ask how extensive is the class of LR(k) languages. We begin with the following result.

THEOREM 2.4

If G is an LR(k) grammar, then L(G) is a deterministic language. □

This theorem is due to Knuth (1965). Its proof depends upon the existence of a parsing algorithm for the LR(k) grammars (which we present later - see Section 2.5) which can be implemented on a DPDA. Thus we see that not every context free language is an LR(k) language; rather, the LR(k) languages are some subset of the deterministic languages. Since it is clear from Definition 2.1 that every LR(k) grammar is an LR(k + 1) grammar, it follows that every LR(k) language is also an LR(k+1) language. We would therefore like to know exactly how closely the extent of the LR(k) languages approaches that of the LR(k+1) languages, and also how closely it approaches that of the deterministic languages. These questions are largely resolved in the following theorem.

THEOREM 2.5

Every deterministic language is generated by some LR(1) grammar. □

This result is also from Knuth (1965) but that source does not present a rigorous proof. The first complete and reasonably direct proof was given by Harrison and Havel (1974).

Theorem 2.5 tells us that the class of LR(k) languages is not enlarged by taking values of k greater that 1. In combination with Theorem 2.4 it also tells us that when $k \geq 1$ the LR(k) languages are co-extensive with the deterministic languages. Theorem 2.5 cannot be sharpened from LR(1) to LR(0) because the LR(0)

languages are known to be a proper subset of the deterministic languages. The LR(0) languages may be characterized, however, in terms of the strict deterministic languages as the following theorem (which is due to Geller and Harrison (1973)) shows.

THEOREM 2.6

L is an LR(0) language if and only if there exists a pair of strict deterministic languages L_1 and L_2 such that $L = L_1 L_2^*$. □

In conclusion, these results show that the LR(k) languages are closely related to the deterministic languages. In particular, when $k \geq 1$, the two classes are equivalent. This is encouraging from a practical point of view because it indicates that the LR(k) grammars have sufficient power to describe the syntax of programming languages. It is also one of the reasons why the LR(k) grammars are so interesting from a theoretical viewpoint for, to quote Knuth (1965) again, it suggests that "the LR(k) condition is a natural analogue, for grammars, of the deterministic condition, for languages."

In the following sections we will examine methods of testing grammars for the LR(k) property, and of parsing the LR(k) grammars.

2.2. Testing for the LR(k) Property - Part 1.

The concept of an LR(k) grammar is of little practical use unless we can find an algorithm to test whether or not a given grammar possesses the LR(k) property. It is by no means obvious that such an algorithm should exist, for the definition of the LR(k) property given in Definition 2.1 involves quantification over the (possibly infinite) set of all rsf's of the given grammar. Our first result is somewhat disheartening.

THEOREM 2.7

The problem of deciding, for a given grammar G, whether or not there exists a k such that G is LR(k), is recursively unsolvable. □

The proof of this theorem involves a reduction to a modified form of Post's Correspondence Problem, and is due to Knuth (1965).

This result is not, of course, the disaster it may seem, for in practice we are not concerned with whether or not a grammar is LR(k) for some k, but rather with whether it is LR(k) for a particular, predetermined, value of k. When stated in this form, the problem becomes solvable.

THEOREM 2.8

Given a grammar G and a natural number k, there is an algorithm to determine whether or not G is LR(k). □

In fact, there are three distinct algorithms for solving this problem. Two of the methods are from Knuth (1965), while the third is due to Hunt, Szymanski and

Ullman (1974,1975). The three methods are distinct, yet related, and each has its own particular merits. Since there is an obvious algorithm to determine whether a grammar satisfies condition (i) of Definition 2.1, we will only concern ourselves with methods for testing condition (ii) of that definition. Accordingly, we will assume for the remainder of this section, and throughout the next two, that $G=(V_N, V_T, P, S)$ is a reduced grammar in which $S \rightarrow^+ S$ does not occur, and that k is a fixed natural number.

The first method which we present is one of those due to Knuth. Its chief virtue lies in the fact that it provides a proof of Theorem 2.8 with a minimum of additional technical apparatus. For this reason it is the method most commonly quoted when it is required to demonstrate the decidability of the LR(k) property for purely theoretical purposes. (See, for example, the books of Salomaa (1973) and Hopcroft and Ullman (1969).) Our construction is a variation upon those employed in these references and since the practical details are of little interest we omit them. We require one additional definition.

DEFINITION 2.9

Let $\perp$ be a symbol not in V. (This symbol will be used as an 'endmarker' and will be reserved for this purpose for the rest of this section.) For each production $q \in P$ we define the set of strings $R_k^G(q)$, called the LR(k) contexts of q, as follows :

$$R_k^G(q) = \{(m+k):\beta\perp^k \mid \beta \text{ is an rsf of G with a handle } (q,m)\}. \square$$

That is, for each rsf β of G which has a handle involving production q, we include in $R_k^G(q)$ that prefix of β extending as far as the k'th symbol of the right context of the handle. If the right context should be less than k symbols long, then it is first augmented by the addition of a suitable number of endmarkers. This use of endmarkers is essential to the argument used in the proof of the next result, which expresses the LR(k) property in terms of the sets $R_k^G(q)$.

LEMMA 2.10

G is LR(k) if and only if, for any $p,q \in P$, $\alpha \in V^*\perp^*$ and $u \in V_T^*\perp^*$; $\alpha \in R_k^G(p)$ and $\alpha u \in R_k^G(q)$ imply $p=q$ and $u = \Lambda$.

PROOF. We prove the result first in the 'if' direction. Suppose that θ and ψ are rsf's of G with handles (p,m) and (q,n) respectively, such that

$$m/\psi \in V_T^* \qquad (1)$$

and

$$(m+k):\theta = (m+k):\psi \qquad (2)$$

and assume that $\alpha \in R_k^G(p)$ and $\alpha u \in R_k^G(q)$ imply $p=q$ and $u=\Lambda$. We must show that these imply $(p,m)=(q,n)$. Define $\alpha = (m+k):\theta\perp^k$ and $\beta = (n+k):\psi\perp^k$. Then $\alpha \in R_k^G(p)$ and $\beta \in R_k^G(q)$. We now distinguish two cases according to the relative magnitudes of m and n.

Case 1 : $m \leq n$. In this case (1) and (2) imply that $\beta = \alpha u$ for some $u \in V_T^*\perp^*$ and so the hypothesis provides $p=q$ and $u=\Lambda$. But $n-m = \text{len}(u)$ and so $u=\Lambda$ implies that $n=m$ and we conclude $(p,m) = (q,n)$ as required.

Case 2 : $m > n$. From (2) and Lemma 2.3 we obtain $n/\alpha \in V_T^*$ and this result, taken together with (2) and the condition $m>n$, implies that $\alpha = \beta u$ for some $u \in V_T^*\perp^*$. Again the hypothesis provides $p=q$ and $u=\Lambda$ and the conclusion $(p,m)=(q,n)$ follows as before and completes the proof in the 'if' direction.

We now turn to the 'only if' direction. Suppose that G is LR(k) and that for some $p,q \in P$, $\alpha \in V^*\perp^*$ and $u \in V_T^*\perp^*$ we have $\alpha \in R_k^G(p)$ and $\alpha u \in R_k^G(q)$. We need to prove that $p=q$ and $u=\Lambda$. Now $\alpha \in R_k^G(p)$ implies there is some rsf θ of G with a handle (p,m) satisfying $(m+k):\theta\perp^k = \alpha$. Similarly, $\alpha u \in R_k^G(q)$ implies that there is an rsf ψ of G with a handle (q,n) satisfying $(n+k):\psi\perp^k = \alpha u$. It follows from these observations that $m/\psi \in V_T^*$ and $(m+k):\theta = (m+k):\psi$ and therefore, since G is LR(k), that $(p,m) = (q,n)$. Hence $p = q$ and (since $\text{len}(u) = n-m$) $u = \Lambda$ and the lemma is proved. □

The reader who doubts the necessity of using endmarkers in Definition 2.9 should consider the following grammar :

$$S \rightarrow Sa \qquad \text{(Grammar G5)}$$
$$S \rightarrow a$$

Take k=1 and it is easily seen that

$$R_1^{G5}(S \rightarrow Sa) = \{Sa\perp, Saa\}$$

and $$R_1^{G5}(S \rightarrow a) = \{a\perp, aa\}$$

and hence, by Lemma 2.10, that G5 is LR (1). If the endmarker were absent, however, we should have

$$R_1^{G5}(S \rightarrow Sa) = \{Sa, Saa\}$$

and then in Lemma 2.10 we could take p=q= $S \rightarrow Sa$, α = Sa and u=a and thereby conclude, incorrectly, that G5 is not LR(1).

It may seem that the reformulation of the LR(k) property which is provided by Lemma 2.10 does not constitute much of an advance; it still involves properties of potentially infinite sets. The crucial result in this development is provided by the next lemma.

Lemma 2.11

If q is a production in P then $R_k^G(q)$ is a regular set.

PROOF. The proof of this result depends upon the construction of a right linear grammar $Q_k^G(q) = (V_N', V_T', P', S')$ such that $L(Q_k^G(q)) = R_k^G(q)$. To say that $Q_k^G(q)$ is right linear means that the right parts

of all the productions in P' are constrained to be strings in $V_T'^* \cup V_T'^* V_N'$. Right linear grammars are well known to generate only regular sets. (See, for example, Salomaa (1973), Chapter II Theorem 5.3).

Recall the convention that $\perp$ represents the end-marker symbol and that $\perp$ is assumed not to be a member of V. The grammar $Q_k^G(q)$ is defined as follows :

(i) $V_N' = \{[A,w] \mid A \in V_N, w \in k{:}V_T^* \perp^k\}$,

(ii) $V_T' = V_N \cup V_T \cup \{\perp\}$,

(iii) $P' = P_1 \cup P_2$ where P_1 and P_2 are disjoint sets of productions given by :

$P_1 = \{[A,w] \to \gamma[B,v] \mid A \to \gamma B \delta$ is a production in P and $v \in \mathrm{FIRST}_k^G(\delta w)\}$,

$P_2 = \{[C,w] \to \theta w \mid C \to \theta$ is production $q\}$,

(iv) $S' = [S,\perp^k]$.

It is clear that $Q_k^G(q)$ is right linear and hence $L(Q_k^G(q))$ is a regular set. It remains to prove that $L(Q_k^G(q)) = R_k^G(q)$. To do this is sufficient to show that $Q_k^G(q)$ contains the derivation

$$[S,\perp^k] \to^* \gamma[B,v]$$

if and only if G contains the derivation

$$S \to^* \gamma Bx$$

for some $x \in V_T^*$ satisfying $v = k{:}\, x\perp^k$. This result may be established by straightforward inductions upon the lengths of the derivations involved. We omit the details. □

Note that, in general, the grammar $Q_k^G(q)$ will not be reduced. However, this does not affect the utility of the construction in any way. Note too that it is only the set of productions P_2 which changes as q ranges over the productions of G.

Armed with Lemmas 2.10 and 2.11 we may now prove Theorem 2.8

PROOF OF THEOREM 2.8 - FIRST METHOD

By virtue of Lemma 2.10 we need to prescribe a method whereby we can determine :

(a) for each production q in P, whether or not there exist $\alpha \in V^*\perp^*$ and $u \in V_T^*\perp^*$ with $u \neq \Lambda$ such that both $\alpha \in R_k^G(q)$ and $\alpha u \in R_k^G(q)$.

(b) for each (ordered) pair of distinct productions p and q in P, whether or not there exist $\alpha \in V^*\perp^*$ and $u \in V_T^*\perp^*$ such that $\alpha \in R_k^G(p)$ and $\alpha u \in R_k^G(q)$.

Now, if L_1 and L_2 are languages, the quotient of L_1 with respect to L_2 is written L_1/L_2 and is defined by :

$$L_1/L_2 = \{x \mid \text{there exists } y \in L_2 \text{ such that } xy \in L_1\}.$$

(In reading the formulae that follow, assume that the quotient operator has higher precedence than intersection.) Using this notion, test (a) above may be restated as the problem of deciding, for each q in P, whether the language $S_k^G(q)$ defined by

$$S_k^G(q) = R_k^G(q) \,/\, (V_T^+\perp^* \cup V_T^*\perp^+) \cap R_k^G(q)$$

is empty or not. Similarly, test (b) reduces to the problem of deciding, for each pair of distinct productions p and q in P, whether the language $T_k^G(p,q)$ defined by :

$$T_k^G(p,q) = R_k^G(q) \,/\, V_T^*\perp^* \cap R_k^G(p)$$

is empty or not.

Now all the sets appearing in the definitions of $S_k^G(q)$ and $T_k^G(p,q)$ are regular. (Recall Lemma 2.11.) Furthermore, the regular sets are closed under the operations of union and intersection (see Hopcroft and Ullman (1969), Theorem 3.6) and quotient (op.cit.Theorem 9.13). Hence, the sets $S_k^G(q)$ and $T_k^G(p,q)$ are regular, and because the proof of Lemma 2.11 is constructive, as are the proofs of the other results cited above, it is in principle possible to construct finite automata which recognise these sets. Since there is a well known algorithm (op.cit.,Theorem 3.11) to determine whether the language recognized by a finite automaton is empty or not, we may conclude the theorem. □

We end this section with an example which illustrates some of the ideas that have been introduced here. We will use the Grammar G1 which, it may be remembered, has productions :

$$\begin{array}{lll} S & \longrightarrow & aAc \\ A & \longrightarrow & bAb \mid b \end{array}$$

We take k = 1 and construct the grammar $Q_1^{G1}(A \to b)$.Let $Q_1^{G1}(A \to b) = (V_N', V_T', P', S')$. Then

$V_N' = \{[S,\perp], [S,a], [S,b], [S,c], [A,\perp], [A,a], [A,b], [A,c]\}$,
$S' = [S,\perp]$,

$V_T' = \{S, A, a, b, c, \perp\}$, and

$P' = P_1 \cup P_2$ where

$$P_1 = \{ [S,\perp] \to a[A,c] \quad [A,c] \to b[A,b] \quad [A,b] \to b[A,b] \}$$ + some useless productions,

$$P_2 = \{ [A,b] \to bb, \quad [A,c] \to bc \}$$ + some useless productions.

It is easily seen that this grammar contains the derivations :

$[S, \perp] \rightarrow a[A, c] \longrightarrow ab[A, b] \longrightarrow abbb$, and

$[S, \perp] \rightarrow a[A, c] \longrightarrow ab[A, b] \longrightarrow abb[A, b] \longrightarrow abbbb$

and so abbb and abbbb are both members of $R_1^{Gl}(A \rightarrow b)$. Hence, by taking α = abbb and u=b in Lemma 2.10, we discover that Gl is not LR(1).

The algorithm for testing for the LR(k) property which is indicated in the proof of Theorem 2.8 is not well suited to practical use. In the next section we will derive a practical algorithm for testing grammars for this property.

2.3. Testing for the LR(k) Property - Part 2.

We have seen one method of testing for the LR(k) property; now we present another. This method is also due to Knuth and is of interest because, in the case of grammars which are LR(k), the construction may be extended to provide a parser for the grammar concerned. Before describing the method, we need several new definitions which are fundamental to this and subsequent developments.

DEFINITION 2.12

An LR(k) item for G is a pair : $[B \rightarrow \beta_1 . \beta_2, v]$,where $B \rightarrow \beta_1\beta_2 \in P$ and $v \in V_T^{*k}$. The set of all LR(k) items for G is denoted I_k^G. □

That is, an LR(k) item consists of a production from P with a dot placed somewhere in its right part (we assume that the dot is not in V) and a terminal string up to k symbols long. Items in which the dot appears at the extreme left of the right part of the production (i.e. those in which $\beta_1 = \Lambda$)are called initial items; those in which the dot appears at the extreme right (i.e. $\beta_2 = \Lambda$) are called final items while those items which are neither initial nor final are called intermediate. Note that any non-initial LR(k) item can be written in the form $[B \rightarrow \beta_1 X . \beta_2, v]$ where $X \in V$. The symbol X which precedes the dot is called the associated symbol of the item.

DEFINITION 2.13

A pair of distinct LR(k) items for G are said to be in conflict (on lookahead u) if they have the form $[A \rightarrow \alpha .,u]$ and $[B \rightarrow \beta_1.\beta_2,v]$ respectively and satisfy $u \in EFF_k(\beta_2 v)$. □

Observe that one of the items in a conflicting pair must be a final item. If the other item is final also (i.e. if in Definition 2.13. we have $\beta_2 = \Lambda$) then we say that we have a "reduce/reduce" conflict; otherwise we have a "shift/reduce" conflict. The reason for this terminology will become apparent when we come to describe the LR(k) parsing algorithm. We note that an obvious algorithm exists to test whether or not a given pair of items are in conflict.

It is the idea of a "valid" LR(k) item which is particularly important.

DEFINITION 2.14

When $\theta \in V^*$, the LR(k) item $[B \rightarrow \beta_1.\beta_2,v]$ is said to be valid for θ (with respect to G and k) if and only if there is a derivation

$$S \xrightarrow[R]{*} \gamma Bx \xrightarrow[R]{} \gamma\beta_1\beta_2 x$$

in G with $\theta = \gamma\beta_1$, and $v = k{:}x$. □

Observe that if there is an LR(k) item which is valid for θ, then θ must be a viable prefix of G. Conversely every viable prefix possesses at least one valid LR(k) item. Observe too that if there is a non-initial LR(k) item which is valid for θ , then $\theta \neq \Lambda$ and the associated symbol of that item must be equal to the last symbol of θ . Thus all non-initial LR(k) items which are valid for a given viable prefix share the same associated symbol.

DEFINITION 2.15

When $\theta \in V^*$, the set of all LR(k) items which are valid for θ is called the LR(k) state for θ and is denoted by $V_K^G(\theta)$. We usually drop the sub and superscripts and write simply $V(\theta)$ when no ambiguity is likely. The set consisting of the LR(k) states of all the viable prefixes of G is called the LR(k) stateset for G and is denoted by S_K^G. Note that $V(\theta)$ is non-empty if and only if θ is a viable prefix of G. Hence

$$S_k^G = \{ V(\theta) \neq \emptyset \mid \theta \in V^* \} . \square$$

We now define the 'adequacy' of LR(k) states and statesets. This notion is the key to the algorithm we are seeking.

DEFINITION 2.16

An LR(k) state (or indeed any set of LR(k) items for G) is adequate if and only if it contains no pair of conflicting items. The LR(k) stateset for G is adequate if and only if each of its component LR(k) states is adequate. States and statesets which are not adequate are said to be inadequate. $\square$

The LR(k) property is closely related to the adequacy of LR(k) statesets. Before we can prove this fact, we need another lemma.

Lemma 2.17

Let $\alpha\beta$ be an rsf of G with a handle (q,n) satisfying $n > \text{len}(\alpha)$. Then there is a non-final LR(k) item $[C \rightarrow \gamma_1 . \gamma_2, v]$ which is valid for α with $EFF_k(\beta) \subseteq EFF_k(\gamma_2 v)$.

PROOF. Since $\alpha\beta$ is an rsf of G with a handle (q,n), there must be an explicit r-derivation $D = \langle (q_i, n_i) \rangle_{i=1}^{r}$ of $\alpha\beta$ from S with $(q_r, n_r) = (q,n)$. Let $\langle \psi_i \rangle_{i=0}^{r}$ be the corresponding implicit derivation. Clearly there exists t in the range $1 \leqslant t \leqslant r$ such that $n_t - \deg(q_t) \leqslant \text{len}(\alpha)$. (Take t=1 for example, since $\psi_0 = S$ we must have $n_1 - \deg(q_1) = 0$.) Now choose the largest such t. We will show that $\text{len}(\alpha) < n_t$. This relation is certainly true when t=r (because $\text{len}(\alpha) < n$ by hypothesis) so assume that $t < r$ and suppose, for the sake of deriving a contradiction, that $\text{len}(\alpha) \geqslant n_t$. Because D is an r-derivation we have $n_{i+1} - \deg(q_{i+1}) \leqslant n_i$ for all i in the range $1 \leqslant i < r$ and so if $t < r$ and $\text{len}(\alpha) \geqslant n_t$ this gives

$$n_{t+1} - \deg(q_{t+1}) < \text{len}(\alpha)$$

which contradicts the choice that t be the <u>largest</u> integer with the prescribed properties. Hence we conclude

$$n_t - \deg(q_t) \leqslant \text{len}(\alpha) < n_t. \qquad (1)$$

We now have the derivation

$$S \xrightarrow{*}_{\pi} \psi_{t-1} \Rightarrow (q_t, n_t) \Rightarrow_{\pi} \psi_t \xrightarrow{*}_{\pi} \alpha\beta \qquad (2)$$

and the choice of t ensures that $n_i - \deg(q_i) > \text{len}(\alpha)$ for all i in the range $t < i \leqslant r$ and so it follows that ψ_t has the form $\psi_t = \alpha\theta$ where θ satisfies $\theta \xrightarrow[\text{RSFS}]{*} \beta$. Let production q_t be $C \rightarrow \gamma$. Then ψ_t may also be written in the form $\psi_t = \sigma\gamma x$ where $\text{len}(\sigma\gamma) = n_t$. The inequalities (1) then become

$$\text{len}(\sigma) \leqslant \text{len}(\alpha) < \text{len}(\sigma\gamma)$$

and so γ can be written as $\gamma = \gamma_1\gamma_2$ where $\text{len}(\sigma\gamma_1) = \text{len}(\alpha)$.

Now $\psi_t = \alpha\theta = \sigma\gamma_1\gamma_2 x$ and the construction ensures that $\sigma\gamma_1 = \alpha$, $\gamma_2 \neq \lambda$ and $\gamma_2 x = \theta$. Therefore, the derivation (2) may be written as

$$S \underset{R}{\rightarrow} \sigma C x \underset{R}{\rightarrow} \sigma\gamma_1\gamma_2 x$$

and so when $v = k{:}x$ it follows that $[C \rightarrow \gamma_1 . \gamma_2, v]$ is a non-final LR(k) item which is valid for α. Recall that we have $\gamma_2 x = \theta$ and $\theta \underset{RIGHT}{\overset{*}{\rightarrow}} \beta$ so that $\gamma_2 x \underset{RIGHT}{\overset{*}{\rightarrow}} \beta$. It then follows that $EFF_k(\beta) \subseteq EFF_k(\gamma_2 v)$ and the proof is complete. □

We can now give the theorem on which our second method of testing for the LR(k) property depends.

THEOREM 2.18

G is LR(k) if and only if its LR(k) stateset is adequate.

PROOF. To establish the result in the 'if' direction we take its contrapositive. That is, we suppose G is not LR(k) and prove that its LR(k) stateset must contain an inadequate state.

Now if G is not LR(k) there exist rsf's α and β of G with handles (p,m) and (q,n) respectively such that

$$m/\beta \in V_T^*, \tag{1a}$$
$$(m+k):\alpha = (m+k):\beta, \tag{1b}$$
and $$(p,m) \neq (q,n). \tag{1c}$$

We distinguish three cases according to the relative magnitudes of m and n.

Case 1 : m=n. Let production p be $D \rightarrow \delta$ and let q be $E \rightarrow \sigma$. Also let $\theta = m:\alpha$ and let $u = k:x$ where x is the string satisfying $\alpha = \theta x$. Then it follows from (1a) and (1b) that β has the form $\beta = \theta y$ where y also satisfies $u = k:y$. Because (p,m) is a handle for α it follows from this construction that $[D \rightarrow \delta.,u]$ is a valid LR(k) item for θ. Similarly, because (q,n) is a handle for β , $[E \rightarrow \sigma.,u]$ is also valid for θ. But when m=n, (1c) can only be satisfied if $p \neq q$. Consequently, since they are distinct, the two items $[D \rightarrow \delta.,u]$ and $[E \rightarrow \sigma.,u]$ are in conflict and so $V(\theta)$ is inadequate.

Case 2 : $m < n$. Again let production p be $D \rightarrow \delta$ and let θ and u be defined as in the previous case. As before, $[D \rightarrow \delta.,u]$ is valid for θ and (1a) and (1b) imply that $\beta = \theta y$ where $u = k:y$. Now because (q,n) is a handle for θy and $n > \text{len}(\theta)$, Lemma 2.17 implies that there is a non-final LR(k) item $[C \rightarrow \gamma_1 . \gamma_2, v]$ which is valid for θ with $EFF_k(y) \subseteq EFF_k(\gamma_2 v)$. It follows that $[C \rightarrow \gamma_1 .. \gamma_2, v]$

conflicts with $[D \to \delta .,u]$ and since both items are valid for θ, this means that $V(\theta)$ is inadequate.

Case 3 : $m > n$. In this case the construction and argument proceed exactly as in the case above, but with the roles of p and q, m and n, and α and β interchanged. Also, Lemma 2.3 provides $n/\alpha \in V_T^*$ which is needed in place of (1a).

We now prove the contrapositive of the theorem in the 'only if' direction. We suppose that S_k^G is inadequate and proceed to deduce that G cannot be LR(k). Now if S_k^G is inadequate there must be some viable prefix θ of G such that $V(\theta)$ contains a pair of conflicting items, say $[D \to \delta .,u]$ and $[C \to \gamma_1 . \gamma_2, v]$. For conflict to occur we must have $u \in EFF_k(\gamma_2 v)$. Because $[D \to \delta .,u]$ is valid for θ there must be a derivation in G with the form.

$$S \xrightarrow[R]{*} \mu D x \xrightarrow[R]{} \mu \delta x$$

where $\mu\delta = \theta$ and $u = k{:}x$. Let $\alpha = \mu\delta x$, $m = \text{len}(\theta)$, and let the production $D \to \delta$ be called p. Then (p,m) is a handle for α and we have

$$(m+k){:}\alpha = \theta u. \qquad (2)$$

Similarly, since $[C \to \gamma_1 . \gamma_2, v]$ is also valid for θ, there is a derivation in G with the form

$$S \xrightarrow[R]{*} \eta C y \xrightarrow[R]{} \eta \gamma_1 \gamma_2 y \qquad (3)$$

where $\theta = \eta\gamma_1$ and $v = k{:}y$. Now we have $u \in EFF_k(\gamma_2 v)$ and thererore also $u \in EFF_k(\gamma_2 y)$ and so there exists $z \in V_T^*$ such that $\gamma_2 y \xrightarrow[REFF]{*} z$ and $u = k{:}z$. We now distinguish two cases according to the number of steps in the derivation $\gamma_2 y \xrightarrow[REFF]{*} z$.

Case 1 : $\gamma_2 y = z$ (i.e. no steps at all). If we let the production $C \to \gamma_1 \gamma_2$ be called q and let $len(\eta\gamma_1\gamma_2) = n$ then (3) implies that (q,n) is a handle of $\eta\gamma_1\gamma_2 y$. But $\eta\gamma_1 = \theta$ and $\gamma_2 y = z$ and so (q,n) is a handle for θz . Since len $(\theta) = m$ and $u = k{:}z$, we also have

$$(m+k){:}\theta z = \theta u \qquad (4)$$

and

$$m/\theta z \ e \ V_T^* \qquad (5)$$

Now if G were LR(k), the facts that (p,m) is a handle for α and (q,n) is a handle for θz , combined with (2), (4) and (5), would imply that (p,m) = (q,n). We now show that these handles must be distinct and hence that G is not LR(k). If (p,m) = (q,n) then obviously p=q and m=n. Observe that $n = m + len(\gamma_2)$ and so m=n implies $\gamma_2 = \Lambda$ which in turn implies u=v (since $u \in EFF_k(\gamma_2 v)$). We now have p=q, $\gamma_2 = \Lambda$ and u=v. But this means that the two items $[D \to \delta\ .,u]$ and $[C \to \gamma_1.\gamma_2,v]$ are the same and so contradicts the hypothesis that they are in conflict. Thus we conclude $(p,m) \neq (q,n)$ and therefore that G is not LR(k).

Case 2 : $\gamma_2 y \xrightarrow[REFF]{+} z$ (i.e. at least one step). Since the derivation contains at least one step, we may distinguish the last and write

$$\gamma_2 y \xrightarrow[REFF]{*} \rho \ \xrightarrow{(q,n)}_{REFF} \ z.$$

Note that since this a reff-derivation we must have $n > 0$. Combining this derivation with that of (3) gives

$$S \xrightarrow[R]{*} \eta\gamma_1\gamma_2 y \xrightarrow[R]{*} \eta\gamma_1\rho \ \xrightarrow{(q,n+len(\eta\gamma_1))}_{R} \ \eta\gamma_1 z$$

and because $\eta\gamma_1 = \theta$, len $(\theta) = m$ and $k{:}z = u$ this provides

$$S \xrightarrow[R]{*} \theta\rho \ \xrightarrow{(q,n+m)}_{R} \ \theta z$$

and so (q,n+m)is a handle for θz . The results (4) and (5) will follow exactly as in the previous case and then

the supposition that G is LR(k) will require that (p,m) = (q,n+m). But this is clearly impossible if n > 0 and so we conclude that G is not LR(k) and the proof of the theorem is complete. □

Given a grammar and its LR(k) stateset, we can certainly test the stateset for adequacy and thereby determine whether the grammar is LR(k). Thus in order to furnish an alternative method of testing for the LR(k) property, it only remains to prescribe an algorithm for constructing LR(k) statesets. To this end, we begin with a new definition.

DEFINITION 2.19

When Δ is any set of LR(k) items for G, we define its closure, denoted $\text{CLOSURE}_k^G(\Delta)$, recursively as the smallest set satisfying :

$$\text{CLOSURE}_k^G(\Delta) = \Delta \cup \{[A \to .\alpha, u] \mid \text{there exists an item } [B \to \beta_1.A\beta_2, v] \in \text{CLOSURE}_k^G(\Delta) \text{ where } A \to \alpha \in P \text{ and } u \in \text{FIRST}_k^G(\beta_2 v)\}$$

and when $X \in V$ we define

$$\text{NEXT}_k^G(\Delta, X) = [D \to \delta_1 X.\delta_2, w] \mid [D \to \delta_1.X\delta_2, w] \in \Delta\}.$$

It is clear that the functions CLOSURE and NEXT are computable. We define a third function by their composition thus :

$$\text{GOTO}_k^G(\Delta, X) = \text{CLOSURE}_k^G(\text{NEXT}_k^G(\Delta, X)).$$

As usual, we will omit the sub and superscripts from these functions when the meaning is clear. □

The algorithm for constructing LR(k) statesets is based on the following theorem.

THEOREM 2.20

Let $\theta \in V^*$ and $X \in V$. Then $V(\theta X) = \text{GOTO}(V(\theta), X)$. □

A proof of this result is provided by Aho and Ullman (1972a, Theorem 5.10)

Implicit in this proof are certain properties of the functions CLOSURE and NEXT which are useful in themselves (and from which Theorem 2.20 may be deduced directly). In order to be able to state these properties conveniently, we partition the items making up an LR(k) state into two groups, called the 'nucleus' and the 'completion' of the state. The basic idea is that the nucleus of an LR(k) state contains all and only the non-initial items in the state; any remaining items form the completion of the state. The formal definition is slightly complicated by the need to deal with the state $V(\Lambda)$ as a special case.

DEFINITION 2.21

The nucleus of the LR(k) state for θ is denoted by $N_k^G(\theta)$ and is defined by :

(i) $N_k^G(\Lambda) = \{[S \rightarrow .\alpha, \Lambda] \mid S \rightarrow \alpha \in P\}$,

(ii) and when $\theta \neq \Lambda$

$$N_k^G(\theta) = \{[B \rightarrow \beta_1 . \beta_2, v] \in V_k^G(\theta) \mid \beta_1 \neq \Lambda\}.$$

The completion of the LR(k) state for θ is denoted by $C_k^G(\theta)$ and is given by :

$$C_k^G(\theta) = V_k^G(\theta) \setminus N_k^G(\theta).$$

When the meaning is clear, we drop the sub and super-scripts and write simply $N(\theta)$ and $C(\theta)$. □

Note that when $\theta \neq \Lambda$ we have, as a consequence of its very definition, that $N(\theta) \subseteq V(\theta)$. It is easy to prove that this relationship remains true when $\theta = \Lambda$. We may now state the properties of the functions NEXT and CLOSURE alluded to earlier.

THEOREM 2.22

Let $\theta \in V^*$ and $X \in V$. Then

(i) $N(\theta X) = \text{NEXT}(V(\theta), X)$, and

(ii) $V(\theta) = \text{CLOSURE}(N(\theta))$. □

At last we can present the algorithm for constructing LR(k) statesets. We start off with just a single state (i.e. $V(\Lambda)$) and add states to the stateset until we determine that no more should be added. Note that during execution of the algorithm a tabulation of the function $\text{GOTO}_k^G(\Delta, X)$ may be produced for all states Δ in the stateset and all symbols $X \in V$. This tabulation will be needed later when we come to construct parsers for the LR(k) grammars.

ALGORITHM 2.23

Evaluation of the LR(k) stateset for G.

Input : The Grammar $G = (V_N, V_T, P, S)$ and a value for k.

Output : S_k^G - the LR(k) stateset for G.

Method : The stateset is built up in the set-valued variable S. A marker flag is considered to be attached to each LR(k) state placed in S; states are 'unmarked' when first added to S.

```
begin
        compute V(Λ) and set S = {V(Λ)} ;
        while S contains any unmarked states do
              select an unmarked state Δ from S and
              mark it;
              for each X ∈ V do
                  compute  Σ = GOTO(Δ,X);
                  if Σ ≠ ∅ and Σ is not in S then
                                add Σ to S endif
              endfor
        endwhile;
        set S_k^G = S
end . □
```

The correctness of this algorithm follows directly from Theorem 2.20. It is plain that the algorithm will terminate after a finite number of steps, provided that the stateset is finite; clearly it <u>is</u> finite for it is no larger than the powerset of I_k^G - which itself is finite.

We now have an alternative algorithm for testing for the LR(k) property, when k is fixed.

<u>PROOF OF THEOREM 2.8 - SECOND METHOD</u>

Construct the LR(k) stateset for G using Algorithm 2.23 and test it for adequacy. By virtue of Theorem 2.18, G will be LR(k) if and only if no inadequacies are found. □

Of course we may save ourselves a certain amount of wasted effort in the case of grammars which are not LR(k) by testing each new LR(k) state for adequacy before we add it to S_k^G in Algorithm 2.23. The enumeration of S_k^G may be abandoned as soon as an inadequate state is encountered. With grammars which <u>are</u> LR(k), however, the enumeration must proceed to the very end.

We illustrate some of the ideas introduced in this section by proving once again that the grammar G1 is not LR(1). We use Algorithm 2.23 to construct the LR(1) stateset for G1 and display the result in Figure 2.1. We number each state in the stateset and tabulate the function GOTO by referring to states via their numbers.

STATE NO.	LR (1) STATES		GOTO				
	NUCLEUS	COMPLETION	S	A	a	b	c
1	[S→.aAc,#]				2		
2	[S→a.Ac,#]	[A→.bAb,c] [A→.b,c]		3		4	
3	[S→aA.c,#]						5
4	[A→b.Ab,c] [A→b.,c]	[A→.bAb,G] [A→.b,b]		6		7	
5	[S—aAc.,#]						
6	[A—bA.b,c]					8	
7	[A→b.Ab,b] [A→b.,b]	[A→.bAb,b] [A→.b,b]		9		7	
8	[A→bAb.,c]						
9	[A→bA.b,G]					10	
10	[A→bAb.,b]						

Figure 2.1. The LR(1) Stateset and GOTO Function for Grammar G1.

We see that state 7 is inadequate because it contains the conflicting items [A → b., b] and [A → .b, b] among others. It follows that grammar G1 is not LR(1).

Although the procedure we have described does provide a practical test for the LR(k) property, it has one important drawback: it may take an unacceptably long time to make its decision. In order to be more precise about this topic we need to introduce the idea of the "complexity" of an algorithm.

Once we have an algorithm for solving a particular problem, it is natural to enquire into the efficiency of the algorithm; that is the amount of time and space it consumes when solving instances of the problem. So that we may achieve a measure of independence from the details of the implementation of the algorithm, and also from the precise model of computation involved, we usually choose to express these quantities as functions of the size of the input to the algorithm. Here we are mainly concerned about the time taken by an algorithm to process an input of a given size in the worst case. Accordingly, we define the time complexity of an algorithm to be that function $f(n)$ which is the maximum, over all inputs of size n, of the time taken by the algorithm. If, for example, an algorithm processess all inputs of size n in time which is bounded above by $c.n^2$, for some constant c, then we say that the complexity of this algorithm is $O(n^2)$ - pronounced "order n^2 ". An algorithm is said to be of polynomial complexity if its time complexity is $O(p(n))$ for some polynomial p, while it is said to be of exponential complexity if its complexity is $O(2^{q(n)})$ for some polynomial q. Algorithms of exponential complexity are unattractive because, as the size of the input increases, the time

taken by the algorithm grows so explosively that it rapidly becomes unacceptably large.

The inputs to the algorithms we are presently considering are descriptions of grammars which are to be tested for the LR(k) property. (We are supposing the value of k to be fixed beforehand.) Therefore, if we wish to examine the complexity of these algorithms, we must be more precise about what we mean by the "size" of a grammar.

Now any description of a grammar must enumerate, in some form or other, the productions of the grammar. Conversely, assuming certain reasonable conventions, a grammar may be completely specified by just its set of productions. Accordingly therefore, we define the size of a grammar to be the space required to list its productions, assuming that each symbol in the grammar's vocabulary occupies just one unit of space. Thus we have :

DEFINITION 2.24

Let $G = (V_N, V_T, P, S)$ be a grammar. Define the size of G, denoted SIZE (G) by

$$\text{SIZE}(G) = \sum_{q \in P} (1 + \deg(q)). \square$$

Observe that if SIZE (G) = n, then $|V| = O(n), |P| = O(n)$ and $|I_k^G| = O(n^{k+1})$.

Now the problem with our second method of testing for the LR(k) property is that it requires the construction and examination of every LR(k) state in the LR(k) stateset of the grammar concerned. (At least it does when the grammar is LR(k).) Since the processing of each LR(k) state must occupy at least one unit of time, it follows that if there exist any families of LR(k) grammars in

which the cardinality of the LR(k) stateset grows exponentially with the size of the grammar, then Algorithm 2.23 will have exponential complexity.

Such families of grammars do exist. One such was found by John Reynolds and was reported by Earley (1968). When n is a positive integer, the n'th member of this family is denoted by EXP(n) and is defined by the following production schema :

$S \rightarrow A_i$	$(1 \leqslant i \leqslant n)$
$A_i \rightarrow c_j A_i$	$(1 \leqslant i,j \leqslant n,\ i \neq j)$
$A_i \rightarrow c_i B_i$	$(1 \leqslant i \leqslant n)$
$A_i \rightarrow d_i$	$(1 \leqslant i \leqslant n)$
$B_i \rightarrow c_j B_i$	$(1 \leqslant i,j \leqslant n)$
$B_i \rightarrow d_i$	$(1 \leqslant i \leqslant n)$

For each $n > 0$, the grammar EXP(n) is LR(0) and its size is O(n). However, the cardinality of the LR(0) stateset for EXP(n) is $O(n.2^n)$. Since, for any grammar G and any $k > 0$, the cardinality of the LR(k) stateset for G is at least as great as that of its LR(0) stateset, this family of grammars demonstrates that our second method of testing for the LR(k) property is of exponential complexity for all values of k.

In the next section we consider a method of testing for the LR(k) property which does not suffer from this disadvantage.

2.4. Testing for the LR(k) Property - Part 3.

We have seen that our second method of testing for the LR(k) property is of exponential complexity. In contrast, the first method we presented can be shown to have polynomial complexity. By combining techniques from each of these two methods it is possible to derive a practical algorithm which tests for the LR(k) property in time $O(n^{3k+3})$, where n is the size of the grammar under test. By refining the technique, Hunt et al. (1974) showed that the complexity of this algorithm can be reduced to $O(n^{2k+2})$ and by means of additional refinements they showed (1975) that it can be reduced still further - to only $O(n^{k+2})$. We shall present the $O(n^{3k+3})$ algorithm since it embodies most of the principal ideas without requiring too extended a development.

Central to this third algorithm is the construction of an extended finite state automaton. The states of this automaton correspond to the LR(k) items of the grammar under test while its input alphabet is the vocabulary of the grammar. Its construction is formally defined thus :

CONSTRUCTION 2.25

When $G = (V_N, V_T, P, S)$ is a grammar and k is a natural number, define the ENFA $M_k^G = (Q, q_0, F, I, \delta)$ as follows :

(a) $Q = I_k^G \cup \{q_0\}$,

(b) F is irrelevant,

(c) $I = V$, and

(d) δ, the transition function is given by :

(i) $\delta(q_0, \Lambda) = \{[S \to .\alpha, \Lambda] \mid S \to \alpha \in P\}$,

(ii) when q is of the form $q = [A \to \theta_1 . B\theta_2, u]$ with $B \in V_N$, then

$$\delta(q, \Lambda) = \{[B \to .\beta, v] \mid B \to \beta \in P \text{ and } v \in FIRST_k(\theta_2 u)\},$$

(iii) when q is of the form $q = [A \to \theta_1 . X\theta_2, u]$ with $X \in V$, then

$$\delta(q, X) = \{[A \to \theta_1 X . \theta_2, u]\}. \quad \square$$

It may be seen that the type (iii) transitions of M_k^G perform the role of the function NEXT used in the previous section : for each LR(k) item Δ and each $X \in V$ we have $\delta(\Delta, X) = NEXT(\{\Delta\}, X)$.

Similarly, the type (ii) transitions effectively perform the CLOSURE operation. Consequently, when the domain of δ is extended to $Q \times I^*$ in the usual way, we obtain the following result :

LEMMA 2.26

Let $\theta \in V^*$. Then in M_k^G we have $\delta(q_0, \theta) \cap I_k^G = V_k^G(\theta)$. $\square$

This lemma may be proved formally by a straightforward induction on the length of θ. (The intersection with I_k^G is needed simply to discard the start state q_0.)

It follows from Lemma 2.26 that we could use M_k^G to evaluate S_k^G if we so wished - but this would be to no advantage, since it is the size of S_k^G which causes our previous algorithm to have exponential complexity. Instead, we shall seek to compute the set $PAIRS_k^G$ which is defined as follows :

DEFINITION 2.27

$PAIRS_k^G = \{(\Delta, \Sigma) \mid \Delta$ and Σ are LR(k) items for G such both $\Delta, \Sigma \in V_k^G(\theta)$ for some $\theta \in V^*\}$. □

That is to say, $PAIRS_k^G$ contains all pairs of LR(k) items which are simultaneously valid for some viable prefix of G. Clearly, as a corollary to Theorem 2.18, we have :

THEOREM 2.28

G is LR(k) if and only if $PAIRS_k^G$ contains no inadequate members. □

Observe that the cardinality of $PAIRS_k^G$ is polynomial in the size of the grammar G (it is $O(n^{2k+2})$) and so we avoid the problem that was the downfall of our previous algorithm.

Now given any ENFA $M = (Q, q_0, F, I, \delta)$ we may define the set STATE-PAIRS(M) to be the set of all pairs of states which are simultaneously accessible from the start state. That is :

DEFINITION 2.29

$STATE\text{-}PAIRS(M) = \{(p,q) \in Q \times Q \mid$ there exists $\theta \in I^*$ such that both $p, q \in \delta(q_0, \theta)\}$. □

Then, by virtue of Lemma 2.26, we have :

LEMMA 2.30

$PAIRS_k^G = STATE\text{-}PAIRS(M_k^G) \cap I_k^G \times I_k^G$. □

Thus, in order to arrive at a new method of testing for the LR(k) property, all we need is an algorithm for evaluating the set STATE-PAIRS(M_k^G). It is, of course, perfectly possible to give an algorithm which will evaluate STATE-PAIRS(M) for an arbitrary ENFA M. However, we are only interested in applying the algorithm to automata of the type given by Construction 2.25 and by exploiting the particular form taken by these automata we can obtain a specialized algorithm which is more efficient than a fully general one.

Our algorithm will use the fact that the nondeterminism in the automaton M_k^G is of a restricted form; although a given state may have several Λ-transitions, there is at most one transition defined on a symbol other than Λ. If a state q does have a transition defined on a symbol from V then, since this symbol must be unique, we may unambiguously refer to it as the 'OUTSYM' of q. States which have only Λ-transitions may be said to have an undefined OUTSYM. Formally, we define OUTSYM as a function thus :

DEFINITION 2.31

Let q be a state of M_k^G. If $q = q_0$ (the special start state) or if q is a final LR(k) item then OUTSYM(q)= ϕ (i.e. 'undefined'). Otherwise q can be written in the form $q = [A \rightarrow \theta_1.X\theta_2, u]$ and in this case we define OUTSYM(q) = X. (Note that X is the unique symbol such that $\delta(q,X) \neq \emptyset$.) □

We may now present an algorithm (which is a slightly adapted version of one due to Hunt et al. (1974)) for evaluating the set STATE-PAIRS(M_k^G).

ALGORITHM 2.32

Evaluation of STATE-PAIRS(M_k^G).

Input : The ENFA $M_k^G = (Q, q_0, F, I, \delta)$.

Output : The set STATE-PAIRS(M_k^G).

Method : The set STATE-PAIRS(M_k^G) will be built up in a $|Q| \times |Q|$ bit matrix called PAIRS, which is indexed by the states of M_k^G. When the algorithm terminates, PAIRS [p, q] will contain 1 if and only if $(p,q) \in$ STATE-PAIRS(M_k^G). A stack, called STACK, is used to contain backlogged pairs (p,q); that is those pairs of states for which PAIRS [p,q] = 1 has been found, but whose successors have not been examined. Initially, PAIRS contains all zeroes, and STACK is empty.

```
procedure INSERT(p,q);
      if PAIRS [p,q] = 0 then
            PAIRS [p,q] : = 1;
            push (p,q) onto STACK
      endif
end  INSERT;
begin comment main algorithm;
     INSERT (q0,q0);
1.   while   STACK is not empty do
2.           pop (p,q) from STACK;
3.           for each q' ∈ δ(p,Λ) do INSERT(p,q') endfor;
4.           for each p' ∈ δ(q,Λ) do INSERT(p',q) endfor;
5.           set X = OUTSYM(p) and Y = OUTSYM(q);
6.           if X ≠ φ and Y ≠ φ and X = Y then
                                  INSERT( δ (p,X), δ (q,Y))
             endif
     endwhile
end. □
```

It should be clear that Algorithm 2.32 computes STATE-PAIRS(M_k^G) correctly. Now let us consider the complexity of the algorithm. We will examine the amount of work charged to each of the numbered steps. Since no (ordered) pair of states is ever stacked twice, steps 1,2,5 and 6 will have work amounting to at most $O(|Q|^2)$ charged to them. In the worst case, steps 3 and 4 will be executed once for every pair $(p,q) \in Q \times Q$. Or to put it another way, these steps will be executed at most $|Q|$ times for each state $p \in Q$. Thus the total work performed in each of steps 3 and 4 is $O(|Q| \cdot \sum_{p \in Q} |\delta(p, \Lambda)|)$ - that is the cost of traversing each of the " Λ - transition lists" $|Q|$ times. Thus the total cost of Algorithm 2.32 is $O(|Q|^2 + |Q| \cdot \sum_{p \in Q} |\delta(p, \Lambda)|)$.

Now consider the cost of applying Algorithm 2.32 when the size of the grammar G is n. We will have :

$$|Q| = O(n^{k+1}) \text{ , and at worst}$$

$$\sum_{p \in Q} |\delta(p, \Lambda)| = O(|Q|^2) = O(n^{2k+2}).$$

Consequently we have :

THEOREM 2.33

When G is a grammar and SIZE (G) = n, the cost of applying Algorithm 2.32 to M_k^G is $O(n^{3k+3})$. □

We can now give our third proof of Theorem 2.8

PROOF OF THEOREM 2.8 - THIRD METHOD

Construct M_k^G and apply Algorithm 2.32 in order to compute STATE-PAIRS (M_k^G) . Then use Lemma 2.30 to construct the set $PAIRS_k^G$ and test each member of this set for adequacy. By virtue of Theorem 2.28, G will be LR(k) if and only if $PAIRS_k^G$ contains no inadequate member. We claim that the overall running time of this testing procedure is dominated

by the time taken by Algorithm 2.32. Thus the complexity of this third method of testing for the LR(k) property is $O(n^{3k+3})$, where n is the size of the grammar under test. □

In order to illustrate these ideas, we once again show that grammar G1 is not LR(1). The state - diagram of M_1^{G1} is given in the following diagram, Figure 2.2. Note that we omit states which cannot be reached from the start - state.

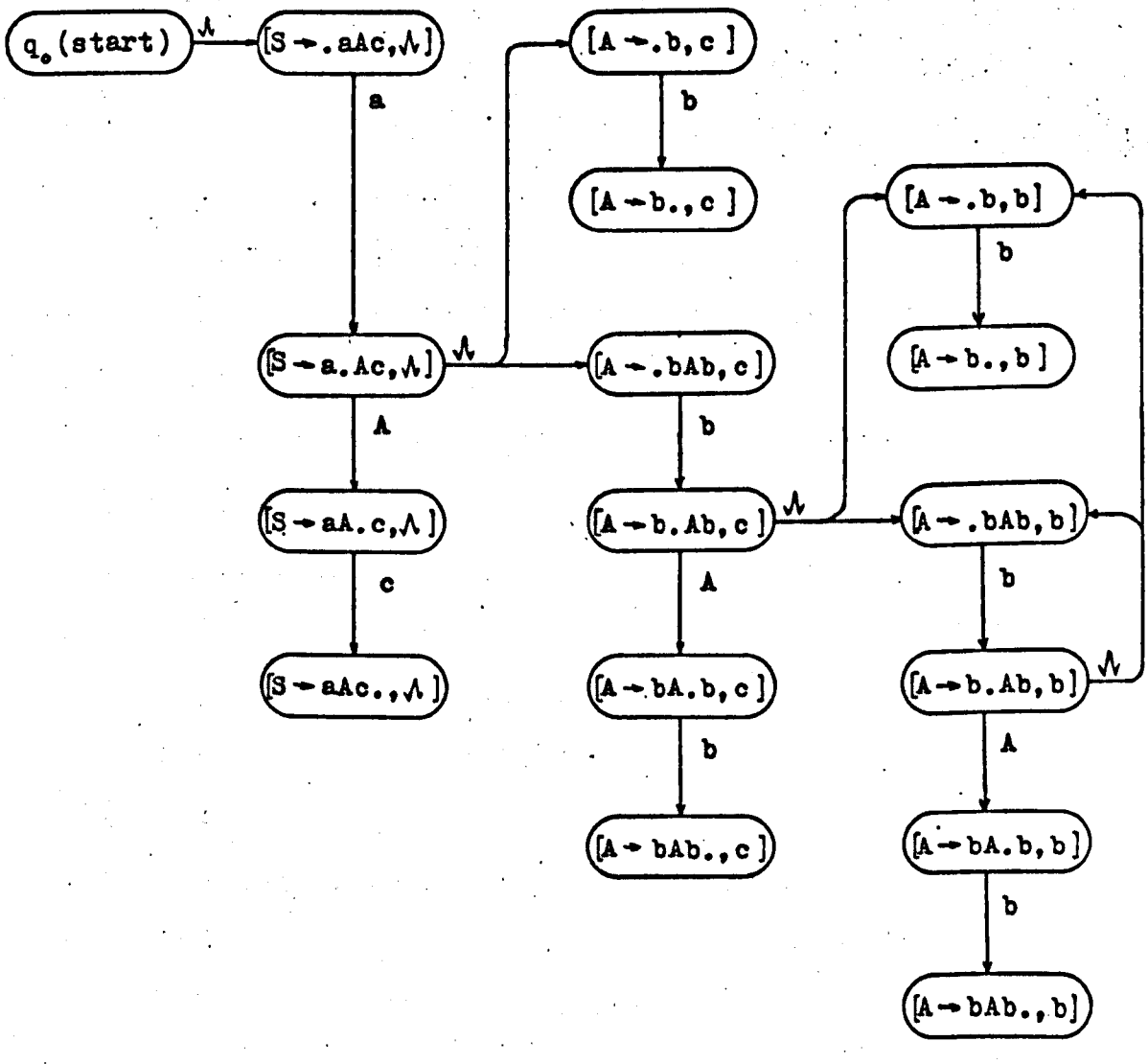


Figure 2.2 : The Transition Diagram of M_1^{G1} - the ENFA Corresponding to Grammar G1 when k = 1.

It may be seen that the input string abb takes the automaton M_1^{G1} from the start-state to the set of states : $\{[A \rightarrow .bAb, b], [A \rightarrow b.Ab, b], [A \rightarrow .b, b], [A \rightarrow b., b]\}$. Thus, in particular, both $[A \rightarrow .b, b], [A \rightarrow b., b] \in PAIRS_1^{G1}$ and since these items conflict, it follows that G1 is not LR(1).

The improved algorithm of Hunt et al.(1974), which has complexity $O(n^{2k+2})$, is similar to the one described in this section but constructs the automaton M_k^G rather more carefully. The dominant factor in the complexity of Algorithm 2.32 is the $O(n^{2k+2})$ term due to the number of ν - transitions in M_k^G. At the expense of adding a number of special states to the automaton (but not so many that the total number of states rises above $O(n^{k+1})$), the number of ν - transitions may be reduced from $O(n^{2k+2})$ to $O(n^{k+1})$. Using this modified form of automaton, the cost of applying Algorithm 2.32 is reduced to $O(n^{2k+2})$.

The fastest known algorithm for testing the LR(k) property, also due to Hunt et al.(1975), works slightly differently. Instead of constructing a single automaton M_k^G which is then used to find all the conflicting pairs of LR(k) items, this method constructs many separate automata. Each automaton is used to find those pairs of LR(k) items which are in conflict for a particular look-ahead string. The cost of applying Algorithm 2.32 to each individual automaton is $O(n^2)$, and since there are $|V_T^{*k}| = O(n^k)$ different look-ahead strings to consider (and therefore $O(n^k)$ separate automata), the overall complexity of this method is $O(n^{k+2})$.

2.5. Parsing the LR(k) Grammars.

Now that we know how to test whether a grammar is LR(k), we may proceed to describe a method for parsing the LR(k) grammars. The method is one of those modelled by Algorithm 1.4 and is distinguished from other such methods by the particular form of its parsing tables. These are known, naturally enough, as 'LR(k) parsing tables' and the parsing algorithm which results when these are used to drive Algorithm 1,4 is known as the 'LR(k) parsing algorithm'.

Recall that a set of parsing tables for Algorithm 1.4 are a 4-tuple $T = (Q, s_o, g, f)$ where Q is set of parsing states, s_o is a distinguished initial state, g is a parsing goto function, and f is a parsing action function. Throughout this section we will suppose that $G = (V_N, V_T, P, S)$ is an LR(k) grammar and we will show that, given G, its LR(k) stateset, and a tabulation of its GOTO function (both provided by Algorithm 2.23), it is always possible to construct a set of parsing tables to drive Algorithm 1.4 correctly.

In order to do this, it is necessary first of all to construct a suitable set of parsing states Q - and to do this we need to partition the viable prefixes of G into a set of equivalence classes and then take each of these equivalence classes to correspond to a parsing state in Q. The technique which is employed in LR(k) tables is to assign viable prefixes to the same equivalence class if they share the same LR(k) state. That is a pair of viable prefixes, say θ and ψ, belong to the same class if (and only if) $V(\theta) = V(\psi)$. In other words, the parsing states

in Q are identified with the LR(k) states in S_k^G. Under this construction, the LR(k) states for G may be regarded as names for equivalence classes of VP^G and it should be clear that this construction does provide an acceptable way of imposing a finite partition over VP^G. There is one small, but vital point here, however, which must not be overlooked. The parsing states in Q are assumed to be simple objects bearing no information in themselves, whereas the LR(k) states in S_k^G are complex objects, being composed of LR(k) items, and carry a considerable information content. Therefore we do not identify the set Q with S_k^G directly; instead we identify it with a set composed of the names of members of S_k^G, where names are supposed to be simple objects bearing no information other than their own identity. This distinction between LR (k) states and their names is not entirely frivolous and will prove significant during constructions which appear in Chapter 4. In order to make this notion of naming precise and uniform, we introduce the following definition :

DEFINITION 2.34

Let X be any finite non-empty set. Then NAMES(X) is an alphabet with the same cardinality as X and $NAMEOF_X$: $X \rightarrow$ NAMES (X) is assumed to be a fixed bijection taking members of X into their'names'. We also allow the function $NAMEOF_X$ to be applied to arguments which are not members of X and in this case the function value is always 'undefined'. When the identity of the set X is clear we omit the subscript from the function $NAMEOF_X$. By convention, we always assume that distinct sets X and Y give rise to disjoint alphabets NAMES(X) and NAMES(Y). □

We may now describe the construction of the parsing tables which drive the LR(k) parsing algorithm.

CONSTRUCTION 2.35

First we need a subsidiary definition. Let Δ be any set of LR(k) items for G and let $u \in V_T^{*k}$. Define the value of the function ACTION (Δ, u) to be :

(a) SHIFT if Δ contains an item $[B \rightarrow \beta_1 . \beta_2, v]$ where $\beta_2 \neq \Lambda$ and $u \in EFF_k(\beta_2 v)$

(b) REDUCE q if Δ contains the item $[A \rightarrow \alpha ., u]$ where $A \rightarrow \alpha$ is production q,

(c) ERROR if neither case (i) nor case (ii) applies.

Note that ACTION(Δ, u) will be multi-valued if (and only if) the set Δ is inadequate. Nowwe can give the main construction.

The LR(k) parsing tables for G are denoted by T_k^G and are given by $T_k^G = (Q, s_o, g, f)$ where :

(i) $Q = NAMES(S_k^G)$,

(ii) $s_o = NAMEOF(V(\Lambda))$,

(iii) for each $\Delta \in S_k^G$ and $X \in V$,
$g(NAMEOF(\Delta), X) = NAMEOF(GOTO(\Delta, X))$, and

(iv) for each $\Delta \in S_k^G$ and $u \in V_T^{*k}$,
$f(NAMEOF(\Delta), u) = ACTION(\Delta, u)$. □

Note that the empty set $\emptyset$ is not a member of S_k^G and therefore, according to Definition 2.34, NAMEOF($\emptyset$) is undefined. This means that if GOTO(Δ,X) = $\emptyset$ then, by part (iii) of the definition above, the value of g(NAMEOF(Δ),X) will be undefined. This is intentional and is consistent with the general notion of parsing tables given in Section 1.7 where, it may be remembered, the parsing goto function g was expressly permitted to be a partial function. The action function f must, in contrast, be total and it may be seen that part (iv) of the definition above ensures that this is so. Also note that because G is required to be LR(k) it follows from Theorem 2.18 that every LR(k) state in S_k^G must be adequate. It is this property which ensures that the parsing action function is truly a function; that is, it is single-valued. If G were not LR(k) then we could still construct the tables T_k^G but f(NAMEOF(Δ),u) would be multi-valued for some $\Delta \in S_k^G$ and $u \in V_T^{*k}$. If Δ contained a shift/reduce conflict on lookahead u, then the value of f(NAMEOF(Δ),u) would be simultaneously both SHIFT and REDUCE q for some production q. Similarly, in the case of a reduce/reduce conflict, the function value would be both REDUCE p and REDUCE q for dissimilar productions p and q. These observations explain the choice of terminology used for the two types of conflict that can occur.

Before discussing the theoretical properties of the LR(k) parsing algorithm, we will work through an illustrative example. For this illustration we will use a rather more realistic grammar than those encountered previously. The productions of this grammar are listed below :

1	S	⟶ E	
2	E	⟶ E + T |	(Grammar G3)
3		T	
4	T	⟶ T + P |	
5		P	
6	P	⟶ (E) |	
7		X	

Grammar G3 is a model fragment from a conventional ALGOL-60 type programming language grammar. It generates a language consisting of simple arithmetic expressions involving the two operators, + and *, and a single operand X. The grammar causes * to have higher precedence than + and parentheses are available to override the normal order of evaluation. Using Algorithm 2.23 the LR(1) stateset and GOTO function for this grammar may be computed and the result is displayed in Figure 2.3. Note that collections of LR(1) items which differ from one another only in their second components are abbreviated by writing them as single 'compound' items. Thus, for example, the three items $[T \rightarrow .P,)]$, $[T \rightarrow .P,+]$ and $[T \rightarrow .P,*]$ are combined and written as $[T \rightarrow .P,),+,*]$. The LR(1) states appearing in Figure 2.3 are assigned integer names (the 'state number') and entries in the GOTO portion of the table refer to states via their numbers. Strictly speaking, this means that it is really the parsing goto function g rather than GOTO which is displayed in the

figure. Clearly,the distinction between GOTO and g is rather a fine one, existing only for expository rather than practical reasons, and no real confusion is caused by the form of presentation used in Figure 2.3.

STATE No.	LR(1) STATES NUCLEUS	COMPLETION	GOTO S	E	T	P	(	X	)	*	+
1	[S→.E,Λ]	[E→.E+T,Λ,+] [E→.T,Λ,+] [T→.T*P,Λ,+,*] [T→.P,Λ,+,*] [P→.(E),Λ,+,*] [P→.X,Λ,+,*]		2	3	4	5	6			
2	[S→E.,Λ] [E→E.+T,Λ,+]										7
3	[E→T.,Λ,+] [T→T.*P,Λ,+,*]									8	
4	[T→P.,Λ,+,*]										
5	[P→(.E),Λ,+,*]	[E→.E+T,),+] [E→.T,),+] [T→.T*P,),+,*] [T→.P,),+,*] [P→.(E),),+,*] [P→.X,),+,*]		9	10	11	12	13			
6	[P→X.,Λ,+,*]										
7	[E→E+.T,Λ,+]	[T→.T*P,Λ,+,*] [T→.P,Λ,+,*] [P→.(E),Λ,+,*] [P→.X,Λ,+,*]			14	4	5	6			
8	[T→T*.P,Λ,+,*]	[P→.(E),Λ,+,*] [P→.X,Λ,+,*]				15	5	6			
9	[P→(E.),Λ,+,*] [E→E.+T,),+]								16		17
10	[E→T.,),+] [T→T.*P,),+,*]									18	
11	[T→P.,),+,*]										
12	[P→(.E),),+,*]	[E→.E+T,),+] [E→.T,),+] [T→.T*P,),+,*] [T→.P,),+,*] [P→.(E),),+,*] [P→.X,),+,*]		19	10	11	12	13			
13	[P→X.,),+,*]										
14	[E→E+T.,Λ,+] [T→T.*P,Λ,+,*]									8	
15	[T→T*P.,Λ,+,*]										
16	[P→(E).,Λ,+,*]										
17	[E→E+.T,),+]	[T→.T*P,),+,*] [T→.P,),+,*] [P→.(E),),+,*] [P→.X,),+,*]			20	11	12	13			
18	[T→T*.P,),+,*]	[P→.(E),),+,*] [P→.X,),+,*]				21	12	13			
19	[P→(E.),),+,*] [E→E.+T,),+]								22		17
20	[E→E+T.,),+] [T→T.*P,),+,*]									18	
21	[T→T*P.,),+,*]										
22	[P→(E).,),+,*]										

Figure 2.3. : The LR(1) Stateset and GOTO Function forGrammar G3

It can be seen from Figure 2.3 that no inadequacies are present in the LR(1) stateset for G3 and so the grammar is LR(1). The parsing tables T_1^{G3} may therefore be constructed and these are given in Figure 2.4. using the

conventions for displaying parsing tables which were established for Figure 1.2.

STATE	ACTION FUNCTION						GOTO FUNCTION								
NO.	Λ	(	X	)	*	+	S	E	T	P	(	X	)	*	+
1		sh	sh					2	3	4	5	6			
2	1					sh									7
3	3				sh	3								8	
4	5				5	5									
5		sh	sh					9	10	11	12	13			
6	7				7	7									
7		sh	sh						14	4	5	6			
8		sh	sh							15	5	6			
9				sh		sh							16		17
10				3	sh	3								18	
11				5	5	5									
12		sh	sh					19	10	11	12	13			
13				7	7	7									
14	2				sh	2								8	
15	4				4	4									
16	6				6	6									
17		sh	sh						20	11	12	13			
18		sh	sh							21	12	13			
19				sh		sh							22		17
20				2	sh	2								18	
21				4	4	4									
22				6	6	6									

Figure 2.4 : T_1^{G3} - the LR(1) Parsing Tables for Grammar G3.

In order to illustrate the behaviour of the LR(k) parsing algorithm we display in Figure 2.5. the various moves executed by the LR(1) parser for G3 (that is to say Algorithm 1.4 driven by the tables of Figure 2.4) while processing the string X*(X+X).

MOVE NO.	SYMBOL STACK CONTENTS	STATE STACK CONTENTS	UNCONSUMED INPUT	ACTION
1	↯	1	X*(X+X)	SHIFT
2	X	1,6	*(X+X)	REDUCE P→X
3	P	1,4	*(X+X)	REDUCE T→P
4	T	1,3	*(X+X)	SHIFT
5	T*	1,3,8	(X+X)	SHIFT
6	T*(	1,3,8,5	X+X)	SHIFT
7	T*(X	1,3,8,5,13	+X)	REDUCE P→X
8	T*(P	1,3,8,5,11	+X)	REDUCE T→P
9	T*(T	1,3,8,5,10	+X)	REDUCE E→T
10	T*(E	1,3,8,5,9	+X)	SHIFT
11	T*(E+	1,3,8,5,9,17	X)	SHIFT
12	T*(E+X	1,3,8,5,9,17,13	)	REDUCE P→X
13	T*(E+P	1,3,8,5,9,17,11	)	REDUCE T→P
14	T*(E+T	1,3,8,5,9,17,20	)	REDUCE E→E+T
15	T*(E	1,3,8,5,9	)	SHIFT
16	T*(E)	1,3,8,5,9,16	↯	REDUCE P→(E)
17	T*P	1,3,8,15	↯	REDUCE T→T*P
18	T	1,3	↯	REDUCE E→T
19	E	1,2	↯	REDUCE S→E and ACCEPT

Figure 2.5. : The Behaviour of the LR(1) Parser for Grammar G3 with Input X*(X+X).

From Figure 2.5 we see that the input string X*(X+X) is accepted by the algorithm and, anticipating results which are given shortly, we claim that this means that the string is a valid sentence of Grammar G3. By inspecting the 'REDUCE' entries in the "ACTION" column of Figure 2.5 we can construct the parse tree shown in Figure 2.6.

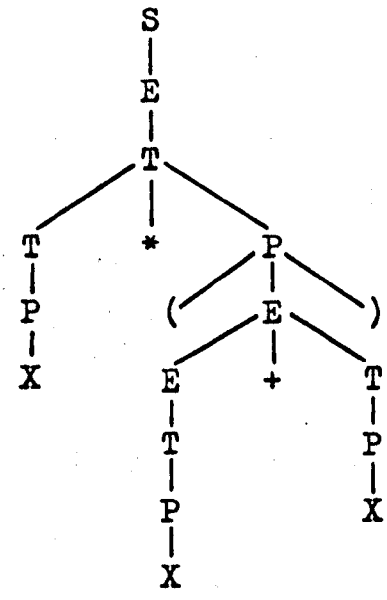


Figure 2.6. : The Parse Tree of X*(X+X) with respect to Grammar G3.

For comparison, we display in Figure 2.7 the behaviour of this parser when presented with the invalid input X(X+X).

MOVE NO.	SYMBOL STACK CONTENTS	STATE STACK CONTENTS	UNCONSUMED INPUT	ACTION
1		1	X(X+X)	SHIFT
2	X	1,6	(X+X)	ERROR

Figure 2.7: The Behaviour of the LR(1) Parser for GrammarG3 with input X(X+X).

From this figure we see that the input X(X+X) is rejected by the LR(1) parser for G3 at the earliest possible moment, that is as soon as the substring it has seen so far (i.e. the two symbols "X(") fails to be a prefix to any valid sentence of G3.

Fortified by this example, we now turn to a theoretical examination of the properties of the LR(k) parsing algorithm. The following theorem assures us that the algorithm performs correctly when presented with valid input.

THEOREM 2.36

The LR(k) parsing algorithm parses all sentences correctly.

PROOF. In the discussion which followed the introduction of Algorithm 1.4 we deduced conditions which its parsing tables must satisfy if the algorithm is to parse sentences correctly. In order to prove that the LR(k) parsing algorithm performs properly it is therefore necessary to prove that its parsing tables $T_k^G = (Q, s_o, g, f)$ satisfy these conditions.

First note that, since the grammar G is supposed to be LR(k), its LR(k) stateset can contain no inadequacies and so the parsing action function f of Definition 2.35 is single-valued. Next we must construct a surjective mapping EQUIV : $VP^G \rightarrow Q$ and show that :

(i) EQUIV (Λ) = s_o,

(ii) whenever $\theta, \psi \in VP^G$ and $X \in V$ are such that

  (a) EQUIV(θ) = EQUIV(ψ) and

  (b) both $\theta X, \psi X \in VP^G$, then
  EQUIV(θX) = EQUIV(ψX),

(iii) whenever $\theta \in VP$ and $X \in V$ are such that $\theta X \in VP^G$ then g(EQUIV(θ),X) = EQUIV(θX), and finally

(iv) whenever θx is an rsf of G with a handle (q,m) satisfying $m \geq len(\theta)$, then the value of f(EQUIV(θ),k:x) is :

  (a) REDUCE q if m = len(θ), and

  (b) SHIFT if m > len(θ).

Now in the case of LR(k) parsing tables we can construct a suitable function EQUIV by the definition

$$\text{EQUIV}(\theta) = \text{NAMEOF}(V(\theta)).$$

Then, since Definition 2.35 specifies that $s_0 = \text{NAMEOF}(V(\Lambda))$, we see that condition (i) above is satisfied immediately by this construction. Condition (ii) is equally obvious and to establish condition (iii) it is only necessary to note that Definition 2.35 gives $g(\text{NAMEOF}(V(\theta)),X) = \text{NAMEOF}(\text{GOTO}(V(\theta),X))$ and that Theorem 2.20 gives $V(\theta X) = \text{GOTO}(V(\theta),X)$. Hence $g(\text{NAMEOF}(V(\theta)),X) = \text{NAMEOF}(V(\theta X))$ as required.

In order to prove the final condition (iv), suppose first that $m = \text{len}(\theta)$ and let production q be $A \rightarrow \alpha$. Then since (q,m) is a handle for θx there must be a derivation in G with the form $S \underset{R}{\rightarrow}^{*} \mu Ax \underset{R}{\rightarrow} \mu\alpha x$ where $\mu\alpha = \theta$. If we put $u = k{:}x$ then this derivation implies that the item $[A \rightarrow \alpha.,u]$ is valid for θ and therefore $f(\text{NAMEOF}(V(\theta)),u) = \text{REDUCE q}$ as required. On the other hand, if $m > \text{len}(\theta)$, then we may use Lemma 2.17 to see that $V(\theta)$ must contain a non-final item of the form $[B \rightarrow \beta_1.\beta_2, v]$ with $\text{EFF}(x) \subseteq \text{EFF}(\beta_2 v)$. Since x is a terminal string, we have $\text{EFF}(x) = \{u\}$ and so $f(\text{NAMEOF}(V(\theta)),u) = \text{SHIFT}$ as required to complete the proof. □

Not only does the LR(k) parsing algorithm parse sentences correctly, but it does so in linear time.

THEOREM 2.37

The number of moves made by the LR(k) parsing algorithm while parsing a sentence of length n is $O(n)$. □

This result is proved by Aho and Ullman (1972a, Theorem 5.13).

Note that Theorem 2.37 expresses the running time of the LR(k) parsing algorithm in terms of its own primitive operations, not of time directly. However, it is clear that each move made by an LR(k) parser can be performed in fixed time by any reasonable model of computation and so the total time taken is indeed linear in the length of the input. The only workspace used by the LR(k) parsing algorithm is that needed for the parse and state stacks and since no individual move can add more than one symbol to each of these stacks it follows from Theorem 2.37 that the space used by the algorithm is also linear in the length of the input.

All that remains now is to examine the ability of the LR(k) parsing algorithm to detect and reject all those inputs which are not valid sentences of the grammar concerned. It can be seen from the description of Algorithm 1.4 that there are five situations in which the LR(k) parsing algorithm can reject its input. These are when

(i) The unconsumed input is found to be empty during a shift move (part (i) of sub-step 3 (a)),

(ii) the parsing goto function yields an undefined value during a shift move (part (iv) of sub-step 3 (a)),

(iii) The symbol stack is found to contain too few symbols during a reduce move (part (i) of sub-step 3 (b)),

(iv) the parsing goto function yields an undefined value during a reduce move (part (vi) of sub-step 3 (b)), and

(v) the parsing action function yields the value ERROR (sub-step 3 (c)).

There is a fundamental difference in the error detection behaviour of the LR(k) parsing algorithm between the cases

$k > 0$ and $k = 0$. The LR(0) action function can never have the value ERROR and so all error detection in the LR(0) parsing algorithm is performed in the first four of the situations listed above. (Actually, situation (iii) cannot occur with any LR(k) parser.) When $k > 0$, however, not only is ERROR in the range of the action function, but it can be shown that <u>all</u> syntactic errors are caught by this mechanism, This means, incidentally, that the efficiency of the LR(k) parsing algorithm may be improved when $k > 0$ by removing from Algorithm 1.4 all those tests which are concerned with detecting situations (i) to (iv) above. We shall concentrate our attention on the case $k > 0$ and in the next lemma we specify the circumstances in which the LR(k) action function has the value ERROR.

<u>LEMMA 2.38</u>

Let $k > 0$ and let $T_k^G = (Q, s_0, g, f)$ be the LR(k) parsing tables for the grammar $G = (V_N, V_T, P, S)$. Also let θ be a viable prefix of G and let $x \in V_T^*$ be a string for which all $y \in V_T^*$ such that θy is an rsf of G satisfy $k{:}x \neq k{:}y$. Then $f(\text{NAMEOF}(V(\theta)), k{:}x) = \text{ERROR}$.

PROOF. Let $u = k{:}x$ and suppose that $f(\text{NAMEOF}(V(\theta)), u) = \text{SHIFT}$. Then according to Definition 2.35 there must be some non-final LR(k) item $[B \rightarrow \beta_1 . \beta_2, v] \in V(\theta)$ such that $u \in \text{EFF}(\beta_2 v)$. Now because $[B \rightarrow \beta_1 . \beta_2, v]$ is valid for θ there must be some derivation in G with the form $S \underset{R}{\rightarrow}^* \mu B z \underset{R}{\rightarrow} \mu \beta_1 \beta_2 z$ with $\mu\beta_1 = \theta$ and $v = k{:}z$. Then, since $u \in \text{EFF}(\beta_2 v)$, we also have $u \in \text{EFF}(\beta_2 z)$ and this means that there exists $y \in V_T^*$ such that $\beta_2 z \underset{\text{REFF}}{\rightarrow}^* y$ and $u = k{:}y$. But because $\theta\beta_2 z$ is an rsf of G, this means that θy is also an rsf of G and since $u = k{:}y$ this contradicts the hypothesis that no such y exists. We

conclude that f (NAMEOF(V(θ)),u) ≠ SHIFT. A similar argument shows that f (NAMEOF(V(θ)),u) ≠ REDUCE q for any q ∈ P and so it follows that value of the function can only be ERROR and the lemma is proved. □

This result will be used shortly to prove that, when k > 0, the LR(k) parsing algorithm rejects all invalid inputs as soon as possible. Before we can do this, it is necessary to be more precise about what we mean by 'as soon as possible'.

DEFINITION 2.39

Let G = (V_N, V_T, P,S) be any grammar and let x ∈ V_T^* be such that x ∉ L(G). Then the error position in x (with respect to G) is denoted by EP^G (x) and is given by $EP^G(x) = len(y)+1$ where y is the longest prefix of x for which some z ∈ V_T^* can be found such that yz ∈ L(G). We will omit the superscript G and write simply EP(x) when no ambiguity can result. □

We claim that the error position is the first point during a strictly left to right scan at which it is possible to determine that a string is not a valid sentence of the grammar concerned. This is because all initial substrings which do not extend as far as the error position are prefixes to valid sentences of the grammar and therefore provide no basis for rejecting the string. Note that this is not the same as saying that the symbol in the error position is "wrong" or that the error was committed at that point. Consider, for example, the following string which is intended to be an ALGOL60 statement :

If X < Y then Z : = 0 ; else Z : = 1 ;

Our experience of ALGOL60 allows us to assert with some certainty that the error in this supposed statement is the presence of the semi-colon *preceding* the else. However, the string up to and including the semi-colon is a valid ALGOL60 statement and so no error can be proclaimed during a left to right scan until the else is encountered. The symbol else occupies the error position in this string because it is the first point at which the error can be detected, even though it is probably neither the source, nor the location, of the true error.

When we speak of the ability of the LR(k) parsing algorithm to detect errors "as soon as possible" we mean "as soon as the symbol in the EP(x)'th position is examined" and we claim that this is the best performance which can be reasonably required of a left to right parsing algorithm. Since the LR(k) parsing algorithm looks k symbols ahead and advances down the input string by one symbol each time a shift move is made, stating that the algorithm detects errors "as soon as possible" is therefore the same as saying that it rejects an invalid input, x say, on the move following the EP(x)-k'th shift move; that is as soon as the symbol (if any) in the EP(x)'th position comes into view. We will now prove this property of the LR(k) parsing algorithm.

THEOREM 2.40

Let $k > 0$ and let $x \in V_T^*$ be a string such that $x \notin L(G)$. Then the LR(k) parsing algorithm rejects x on :

(i) the first move if $EP(x) \leqslant k$, or

(ii) the move following the $EP(x)-k$ 'th shift move if $EP(x) > k$.

PROOF. Suppose that $EP(x) \leqslant k$. Then there can be no $y \in L(G)$ such that $k:x = k:y$ and so, by virtue of Lemma 2.38, we have $f(NAMEOF(V(\Lambda)), k:x) = ERROR$. Since the initial state of the LR(k) parsing algorithm is s_0 and $s_0 = NAMEOF(V(\Lambda))$, this means that x will be rejected on the first move made by the algorithm.

Now consider the case $EP(x) > k$. We can write x in the form $x = yz$ where $len(y) = EP(x)-k$ and the definition of $EP(x)$ means that there can be no $w \in V_T^*$ with $k:z = k:w$ such that $yw \in L(G)$ but there must be some $v \in V_T^*$ with $(k-1):z = (k-1):v$ and $yv \in L(G)$. Consequently, until it has made its $EP(x)-k$'th shift move, the LR(k) parsing algorithm has no way of knowing that its input is in fact yz (ie.x) rather than the valid sentence yv. Certainly therefore, the algorithm cannot reject x before it has made its $EP(x)-k$'th shift move. Immediately after that move the parse stack will contain some viable prefix θ such that $\theta \xrightarrow[R]{*} y$ and the lookahead string will be $u = k:z$. Now there can be no $w \in V_T^*$ such that θw is an rsf of G which also satisfies $k:w = k:z$, for if there were, then yw would be a sentence of G - contradicting the observations made earlier concerning the non-existence of

such a w. It follows, again from Lemma 2.38, that f(NAMEOF(V(θ)),u) = ERROR and since the state on top of the state stack at this time must be NAMEOF(V(θ)) it follows that the algorithm will reject x on its next move. □

As we remarked earlier, the error detection of the LR(0) parsing algorithm is rather different to that of the case k > 0 discussed above. Because the LR(0) parser uses no lookahead, it is not until it has committed itself to a move that this parser is able to inspect the next symbol of its input string. In spite of this, LR(0) parsers do detect and reject all invalid inputs, though not quite a soon as, say, an LR (1) parser would. Although we shall not prove it, it can be shown that when an LR(0) parser is presented with an invalid input string x, it will always halt and reject the string either during or before making its EP(x)'th shift move. This means that an LR(0) parser may make some moves, but only of the reduce type, after an LR(1) parser presented with the same input would have halted, but will itself halt and declare ERROR before making another shift move.

2.6. Summary

By virtue of their very definition, the LR (k) grammars are the largest class which can be parsed deterministically from left to right. This gives them an immediate theorectical appeal which is heightened by the discovery that they generate exactly the deterministic languages (at least they do when $k > 0$). Although the LR(k) property is undecidable in general, we have seen three different algorithms for deciding whether a given grammar is LR(k) for some particular, predetermined, value of k. The method of Section2.2 though conceptually straightfoward and useful for proving Theorem 2.8 was not developed sufficiently to yield a practical algorithm. The methods of Sections 2.3 and 2.4 both yield practical algorithms but the former suffers from the disadvantage that its worst-case complexity is exponential in the size of the grammar under test. However, it seems that for conventional programming language grammars the size of the LR(k) stateset grows only linearly with the size of the grammar (see Purdom (1974)). Thus the poor worst-case performance of this algorithm is unlikely to be a serious drawback in practice. As far as we know, the efficient algorithms of Section 2.4 are untried in practice.

A parser for an LR(k) grammar is easily constructed from its LR(k) stateset. The performance of these parsers is superior in many respects to almost all other bottom up methods. One of the great advantages of the LR(k) parsing algorithm over most of its rivals is the fact that, when a reduce move is called for, an LR(k) parser knows

immediately which production is to be used in the reduction. In contrast, most other bottom up methods know only that a reduce move involving some production is required, and must spend time examining their parse stacks in order to discover the identity of the production to be used. The LR(k) parsing algorithm is therefore, in general, faster (that is it has a smaller constant of proportionality) than other linear-time parsing methods.

Furthermore, the ability of the LR(k) parsing method to detect syntactic errors at the earliest possible moment is vastly superior to the error detection facilities afforded by other bottom up methods. Although, as we have seen, the "error position" within a syntactically incorrect string is not necessarily the point at which the error was committed, it is almost certainly the best place to which to direct the user's attention, and it is also the point at which the contents of the parse stack may best be used to enable the automatic generation of meaningful diagnostic messages and to initiate automatic error recovery procedures.

The price paid for the generality, speed, and excellent error detection of the LR(k) parsing method is in the great size of the tables which drive the parser. It is ultimately the cardinality of the LR(k) stateset for a grammar which determines the size of its LR(k) parsing tables and this value grows dramatically with increasing k. Although LR(0) statesets are quite modest in size, the LR (0) grammars are too restrictive to be useful in practice. The LR(1) grammars, however, appear sufficiently general to model the syntax of most

conventional programming languages. Those programming language grammars which are not LR(1) are quite likely to be ambiguous or to present other difficulties to a human reader. Practical confirmation of the utility of the LR(1) grammars is provided by the fact that many more restricted classes of grammars have found widespread and successful application. Sadly, though, the LR(1) parsing algorithm founders in practice because its parsing tables are intolerably large - the LR(1) state-set for a grammar/of the size ALGOL60 will contain many thousands of states and will give rise to parsing tables requiring tens of thousands of machine words for storage.

Fortunately, methods derived from the LR(k) parsing algorithm (or, more precisely, from the LR(1) algorithm) have been found which reduce the space required by the parsing tables to acceptable proportions, while retaining almost all the advantages of the basic algorithm. These methods are discussed in Chapter 6.

There is one parsing method which does outperform the LR(k) algorithm in one important respect. This is the "operator precedence" method of Floyd (1963), and while it is applicable to only a very restricted class of grammars and affords appalling error detection facilities, it is very fast indeed. The reason for this is that the operator precedence method does not really parse according to the original grammar at all, but parses with respect to an abbreviated "skeletal grammar". This enables it to bypass completely many of the reduction steps needed by conventional parsing algorithms. Since matters are arranged so that these bypassed reductions are without semantic significance, the "sparse parses" produced by the

operator precedence method are just as acceptable for the purpose of translation as ordinary parses. In the next chapter we shall generalize the LR(k) property so that the LR(k) parsing algorithm too may bypass certain reduction steps, thereby obtaining a considerable gain in parsing speed, without sacrificing any of the other attractive properties of this parsing method.

CHAPTER 3.

THE CFLR (k) PROPERTY

In order to motivate the subject matter of this chapter, we invite the reader to consider the parse tree shown in Figure 3.1 which displays a derivation in the programming language EULER (Wirth and Weber, (1966)).

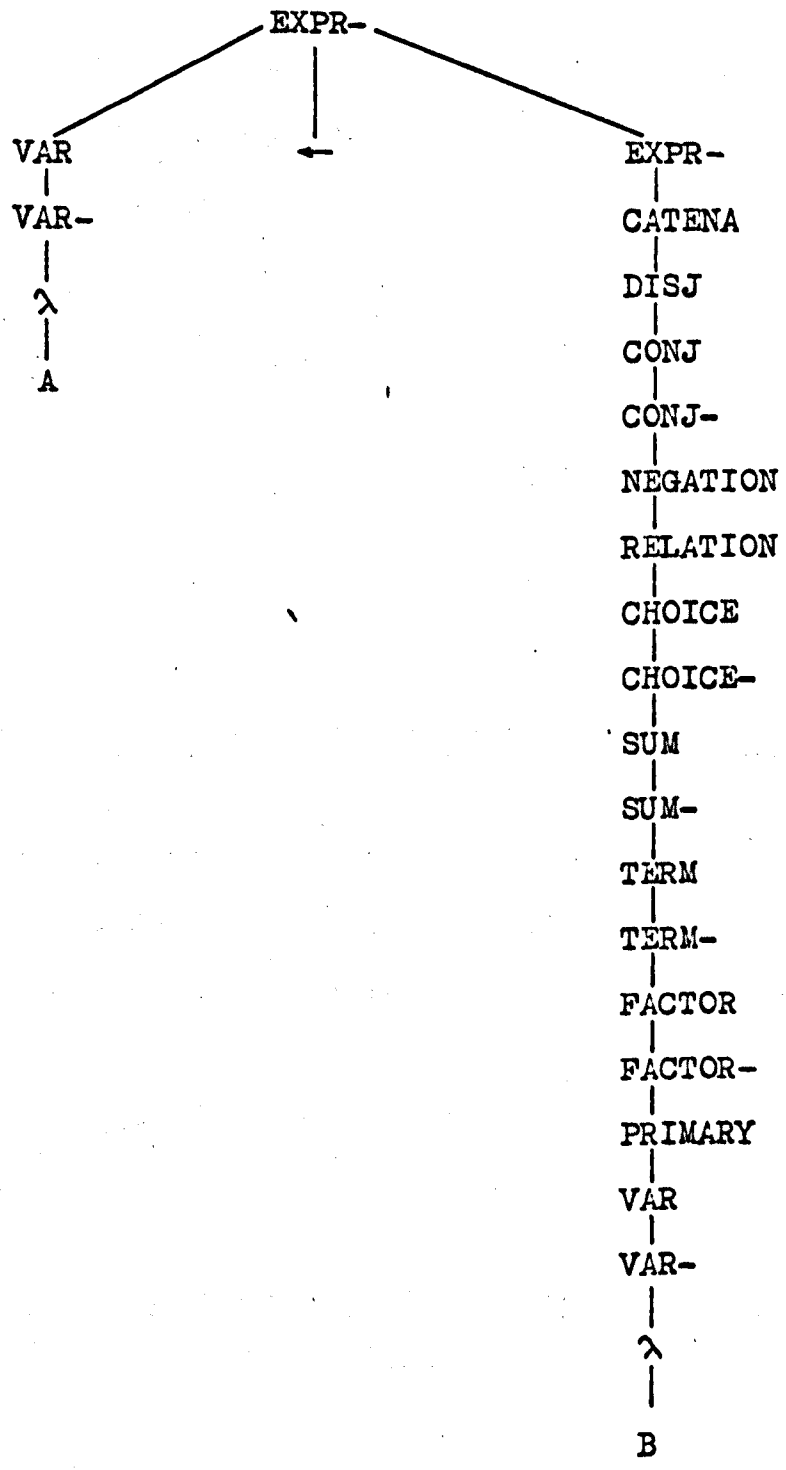


Figure 3.1 : A Derivation from EULER.

The only productions involved in this derivation which have any semantic significance are :

EXPR- ⟶ VAR ← EXPR-

VAR- ⟶ λ

PRIMARY ⟶ VAR

λ ⟶ A

λ ⟶ B

Consequently, the "sparse parse tree" of Figure 3.2 is just as satisfactory for the purpose of translation as that of Figure 3.1.

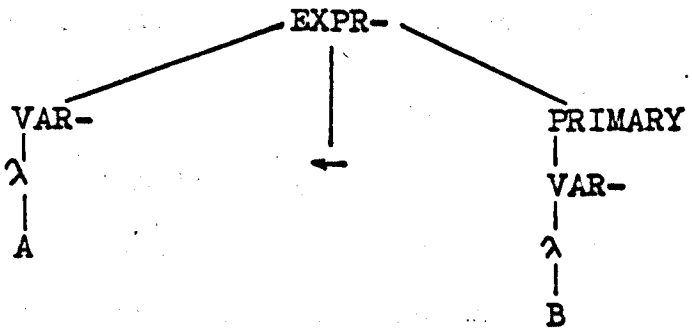


Figure 3.2: A Sparse Parse of the Generation in Figure 3.1.

However, the tree shown in Figure 3.3 is not a satisfactory replacement for that of Figure 3.1 because some derivation steps with semantic significance have been omitted.

Figure 3.3 : An Unsatisfactory Sparse Parse of the Generation in Figure 3.1.

A conventional parsing algorithm for the language EULER would, of course, produce the parse corresponding to Figure 3.1. In particular, an LR(k) parser (or any other shift-reduce bottom up parser) would require 3 shift moves and 23 reduce moves in order to produce that parse. Of the reduce moves, only 6 involve productions with semantic significance; the other 17 (i.e 74% of the total) are of no interest for the purpose of translation and the proportion of the parser's effort which is expended on these moves may be considered wasted. It would be very interesting and useful therefore, to seek parsing algorithms capable of producing the "sparse parse" of Figure 3.2 directly. We would expect such a "sparse parser" to be very much faster than a conventional parser.

Because the LR(k) grammars and their associated parsing algorithm are generally very attractive, it is with them that we shall concentrate our search for sparse parsing techniques. Before proceeding further we need to formalise the notion of a sparse parse. Following Gray and Harrison (1972) we suppose that, independently of context, a production either does or does not have semantic significance. Accordingly we make the following definition.

DEFINITION 3.1

Let $G = (V_N, V_T, P, S)$ be a grammar and $H \subseteq P$. If $\alpha, \beta \in V^*$ satisfy $\alpha \Rightarrow^* \beta$ and $D = \langle (q_i, m_i) \rangle_{i=1}^{n}$ is an explicit derivation of β from α then the H-sparse derivation corresponding to D is :

$$D_H = \langle (q_i, m_i) \mid q_i \in H \rangle_{i=1}^{n}$$

Naturally, when D is an r-derivation, D_H is said to be an

H-sparse r-derivation and P_H, the H-sparse parse corresponding to D, is defined as the sequence of productions which appear in D_H ; that is :

$$P_H = \langle q_i \mid q_i \in H \rangle_{i=1}^{n}. \quad \square$$

The intended interpretation here is that the productions in H have semantic significance while those in $P \setminus H$ do not. An H-sparse derivation is the subsequence of an ordinary derivation which contains only those steps which have semantic signficance. Note that, unlike the case with ordinary derivations, different sentences may share the same sparse derivation.

We would like to discover shift-reduce bottom up parsing algorithms which ignore the productions in $P \setminus H$ totally. However, it is difficult to see how this can be accomplished in general. In particular, if any productions in $P \setminus H$ have degree other than 1, then ignoring reductions by these productions will surely cause the parse stack to have the "wrong length" during the subsequent activity of the parser. If we undertake only to ignore productions of degree 1, then this difficulty at least does not arise. This is because the parse stack in the shift-reduce bottom up parsing method has the same length both before and after reduction by productions of degree 1. Productions of degree 1 and without semantic significance are called "chain" productions (because they partake in long chains of reductions such as that from PRIMARY to EXPR- in Figure 3.1). We make this notion precise and introduce some additional terminology in the next definition.

DEFINITION 3.2

Let $G = (V_N, V_T, P, S)$ be a grammar. The production $A \rightarrow \theta \in P$ is said to be a chain production if

(i) it has no semantic significance, and

(ii) $A \neq S$ and len $(\theta) = 1$.

When $C \subseteq P$ is a set of chain productions, we say that C is a chain set for G and the pair (G,C) is called a chain specified grammar, or cs-grammar for short. If we define $H = P \setminus C$, then the H-sparse derivations in G are more conveniently called the chain free derivations in (G,C). Similarly the H-sparse r-derivations and H-sparse parses in G will be called the chain-free r-derivations and chain free parses respectively. □

Thus a chain-free derivation is a subsequence of an ordinary derivation from which all steps involving chain productions have been deleted.

In future we will often abbreviate "chain-free" to the hyphenated prefix "cf-". Chain-free r-derivations will be called cfr-derivations. When D is a derivation in G we will denote the cf-derivation in (G,C) corresponding to D by D_{cf}. Observe that Definition 3.2 requires that no chain production has the goal symbol as its left part. We make this stipulation because it is a simple way to ensure that no sentence in L(G) has a null cf-derivation. This is convenient for technical reasons. Note that this is the only condition which we place on chain productions. In particular, we do not exclude chain

productions whose right parts are terminal symbols, nor do we preclude sets of chain productions which share the same left part.

Since we have now retreated from our original objective, which was to find a fully general sparse parsing technique, in favour of a lesser goal, namely the discovery of a chain-free parsing method, we should enquire whether the likely benefits remain worthwhile. In practice it seems that programming language grammars contain relatively few chain productions. For example, an ALGOLW grammar containing 183 productions has but 13 chain productions; that is about 7% of the total. However, the frequency of occurrence of chain productions within grammars is not the real issue. What really matters is the frequency with which chain productions appear in derivations. If chain productions tend to occur in the "heavily used" portions of grammars then they may well appear in derivations with considerably greater frequency than their comparatively infrequent appearance within grammars might suggest. This does indeed seem to be the case, for within a programming language grammar chain productions are typically used for two different purposes, both of which concern portions of the grammar which are likely to be heavily used. Firstly, they are used to collect together several syntactic categories under a single heading. In ALGOL60 for example we have :

⟨STATEMENT⟩ ⟶ ⟨UNCONDITIONAL STATEMENT⟩ |
⟨CONDITIONAL STATEMENT⟩ |
⟨FOR STATEMENT⟩ .

Secondly, they are used to enforce precedence among the operators in expressions. In the case of Grammar G3 for instance, the productions E → T and T → P are chain productions which cause parenthesized sub-expressions to be evaluated before unparenthesized ones and also cause the operator * to have higher precedence than +. It is especially this second use of chain productions which causes them to occur disproportionately often in derivations. Measurements by Anderson (1972) on several ALGOLW programs showed that, on average, 70% of all the productions appearing in derivations were chain productions (even though chain productions accounted for only 7% of the productions in the grammar). When an ad-hoc method for bypassing reductions by chain productions was incorporated, Anderson found that the parsing phase of the ALGOLW compiler was speeded up by almost 50% and that total compilation time was reduced by about 15%. Further evidence is provided by Aho and Ullman (1973b) who report a private communication of Horning relating to experiments at Toronto which showed that the parser in the XPL compiler was speeded up about 2½ times when reductions involving chain productions were bypassed. Since compilers are usually well constructed and carefully optimised programs, these improvements are to be regarded as very substantial and worthwhile. It is worth noting that almost all the productions without semantic significance in programming language grammars do turn out to be chain

productions; and those semanticless productions which are not chain productions do not seem to occur all that frequently in derivations. In summary, chain-free parsing is likely to retain almost all the benefit of fully general sparse-parsing and this benefit is considerable.

There have been several attempts to modify the LR(k) parsing algorithm so that it bypasses reductions by chain productions. We have already referred to the work of Anderson, that of Aho and Ullman (1973b) is also notable. These methods suffer from certain practical disadvantages and because of their ad-hoc nature they provide no theoretical insights into the nature and properties of cf-parsing. In contrast, we shall seek to construct LR(k)-type chain-free parsers from "first principles" in the hope of obtaining both a better understanding of the underlying processes and also a better chain-free parsing method.

3.1. Bottom up Chain-Free Parsing.

We begin our investigation of chain-free parsing by modifying some of our existing concepts in order to take account of chain productions. First of all we must revise our idea of syntactic ambiguity. Consider the following grammar :

$$S \longrightarrow A$$
$$A \longrightarrow A \qquad \text{(Grammar G4)}$$
$$A \longrightarrow x$$

This grammar generates the language $\{x\}$ and plainly it is an ambiguous grammar; in fact its single sentence has an infinite number of r-derivations. However, if $A \rightarrow A$ is a chain production, then all the r-derivations of x yield but a single cfr-derivation, namely $\langle (S \rightarrow A,1),\ (A \rightarrow x,\ 1)\rangle$. For the purposes of translation it is only the cfr-derivation which is significant, and since this is unique, the grammar is "as good as" unambiguous. Accordingly we will now consider a grammar to be ambiguous only if some of its sentences possess more than one chain-free r-derivation. The next definition makes this notion precise.

DEFINITION 3.3

The chain specified grammar (G,C) is cf-unambiguous if and only if every sentence in L(G) has a unique cfr-derivation. □

The next result is an obvious corollary to this definition.

THEOREM 3.4

(G,C) is cf-unambiguous if G is unambiguous. □

The next candidate for revision is the concept of a "handle". Recall that the handle of an rsf is effectively the last step to appear in an explicit r-derivation of that rsf. By analogy, we may provisionally define the "chain-free handle" (or "cf-handle" for short) of an rsf to be the last step appearing in some explicit chain-free r-derivation of the rsf. Associated with the concept of a handle was the idea of "reducing" an rsf. Remember that reducing an rsf meant replacing its handle phrase by the left part of its handle production. We may suppose, by analogy, that "cf-reducing" an rsf means replacing its cf-handle phrase by the left part of its cf-handle production. The general bottom up parsing algorithm (Algorithm 1.2) was expressed in terms of these two primitive operations - finding the handle of an rsf and then reducing the rsf. We may tentatively construct a bottom up chain-free parsing algorithm by replacing the terms "handle" and "reduce" in Algorithm 1.2 by "cf-handle" and "cf-reduce" respectively. Let us see how this works in practice.

We will use the grammar G3 given in Section 2.5 and will choose C3 = $\{E \rightarrow T, T \rightarrow P, P \rightarrow X\}$ as the chain set. Suppose we wish to cf-parse the sentence X+X*X. The explicit r-derivation of this sentence is :

$$\langle (S \rightarrow E,1)\ ,\ (E \rightarrow E + T,3)\ ,\ (T \rightarrow T * P,5),$$
$$(P \rightarrow X,5)\ ,\ (T \rightarrow P,\ 3)\ ,\ (P \rightarrow X,\ 3),$$
$$(E \rightarrow T,1)\ ,\ (T \rightarrow P,\ 1)\ ,\ (P \rightarrow X,\ 1) \rangle .$$

The first step of our proposed bottom up cf-parsing algorithm requires that we find the cf-handle of X+X*X. Inspection of the r-derivation above reveals that (T→T*P,5) is the cf-handle of this sentence. We should now cf-reduce the string X+X*X, that is to say we should replace its cf-handle phrase by the left part of its cf-handle production. A problem now arises because we have not specified the notion of a cf-handle phrase sufficiently precisely. If (T→T*P,5) were the ordinary handle of some string, we would expect that string to contain the right part of the handle production, that is T*P, as a substring occupying the 3rd,4th and 5th symbol positions and that occurrence of T*P would be the handle phrase. Here, however, we find the string X*X in the position where we would expect to find the handle phrase. However, X*X is derived from T*P by a sequence of chain productions so let us suppose that the correct interpretation of "cf-handle phrase" here is the substring X*X. Then, suppressing any doubts that this may occasion, it seems that to cf-reduce X+X*X we should replace the substring X*X by the symbol T (that is the left part of the cf-handle production). This operation yields the string X+T and the proposed cf-parsing algorithm now requires us to find the cf-handle of this new string. But at this point everything collapses in disarray. The string X+T is not an rsf of the grammar and so we cannot speak of its r-derivation, let alone its cfr-derivation or its cf-handle. We must re-examine our constructions.

One dubious aspect of our failed cf-parsing attempt was the interpretation of "cf-handle phrase" and the way in which we cf-reduced the string X+X*X. But this is not the real cause of our failure. The true roots of the problem lie in the notions of rsf's and r-derivations, for these are inappropriate to the cf-parsing context. As presently defined, a cfr-derivation is a subsequence of an r-derivation. This means that not only are non-chain productions constrained to be applied right - canonically, but so too are chain productions. Surely there is an inconsistency here. We profess no interest whatsoever in chain productions and do not care to be told whether they are used or not; it must be unnatural, therefore, to require, as we do above, that when chain productions are used then they are used right-canonically. Either we care about chain productions or we do not. We should not take a middle course.

We escape this dilemma by defining a new type of derivation (called a "rorc-derivation") in which non-chain productions must be used right canonically while chain productions are allowed to be used "anywhere". This is the import of the next definition in which we also consolidate some of our earlier provisional definitions and introduce sundry other related concepts and notations.

DEFINITION 3.5

Let (G,C) be a cs-grammar. If $\alpha, \beta \in V^*$ have the form $\alpha = \gamma A \delta$ and $\beta = \gamma X \delta$ where $A \to X \in C$ then we say that α directly chain derives β (with respect to (G,C)) and write $\alpha \to_c \beta$. Clearly $\to_c$ is a relation on V^*. We use it to define further relations on V^* as follows :

(i) $\xrightarrow[RC]{} = \to_R \cap \to_c$,

(ii) $\xrightarrow[RORC]{} = \to_R \cup \to_c$,

(iii) $\xrightarrow[RCF]{} = \to_R \setminus \to_c$.

These relations are pronounced 'directly right-canonically chain derives' ('rc-derives' for short), 'directly right-canonically or chain derives'(' rorc-derives ' for short), and 'directly right-canonically chain free derives' ('rcf-derives' for short) respectively. Their closures are denoted by $\xrightarrow[RC]{}^+$, $\xrightarrow[RORC]{}^*$ etc. in the usual way.

When $A \to \theta$ is production q, $\alpha = \gamma A \delta$, $\beta = \gamma \theta \delta$, and $m = \text{len}(\gamma\theta)$ we extend our previous notation and write :

(i) $\alpha \to_c \beta$ if $q \in C$,

(ii) $\alpha \to_{RC} \beta$ if $q \in C$ and $\delta \in V_T^*$,

(iii) $\alpha \to_{RORC} \beta$ if $q \in C$ or $\delta \in V_T^*$, and

(iv) $\alpha \to_{RCF} \beta$ if $q \in P \setminus C$ and $\delta \in V_T^*$.

Observe that if $\alpha \to_{RORC} \beta$ then $q \in C$ implies $\alpha \to_c \beta$ and $q \in P \setminus C$ implies $\alpha \to_{RCF} \beta$. When $\alpha \xrightarrow[RORC]{}^* \beta$ there must exist a sequence of strings $\langle \psi_i \rangle_{i=0}^{r}$ and sequence of reductions $D = \langle (q_i, m_i) \rangle_{i=1}^{r}$ such that $\alpha = \psi_0 \to_{RORC} \psi_1 \to_{RORC} \psi_2 \cdots \psi_{r-1} \to_{RORC} \psi_r = \beta$.

The sequence D is called an explicit rorc-derivation of β from α (with respect to (G,C)) and we may write $\alpha \xrightarrow{[D]}_{RORC} \beta$. We define explicit c- , rc-, and rcf-derivations similarly.

The rorc-sentential forms of (G,C) (called rorcsf's for short) are the members of the set $\{\alpha \in V^* \mid S \xrightarrow[RORC]{}^* \alpha\}$ When γ is a rorcsf of (G,C) there must exist further rorcsf's α and β and a derivation step (q,m) such that

$$S \xrightarrow[RORC]{}^* \alpha \xrightarrow{(q,m)}_{RCF} \beta \xrightarrow{}_{C}^* \gamma$$

and then (q,m) is said to be a cf-handle of γ. Note that we will have $m/\beta \in V_T^*$, $m/\beta = m/\gamma$, and $len(\beta) = len(\gamma)$. Provided that (q,m) is the only cf-handle of γ we can unambiguously refer to q as the cf-handle production of γ and to m as the cf-handle position in γ. Also the strings $m:\gamma$ and m/γ may be called the left and right contexts of the cf-handle of γ respectively. If production q is $A \to \theta$ then α, β and γ can be written in the form $\alpha = \delta Ax$, $\beta = \delta\theta x$ and $\gamma = \mu\rho x$ where $len(\delta\theta) = m$, $\delta \xrightarrow{}_{C}^* \mu$ and $\theta \xrightarrow{}_{C}^* \rho$. The substring ρ of γ is then called the cf-handle phrase of γ. The act of replacing the cf-handle phrase of a rorcsf by the left part of its cf-handle production is known as cf-reducing the rorcsf. It can be seen that cf-reducing γ above yields the string μAx. Observe that this string is also a rorcsf of (G,C).□

The definitions above are crucial to all the developments which follow. In these developments, the rorc-derivations and rorcsf's will fulfill the roles previously occupied by r-derivations and rsf's. Similarly, the notion of a cf-handle and the operation of cf-reduction will replace those of handles and of ordinary reduction. The vital property of rorcsf's is that they are closed under the operation of cf-reduction. Observe that when the chain set is empty, these notions of rorcsf's, rorc-derivations and cf-handles etc., revert to the corresponding "old" notions of rsf's, r-derivations and ordinary handles. Thus the new theory which we are constructing is a true generalisation of the established theory.

There is one concept, the most fundamental of all, which is still bound to the old ideas. As defined by Definition 3.2, a cfr-derivation is a subsequence of an r-derivation, and we have discovered that r-derivations are an inappropriate notion in the context of cf-parsing. It seems likely that cfr-derivations should be redefined as the chain-free derivations corresponding to rorc-derivations. Happily this redefinition is unnecessary for it is equivalent to the original one. To prove this important fact we first need a technical lemma which says, in effect, that all the steps in a rorc-derivation which are not right-canonical can be pushed to the tail end of the derivation.

LEMMA 3.6

Let (G,C) be a cs-grammar. If $\alpha, \gamma \in V^*$ satisfy $\alpha \xrightarrow[rorc]{}{}^* \gamma$ and D is an explicit rorc-derivation of γ from α then there exists $\beta \in V^*$ such that $\alpha \rightarrow_r^* \beta \rightarrow_c^* \gamma$ and, furthermore, there is an explicit r-derivation E of β from α such that $D_{cf} = E_{cf}$.

PROOF. The proof is by induction on the length of the derivation D. The basis of the induction is the case where D contains no steps at all and is trivial. For the inductive step we assume the result to be true of all rorc-derivations containing n steps ($n \geq 0$) and then suppose that D contains n + 1 steps. We can distinguish the last step in D as (q,m) and write $D = D' \bullet \langle (q,m) \rangle$ where the derivation D' contains n steps. For some $\delta \in V^*$ we will then have :

$$\alpha \xrightarrow{[D']}_{rorc} \delta \xrightarrow{(q,m)}_{rorc} \gamma.$$

Applying the inductive hypothesis to the derivation D', we deduce that there exists $\mu \in V^*$ and an r-derivation E' of μ from α such that $E'_{cf} = D'_{cf}$ and

$$\alpha \xrightarrow{[E']}_r \mu \rightarrow_c^* \delta \xrightarrow{(q,m)}_{rorc} \gamma.$$

There are now two cases to consider according to whether or not q is a chain production.

Case 1 : $q \in C$. Then we have $\delta \xrightarrow{(q,m)}_c \gamma$ and so $\mu \rightarrow_c^* \gamma$.

We may define $E = E'$ and $\beta = \mu$ and the inductive step is complete for this case.

Case 2 : $q \in P \setminus C$. Then we have $\delta \xrightarrow{(q,m)}_{rcf} \gamma$ and so we can write δ and γ in the form $\delta = \eta A x$ and $\gamma = \eta \theta x$ where $A \to \theta$ is production q and $len(\eta\theta) = m$. Since we also have $\mu \overset{*}{\to}_{c} \delta$ it follows that we can write $\mu = \sigma B \pi$ where $\sigma \overset{*}{\to}_{c} \eta$, $B \overset{*}{\to}_{c} A$ and $\pi \overset{*}{\to}_{c} x$. Since $x \in V_T^*$, $\pi \overset{*}{\to}_{c} x$ implies $\pi \overset{*}{\to}_{rc} x$. Also, because $B \in V_N$, $B \overset{+}{\to}_{c} A$ implies $B \overset{+}{\to}_{rc} A$. Therefore there exist explicit rc-derivations E^2 and E^3 such that

$$\mu = \sigma B \pi \xrightarrow{[E^2]}_{rc} \sigma B x \xrightarrow{[E^3]}_{rc} \sigma A x.$$

We also have

$$\sigma A x \xrightarrow{(q,m)}_{rcf} \sigma \theta x \overset{*}{\to}_{c} \eta \theta x = \gamma .$$

Hence, if we define E by $E = E' \bullet E^2 \bullet E^3 \bullet \langle (q,m) \rangle$ and β by $\beta = \sigma \theta x$, then we have $\alpha \xrightarrow{[E]}_{r} \beta \overset{*}{\to}_{c} \gamma$ as required and it only remains to verify that $E_{cf} = D_{cf}$. Clearly we have $E_{cf} = E'_{cf} \bullet E^2_{cf} \bullet E^3_{cf} \bullet \langle (q,m) \rangle$ and since E^2 and E^3 are rc-derivations, we must have that both E^2_{cf} and E^3_{cf} are null sequences. Thus $E_{cf} = E'_{cf} \bullet \langle (q,m) \rangle$. But we also have $D_{cf} = D'_{cf} \bullet \langle (q,m) \rangle$ and $E'_{cf} = D'_{cf}$. It follows that $E_{cf} = D_{cf}$ and so the inductive step and the proof of the lemma are complete. □

The result we seek is a straightforward corollary to this lemma.

COROLLARY 3.7

Let (G,C) be a cs-grammar. If $x \in L(G)$ and D is an explicit rorc-derivation of x from S then D_{cf} is a cfr-derivation of x from S.

PROOF. We have $S \xrightarrow{[D]}_{RorC} x$ and so by the preceding lemma there exists $\alpha \in V^*$ and an explicit r-derivation E of α from S such that $D_{cf} = E_{cf}$ and $S \xrightarrow{[E]}_{R} \alpha \xrightarrow{*}_{C} x$. Since $x \in V_T^*$ and $\alpha \xrightarrow{*}_{C} x$ we must also have $\alpha \xrightarrow{*}_{RC} x$ and so there exists an explicit rc-derivation F of x from α. We now have

$$S \xrightarrow{[E]}_{R} \alpha \xrightarrow{[F]}_{RC} x$$

and if we define EF = E • F we see that EF is an explicit r-derivation of x from S. The corresponding cfr-derivation is $EF_{cf} = E_{cf} \bullet F_{cf}$. But F is an rc-derivation and so F_{cf} is the null sequence. It follows that $EF_{cf} = E_{cf}$ and since we know that $E_{cf} = D_{cf}$ it follows that $EF_{cf} = D_{cf}$ and hence that D_{cf} is indeed an explicit cfr-derivation of x from S. □

Another useful corollary of Lemma 3.6 is the following result.

COROLLARY 3.8

If γ is a rorcsf of a cs-grammar (G,C) with a cf-handle (q,m), then there exist $\alpha, \beta \in V^*$ such that

$$S \xrightarrow[R]{*} \alpha \dashv(q,m)\vdash_{RcF} \beta \xrightarrow[c]{*} \gamma.$$

PROOF. By definition, if γ is a rorcsf of (G,C) with a cf-handle (q,m) there must exist $\bar{\alpha}$ and $\bar{\beta} \in V^*$ such that

$$S \xrightarrow[RcRc]{*} \bar{\alpha} \dashv(q,m)\vdash_{RcR} \bar{\beta} \xrightarrow[c]{*} \gamma.$$

In other words, there is an explicit rorc-derivation D of γ from S such that (q,m) is the last member of D_{cf}. (To see this, note that $\xrightarrow[c]{} \subseteq \xrightarrow[RcRc]{}$.) Therefore, by Lemma 3.6, there exists $\mu \in V^*$ and an explicit r-derivation E such that $E_{cf} = D_{cf}$ and

$$S \dashv[E]\vdash_{R} \mu \xrightarrow[c]{*} \gamma.$$

Since (q,m) is the last member of the sequence D_{cf} and $D_{cf} = E_{cf}$, we may write E in the form

$$E = E' \bullet \langle (q,m) \rangle \bullet E^2$$

where the sequence E^2 is an rc-derivation. (That is to say E^2_{cf} is the null sequence). Clearly, there exist $\alpha, \beta \in V^*$ such that

$$S \dashv[E']\vdash_{R} \alpha \dashv(q,m)\vdash_{RcF} \beta \dashv[E^2]\vdash_{Rc} \mu \xrightarrow[c]{*} \gamma$$

and thus we have

$$S \xrightarrow[R]{*} \alpha \dashv(q,m)\vdash_{RcF} \beta \xrightarrow[c]{*} \gamma$$

and the result is proved. □

Before returning to a re-examination of the bottom up cf-parsing algorithm we prove an important result which relates cf-ambiguity to the uniqueness of cf-handles.

THEOREM 3.9 (cf. Theorem 1.1)

Let $G = (V_N, V_T, P, S)$ be a reduced grammar in which $S \rightarrow^+ S$ does not occur and let C be a chain set for G. Then (G,C) is cf-unambiguous if and only if every rorcsf of (G,C) has exactly one cf-handle, except S which has none.

PROOF. We prove the result in the "only if" direction by showing that if any rorcsf has two distinct cf-handles then (G,C) is cf-ambiguous. Let α be a rorcsf of (G,C) with a pair of distinct cf-handles (p,m) and (q,n). There must be two explicit rorc-derivations D and E of α from S such that (p,m) is the last member of D_{cf} and (q,n) is the last member of E_{cf}. Since $(p,m) \neq (q,n)$ it follows that $D_{cf} \neq E_{cf}$. Now because G is reduced, there must be some $x \in V_T^*$ such that $\alpha \rightarrow_R^* x$, so let F be an explicit r-derivation of x from α. Then if we define $DF = D \bullet F$ and $EF = E \bullet F$ it follows that DF and EF are both rorc-derivations of x from S. By Corollary 3.7 this means that both DF_{cf} and EF_{cf} are cfr-derivations of x from S. But we have :

$$DF_{cf} = D_{cf} \bullet F_{cf} \quad \text{and}$$

$$EF_{cf} = E_{cf} \bullet F_{cf}$$

and since $D_{cf} \neq E_{cf}$, this implies that $DF \neq EF$. Hence the sentence x has two distinct cfr-derivations and so (G,C) is cf-ambiguous and the proof is complete for the "only if" direction.

For the proof in the "if" direction we show that if (G,C) is cf-ambiguous, then some rorcsf of (G,C) has a pair of distinct cf-handles. Suppose that (G,C) is cf-ambiguous. Then some sentence $x \in L(G)$ possesses

a pair of explicit r-derivations D and E such that $D_{cf} \neq E_{cf}$. Three cases need to be considered.

Case 1 : $D_{cf} = Q \bullet E_{cf}$ for some non-null sequence Q. We will show that this case is impossible. We have $S \xrightarrow{[E]}_R x$ and since no chain production may have S as its left part this means that the first member of the sequence E is the same as that of the subsequence E_{cf}. Furthermore, this first member has the form $(S \to \alpha, \text{len}(\alpha))$ where $S \to \alpha \in P$. We can therefore write E_{cf} in the form $E_{cf} = \langle (S \to \alpha, \text{len}(\alpha)) \rangle \bullet R$. Since $D_{cf} = Q \bullet E_{cf}$ we can partition the sequence D and write

$$D = D' \bullet \langle (S \to \alpha, \text{len}(\alpha)) \rangle \bullet D^2$$

where $D'_{cf} = Q$ and $D^2_{cf} = R$. Clearly there exist $\gamma, \delta \in V^*$ such that

$$S \xrightarrow{[D']}_R \gamma \xrightarrow{(S \to \alpha, \text{len}(\alpha))}_R \delta \xrightarrow{[D^2]}_R x.$$

It follows that γ has the form $\gamma = Sz$ for some $z \in V_T^*$. Now define D^3 by $D^3 = \langle (S \to \alpha, \text{len}(\alpha)) \rangle \bullet D^2$ and we have $Sz \xrightarrow{[D^3]}_R x$. But we also have $S \xrightarrow{[E]}_R x$ and $E_{cf} = D^3_{cf}$. This implies $z = \Lambda$ and so $S \xrightarrow{[D']}_R \gamma$ becomes $S \xrightarrow{[D']}_R S$. Now $D_{cf} = Q$ and Q is not null. Hence $S \to^+ S$. But this contradicts the requirement that $S \to^+ S$ does not occur in G and so we conclude that this case is impossible.

Case 2 : $E_{cf} = Q \bullet D_{cf}$ for some non-null sequence Q. This case is symmetrical with Case 1 and may be shown by the same argument to be impossible.

Case 3 : Neither Case 1 nor Case 2 obtains. In this case, since $D_{cf} \neq E_{cf}$ we must be able to write D and E in the form :

$$D = D' \bullet \langle (p,m) \rangle \bullet D^2 , \quad \text{and}$$
$$E = E'' \bullet \langle (q,n) \rangle \bullet E^2$$

where $p,q \in P \setminus C$, $(p,m) \neq (q,n)$ and $D^2_{cf} = E^2_{cf}$. There will exist $\alpha,\beta,\gamma,\delta \in V^*$ such that

$$S \xrightarrow{[D']}_R \alpha \xrightarrow{(p,m)}_{RCF} \beta \xrightarrow{[D^2]}_R x \quad \text{and}$$
$$S \xrightarrow{[E']}_R \gamma \xrightarrow{(q,n)}_{RCF} \delta \xrightarrow{[E^2]}_R x.$$

Now although $D^2_{cf} = E^2_{cf}$, it is not necessarily the case that $D^2 = E^2$ and so we cannot be sure that $\beta = \delta$. However, by a straightfoward induction on the length of the sequence D^2_{cf} (we omit the details) it can be shown that there exists $\theta \in V^*$ such that both $\beta \xrightarrow{*}_c \theta$ and $\delta \xrightarrow{*}_c \theta$. We therefore have

$$S \xrightarrow{*}_R \alpha \xrightarrow{(p,m)}_{RCF} \beta \xrightarrow{*}_c \theta$$

and $S \xrightarrow{*}_R \gamma \xrightarrow{(q,n)}_{RCF} \delta \xrightarrow{*}_c \theta$ and so we see that θ is a rorcsf of (G,C) with a pair of distinct handles (p,m) and (q,n). This completes the proof in the "if" direction and so we may conclude the theorem. □

We saw earlier how our first attempt to formulate a bottom up cf-parsing algorithm broke down due to an inadequate definitional framework. We have now repaired these deficiencies and may proceed to re-present the algorithm.

ALGORITHM 3.10 (cf. Algorithm 1.2)

Bottom up cf-parsing algorithm.

Input : A cf-unambiguous cs-grammar (G,C) and a sentence $x \in L(G)$ which is to be cf-parsed.

Output: The explicit cfr-derivation (in reverse order) of x with respect to (G,C).

Method:

1. Set $\alpha = x$.
2. Repeat steps 3 and 4 until $\alpha = S'$ (S is the goal symbol of G).
3. Determine the cf-handle of α and output it.
4. Cf-reduce α by its cf-handle and let the result replace α. □

The close similarity between Algorithms 3.10 and 1.2 should be noted. The present algorithm is in fact identical to Algorithm 1.2 except that cf-handles have been substituted for handles and the operation of cf-reduction has replaced that of ordinary reduction. Unlike that of Algorithm 1.2, however, the correctness of Algorithm 3.10 is not at all obvious and must be established by a formal proof. This we proceed to do.

THEOREM 3.11

Algorithm 3.10 terminates after a finite number of repetitions of steps 3 and 4 and its output is the correct cfr-derivation (in reverse order) of the input sentence x.

PROOF. Let r be the number of times that steps 3 and 4 of Algorithm 3.10 are executed (note that r will be infinite if the algorithm fails to terminate) and let (p_i, m_i) be the cf-handle found during the i'th execution of Step 3 of the algorithm and let α_i denote the contents of the string α <u>after</u> the i'th execution of step 4. Let $\alpha_0 = x$. Since $x \in L(G)$ and (G,C) is cf-unambiguous, there is a unique explicit cfr-derivation $D = \langle (q_i, n_i) \rangle_{i=s}^{1}$ of x from S. Note that for convenience we have numbered the steps in D in the reverse of the normal order. Let $\langle \psi_i \rangle_{i=s}^{0}$ and $\langle \phi_i \rangle_{i=s}^{0}$ be sequences of rorcsf's of (G,C) such that

$$S = \psi_s \xrightarrow{*}_c \phi_s \dashv (q_s, n_s) \rightarrow_{nc} \psi_{s-1} \xrightarrow{*}_c \phi_{s-1} \dashv (q_{s-1}, n_{s-1}) \rightarrow_{nc} \psi_{s-2} \xrightarrow{*}_c \cdots$$
$$\cdots \xrightarrow{*}_c \phi_1 \dashv (q_1, n_1) \rightarrow_{nc} \psi_0 \xrightarrow{*}_c \phi_0 = x$$

and let t be the minimum of s and r. Observe that since no chain production has S as its left part, we in fact have $S = \psi_s = \phi_s$. We will prove that for all i in the range $0 \leq i \leq t$ we have $\psi_i \xrightarrow{*}_c \alpha_i$ and for $1 \leq i \leq t$ we have $(p_i, m_i) = (q_i, n_i)$. The proof is by induction on i. For the basis we merely observe that $\psi_0 \xrightarrow{*}_c \phi_0$ and $\phi_0 = \alpha_0 = x$ and so $\psi_0 \xrightarrow{*}_c \alpha_0$. For the inductive step we suppose that for some i in the range $0 \leq i < t$ we have $\psi_i \xrightarrow{*}_c \alpha_i$ and proceed to prove that $\psi_{i+1} \xrightarrow{*}_c \alpha_{i+1}$ and $(p_{i+1}, m_{i+1}) = (q_{i+1}, n_{i+1})$.

By construction we have

$$S \xrightarrow{*}_{nonc} \psi_{i+1} \xrightarrow{*}_c \phi_{i+1} \dashv (q_{i+1}, n_{i+1}) \rightarrow_{nc} \psi_i$$

and by the inductive hypothesis we have $\psi_i \xrightarrow{*}_c \alpha_i$. Therefore (q_{i+1}, n_{i+1}) is the cf-handle of α_i. Algorithm 3.10 finds (p_{i+1}, m_{i+1}) as the cf-handle of α_i and since (G,C) is cf-unambiguous this means that $(q_{i+1}, n_{i+1}) = (p_{i+1}, m_{i+1})$ as required. Now we can write ϕ_{i+1} and ψ_i

in the form $\phi_{i+1} = \mu Ax$ and $\psi_i = \mu\theta x$ where $A \rightarrow \theta$ is production q_{i+1} and len $(\mu\theta) = n_{i+1}$. And since $\psi_i \rightarrow_c^* \alpha_i$ we can write α_i in the form $\alpha_i = \bar{\mu}\bar{\theta}x$ where $\mu \rightarrow_c^* \bar{\mu}$ and $\theta \rightarrow_c^* \bar{\theta}$. The act of cf-reducing α_i clearly yields the string $\alpha_{i+1} = \bar{\mu}Ax$ and we see immediately that $\phi_{i+1} \rightarrow_c^* \alpha_{i+1}$. Since $\psi_{i+1} \rightarrow_c^* \phi_{i+1}$ we therefore have $\psi_{i+1} \rightarrow_c^* \alpha_{i+1}$ which is the result required to complete the induction.

We prove that Algorithm 3.10 terminates by demonstrating that $r \leq s$. Suppose that $r > s$. Then from the forgoing induction we know that $\psi_s \rightarrow_c^* \alpha_s$. But $\psi_s = S$ and no chain production has S as its left part. Therefore $\alpha_s = S$ and so Algorithm 3.10 terminates after step 4 has been executed for the s'th time. We conclude that $r \leq s$ It remains to show that $r = s$. Suppose that $r < s$. This implies that $\alpha_r = S$ and hence that $\psi_r \rightarrow_c^* S$. But if $r < s$ we have $S \rightarrow^+ \psi_r$ and therefore $S \rightarrow^+ S$. But (G,C) is cf-unambiguous and so it cannot be that $S \rightarrow^+ S$ in G. We conclude that $r = s$ and hence that Algorithm 3.10 is a correct cf-parsing algorithm. □

Let us now return to our abortive attempt to cf-parse the string X+X*X with respect to (G3, C3). Observe that, as far as it went, the method employed in that attempt was consistent with Algorithm 3.10. The cf-handle of X+X*X is indeed (T → T*P,5), and its cf-handle phrase is indeed X*X. Therefore cf-reduction of X+X*X yields the string X+T as before.

We previously gave up at this point because X+T is not an rsf of the grammar. We now know that X+T is a rorcsf and that the algorithm may proceed. A rorc-derivation of X+T is

$$\langle (S \to E, 1)\ ,\quad (E \to E+T, 3)\ ,\ (E \to T, 1)$$
$$(T \to P, 1)\ ,\quad (P \to X, 1)\rangle$$

and so we see that its cf-handle is $(E \to E+T, 3)$. This means that when X+T is cf-reduced it yields the single symbol E. One more iteration of step 3 of Algorithm 3.10 gives the last cf-handle of the parse, namely $(S \to E, 1)$ and after cf-reducing E to S the algorithm terminates. The cfr-derivation found by this process is of course the correct one :

$$\langle (S \to E, 1)\ ,\ (E \to E + T, 3)\ ,\ (T \to T * P, 5)\rangle.$$

Just as Algorithm 1.2 can be developed to yield the shift-reduce bottom up parsing method of Algorithm 1.3, so Algorithm 3.10 may be recast as a shift-reduce method. We do not reproduce the development in detail since it exactly parallels that in Section 1.7. It is easy to see that the parsing method of Algorithm 1.3. becomes a cf-parsing method on simply substituting an appropriate chain-free parsing action function for the ordinary action function f.

Note that this modification of Algorithm 1.3 depends crucially upon the fact that chain productions have degree 1 and hence that the length of the parse stack is not disturbed by failing to make reductions involving chain productions.

Recall that Algorithm 1.3 demands that the grammar G be unambiguous and that when α denotes the contents of the parse stack and x denotes the unconsumed input then αx must be an rsf of the grammar whose handle (q,m) satisfies $m \geq \text{len}(\alpha)$. Also the parsing action function must satisfy :

$$f(\alpha,x) = \text{REDUCE } q \quad \text{if} \quad m = \text{len}(\alpha),$$
$$\text{and} \quad f(\alpha,x) = \text{SHIFT} \quad \text{if} \quad m > \text{len}(\alpha).$$

To produce a cf-parser from Algorithm 1.3 we simply require that (G,C) be <u>cf-unambiguous</u>, that αx be a <u>rorcsf</u> of (G,C) whose <u>cf-handle</u> (p,n) satisfies $n \geq \text{len}(\alpha)$ and that

$$f(\alpha,x) = \text{REDUCE } p \quad \text{if } n = \text{len}(\alpha),$$
$$\text{and } f(\alpha,x) = \text{SHIFT} \quad \text{if } n > \text{len}(\alpha).$$

The places where the cf-parsing requirements differ from the ordinary ones are underlined.

In the ordinary shift-reduce parsing algorithm, the strings that could appear in the parse stack were called the "viable prefixes" of the grammar. Their counterparts in the chain-free shift-reduce parsing method are naturally called the "chain free viable prefixes". They are defined as follows :

<u>DEFINITION 3.12</u>

A string $\alpha \in V^*$ is a <u>chain-free viable prefix</u> for (G,C) (a <u>cf-viable prefix</u> for short) if and only if there exists $\beta \in V^*$ such that $\alpha\beta$ is a rorcsf of (G,C) with a cf-handle (q,m) satisfying $m \geq \text{len}(\alpha)$. The set of all cf-viable prefixes for (G,C) is denoted by $\text{CFVP}^{(G,C)}$. □

When the chain set C is empty, the cf-viable prefixes of (G,C) are just the ordinary viable prefixes of G.

From Algorithm 1.3 we earlier developed the general form of a "table driven bottom up parser using k symbol lookahead" which was modelled by Algorithm 1.4. Remember that this latter algorithm requires that the viable prefixes of the grammar be partitioned into a finite number of equivalence classes and that is precise behaviour is governed by a set of parsing tables of the form $T = (Q, s_0, g, f)$. In an identical fashion we can derive table-driven cf-parsing methods of the same type.

To model these methods we retain Algorithm 1.4 completely unchanged but drive it with a set of cf-parsing tables instead of with ordinary parsing tables. As with ordinary tables, cf-parsing tables will consists of a set Q of states, an initial state s_0, and a pair of functions f and g which are the action and goto functions respectively. Whereas in ordinary tables the states Q are the names of the equivalence classes into which the viable prefixes of the grammar are partitioned, in cf-tables it is the cf-viable prefixes which are partitioned and assigned to the states in Q. Also, in cf-tables the action function f will never have the value REDUCE q when q is a chain production since the intention is that all such reductions are bypassed. Following the specification of Algorithm 1.4 we deduced conditions which ordinary parsing tables must satisfy if they are to drive the algorithm correctly. These conditions extend to the case of cf-parsing tables in a natural manner and we incorporate them into the formal definition of a set of cf-parsing tables as follows :

DEFINITION 3.13

Let (G,C) be a cf-unambiguous cs-grammar where $G = (V_N, V_T, P, S)$ and let k be a natural number. A set of cf-parsing tables (using k symbol lookahead) for (G,C) is a 4-tuple $CFT = (Q, s_0, g, f)$ where :

(i) Q is a finite, non-empty set of cf-parsing states,

(ii) $s_0 \in Q$ is a distinguished initial state,

(iii) $g : Q \times V \to Q$ is the cf-parsing goto function, and

(iv) $f : Q \times V_T^{*k} \to$ CF-ACTIONS$^{(G,C)}$ is the cf-parsing action function, where CF-ACTIONS$^{(G,C)}$ is the set of possible cf-parsing actions for (G,C), that is:

CF-ACTIONS$^{(G,C)}$ = {ERROR, SHIFT} $\cup$ {REDUCE q | q $\in$ P\C}.

For these tables to drive Algorithm 1.4 as a correct cf-parser, there must exist a surjective mapping CF-EQUIV : $CFVP^{(G,C)} \to Q$ such that :

(i) CF-EQUIV(Λ) = s_0,

(ii) whenever θ and ψ are cf-viable prefixes and $X \in V$ such that

(a) CF-EQUIV(θ) = CF-EQUIV(ψ) and

(b) both θX and ψX are cf-viable prefixes

then CF-EQUIV(θX) = CF-EQUIV(ψX),

(iii) whenever θ and θX are both cf-viable prefixes then g(CF-EQUIV(θ),X) = CF-EQUIV(θX), and

(iv) whenever θx is a rorcsf of (G,C) with a cf-handle (q,m) satisfying $m \geq len(\theta)$, then the value of f(CF-EQUIV(θ),k:x) is :

(a) REDUCE q if $m = len(\theta)$, and

(b) SHIFT if $m > len(\theta)$. □

To illustrate these ideas we show in Figure 3.4 a set of cf-parsing tables for the cs-grammar (G3,C3) using 1 symbol lookahead. Although we do not care to explain

yet how these tables were constructed, we claim that they satisfy all the conditions of Definition 3.13.

STATE	CF-ACTION FUNCTION						CF-GOTO FUNCTION								
NO.	∧	(	X	)	*	+	S	E	T	P	(	X	)	*	+
1		sh	sh					2	2	2	3	2			
2	1				sh	sh				a				4	5
3		sh	sh					6	6	6	3	6			
4		sh	sh					7	7	7	3	7			
5		sh	sh					8	8	8	3	8			
6				sh	sh	sh							9	4	5
7	4			4	4	4									
8	2			2	sh	2								4	
9	6			6	6	6									

Figure 3.4 : Cf-Parsing Tables for (G3,C3) using 1 Symbol Lookahead

In Figure 3.5 we display the moves made by Algorithm 1.4 when driven by the tables of Figure 3.4 and presented with the input string X*X+X.

MOVE NO.	SYMBOL STACK CONTENTS	STATE STACK CONTENTS	UNCONSUMED INPUT	ACTION
1	∧	1	X*X+X	SHIFT
2	X	1,2	*X+X	SHIFT
3	X*	1,2,4	X+X	SHIFT
4	X*X	1,2,4,7	+X	REDUCE T→T*P
5	T	1,2	+X	SHIFT
6	T+	1,2,5	X	SHIFT
7	T+X	1,2,5,8	∧	REDUCE E→E+T
8	E	1,2	∧	REDUCE S→E and ACCEPT

Figure 3.5 : Behaviour of Algorithm 1.4 when Driven by the Tables of Figure 3.4 and Presented with Input X*X+X.

Inspection of the reduce entries in the action column of Figure 3.5 shows that the correct cf-parse, namely $\langle S \rightarrow E, E \rightarrow E+T, T \rightarrow T*P \rangle$, is output by the algorithm.

The motivation behind the introduction of the LR(k) grammars was to capture and explore the properties of the widest class of grammars for which bottom up parsing tables using k symbol lookahead can be constructed. The next step is naturally to extend this idea by attempting to characterize those cs-grammars which can be cf-parsed from left to right using k symbol lookahead.

3.2 The CFLR(k) Property.

The essential property of LR(k) grammars is that any handle is uniquely determined by its left context and the first k symbols of its right context. This property was given precise expression in Definition 2.1 and some alternative formulations of the property were briefly considered and rejected in Section 2.1 .We now define an analogous property for cs-grammars.We say that a cs-grammar is $CF^{LR}(k)$ if every cf-handle is uniquely determined by its left context and the first k symbols of its right context. We define this property more exactly as follows :

DEFINITION 3.14 (cf. Definition 2.1.)

Let (G,C) be a cs-grammar and k a natural number. Then (G,C) is CFLR(k) if and only if the following conditions are satisfied:

(i) G is reduced and $S \rightarrow^+ S$ does not occur in G, and

(ii) whenever α and β are rorcsf's of (G,C) having cf-handles (p,m) and (q,n) respectively and satisfying $m/\beta \subset V_T^*$ and $(m+k):\alpha = (m+k):\beta$, then necessarily (p,m) = (q,n).

We say that G is CFLR(k) if there is a chain set C for G such that (G,C) is CFLR(k). A language is CFLR(k) if it is generated by some CFLR(k) grammar. □

This definition is the natural chain free generalisation of the LR(k) property (Definition 2.1). Observe that when the chain set is empty, the CFLR(k) property degenerates into the LR(k) property. Thus the CFLR(k) property is a true generalisation of the LR(k) property : to every LR(k)

grammar G there corresponds the CFLR(k) cs-grammar $(G,\emptyset)$.

A property which is needed to establish certain results concerning CFLR(k) grammars is the following.

LEMMA 3.15 (cf. Lemma 2.3)

If α and β are rorcsf's of a cs-grammar (G,C) and have cf-handles (p,m) and (q,n) respectively such that $m{:}\alpha = m{:}\beta$, then $n/\alpha \in V_T^*$.

PROOF. This result may be proved by the same argument as that used in Lemma 2.3. □

Now let us look at some simple examples. The grammar G4 given at the beginning of Section 3.1 is ambiguous and is therefore not LR(k) for any k. However G4 is CFLR(0) if $\{A \rightarrow A\}$ is taken as the chain set. This is easily seen to be so because $(G4, \{A \rightarrow A\})$ possesses only three rorcsf's namely, S, A, and x. The goal symbol S has no cf-handle, while A and x have cf-handles $(S \rightarrow A,1)$ and $(A \rightarrow x,1)$ respectively. Clearly these satisfy the CFLR(0) property. This example plainly shows that the CFLR(k) grammars include some that are not LR(k), nor even unambiguous. Note however, that although G4 is ambiguous, $(G4, \{A \rightarrow A\})$ is cf-unambiguous. This is no accident; just as all LR(k) grammars are unambiguous, so all CFLR(k) cs-grammars are cf-unambiguous.

THEOREM 3.16 (cf. Theorem 2.2)

If (G,C) is a CFLR(k) cs-grammar, then it is cf-unambiguous.

PROOF : This result follows from Definition 3.14 and Theorem 3.9 by just the same argument as that used to establish the corresponding result (Theorem 2.2) for the LR(k) grammars. □

Ambiguous grammars are not the only ones which may fail to be LR(k) and yet be CFLR(k) with respect to certain chain sets. Consider the following grammar :

$$S \rightarrow Ay \mid ax \qquad \text{(Grammar G5)}$$

$$A \rightarrow a$$

This grammar is LR(1) but not LR(0). Intuitively, the reason for this is that having seen the symbol a, it is impossible to decide whether or not to reduce it to A without inspecting the next symbol. Grammar G5 is CFLR(0), however, if $\{A \rightarrow a\}$ is taken as the chain set. This is because the decision which needs lookahead to resolve it during an ordinary parse just does not occur during a cf-parse since the reduction causing the difficulty is a chain production. Thus the grammar can be cf-parsed without lookahead and so it is CFLR(0).

We have now seen two examples of non-LR(k) grammars which become CFLR(k) when suitable chain sets are chosen. Naturally we should now ask whether any LR(k) grammars fail to be CFLR(k) for certain chain sets. This question is answered below and provides an important and interesting theorem.

We shall prove that if G is an LR(k) Grammar and C is <u>any</u> chain set in G, then (G,C) is CFLR(k). The proof is quite lengthy and difficult. First we need a lemma concerning LR(k) grammars which is rather similar to a result quoted (but not proved) by Geller and Harrison (1973) and which they call the "Extended LR(k) Theorem".

LEMMA 3.17

Let $G = (V_N, V_T, P, S)$ be an LR(k) grammar with $\alpha, \beta \in V^*$ and $x, y \in V_T^*$ such that

(i) $S \xrightarrow[R]{*} \alpha x$,

(ii) $S \xrightarrow[R]{*} \beta y$,

(iii) $\alpha \xrightarrow[R]{*} \beta$ and

(iv) $k{:}x = k{:}y$.

Then $S \xrightarrow[R]{*} \alpha y$.

PROOF. The proof is by induction on the number of steps in the r-derivation of β from α. The basis is the case where there are no steps at all, that is to say when $\alpha = \beta$, and the conclusion of the lemma is trivial in this case.

For the inductive step, assume the result to be true whenever the number of steps is n $(n \geqslant 0)$ and then suppose that $\alpha \xrightarrow[R]{n+1} \beta$. We may distinguish the last step in this derivation as (p,m) and then for some $\gamma \in V^*$ we will have

$$\alpha \xrightarrow[R]{n} \gamma \dashrightarrow_{R}^{(p,m)} \beta. \qquad (1)$$

It then follows from (i) that

$$S \xrightarrow[R]{*} \alpha x \xrightarrow[R]{n} \gamma x \dashrightarrow_{R}^{(p,m)} \beta x$$

and so (p,m) is the handle of βx. Now (1) implies that $m \leqslant \text{len}(\beta)$ and that $m/\beta \in V_T^*$. Therefore $m/\beta y \in V_T^*$ and we also have, by virtue of (iv), that $(m+k){:}\ \beta x = (m+k){:}\ \beta y$. Because G is LR(k), it must follow from these observations that (p,m) is also the handle of βy and so we have

$$S \xrightarrow[R]{*} \gamma y \dashrightarrow_{R}^{(p,m)} \beta y. \qquad (2)$$

Now (1) gives $\alpha \xrightarrow[R]{n} \gamma$, (2) gives $S \xrightarrow[R]{*} \gamma y$ and we still have (i) and (iv). Therefore we may apply the

inductive hypthesis (to $\alpha \xrightarrow{n}_R \gamma$) and conclude that $S \xrightarrow{*}_R \alpha y$, which is the result needed to complete the inductive step and the proof of the lemma. □

We also find it convenient to introduce a concept of "distance" between strings in a cs-grammar.

DEFINITION 3.18

Let (G,C) be a cf-unambiguous, reduced cs-grammar. If $\alpha, \beta \in V^*$ satisfy $\alpha \xrightarrow[\text{non-c}]{*} \beta$ then define the cf-distance from α to β to be the number of steps in the cfr-derivation of β from α . □

Note that the requirements that G be reduced and that (G,C) be cf-unambiguous ensure that the idea of cf-distance is well defined. Note too that if $\alpha \xrightarrow{*}_c \beta$ then the cf-distance from α to β is zero. And observe that if $\alpha \xrightarrow[\text{non-c}]{*} \beta \xrightarrow[\text{non-c}]{*} \gamma$ then the cf-distance from α to γ is equal to the sum of the cf-distances from α to β and from β to γ .

We may now state and prove an important theorem.

THEOREM 3.19

Let G be LR(k). Then for any chain set C in G, (G,C) is CFLR(k).

PROOF. Because it is LR(k), G must be reduced and $S \rightarrow^{+} S$ cannot occur in G. Also, by Theorem 2.2, G must be unambiguous. Let C be any chain set in G. Then (G,C) is cf-unambiguous by Theorem 3.4. These observations will be needed in the proof.

Now in order to prove that (G,C) is CFLR(k), we suppose that α and β are rorcsf's of (G,C) with cf-handles (p,m) and (q,n) respectively such that :

$$m/\beta \in V_T^* \qquad (1a)$$

and $$(m+k):\alpha = (m+k):\beta \qquad (1b)$$

and proceed to show that (p,m) = (q,n).

If (p,m) is a cf-handle for α, then by Corollary 3.8 there exist $\gamma, \delta \in V^*$ such that

$$S \underset{R}{\rightarrow}^{*} \gamma \text{-}(p,m)\rightarrow_{RCP} \delta \underset{C}{\rightarrow}^{*} \alpha. \qquad (2)$$

Owing to the definition of the relation $\rightarrow_{RCP}$ we must have $m/\delta \in V_T^*$ and so we may write $\delta = \mu x$ and $\alpha = \theta x$ where m = len(μ) = len(θ) and $\mu \underset{C}{\rightarrow}^{*} \theta$. We may therefore rewrite (2) as :

$$S \underset{R}{\rightarrow}^{*} \gamma \text{-}(p,m)\rightarrow_{RCP} \mu x \underset{C}{\rightarrow}^{*} \theta x. \qquad (3)$$

Since G is reduced, there must exist $z \in V_T^*$ such that $\theta \underset{R}{\rightarrow}^{*} z$. Let the cf-distance from θ to z be d. Because $\mu \underset{C}{\rightarrow}^{*} \theta$ and $\theta \underset{R}{\rightarrow}^{*} z$ we must also have $\mu \underset{R}{\rightarrow}^{*} z$ and the cf-distance from μ to z must be d also (this is because the cf-distance from μ to θ is zero). We may now extend (3) to give :

$$S \underset{R}{\rightarrow}^{*} \gamma \text{-}(p,m)\rightarrow_{RCP} \mu x \underset{R}{\rightarrow}^{*} zx. \qquad (4)$$

We also have that (q,n) is a cf-handle for β and so, again by Corollary 3.8, there exist $\rho, \pi \in V^*$ such that

$$S \underset{R}{\overset{*}{\rightarrow}} \rho \dashv(q,n)\underset{R\subset\beta}{\rightarrow} \pi \underset{L}{\overset{*}{\rightarrow}} \beta. \qquad (5)$$

From (1a) and (1b) it follows that β has the form $\beta = \theta y$ where $y \in V_T^*$ satisfies $k{:}y = k{:}x$ and since we already have $\theta \underset{R}{\overset{*}{\rightarrow}} z$, it follows that $\pi \underset{R}{\overset{*}{\rightarrow}} zy$. We can therefore extend (5) to give

$$S \underset{R}{\overset{*}{\rightarrow}} \rho \dashv(q,n)\underset{R\subset\beta}{\rightarrow} \pi \underset{R}{\overset{*}{\rightarrow}} zy. \qquad (6)$$

Note that the cf-distance from π to zy is again d. (This is because the cf-distance from π to θy is zero, and the cf-distance from θ to z is d.)

Now (4) gives $S \underset{R}{\overset{*}{\rightarrow}} \mu x$

and (6) gives $S \underset{R}{\overset{*}{\rightarrow}} zy$

and we know that $\mu \underset{R}{\overset{*}{\rightarrow}} z$ and that $k{:}x = k{:}y$.

Therefore Lemma 3.17 gives $S \underset{R}{\overset{*}{\rightarrow}} \mu y$ and so we obtain

$$S \underset{R}{\overset{*}{\rightarrow}} \mu y \underset{R}{\overset{*}{\rightarrow}} zy. \qquad (7)$$

From (4) we see that the handle of μx is (p,m). Recall that $m = \mathrm{len}(\mu)$ and $k{:}x = k{:}y$. It follows that $m/\mu y \in V_T^*$ and that $(m+k){:}\mu x = (m+k){:}\mu y$. Therefore, since G is LR(k), the handle of μy must be (p,m) also. Thus for some $\sigma \in V^*$, (7) becomes

$$S \underset{R}{\overset{*}{\rightarrow}} \sigma \dashv(p,m)\underset{R\subset\sigma}{\rightarrow} \mu y \underset{R}{\overset{*}{\rightarrow}} zy. \qquad (8)$$

But then both (6) and (8) are r-derivations of zy. Since G is unambiguous there can be only one r-derivation of zy and so (6) and (8) must simply be different ways of writing this unique r-derivation. Let $\langle (q_i, m_i) \rangle_{i=1}^{r}$ be the explicit r-derivation of zy from S and let $\langle \psi_i \rangle_{i=1}^{r}$ be the corresponding implicit derivation. From (6) and (8)

it follows that there exist i,j in the range $1 \leq i,j \leq r$ such that

$$\text{(from 6)}: \quad (q_i, m_i) = (q,n), \quad \psi_{i-1} = \rho, \quad \psi_i = \pi \quad \text{and} \tag{9}$$

$$\text{(from 8)}: \quad (q_j, m_j) = (p,m) \quad \psi_{j-1} = \sigma, \quad \psi_j = \mu y.$$

Remember that our goal is to show that $(p,m)=(q,n)$. We can do this by showing that $i=j$. Clearly exactly one of the relations $i=j$, $i<j$ and $i>j$ must be true. Suppose first that $i<j$. Then we have

$$\psi_0 \xrightarrow[R]{*} \psi_i \xrightarrow[R]{*} \psi_{j-1} \dashv (q_j, m_j) \rightarrow_R \psi_j \xrightarrow[R]{*} \psi_r.$$

Substituting S for ψ_0, zy for ψ_r and using the identities from (9) gives :

$$S \xrightarrow[R]{*} \pi \xrightarrow[R]{*} \sigma \dashv (p,m) \rightarrow_R \mu y \xrightarrow[R]{*} zy. \tag{10}$$

Now we know that $p \in P \setminus C$ (since (p,m) is the cf-handle of α) and so it follows from (10) that the cf-distance from π to zy is at least one greater than the cf-distance from μy to zy. But this is not so, for both these cf-distances are known to be d. From this contradiction we conclude that $i \geq j$. By an exactly similar argument it may be shown that the supposition $i > j$ is also untenable and so we deduce that $i = j$. Then (9) gives $(p,m) = (q,n)$ and we may conclude the theorem. □

This result is of great practical significance, for it means that the speed benefits of cf-parsing can be obtained (anticipating for the moment that it is possible to construct cf-parsers for the CFLR(k) cs-grammars) without sacrificing the attractive generality of the LR(k) grammars.

Intuitively, the conclusion to be drawn from Theorem 3.19 seems to be that cf-parsing is easier than ordinary parsing. While this is gratifying, it is hardly surprising, since a cf-parser is required to provide less information than an ordinary parser. The proof of Theorem 3.19 suggests the following generalisation.

CONJECTURE 3.20

If (G,C) is a CFLR(k) cs-grammar and C' is any chain set in G such that C' $\supseteq$ C then (G,C') is CFLR(k) also. □

The obstacle to proving this conjecture seems to be purely one of notational complexity.

While Theorem 3.19 goes some part of the way towards relating the LR(k) and CFLR(k) properties, it tells us nothing about those CFLR(k) cs-grammars (G,C) where G is not LR(k). It may seem plausible that if a grammar is CFLR(k) but not LR(k) then "all the places where the grammar is not LR(k) involve chain productions!" This idea certainly fits the behaviour of grammars G4 and G5 considered earlier and when expressed more precisely it provides the following proposition.

PROPOSITION 3.21

Let $G = (V_N, V_T, P, S)$ be a reduced grammar in which $S \rightarrow^+ S$ does not occur but which is not LR(k). Let C be a chain set for G. Then (G,C) is CFLR(k) if and only if whenever α and β are rsf's of G with handles (p,m) and (q,n) respectively such that

(i) $m/\beta \in V_T^*$,

(ii) $(m+k):\alpha = (m+k):\beta$ and

(iii) $(p,m) \neq (q,n)$

then either $p \in C$, or $q \in C$, or both.

REFUTATION. While the proposition is clearly true in the less interesting "only if" direction, it is false in the "if" direction as the following counter-example shows.

Consider the grammar :

$$\begin{aligned} S &\rightarrow Ax \mid \\ &\quad\ B \qquad\qquad \text{(Grammar G6)} \\ A &\rightarrow a \\ B &\rightarrow a \end{aligned}$$

and take $\{A \rightarrow a, B \rightarrow a\}$ as the chain set. This grammar is not LR(0) and the only circumstance in which (i), (ii) and (iii) above obtain is when we take $\alpha = a$ and $\beta = ax$. The handle of a is $(B \rightarrow a, 1)$ while that of ax is $(A \rightarrow a, 1)$. Both of these handles involve chain productions and so our proposition claims that the grammar should be CFLR(0). But this is not so, for the cf-handle of a is $(S \rightarrow B, 1)$ while the cf-handle of ax is $(S \rightarrow Ax, 2)$ and these clearly violate the CFLR(0) property. □

Since Proposition 3.21 is false, we still know little about those CFLR(k) grammars which are not LR(k). In particular, we do not know whether the sets of CFLR(k) and LR(k) languages are the same. In the next section we will resolve these questions and others by an indirect approach using the concept of a "cover grammar".

3.3. Indirect Approaches to the CFLR(k) Property.

The CFLR(k) property was introduced as the natural way of extending the LR(k) property to the context of chain-free parsing. It has been shown that every LR(k) grammar is a CFLR(k) grammar and hence that every LR(k) language is an CFLR(k) language. It has also been demonstrated that some CFLR(k) grammars are not LR(k) and we now ask whether any or all of these grammars generate LR(k) languages. If they do, then we ask whether it is possible to prescribe a method for constructing LR(k) grammars for these languages directly from the CFLR(k) cs-grammars concerned. This investigation will involve the use of "cover grammars" and leads on to indirect methods of testing for the CFLR(k) property and for cf-parsing. These methods are called indirect because they reduce the problem of testing a given cs-grammar for the CFLR(k) property to that of testing another grammar (the cover grammar) for the LR(k) property. Similarly the problem of cf-parsing with respect to the original cs-grammar is replaced by that of ordinary parsing with respect to the cover grammar.

The notion of grammatical covering which we shall employ is from Gray and Harrison (1972) but the full generality of their definition is unnecessary in the present context. Roughly speaking, one grammar covers another if the ability to parse according to the covering grammar confers the ability to parse according to the covered grammar by a table look-up technique.

We will show that every CFLR(k) cs-grammar is covered by an LR(k) grammar. This provides the principal tool needed for this investigation. We begin by specialising Gray and Harrison's definition to the case of cs-grammars covered by ordinary grammars.

DEFINITION 3.22

Let $G = (V_N, V_T, P, S)$ and $G' = (V'_N, V_T, P', S')$ be grammars (note that these grammars share the same terminal vocabulary) and let C be a chain set for G. Let h be a mapping from P' into $P \setminus C$. For any explicit derivation $D' = \langle (q_i, m_i) \rangle_{i=1}^{r}$ in G', define the image of D' under h to be :

$$h(D') = \langle (h(q_i), m_i) \rangle_{i=1}^{r}$$

and observe that because the range of h is $P \setminus C$, $h(D')$ is a chain-free derivation in (G,C). We say that G' covers (G,C) under h if and only if :

(i) $L(G) = L(G')$, and

(ii) for every $x \in L(G)$,

(a) if D is a cfr-derivation of x in (G,C), there exists an r-derivation D' of x in G' such that $D = h(D')$, and

(b) if D' is an r-derivation of x in G' then $h(D')$ is a cfr-derivation of x in (G,C).

We say simply that G' covers (G,C) if there exists some h such that G' covers (G,C) under h. □

Observe that if G' covers (G,C) under h, then any r-derivation in G' can be converted (by taking its image under h) into a cfr-derivation in (G,C). This means that parsing with respect to G' is as good as (and as fast as) chain-free parsing with respect to (G,C). Our next goal is to develop a method for constructing cover grammars with useful properties. There is a well known method for removing chain productions from a grammar while preserving the language generated (see, for example, Hopcroft and Ullman (1969), Theorem 4.4) and it might seem reasonable to seek suitable cover grammars in this construction. It turns out that this will not do. The resulting grammars do not have the properties we desire and, in any case, the method cannot deal with chain productions whose right parts are terminal symbols. We will employ a quite different construction. In order that this may proceed satisfactorily, it is necessary to exclude cs-grammars possessing a certain simple type of ambiguity.

DEFINITION 3.23

The cs-grammar (G,C) is said to be chain-ambiguous if there exist distinct productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ in $P \setminus C$ and a string $\theta \in V^*$ such that both $\alpha \rightarrow_c^* \theta$ and $\beta \rightarrow_c^* \theta$. □

Our construction will be restricted to cs-grammars which are not chain-ambiguous. Note that it is elementary to decide whether a given cs-grammar is chain-ambiguous or not (unlike cf-ambiguity, which is undecidable in general) and that if (G,C) is chain-ambiguous and G is reduced, then (G,C) is cf-ambiguous. It follows that no CFLR(k) cs-grammar can be chain-ambiguous.

CONSTRUCTION 3.24

Let (G,C) be a cs-grammar which is not chain-ambiguous where $G = (V_N, V_T, P, S)$. The grammar COVER(G,C)= (V_N', V_T, P', S) is constructed as follows :

$$V_N' = \{A \in V_N \mid A \text{ is the left part of some production in } P \setminus C\},$$

$$P' = \{A \to \theta \mid A \to \alpha \in P \setminus C \text{ and } \alpha \xrightarrow{*}_{\varepsilon} \theta \text{ in } (G,C)\}.$$

We define the surjective mapping $h : P' \to P \setminus C$ as follows :

if $A \to \theta \in P'$, then $h(A \to \theta) = A \to \alpha$ where $A \to \alpha \in P \setminus C$ and $\alpha \xrightarrow{*}_{\varepsilon} \theta$ in (G,C).

Note that if (G,C) were chain ambiguous, then h would not be a mapping. □

Some examples using the grammars introduced in this chapter may help clarify the construction. In these examples we follow each production in COVER(G,C) by an indication of the production in G from which it is obtained (that is to say, its image under the mapping h).

(1) Take Grammar G4 with $\{A \rightarrow A\}$ as the chain set.

COVER (G4, $\{A \rightarrow A\}$) is: $S \rightarrow A$ (from $S \rightarrow A$)

$A \rightarrow x$ (from $A \rightarrow x$).

(2) Take Grammar G5 with $\{A \rightarrow a\}$ as the chain set.

COVER (G5, $\{A \rightarrow a\}$) is: $S \rightarrow Ay$ (from $S \rightarrow Ay$)

$S \rightarrow ay$ (from $S \rightarrow Ay$)

$S \rightarrow ax$ (from $S \rightarrow ax$).

(3) Take Grammar G6 with $\{A \rightarrow a, B \rightarrow a\}$ as the chain set.

COVER (G6, $\{A \rightarrow a, B \rightarrow a\}$) is:

$S \rightarrow Ax$, $S \rightarrow ax$ } (from $S \rightarrow Ax$)

$S \rightarrow B$, $S \rightarrow a$ } (From $S \rightarrow B$)

(4) For a slightly more realistic example, take the Grammar G3 with C3 as the chain set. COVER(G3,C3) is:

$S \rightarrow E \mid T \mid P \mid X$ } (from $S \rightarrow E$)

$E \rightarrow E + T \mid E + P \mid E + X \mid T + T \mid T + P \mid T + X \mid P + T \mid P + P \mid P + X \mid X + T \mid X + P \mid X + X$ } (from $E \rightarrow E + T$)

$$T \to \left.\begin{array}{l} T * P \mid \\ T * X \mid \\ P * P \mid \\ P * X \mid \\ X * P \mid \\ X * X \end{array}\right\} \quad \text{(from } T \to T * P \text{)}$$

$$P \to \left.\begin{array}{l} (E) \mid \\ (T) \mid \\ (P) \mid \\ (X) \end{array}\right\} \quad \text{(from } P \to (E) \text{)}$$

As its name suggests, the grammar COVER(G,C) does indeed cover (G,C). We now prove this fact.

THEOREM 3.25

Let (G,C) be a cs-grammar which is not chain-ambiguous and let COVER(G,C) and h be the grammar and mapping defined by Construction 3.24. Then COVER(G,C) covers (G,C) under h.

PROOF. We state and prove two claims from which it is easy to deduce the theorem.

Claim 1: If θ is an rsf of COVER(G,C) with an explicit r-derivation D in COVER(G,C), then θ is a rorcsf of (G,C) and h(D) is a cfr-derivation of θ in (G,C).

Proof of claim : We use induction on the length of the derivation D. The basis of the induction is the case in which D contains no steps and in this situation the result is trivial. For the inductive step, assume the claim to be true of derivations containing n steps ($n \geq 0$) and suppose that D contains n + 1 steps. We may distinguish the last step of D and write.

$$D = D' \bullet \langle (q,m) \rangle$$

where D' contains n steps. There clearly exists $\propto$ such that

$$S \xrightarrow{[D']}_R \propto \xrightarrow{(q,m)}_R \theta \text{ in COVER(G,C).}$$

Applying the inductive hypothesis to the rsf α and the derivation D', we conclude that α is a rorcsf of (G,C) and that $h(D')$ is a cfr-derivation of α in (G,C). Now since $\alpha \xrightarrow{(q,m)}_{R} \theta$ in COVER(G,C), we may write α and θ in the form $\alpha = \mu Ax$ and $\theta = \mu\gamma x$ where $m=len(\mu\gamma)$ and $A \rightarrow \gamma$ is production q in COVER(G,C). By construction, we will have $h(A \rightarrow \gamma) = A \rightarrow \beta$ where $A \rightarrow \beta \in P \backslash C$ and $\beta \xrightarrow{*}_{c} \gamma$ in (G,C). In (G,C) we therefore have the derivation

$$\alpha = \mu Ax \xrightarrow{(h(q),m)}_{rcf} \mu\beta x \xrightarrow{*}_{c} \mu\gamma x = \theta$$

and so it follows that $\alpha \xrightarrow[rcf]{*} \theta$ in (G,C) and that $\langle (h(q),m) \rangle$ is an explicit cfr-derivation of θ from α in (G,C). Thus $h(D') \bullet \langle (h(q),m) \rangle = h(D)$ is a cfr-derivation of θ in (G,C) and this is the result needed to complete this inductive step and the proof of the first claim. The next result is the inverse of this one.

Claim 2 : If θ is a rorcsf of (G,C) with an explicit cfr-derivation D in (G,C) then θ is an rsf of COVER(G,C) and there is an explicit r-derivation E of θ in COVER(G,C) such that $h(E) = D$.

Proof of claim : The proof is by induction on the length of the cfr-derivation D. Again the basis is the case where D contains no steps at all and is trivial. For the inductive step, assume the result to be true of all cfr-derivations containing n steps ($n \geq 0$) and suppose that D contains n+1 steps. By Definition 3.2. there must be some r-derivation F of θ in (G,C) such that $F_{cf} = D$. We may isolate the last step of F which does not involve a chain

production and write $F = F' \bullet \langle (q,m) \rangle \bullet F^2$ where $q \in P \setminus C$, F'_{cf} contains n steps and F^2_{cf} is null. For some $\alpha, \beta \in V^*$ we will have a derivation in (G,C) with the form :

$$S \xrightarrow{[F']}_R \alpha \xrightarrow{(q,n)}_{RCF} \beta \xrightarrow{[F^2]}_{Rc} \theta. \quad (1)$$

If production q is $A \to \gamma$, we may write α, β and θ in the form $\alpha = \mu A x$, $\beta = \mu \gamma x$ and $\theta = \rho \delta x$ where len $(\mu\gamma) = m$, $\mu \to_c^* \rho$, and $\gamma \to_c^* \delta$.

From (1) we then obtain :

$$S \xrightarrow{[F']}_R \alpha = \mu A x \to_c^* \rho A x \xrightarrow{(q,m)}_{RCF} \rho \gamma x \to_c^* \rho \delta x = \theta.$$

We see from Corollary 3.7 that F'_{cf} is a cfr-derivation of $\rho A x$ in (G,C) and since F'_{cf} contains n steps it follows by the inductive hypthesis that $\rho A x$ is an rsf of COVER(G,C) and that there is an r-derivation E' of $\rho A x$ in COVER(G,C) such that $F'_{cf} = h(E')$. Also, since $A \to \gamma$ is production q in $P \setminus C$ and $\gamma \to_c^* \delta$ in (G,C), it follows that COVER(G,C) contains the production $A \to \delta$ and that $h(A \to \delta) = q$. COVER(G,C) therefore contains the derivation

$$S \xrightarrow{[E']}_R \rho A x \xrightarrow{(A \to \delta, m)}_R \rho \delta x = \beta.$$

If we define $E = E' \bullet \langle (A \to \delta, m) \rangle$ then clearly

$$\begin{aligned} h(E) &= h(E') \bullet \langle (h(A \to \delta), m) \rangle \\ &= F'_{cf} \bullet \langle (q,m) \rangle \\ &= F_{cf} \\ &= D. \end{aligned}$$

This completes the inductive step, the proof of the second claim and the proof of the theorem. □

By way of illustration, observe that the r-derivation of the sentence X+X*X with respect to COVER(G3,C3) is $\langle (S \rightarrow E,1)\ ,\ (E \rightarrow X+T,3)\ ,\ (T \rightarrow X*X,5) \rangle$. The image of this derivation under the mapping associated with the cover grammar is $\langle (S \rightarrow E,1)\ ,\ (E \rightarrow E+T,3)\ ,\ (T \rightarrow T*P,5) \rangle$ which, as we saw earlier, is the correct cfr-derivation of this sentence with respect to the cs-grammar (G3,C3).

The relationship between a cs-grammar and its cover grammar is very interesting. The properties of the cover grammar are mirrored in the cs-grammar by the chain-free counterparts of these properties, and vice-versa. For instance, it is quite easy to prove that a cs-grammar is of cf-ambiguous if and only if its cover grammar is ambiguous in the ordinary sense. A much more interesting result of this type is the following.

THEOREM 3.26

If(G,C) is CFLR(k) then COVER(G,C) is LR(k).

PROOF. First observe that since (G,C) is CFLR(k) it is also cf-unambiguous. Thus (G,C) is not chain ambiguous and so the cover grammar may indeed be constructed. Let $G = (V_N, V_T, P, S)$. Because (G,C) is CFLR(k), G must be reduced and $S \rightarrow^+ S$ cannot occur in G. It is easy to see that these properties imply that the cover grammar is reduced also and does not admit the derivation $S \rightarrow^+ S$. In order to prove that the cover grammar is LR(k) it remains to show that whenever α and β are rsf's of COVER(G,C) with handles (p,m) and (q,n) respectively such that $m/\beta \in V_T^*$ and $(m+k):\alpha = (m+k):\beta$, then (p,m)=(q,n).

Now if α is an rsf of COVER(G,C) with a handle (p,m), it follows from Claim 1 of the proof of Theorem 3.25 that α is a rorcsf of (G,C) with (h(p),m) as a cf-handle. Similarly, if the handle of β in COVER(G,C) is (q,n) then (h(q),n) is the cf-handle of β in (G,C). Then, if $m/\beta \in V_T^*$ and $(m+k):\alpha = (m+k):\beta$, the CFLR(k) property of (G,C) will ensure that (h(p),m)= (h(q),n). It only remains to show that p = q. Let the production h(p) in P\C be $A \rightarrow \gamma$. Then since h(p) = h(q), the productions p and q in COVER(G,C) must be of the form $A \rightarrow \delta$ and $A \rightarrow \sigma$ respectively, where both $\gamma \rightarrow_c^* \delta$ and $\gamma \rightarrow_c^* \sigma$ in (G,C). Thus p and q have the same left parts and the lengths of their right parts are the same. Because (p,m) is a handle of α, we can write $\alpha = \mu\delta x$ where $len(\mu\delta) = m$. And because (q,n) is a handle of β, we can write $\beta = \rho\sigma y$ where $len(\rho\sigma) = n = m$. Then since $(m+k):\alpha = (m+k):\beta$ we must have $m:\alpha = m:\beta$, that is $\mu\delta = \rho\sigma$. But since $len(\delta) = len(\sigma)$, this implies that $\delta = \sigma$. Thus p = q and so we may conclude the theorem. □

A number of important conclusions may be drawn from Theorems 3.25 and 3.26. The first of these is :

COROLLARY 3.27

The families of LR(k) and CFLR(k) languages are co-extensive.

PROOF. To each LR(k) grammar G there trivially corresponds the CFLR(k) cs-grammar (G,∅). Thus each LR(k) language is a CFLR(k) language. And by virtue of Theorem 3.26 to each CFLR(k) cs-grammar (G,C) there corresponds the LR(k) grammar COVER(G,C) generating the same language. Hence each CFLR(k) language is also an LR(k) language and the proof is complete. □

Thus although the CFLR(k) grammars are more extensive than the LR(k) grammars, the generative power of the two families is identical. Furthermore, given a CFLR(k) grammar which is not LR(k),we can mecahnically construct an LR(k) grammar generating the same language by just taking the cover grammar described by Construction 3.24.

Theorem 3.26 also indicates how table-driven cf-parsers may be constructed for the CFLR(k) cs-grammars.

COROLLARY 3.28

If(G,C) is a CFLR(k) cs-grammar, then a table driven cf-parser using κ symbol lookahead can be constructed for (G,C).

PROOF. Because (G,C) is CFLR(k), COVER(G,C) must be LR(k) by virtue of Theorem 3.26. A parser using k symbol look-ahead can certainly be constructed ror COVER(G,C) and the parses which it produces become cf-parses with respect to (G,C) on simply taking their images under the mapping h associated with the covering relationship. □

We have yet to prescribe a method of testing for the CFLR(k) property. For fixed k, (the property is, of course, undecidable unless k is fixed) Theorem 3.26 provides a necessary condition for the CFLR(k) property, namely that

cover grammar be LR(k). The converse of Theorem 3.26 supplies the sufficient condition that is presently lacking.

THEOREM 3.29

The cs-grammar (G,C) is CFLR(k) if COVER(G,C) is defined and LR(k).

PROOF. Note that COVER(G,C) is undefined if and only if (G,C) is chain-ambiguous and that (G,C) cannot be CFLR(k)in this case. Let $G = (V_N, V_T, P, S)$ and suppose that COVER(G,C) is defined and LR(k). Then COVER(G,C) is reduced and does not admit $S \rightarrow^+ S$. This is easily seen to imply that G has these properties also. Now suppose that α and β are rorcsf's of (G,C) with cf-handles (p,m) and (q,n) respectively and that $m/\beta \in V_T^*$ and $(m+k):\alpha = (m+k):\beta$. In order to prove that (G,C) is CFLR(k) it is necessary to show that (p,m) =(q,n). Now by Claim 2 of Theorem 3.25 we know that α and β are rsf's of COVER(G,C) with handles (p',m) and (q',n) respectively where h(p') = p and h(q')= q. Since COVER(G,C) is LR(k), the conditions $m/\beta \in V_T^*$ and $(m+k):\alpha = (m+k):\beta$ imply that (p',m)=(q',n) and so h(p')= h(q')-from which we conclude that (p,m) = (q,n) as required to complete the proof of the theorem. □

In combination, Theorems 3.26 and 3.29 provide a method of testing for the CFLR(k) property. They reduce the problem of testing a cs-grammar for the CFLR(k) property to that of testing its cover grammar for the LR(k) property, and that test may be performed by any of three different methods described in Chapter 2. Since Corollary 3.28 provides a method for cf-parsing the CFLR(k) cs-grammars, it seems that all the important questions concerning the CFLR(k) property have been resolved. In theory this is indeed so; in practice, unfortunately, it is not.

The objection to using these indirect techniques in practice is that the cover grammar is usually larger, and often very much larger, than the cs-grammar which it covers. Whereas, for example, the size of Grammar G3 is 20, that of COVER(G3,C3) is 96 - an increase in size of almost five-fold. The behaviour of G6 indicates that cover grammars for realistic programming language grammars (where "chains" are typically 10 or 12 productions long) are likely to assume sizes of breathtaking proportions. In fact, the size of COVER(G,C) can be exponentially larger than that of the underlying grammar G. To demonstrate this, we exhibit the following family of grammars.

When n is a positive integer, the n'th member of the family is denoted by EXPCOVER(n) and is defined thus :

$$S \rightarrow X_1\ X_2\ X_3\ \ldots\ X_n$$
$$X_i \rightarrow a \qquad (1 \leqslant i \leqslant n)$$
$$X_i \rightarrow b \qquad (1 \leqslant i \leqslant n)$$

The chain set of EXPCOVER(n) is taken as

$$CEXPCOVER(n) = \{ X_i \rightarrow a \mid 1 \leqslant i \leqslant n \}.$$

For example, when n = 3 we have :

$$S \rightarrow X_1\ X_2\ X_3 \qquad \text{(Grammar EXPCOVER(3))}$$
$$X_1 \rightarrow a \mid b$$
$$X_2 \rightarrow a \mid b$$
$$X_3 \rightarrow a \mid b$$

and $CEXPCOVER(3) = \{ X_1 \rightarrow a,\ X_2 \rightarrow a,\ X_3 \rightarrow a \}$

The cover grammar COVER (EXPCOVER(3), CEXPCOVER(3)) is :

$$
\begin{array}{lllll}
S \rightarrow X_1 & X_2 & X_3 & \mid & \\
X_1 & X_2 & a & \mid & \\
X_1 & a & X_3 & \mid & \\
X_1 & a & a & \mid & \\
a & X_2 & X_3 & \mid & \\
a & X_2 & a & \mid & \\
a & a & X_3 & \mid & \\
a & a & a & &
\end{array}
\qquad \text{(from } S \rightarrow X_1\ X_2\ X_3 \text{)}
$$

$$
\begin{array}{ll}
X_1 \rightarrow b & \text{(from } X_1 \rightarrow b \text{)} \\
X_2 \rightarrow b & \text{(from } X_2 \rightarrow b \text{)} \\
X_3 \rightarrow b & \text{(from } X_3 \rightarrow b \text{).}
\end{array}
$$

The size of EXPCOVER(3) is 16, that of itscover grammar is 38. In general, the size of EXPCOVER(n) is 5n + 1 while that of COVER(EXPCOVER(n), CEXPCOVER(n)) is $2n + (n+1).2^n$. Thus the size of the cover grammar grows exponentially in n while that of the basic grammar grows only linearly.

This means that, even if cover grammars are tested for the LR(k) property using the polynomially time-bounded algorithms of Section 2.4, the worst-case time-complexity of the indirect approach to CFLR(k) testing remains exponential in the size of the grammar under test. The behaviour of programming languages is unlikely to be as bad as that of the family EXPCOVER(n), but even so, it will probably be quite bad enough to render the indirect approach to the determination of the CFLR(k) property unattractive. Of course, by virtue of Theorem 3.19, the need to test for the CFLR(k) property can be avoided altogether if we are content to restrict ourselves to cs-grammars based only on LR(k) grammars. This

restriction is probably acceptable in practice. However, when we wish to actually build cf-parsers using the indirect approach, there is no escaping the necessity to construct the cover grammar and its LR(k) parser. It is here that the large size of the cover grammar becomes really objectionable. The LR(1) parser for Grammar G3, for example, has 22 states, while the LR(1) parser for COVER (G3,C3) has 73 states. Using the indirect approach, therefore, the speed benefits of cf-parsing are obtained at the cost of a considerable increase in the (already substantial)size of the parsing tables.

In summary, the indirect approach to cf-parsing via the ordinary parsing of cover grammars is enlightening and yields important theoretical results. But it does not yield practical methods either for CFLR(k) testing or for cf-parsing the CFLR(k) cs-grammars.We will now turn our attention to direct methods for solving these problems. In the next three sections we will construct CFLR(k) versions of the three LR(k) tests introduced in Chapter 2. Later we will consider the problem of constructing direct cf-parsers for the CFLR(k) cs-grammars.

3.4. Testing for the CFLR(k) Property Directly - Part 1.

Throughout this and the two sections following we suppose that k is a fixed natural number, and that $G = (V_N, V_T, P, S)$ is a reduced grammar in which $S \rightarrow^+ S$ does not occur, and that C is a chain set for G. Additionally, in this present section, we reserve the special symbol $\perp$ for use as an endmarker. We require that $\perp$ is not in V.

We begin our investigation of direct methods of testing for the CFLR(k) property by adapting the LR(k) test of Section 2.2. Because the method is of little practical significance, we will provide only the theoretical results necessary to justify the extension of the method to the CFLR(k) context. We will omit all constructional details; the reader can supply them easily. In Section 2.2 we began by defining, for each production q of the grammar G, a set which we called the "LR(k) contexts" of q and which we denoted by $R_k^G(q)$. The LR(k) test developed in that section depended upon the properties of these sets of LR(k) contexts. By analogy, for each non-chain production q of a cs- grammar (G,C), we will define the <u>CFLR(k) contexts</u> of q, denoted by $CFR_k^{(G,C)}(q)$ as follows :

<u>DEFINITION 3.30</u> (cf. Definition 2.9)

For each production $q \in P \setminus C$ we define

$$CFR_k^{(G,C)}(q) = \{ (m+k) : \beta \perp^k \mid \beta \text{ is a rorcsf of } (G,C) \text{ with cf-handle } (q,m) \}. \quad \square$$

Just as the LR(k) property can be stated in terms of conditions upon the sets $R_k^G(q)$, so can the CFLR(k)

property be expressed in terms of the sets $CFR_k^{(G,C)}(q)$.

LEMMA 3.31 (cf. Lemma 2.10)

(G,C) is CFLR(k) if and only if, for any $p,q \in P\backslash C$, $\alpha \in V^* \perp^{*k}$ and $u \in V_T^* \perp^{*k}$; $\alpha \in CFR_k^{(G,C)}(p)$ and $\alpha u \in CFR_k^{(G,C)}(q)$ imply $p = q$ and $u = \Lambda$.

PROOF. This result may be proved by a straightfoward adaption of the argument used to prove Lemma 2.10. □

For typographical convenience we will henceforth omit the sub and superscripts from names of the sets $CFR_k^{(G,C)}(q)$ and $R_k^G(q)$ and write simply CFR(q) and R(q). The LR(k) test of Section 2.2 depends upon the fact that the sets R(q) are regular. We shall now prove that the sets CFR(q) share this property. The regularity of R(q) was established by a construction involving right linear grammars and this construction can be adapted quite straightfowardly to the present situation. However, a more interesting way to prove that the sets CFR(q) are regular is to exploit their relationship to the sets R(q). This relationship is exposed in the next Lemma.

LEMMA 3.32

Let q be a production in $P\backslash C$. Then

$CFR(q) = \{\beta \mid \text{there exists } \alpha \in R(q) \text{ such that } \alpha \Rightarrow_c^* \beta\}$.

PROOF. First suppose that $\beta \in CFR(q)$. Then there is some rorcsf δ of (G,C) with a cf-handle (q,m) such that $\beta = (m+k):\delta \perp^k$. Now because (q,m) is a cf-handle of δ it follows from Corollary 3.8 that there exist $\delta, \mu \in V^*$ such that $S \Rightarrow_R^* \delta \Rightarrow_{(q,m)} \mu \Rightarrow_c^* \delta$.

Thus (q,m) is an ordinary handle of μ and so if we define $\alpha = (m+k):\mu\perp^k$ then it follows that $\alpha \in R(q)$. But $\mu \rightarrow_c^* \gamma$ and so it must be that $\alpha \rightarrow_c^* \beta$. (Note that we are taking liberties with the use of the relation $\rightarrow_c^*$ here, since α and β may include the endmarker symbol $\perp$, which is not in V. However, the intended interpretation should be clear).

Conversely, suppose that $\alpha \in R(q)$ and that $\alpha \rightarrow_c^* \beta$. Then there exists an rsf γ of G with a handle (q,m) such that $\alpha = (m+k):\gamma\perp^k$. For simplicity we will suppose that len $(\gamma) \geq m+k$, so that $\alpha = (m+k):\gamma$. (If len$(\gamma) <$ m+k then the endmarker symbol $\perp$ will appear in α and the argument that follows will be complicated by some tedious details that are needed to take account of this fact.) Let $x \in V_T^*$ be the string such that $\gamma = \alpha x$. Then since (q,m) is a handle of γ and $\alpha \rightarrow_c^* \beta$ we have

$$S \rightarrow_R^* \theta A(q,m) \underset{A\to\alpha\in F}{\Rightarrow} \gamma = \alpha x \rightarrow_c^* \beta x$$

for some $\theta \in V^*$. Thus βx is a rorcsf of (G,C) with (q,m) as a cf-handle. Since $\beta = (m+k):\beta x\perp^k$ we therefore have $\beta \in CFR(q)$.

We have now demonstrated the mutual inclusion of the two sets appearing in the statement of the lemma. It follows that these two sets are equal and so we may conclude the lemma. □

We may now prove that CFR(q) is regular.

LEMMA 3.33 (cf. Lemma 2.11)

Let $q \in P \setminus C$. Then CFR(q) is a regular set.

PROOF. A substitution f is a mapping from an alphabet A onto subsets of B^* for some alphabet B. Thus f associates some language over B with each symbol of A. The mapping f can be extended to strings in A^* as follows :

(i) $f(\Lambda) = \{\Lambda\}$,

(ii) $f(\alpha x) = f(\alpha)f(x)$ for $\alpha \in A^*$ and $x \in A$.

We can further extend f to languages by defining f(L) to be the set : $f(L) = \bigcup_{\alpha \in L} f(\alpha)$ where L is a language over A.

Now consider the particular substitution from V onto subsets of V^* defined by :

$$f(a) = \{a\} \text{ when } a \in V_T \text{ and}$$

$$f(A) = \{X \in V \mid A \xrightarrow{*}_{\cdot} X\} \text{ when } A \in V_N.$$

It is easy to see that for $\alpha \in V^*$ we have

$$f(\alpha) = \{\beta \in V^* \mid \alpha \xrightarrow{*}_{c} \beta\}.$$

Then, by virtue of Lemma 3.32 it follows that

$$CFR(q) = f(R(q)).$$

Now the regular sets are known to be closed under substitution (see Hopcroft and Ullman (1969), Theorem 9.7) and so the regularity of CFR(q) follows directly from that of R(q). □

Because Lemmas 3.31 and 3.33 parallel Lemmas 2.10 and 2.11 exactly, a method of testing for the CFLR(k) property may be constructed along the same lines as the LR(k) testing method indicated in the first proof of Theorem 2.8 (see Section 2.2). This concludes the derivation of our first direct method of testing for the CFLR(k) property.

3.5. Testing for the CFLR(k) Property Directly - Part 2.

We continue our investigation of direct methods of testing for the CFLR(k) property by adapting the LR(k) test of Section 2.3 to this new task. First we need to develop appropriate generalisations for the notions of LR(k) items, states, and statesets. All the constructions and definitions to be introduced in this section will possess the property that they become equivalent to the corresponding ones from Section 2.3 when C, the chain set, is empty. Thus the new constructs may truly be considered as generalisations of the old ones.

We begin by defining the CFLR(k) items for (G,C) to be the same as the LR(k) items for G except that items involving chain productions are excluded.

DEFINITION 3.34 (cf. Definition 2.12)

A CFLR(k) item for (G,C) is a pair $[B \rightarrow \beta_1 . \beta_2, v]$ where $B \rightarrow \beta_1\beta_2 \in P \setminus C$ and $v \in V_T^{*k}$. The set of all CFLR(k) items for (G,C) is denoted $CFI_k^{(G,C)}$. □

Because CFLR(k) items are also LR(k) items, we may speak of initial, intermediate, and final CFLR(k) items in just the same way as with LR(k) items. It is useful to have a name for those LR(k) items which are not CFLR(k) items; we will call them LR(k) chain items. Observe that no chain item is intermediate.

Next we introduce a function $CF\text{-}STRIP_k^{(G,C)}$ from sets of LR(k) items to sets of CFLR(k) items as follows :

DEFINITION 3.35

Let Δ be a set of LR(k) states for G. Then

$$\text{CF-STRIP}_k^{(G,C)}(\Delta) = \Delta \cap \text{CFI}_k^{(G,C)}.$$

We usually omit the sub and superscripts and write this function as simply CF-STRIP. CF-STRIP(Δ) merely discards all chain items from Δ and retains the CFLR(k) items. It will prove convenient to allow CF-STRIP to be applied to arguments which are not simply sets of LR(k) items, but sets of such sets (such as sets of LR(k) states). In this case we require that CF-STRIP first form the union of the components of its argument. That is, when M is contained in the powerset of I_k^G, define

$$\text{CF-STRIP}(M) = \text{CF-STRIP}\left(\bigcup_{\Delta \in M} \Delta\right). \square$$

Now we generalise the notion of 'valid' LR(k) items.

DEFINITION 3.36 (cf. Definition 2.14.)

When $\theta \in V^*$, the CFLR(k) item $[B \rightarrow \beta_1 . \beta_2, v]$ is said to be <u>cf-valid</u> for θ (with respect to (G,C) and k) if and only if there is a derivation

$$S \xrightarrow[Rmd_c]{*} \gamma B x \xrightarrow[RCF]{} \gamma \beta_1 \beta_2 x \xrightarrow[c]{*} \theta \beta_2 x$$

in (G,C) with $v = k{:}x$. $\square$

The notions of LR(k) states and statesets are generalized straightforwardly.

DEFINITION 3.37 (cf. Definition 2.15)

When $\theta \in V^*$, the set of all CFLR(k) items which are cf-valid for θ is called the CFLR(k) state for θ and is denoted by $CFV_k^{(G,C)}(\theta)$ or, more briefly, by $CFV(\theta)$. The set consisting of the CFLR(k) states of all the cf-viable prefixes of (G,C) is called the CFLR(k) stateset for (G,C) and is denoted by $CFS_k^{(G,C)}$. Note that $CFV(\theta)$ is non-empty if and only if θ is a cf-viable prefix of (G,C). Hence :

$$CFS_k^{(G,C)} = \{ CFV(\theta) \neq \emptyset \mid \theta \in V^* \} . \square$$

The idea of pairs of LR(k) items being in "conflict" (see Definition 2.13) applies unchanged to pairs of CFLR(k) items and, just as with LR(k) states, we say that a CFLR(k) state is "adequate" if and only if it contains no pairs of conflicting items. Similarly, a CFLR(k) stateset is said to be adequate if and only if each of its component states is adequate. The CFLR(k) property is related to the adequacy of CFLR(k) statesets in exactly the same way as the LR(k) property is related to the adequacy of LR(k) statesets. Before we can prove this generalisation of Theorem 2.18 we first need to generalise Lemma 2.17.

LEMMA 3.38 (cf. Lemma 2.17.)

Let $\alpha\beta$ be a rorcsf of (G,C) with a cf-handle (q,n) satisfying $n > len(\alpha)$. Then there is a non-final CFLR(k) item $[C \rightarrow \gamma_1 . \gamma_2, v]$ which is cf-valid for α with $EFF_k(\beta) \subseteq EFF_k(\gamma_2 v)$.

PROOF. This result is proved by a generalisation of the argument used to prove Lemma 2.17. The generalisation is not completely straightforward and so we present it in full.

Suppose that $\alpha\beta$ is a rorcsf of (G,C) with a cf-handle (q,n) satisfying $n > len(\alpha)$. Then it follows from Corollary 3.8 that (q,n) is the ordinary handle of some rsf μ of G satisfying $\mu \rightarrow_r^* \alpha\beta$. This latter relation implies that μ has the form $\mu = \alpha'\beta'$ where $\alpha' \rightarrow_r^* \alpha$ and $\beta' \rightarrow_r^* \beta$. Note that $len(\alpha) = len(\alpha')$. Let $D = \langle (q_i, n_i) \rangle_{i=1}^{r}$ be an explicit r-derivation of $\alpha'\beta'$ from S with $(q_r, n_r) = (q,n)$ and let $\langle \psi_i \rangle_{i=0}^{r}$ be the corresponding implicit derivation. Clearly there exists t in the range $1 \leq t \leq r$ such that both $n_t - deg(q_t) \leq len(\alpha)$ and $q_t \in P \setminus C$. (Take t = 1 for example; we must have $n_1 - deg(q_1) = 0$ and since the left part of production q_1 is S, q_1 cannot be a chain production). Now choose the largest such t. We will show that $len(\alpha) < n_t$. This is true by hypothesis if t = r, so assume that t < r and suppose, for the sake of contradiction, that $len(\alpha) \geq n_t$. Let s be the least integer in the range $t < s \leq r$ such that $q_s \in P \setminus C$. (Such an s must exist because $q_r = q$ and $q_r \in P \setminus C$ since it figures in the cf-handle of $\alpha\beta$.) Because D is an r-derivation we have :

$$\left.\begin{array}{l} n_{t+1} - \deg(q_{t+1}) < n_t \\ n_{t+2} - \deg(q_{t+2}) < n_{t+1} \\ \quad\vdots \\ n_s - \deg(q_s) < n_{s-1} \end{array}\right\} \qquad (1)$$

and, by construction, for i in the range $t < i < s$ we have $q_i \in C$ - which implies $\deg(q_i) = 1$ for i in this range. It therefore follows from (1) that

$$n_s - \deg(q_s) < n_{s-1} \leqslant n_{s-2} \leqslant \ldots \leqslant n_{t+1} \leqslant n_t$$

and so the supposition that $\text{len}(\alpha) \geqslant n_t$ implies that $n_s - \deg(q_s) \leqslant \text{len}(\alpha)$. But this contradicts the choice that t be the <u>largest</u> integer with the prescribed properties. Hence we conclude

$$n_t - \deg(q_t) \leqslant \text{len}(\alpha) < n_t . \qquad (2)$$

We now have the derivation

$$S \xrightarrow[R]{*} \psi_{t-1} \Rightarrow (q_t, n_t) \xrightarrow[R\leqslant P]{} \psi_t \xrightarrow[R]{*} \alpha'\beta' \xrightarrow[C]{*} \alpha\beta \qquad (3)$$

and the choice of t ensures that either $n_t - \deg(q_t) > \text{len}(\alpha)$ or $q_i \in C$ for all i in the range $t < i \leqslant r$. It follows that ψ_t has the form $\psi_t = \alpha''\theta$ where $\alpha'' \xrightarrow[C]{*} \alpha'$ and $\theta \xrightarrow[EFF]{*} \beta'$. Now let production q_t be $C \to \gamma$. Then ψ_t can also be written in the form $\psi_t = \sigma\gamma x$ where $\text{len}(\sigma\gamma) = n_t$. The inequalities (2) then become

$$\text{len}(\sigma) \leqslant \text{len}(\alpha) < \text{len}(\sigma\gamma)$$

and we can therefore rewrite γ as $\gamma = \gamma_1\gamma_2$ where $\text{len}(\sigma\gamma_1) = \text{len}(\alpha)$. This gives $\sigma\gamma_1 = \alpha''$, $\gamma_2 \neq \Lambda$ and $\gamma_2 x = \theta$ and so (3) provides

$$S \xrightarrow[R]{*} \sigma Cx \xrightarrow[R\leqslant P]{} \sigma\gamma_1\gamma_2 x = \alpha''\gamma_2 x \xrightarrow[C]{*} \alpha\gamma_2 x .$$

It follows when $v = k{:}x$ that $[C \to \gamma_1 . \gamma_2, v]$ is a non-final CFLR(k) item which is cf-valid for α. Since we also have $\theta = \gamma_2 x$ and $\theta \xrightarrow[EFF]{*} \beta'$ it further follows that $\gamma_2 x \xrightarrow[EFF]{*} \beta$ and so we finally conclude that $EFF_k(\beta) \subseteq EFF_k(\gamma_2 v)$ and the proof is complete. □

Now we can generalize Theorem 2.18.

THEOREM 3.39 (cf. Theorem 2.18)

(G,C) is CFLR(k) if and only if its CFLR(k) stateset is adequate.

PROOF. As with the preceeding lemma, the proof of this result is based on the argument used to prove the corresponding result in the LR(k) case. The proof in the 'if' direction is based upon Lemma 3.38 in just the same way as the proof in the 'if' direction of Theorem 2.18 is based upon Lemma 2.17. The details are straightforward and we omit them.

We establish the result in the only if' direction by proving its contrapositive. Suppose that the CFLR(k) stateset for (G,C) is inadequate. Then there is a cf-viable prefix θ of (G,C) whose CFLR(k) state, CFV(θ), contains a pair of conflicting items, say $[D \rightarrow \delta., u]$ and $[C \rightarrow \gamma_1.\gamma_2, v]$. For conflict to occur we must have $u \in EFF_k(\gamma_2 v)$. Now if $[D \rightarrow \delta., u]$ is cf-valid for θ, there must be a derivation in (G,C) of the form

$$S \underset{Rmc}{\overset{*}{\Rightarrow}} \mu Dx \underset{RCF}{\Rightarrow} \mu\delta x \overset{*}{\underset{c}{\Rightarrow}} \theta x$$

where $u = k{:}x$. Let $\alpha = \theta x$, $m = \text{len}(\theta)$ and let production $D \rightarrow \delta$ be called p. Then (p,m) is a cf-handle for α and $(m+k){:}\alpha = \theta u$. Similarly since $[C \rightarrow \gamma_1.\gamma_2, v]$ is cf-valid for θ also, there is a derivation in (G,C) of the form

$$S \underset{Rmc}{\overset{*}{\Rightarrow}} \eta Cy \underset{RCF}{\Rightarrow} \eta\gamma_1\gamma_2 y \overset{*}{\underset{c}{\Rightarrow}} \theta\gamma_2 y \qquad (1)$$

with $v = k{:}y$. Now we have $u \in EFF_k(\gamma_2 v)$ and therefore also $u \in EFF_k(\gamma_2 y)$. Hence $\gamma_2 y \overset{*}{\underset{EFF}{\Rightarrow}} z$ for some

$z \in V_T^*$ satisfying $u = k{:}z$ where it may be that some, all, or none of the steps in the eff-derivation of z from $\gamma_2 y$ involve chain productions. We distinguish two cases.

Case 1: $\gamma_2 y \xrightarrow{*}_{c} z$. In this case we can extend (1) to obtain

$$S \xrightarrow[RMC]{*} \eta C y \xrightarrow[REF]{} \eta \gamma_1 \gamma_2 y \xrightarrow{*}_{c} \theta z .$$

Now let production $C \rightarrow \gamma_1 \gamma_2$ be called q and define $n = \text{len}(\eta \gamma_1 \gamma_2)$ and $\beta = \theta z$. Clearly (q,n) is a cf-handle for β and we also have $m/\beta \in V_T^*$ and $(m+k){:}\beta = \theta u$.

This latter identity gives $(m+k){:}\alpha = (m+k){:}\beta$ and so if (G,C) were CFLR(k) we should have to have $(p,m) = (q,n)$. We show that this is impossible and hence that (G,C) is not CFLR(k). Suppose $(p,m) = (q,n)$. Obviously this implies $p = q$ and $m = n$. Now $n = m + \text{len}(\gamma_2)$ and so $m = n$ implies $\gamma_2 = \Lambda$ and therefore $u \in \text{EFF}_k(\gamma_2 v)$ implies $u = v$. We now have $p = q$, $\gamma_2 = \Lambda$ and $u = v$ and so $[D \rightarrow \delta ., u] = [C \rightarrow \gamma_1 . \gamma_2, v]$. But this contradicts the hypothesis that these items are in conflict (and are therefore distinct). We conclude that $(p,m) \neq (q,n)$ and therefore that (G,C) is not CFLR(k).

Case 2: $\gamma_2 y \not\xrightarrow{*}_{c} z$. In this case, any eff-derivation of z from $\gamma_2 y$ must involve at least one non-chain production. Also, since z is a terminal string, $\gamma_2 y \xrightarrow[GFF]{*} z$ implies $\gamma_2 y \xrightarrow[REFF]{*} z$. We may therefore distinguish the last non-chain step in this reff-derivation as (q,j) and write

$$\gamma_2 y \xrightarrow[REFF]{*} \rho \xrightarrow{(q,j)}_{REFF} \sigma \xrightarrow{*}_{c} z$$

where $q \in P \setminus C$ and (since this is a reff-derivation)

$j > 0$. We can append this derivation to (1) and thereby obtain

$$S \underset{RoRc}{\overset{*}{\Longrightarrow}} \theta\gamma_2 y \underset{RoRL}{\overset{*}{\Longrightarrow}} \theta\rho \dashv(q, j+\text{len}(\theta))\rightarrow_{RCF} \theta\sigma \overset{*}{\rightarrow}_{\varepsilon} \theta z.$$

Now put $n = j + \text{len}(\theta)$ and $\beta = \theta z$ and it follows that (q,n) is a cf-handle for β. We again have $m/\beta \in V_T^*$ and, since $u = k{:}z$, $(m+k){:}\alpha = (m+k){:}\beta$. But then, since $\text{len}(\theta) = m$ and $j > 0$, it follows from $n = j + \text{len}(\theta)$ that $m \neq n$. Therefore $(p,m) \neq (q,n)$ and so (G,C) is not CFLR(k) in this case either and the proof is complete. □

Given a cs-grammar (G,C) and its CFLR(k) stateset we can easily test the stateset for adequacy and thereby determine whether (G,C) is CFLR(k). All we need now is an algorithm for computing the CFLR(k) stateset corresponding to a given cs-grammar. We will develop such an algorithm by constructing appropriate generalizations of the function CLOSURE, NEXT and GOTO which were introduced in Definition 2.19 and then use these to generalize Algorithm 2.23

DEFINITION 3.40 (cf. Definition 2.19)

When Δ is any set of CFLR(k) items for (G,C), its chain free closure is given by the definition

$$\text{CF-CLOSURE}_k^{(G,C)}(\Delta) = \text{CF-STRIP}_k^{(G,C)}(\text{CLOSURE}_k^G(\Delta))$$

and when $X \in V$ we define

$$\text{CF-NEXT}_k^{(G,C)}(\Delta,X) = \bigcup_{Y \overset{*}{\rightarrow}_{\varepsilon} X} \text{NEXT}_k^G(\Delta,Y) \text{ and}$$

$$\text{CF-GOTO}_k^{(G,C)}(\Delta,X) = \text{CF-CLOSURE}_k^{(G,C)}(\text{CF-NEXT}_k^{(G,C)}(\Delta,X))$$

As usual, we omit the sub and superscripts from the names of these functions whenever possible. □

Because these new functions are defined in terms of the functions CLOSURE and NEXT, we can establish their properties directly from those of these familiar functions. First we need to distinguish the 'nucleus' and the 'completion' of a CFLR(k) state. As in the ordinary LR(k) case, the nucleus of a CFLR(k) state contains all the non-initial items from the state, while the initial items comprise the completion of the state.

DEFINITION 3.41 (cf. Definition 2.21)

The nucleus of the CFLR(k) state for θ is denoted by $CFN_k^{(G,C)}(\theta)$ and defined by :

(i) $CFN_k^{(G,C)}(\Lambda) = N_k^G(\Lambda)$,

(ii) and when $\theta \neq \Lambda$,

$$CFN_k^{(G,C)} = \{ [B \rightarrow \beta_1 . \beta_2, v] \in CFV_k^{(G,C)}(\theta) \mid \beta_1 \neq \Lambda \}.$$

The completion of the CFLR(k) state for θ is denoted by $CFC_k^{(G,C)}(\theta)$ and is given by :

$$CFC_k^{(G,C)}(\theta) = CFV_k^{(G,C)}(\theta) \setminus CFN_k^{(G,C)}(\theta)$$

Whenever possible we write nuclei and completions as simply CFN(θ) and CFC(θ) respectively. □

Next we relate CFLR(k) states and their nuclei to ordinary LR(k) states and nuclei. We do this in an important theorem which is established as a corollary to the following lemma.

LEMMA 3.42

Let $\theta \in V^*$. Then the CFLR(k) item $[B \rightarrow \beta_1 . \beta_2 , v]$ is cf-valid for θ if and only if it is valid in the ordinary LR(k) sense for some $\mu \in V^*$ satisfying $\mu \rightarrow_c^* \theta$.

PROOF. For the proof in the "if" direction, suppose that $[B \rightarrow \beta_1 . \beta_2 , v]$ is valid for μ and that $\mu \rightarrow_c^* \theta$. By the definition of a valid LR(k) item we have that G contains a derivation of the form

$$S \rightarrow_R^* \gamma B x \rightarrow_R \gamma \beta_1 \beta_2 x$$

where $\gamma\beta_1 = \mu$ and $v = k{:}x$. Since $[B \rightarrow \beta_1 . \beta_2 , v]$ is required to be a CFLR(k) item we have $B \rightarrow \beta_1\beta_2 \in P \setminus C$, and since $\mu \rightarrow_c^* \theta$ the derivation above yields

$$S \rightarrow_R^* \gamma B x \rightarrow_{R \in F} \gamma \beta_1 \beta_2 x \rightarrow_c^* \theta \beta_2 x .$$

From this it follows immediately that $[B \rightarrow \beta_1 . \beta_2 , v]$ is cf-valid for θ and the proof is complete for this direction.

For the "only if" direction, suppose that $[B \rightarrow \beta_1 . \beta_2 , v]$ is cf-valid for θ. Then (G,C) contains a derivation of the form

$$S \rightarrow_{R \cap C}^* \gamma B x \rightarrow_{R \in F} \gamma \beta_1 \beta_2 x \rightarrow_c^* \theta \beta_2 x$$

where $v = k{:}x$. Let the production $B \rightarrow \beta_1\beta_2$ be called q and let $n = \text{len}(\theta\beta_2)$. It follows that (q,n) is a cf-handle for the rorcsf $\theta\beta_2 x$ and hence, by Corollary 3.8, there exists an rsf α of G such that $\alpha \rightarrow_c^* \theta\beta_2 x$ and (q,n) is a handle for α. Clearly, α can be written in the form $\alpha = \delta\beta_1\beta_2 x$ where $\delta\beta_1 \rightarrow_c^* \theta$ and so

G must contain the derivation.

$$S \underset{R}{\rightarrow}^{*} \delta Bx \underset{R}{\rightarrow} \delta\beta_1\beta_2 x .$$

Now put $\mu = \delta\beta_1$ and it follows immediately that $\mu \underset{L}{\rightarrow}^{*} \theta$ and that $[B \rightarrow \beta_1 . \beta_2 , v]$ is valid for μ. □

THEOREM 3.43

Let $\theta \in V^{*}$. Then

(i) $CFV(\theta) = CF\text{-}STRIP(\{ V(\mu) \mid \mu \underset{L}{\rightarrow}^{*} \theta \})$ and

(ii) $CFN(\theta) = CF\text{-}STRIP(\{ N(\mu) \mid \mu \underset{L}{\rightarrow}^{*} \theta \})$.

PROOF. These results are immediate consequences of the preceeding lemma. Note that the statement of this theorem employs the notational trick of allowing CF-STRIP to be applied to sets of sets of LR(k)items (see Definition 3.35). Note also that these results are true whether or not θ is a cf-viable prefix of (G,C); in the case that θ is not a cf-viable prefix, all the sets appearing in the statement of the theorem are empty. □

Next we present two lemmas which expose the properties of the functions CF-NEXT and CF-CLOSURE.

LEMMA 3.44 (cf. Part (i) of Lemma 2.22)

Let $\theta \in V^{*}$ and $X \in V$. Then $CFN(\theta X) = CF\text{-}NEXT(CFV(\theta), X)$.

PROOF. By part (ii) of Theorem 3.43 we have

$$CFN(\theta X) = CF\text{-}STRIP(\{ N(\gamma) \mid \gamma \underset{L}{\rightarrow}^{*} \theta X \})$$

which may be rewritten as

$$CFN(\theta X) = CF\text{-}STRIP(\{ N(\mu Y) \mid \mu \underset{L}{\rightarrow}^{*} \theta, Y \underset{L}{\rightarrow}^{*} X \}). \quad (1)$$

Then from part (i) of Lemma 2.22 we have

$$N(\mu Y) = NEXT(V(\mu), Y) \quad (2)$$

and using (2) in(1) gives

$$CFN(\theta X) = CF\text{-}STRIP(\{ NEXT(V(\mu),Y) \mid \mu \rightarrow_c^* \theta,\ Y \rightarrow_c^* X \}). \quad (3)$$

The properties of CF-STRIP and NEXT clearly allow (3) to be rewritten as

$$CFN(\theta X) = \bigcup_{Y \rightarrow_c^* X} NEXT(CF\text{-}STRIP(\{V(\mu) \mid \mu \rightarrow_c^* \theta\}), Y) \quad (4)$$

and part (i) of Theorem 3.43 gives

$$CF\text{-}STRIP(\{ V(\mu) \mid \mu \rightarrow_c^* \theta\}) = CFV(\theta).$$

Using this in(4) gives

$$CFN(\theta X) = \bigcup_{Y \rightarrow_c^* X} NEXT(CFV(\theta), Y) \quad (5)$$

and by Definition 3.40, the right hand side of (5) is just CF-NEXT(CFV(θ),X) from which we conclude the lemma. □

LEMMA 3.45 (cf. Part(ii) of Lemma 2.22)

Let $\theta \in V^*$. Then CFV(θ) = CF-CLOSURE(CFN(θ)).

PROOF. By part (i) of Theorem 3.43 we have

$$CFV(\theta) = CF\text{-}STRIP(\{ V(\mu) \mid \mu \rightarrow_c^* \theta\}) \quad (1)$$

and by part (ii) of Lemma 2.22 we have

$$V(\mu) = CLOSURE(N(\mu)). \quad (2)$$

Using (2) in (1) gives

$$CFV(\theta) = CF\text{-}STRIP(\{ CLOSURE(N(\mu)) \mid \mu \rightarrow_c^* \theta\}). \quad (3)$$

Now only non-final LR(k) items can contribute items other than themselves to the CLOSURE operation, and all chain items which appear in the nucleus of an LR(k) state must be final items. (This is because $N(\Lambda)$ contains no chain items at all, and when $\mu \neq \Lambda$, $N(\mu)$ contains only non-initial items- and a chain item which is non-initial must be final). Hence, for any $\mu \in V^*$ we have

$$CF\text{-}STRIP(CLOSURE(N(\mu))) = CF\text{-}STRIP(CLOSURE(CF\text{-}STRIP(N(\mu)))). \quad (4)$$

Using (4) and (3) and recalling the definition of the function CF-CLOSURE (Definition 3.40) we obtain

$$CFV(\theta) = CF\text{-}CLOSURE(CF\text{-}STRIP(\bigcup_{\mu \in \theta} N(\mu))). \qquad (5)$$

Then using part (ii) of Theorem 3.43 the argument of CF-CLOSURE in (5) may be simplified to yield

$$CFV(\theta) = CF\text{-}CLOSURE(CFN(\theta))$$

which concludes the lemma. □

Using these results we obtain the theorem upon which the algorithm for constructing CFLR(k) statesets depends.

THEOREM 3.46 (cf. Theorem 2.20)

Let $\theta \in V^*$ and $X \in V$. Then $CFV(\theta X) = CF\text{-}GOTO(CFV(\theta),X)$.

PROOF. This result is immediate from the two preceeding lemmas and the definition of the function CF-GOTO. □

We can now present an algorithm for computing CFLR(k) statesets. This algorithm is a straightforward adaption of the algorithm used for constructing ordinary LR(k) statesets.

ALGORITHM 3.47 (cf. Algorithm 2.23.)

Evaluation of the CFLR(k) statesets for (G,C).

Input : The cs-grammar (G,C) and a value for k.

Output : $CFS_k^{(G,C)}$ - the CFLR(k) stateset for (G,C).

Method : Just as in Algorithm 2.23, the stateset is built up in a set-valued variable S. Again, a marker flag is considered to be attached to each CFLR(k) state added to S; states are 'unmarked' when first added to S. During execution of the algorithm a tabluation of the function CF-GOTO(Δ,X) may be built up for all CFLR(k) states Δ in the stateset and all $X \in V$. As in the LR(k) case this tabluation will be needed during construction of the cf-parsing tables for (G,C).

```
begin
    compute CFV (λ) and set S = { CFV(λ) } ;
    while S contains any unmarked states do
        select an unmarked state Δ from S and mark it;
        for each X ∈ V do
            compute  Σ = CF-GOTO(Δ,X);
            if Σ ≠ ∅ and Σ is not in S then add
                                    Σ to S endif
        endfor
    endwhile;
    set CFS_k^(G,C) = S
end. □
```

In combination, Theorem 3.39 and Algorithm 3.47 provide a second direct method of testing for the CFLR(k) property along exactly the same lines as the corresponding LR(k) test outlined in the second proof of Theorem 2.8 (see Section 2.3). Observe that the notion of the adequacy of statesets is identical in both the LR(k) and CFLR(k) cases and that Algorithm 3.47 is basically identical to Algorithm 2.23. Consequently, if an implementation of the LR(k) test of Section 2.3 is available, it can be converted to a CFLR(k) test with very little programming effort - all that need be done is to substitute code for the evaluation of the function CF-GOTO in place of that for the function GOTO.

Also note that when C, the chain set, is empty, the CFLR(k) stateset degenerates into the ordinary LR(k) stateset and Algorithms 3.47 and 2.23 become identical. It therefore follows that the worst-case time complexity of this method of CFLR(k) testing is at least as bad as that of the corresponding LR(k) test - and that is exponential in the size of the grammar under test.

To conclude this section we display in Figure 3.6 the CFLR(1) stateset and CF-GOTO function for (G3,C3). Since no inadequacies are present in the stateset we conclude that this cs-grammar is CFLR(1) - as it must be since G3 is LR(1) . Observe that whereas the LR(1) stateset for G3 (see Figure 2.3) has 22 states, the CFLR(1) stateset for (G3,C3) has but 19. This reduction in the number of states is unusual; CFLR(k) statesets generally contain more states than their LR(k) counterparts.

We shall have more to say about the relative sizes of LR(k) and CFLR(k) statesets in Chapter 5.

STATE No.	CFLR(1) STATES NUCLEUS	COMPLETION	CF-GOTO S	E	T	P	(	X	)	*	+
1	[S→.E,Λ]	[E→.E+T,Λ,+] [T→.T*P,Λ,+,*] [P→.(E),Λ,+,*]		2	3	3	4	3			
2	[S→E.,Λ] [E→E.+T,Λ,+]										5
3	[S→E.,Λ] [E→E.+T,Λ,+] [T→T.*P,Λ,+,*]									6	5
4	[P→(.E),Λ,+,*]	[E→.E+T,),+] [T→.T*P,),+,*] [P→.(E),),+,*]		7	8	8	9	8			
5	[E→E+.T,Λ,+]	[T→.T*P,Λ,+,*] [P→.(E),Λ,+,*]			10	10	4	10			
6	[T→T*.P,Λ,+,*]	[P→.(E),Λ,+,*]				11	4	11			
7	[P→(E.),Λ,+,*] [E→E.+T,),+]								12		13
8	[P→(E.),Λ,+,*] [E→E.+T,),+] [T→T.*P,),+,*]								12	14	13
9	[P→(.E),),+,*]	[E→.E+T,),+] [T→.T*P,),+,*] [P→.(E),),+,*]		15	16	16	9	16			
10	[E→E+T.,Λ,+] [T→T.*P,Λ,+,*]									6	
11	[T→T*P.,Λ,+,*]										
12	[P→(E).,Λ,+,*]										
13	[E→E+.T,),+]	[T→.T*P,),+,*] [P→.(E),),+,*]			17	17	9	17			
14	[T→T*.P,),+,*]	[P→.(E),),+,*]				18	9	18			
15	[P→(E).,),+,*] [E→E.+T,),+]								19		13
16	[P→(E).,),+,*] [E→E.+T,),+] [T→T.*P,),+,*]								19	14	13
17	[E→E+T.,),+] [T→T.*P,),+,*]									14	
18	[T→T*P.,),+,*]										
19	[P→(E).,),+,*]										

Figure 3.6 : The CFLR(1) Stateset and CF-GOTO Function for the cs-grammar (G3,C3).

3.6. Testing for the CFLR(k) Property Directly - Part 3.

In this section we wish to develop a practical algorithm for testing for the CFLR(k) property in polynomial time. The algorithm which we construct will be akin to the LR(k) testing algorithm of Section 2.4. First of all we will adapt the LR(k) constructions in a straightforward manner and show that, while this approach does provide an acceptable algorithm, it also has certain drawbacks. We will then modify the construction slightly in order to achieve a more satisfactory solution. We choose this step by step approach to the final solution because it seems more perspicuous than a direct attack.

Recall that the techniques of Section 2.4 were based upon the enumeration of a set called $PAIRS_k^G$ consisting of all those pairs of LR(k) items which are simultaneuously valid for some viable prefix of the grammar. The next definition generalizes this concept in the natural manner.

DEFINITION 3.48 (cf. Definition 2.27)

The set $CF\text{-}PAIRS_k^{(G,C)}$ is defined by :

$$CF\text{-}PAIRS_k^{(G,C)} =$$

$$\{(\Delta, \Sigma) \mid \Delta \text{ and } \Sigma \text{ are CFLR(k) items for } (G,C) \text{ such that both } \Delta, \Sigma \in CFV(\theta) \text{ for some } \theta \in V^*\}. \square$$

The algorithms we shall develop will exploit the following result - which is an immediate corollary to Theorem 3.39.

THEOREM 3.49 (cf. Theorem 2.28)

(G,C) is CFLR(k) if and only if CF-PAIRS$_k^{(G,C)}$ contains no inadequate members. □

We now require a method for constructing the set CF-PAIRS$_k^{(G,C)}$.

In Section 2.4 we constructed a nondeterministic finite automaton M_k^G in order to enumerate the corresponding set PAIRS$_k^G$.

We adopt a similar construction in the present case and define an automaton $CFM_k^{(G,C)}$ to enable the enumeration of CF-PAIRS$_k^{(G,C)}$.

CONSTRUCTION 3.50 (cf. Construction 2.25)

The ENFA $CFM_k^{(G,C)} = (Q, q_0, F, I, \delta)$ is defined as follows :

(a) $Q = I_k^G \cup \{q_0\}$,

(b) F is irrelevant,

(c) $I = V$ (remember V is the vocabulary of G), and

(d) δ, the transition function is given by :

(i) $\delta(q_0, \Lambda) = \{ [S \rightarrow . \alpha, \Lambda] \mid S \rightarrow \alpha \in P \}$,

(ii) when q is of the form $q = [A \rightarrow \theta_1 . B\theta_2, u]$ with $B \in V_N$ then $\delta(q, \Lambda) =$

$$\{[B \rightarrow . \beta, v] \; B \rightarrow \beta \in P \text{ and } v \in FIRST_k(\theta_2 u)\}$$

(iii) when q is of the form $q = [A \rightarrow \theta_1 X \theta_2, u]$ with $X \in V$ then $\delta(q, Y) = \{[A \rightarrow \theta_1 X . \theta_2, u]\}$ for each $Y \in V$ such that $X \xrightarrow{*}_{\epsilon} Y$.

During the more informal parts of the subsequent discussion we shall write CFM rather than $CFM_k^{(G,C)}$ and M rather than M_k^G. □

Comparison of Constructions 2.25 and 3.50 will reveal that CFM differs from M only in its type (iii) transitions; whereas in M we have $\delta(\Delta, X) = \text{NEXT}(\{\Delta\}, X)$ for each LR(k) item Δ and symbol $X \in V$, in CFM we have $\delta(\Delta, X) = \text{CF-NEXT}(\{\Delta\}, X)$. Since the type (ii) transitions continue to fulfill the role of the CLOSURE function, it is easily seen that CFM possesses the following property :

LEMMA 3.51 (cf. Lemma 2.26)
Let $\theta \in V^*$. Then in $CFM_k^{(G,C)}$ we have

$$\delta(q_0, \theta) \frown CFI_k^{(G,C)} = CFV_k^{(G,C)}(\theta). \square$$

This result may be proved formally by a straightforward induction on the length of θ. From this lemma and the definition of the function STATE-PAIRS (Definition 2.29) we immediately obtain the following result :

LEMMA 3.52 (cf. Lemma 2.30)

$$CF\text{-}PAIRS_k^{(G,C)} =$$

$$\text{STATE-PAIRS}(CFM_k^{(G,C)}) \frown (CFI_k^{(G,C)} \times CFI_k^{(G,C)}). \square$$

The reader may wonder why the states of CFM comprise all the LR(k) items for G when we are ultimately interested only in the CFLR(k) items. The explanation is that the chain items (the LR(k) items which are not CFLR(k) items) are needed because of their contribution to the type (ii) transitions in CFM; although of no interest in themselves, they serve as intermediaries in sequences of λ-transitions from one CFLR(k) item to another. Strictly speaking, it is only the initial chain items which are needed for this purpose. However, it does no harm to also include the final chain items among the state of CFM since they are

discarded in Lemmas 3.51 and 3.52 by virtue of the intersections with $CFI_k^{(G,C)}$. By allowing the states of CFM to include all LR(k) items, the construction of this automaton is kept more uniform with that of M. We shall see later that this uniformity is useful.

In order to illustrate the construction in general and the points raised in the last paragraph in particular, we now introduce a simple example. We will use the following grammar :

S → Ax (Grammar G7)

A → B | x

B → y

and take C7 = {A → B} as the chain set. We use k=0 and display the transition diagram of $CFM_0^{(G7,C7)}$ in Figure 3.7. Note that in this figure we omit the second element (that is the lookahead string) when writing LR(0) items. This is because the lookahead string in LR(0) items is always $\wedge$ and so there is no need to indicate it explicitly.

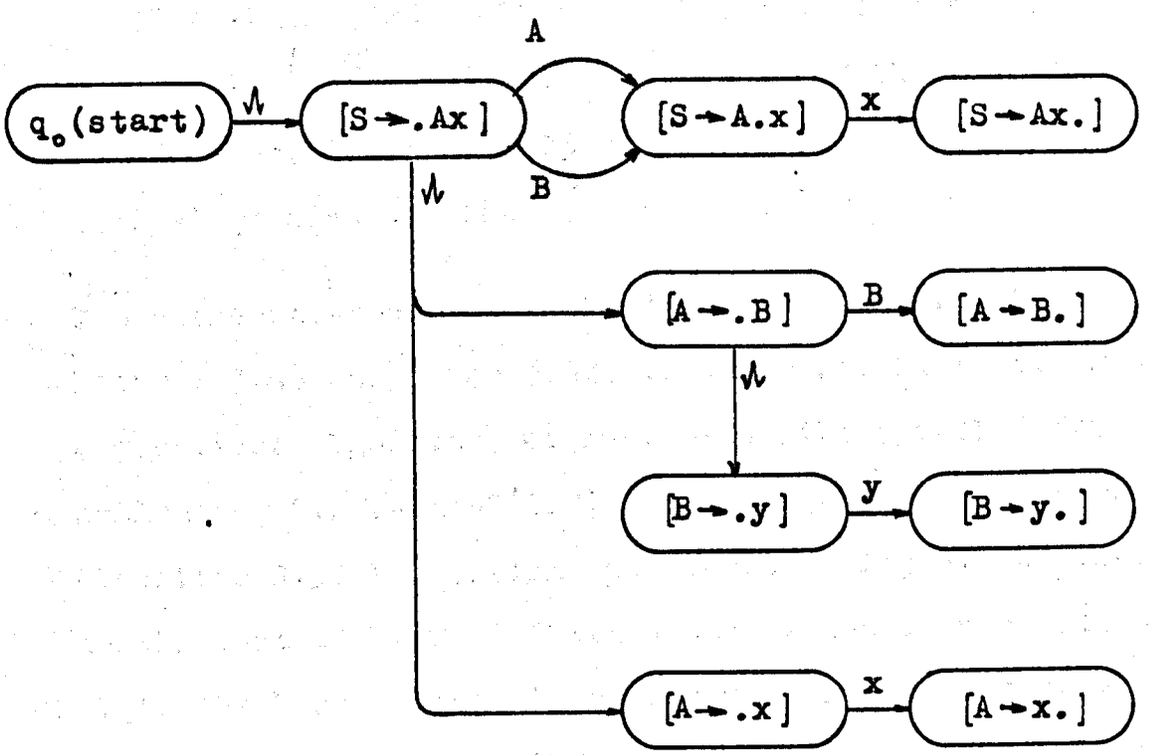


Figure 3.7 : The Transition Diagram of $CFM_0^{(G7,C7)}$ - the ENFA Corresponding to the cs-grammar (G7,C7) when k=0.

Observe in Figure 3.7 that the LR(0) chain item [A → .B] is needed in order to make CFLR(0) item [B → .y] accessible from the start state. Notice also that $CFM_0^{(G7,C7)}$ differs from M_0^{G7} only in the presence of the transition on B from [S → .Ax] to [S → A.x].

Our new method of testing for the CFLR(k) property is the following :

> Construct the automaton CFM and evaluate the set STATE-PAIRS(CFM). Then use Lemma 3.52 to produce $\text{CF-PAIRS}_k^{(G,C)}$ and test each member of this set for adequacy. Declare (G,C) to be CFLR(k) if no inadequacies are found.

The correctness of this method follows immediately from Theorem 3.49 and Lemma 3.52. In order for it to become a practical algorithm, we must prescribe a method for evaluating STATE-PAIRS(CFM). Now in Section 2.4 we used Algorithm 2.32 to perform the corresponding evaluation, namely that of STATE-PAIRS(M), but we cannot use this same algorithm here because it relies upon a property of M which is not shared by CFM. This property is that no state in M has transitions on more than one symbol from V. It is this property which allows us to speak of the OUTSYM of a state and it is used in steps 5 and 6 of Algorithm 2.32. In contrast, states in CFM may have transitions defined on many symbols from V. For example, in Figure 3.7 the state [S → .Ax]has transitions on both the symbols A and B. We will call the set of symbols in V for which a state in CFM has transitions defined the 'CF-OUTSYM' of the state.

DEFINITION 3.53 (cf. Definition 2.31)

Let q be a state in $\text{CFM}_k^{(G,C)}$. Then

$$\text{CF-OUTSYM}(q) = \{X \in V \mid \delta(q,X) \neq \emptyset\}. \quad \square$$

Note that although each state in CFM may have transitions defined for several symbols in V, there is no nondeterminism involved; for each state q and each symbol X ∈ CF-OUTSYM(q), there is but a single state in $\delta(q,X)$. Thus, just like M, all the nondeterminism in CFM is on the Λ-transitions and so it is only steps 5 and 6 of Algorithm 2.32 which need to be changed in order to cope with this new type of automaton. In this way we arrive at the following algorithm for evaluating STATE-PAIRS(CFM).

ALGORITHM 3.54 (cf. Algorithm 2.32)

Evaluation of STATE-PAIRS($CFM_k^{(G,C)}$).

Input : The ENFA $CFM_k^{(G,C)} = (Q,q_0,F,I,\delta)$.

Output: The set STATE-PAIRS($CFM_k^{(G,C)}$).

Method: The data structures and the procedure INSERT are retained unchanged from Algorithm 2.32. The output is represented by the bit matrix PAIRS.

```
begin
     INSERT (q0,q0 );
1.   while STACK is not empty do
2.        pop (p,q) from STACK;
3.        for each q' ∈ δ(p, Λ ) do INSERT(p,q') endfor;
4.        for each p' ∈ δ(q, Λ ) do INSERT(p',q) endfor;
5.        for each X ∈ V do
              if X ∈ CF-OUTSYM(p) and X ∈ CF-OUTSYM(q) then
                        INSERT(δ(p,X), δ(q,X))
              endif
          endfor
     endwhile
end. □
```

Observe that Algorithms 2.32 and 3.54 are identical except that the two steps 5 and 6 in the former are replaced by the single step 5 in the latter. The correctness of Algorithm 3.54 should be clear from the remarks which preceeded its introduction, so let us now consider its complexity. The work charged to each of its steps 1 to 4 will be exactly the same as that charged to these steps in Algorithm 2.32, that is $O(|Q|^2)$ to steps 1 and 2 and $O(|Q| . \sum_{q \in Q} |\delta(q,\Lambda)|)$ to steps 3 and 4. Since the numbers of states and Λ-transitions in CFM are the same as those in M, it follows that when the size of the grammar G is n, the work charged to steps 1 and 2 in Algorithm 3.54 is $O(n^{2k+2})$ while that charged to steps 3 and 4 is $O(n^{3k+3})$. These costs are just the same as those of the corresponding steps in Algorithm 2.32. However, the cost of each execution of step 5 in Algorithm 3.54 is $O(|I|)$ and so the total work charged to this step is $O(|Q| . |I|)$. Since I = V and $|V| = O(n)$ the cost of this step is therefore $O(n^{2k+3})$ and this should be compared with the $O(n^{2k+2})$ cost of steps 5 and 6 in Algorithm 2.32. Nevertheless, the overall complexity of Algorithm 3.54 is clearly dominated by the cost of steps 3 and 4 and so we see that this algorithm has the same complexity, that is $O(n^{3k+3})$, as Algorithm 2.32.

We claim that just as the overall complexity of the LR(k) test given in the third proof of Theorem 2.8 is dominated by the cost of evaluating the set STATE-PAIRS(M), so the complexity of the corresponding CFLR(k) test is dominated by the cost of evaluating STATE-PAIRS(CFM). If this evaluation is performed using Algorithm 3.54 then the

overall complexity of this method of testing for the CFLR(k) property will be $O(n^{3k+3})$. This is the same cost as that of testing for the ordinary LR(k) property - which seems a very satisfactory result. The situation becomes somewhat less satisfactory, however, when we consider the more efficient LR(k) tests of Hunt et al. (1974 and 1975).

As has been explained before, the method from the earlier of these references obtains its $O(n^{2k+2})$ time bound by dint of constructing a modified form of the automaton M in which the number of λ-transitions is reduced from $O(n^{2k+2})$ to $O(n^{k+1})$. This reduces the cost of steps 3 and 4 in Algorithm 2.32 from $O(n^{3k+3})$ to $O(n^{2k+2})$ and thereby reduces the overall complexity of the Algorithm, and hence of the entire LR(k) test, to $O(n^{2k+2})$. If we attempt to improve the efficiency of our CFLR(k) test in the same way, that is by reducing the number of λ-transitions, then the overall complexity of Algorithm 3.54 will become dominated by the cost of its step 5. Thus we would reduce the complexity of the CFLR(k) test to only $O(n^{2k+3})$ rather than to the target of $O(n^{2k+2})$. The $O(n^{k+2})$ LR(k) test of Hunt et al. (1975) will likewise yield an $O(n^{k+3})$ CFLR(k) test. It therefore seems necessary to examine Algorithm 3.54 more closely in order to see whether the $O(n^{2k+3})$ cost of step 5 is really necessary.

The costliness of this step is clearly due to its iterative nature. Now it is apparent from the definition of the type (iii) transitions in CFM that from any given state <u>all</u> the non λ-transitions lead to but <u>one</u> destination

state. That is, in each state q, $\delta(q,X) = \delta(q,Y)$ for all X,Y ∈ CF-OUTSYM(q). It follows that the effect of step 5 in Algorithm 3.54 can be accomplished more economically by the following non-iterative step :

> **if** (CF-OUTSYM(p) ∩ CF-OUTSYM(q)) ≠ ∅ **then**
> select any X ∈ CF-OUTSYM(p) and any
> Y ∈ CF-OUTSYM(q) and INSERT($\delta(p,X)$,$\delta(q,Y)$)
> **endif**

In this new step, instead of following all the transitions from p and q,we follow just one representative from each, having first ensured that there is at least one symbol on which a transition is defined in both states.

We now need some rapid means for testing the emptiness of the intersection CF-OUTSYM(p) ∩ CF-OUTSYM(q) and for selecting the representative symbols X and Y. Now when p is a state of CFM, its CF-OUTSYM set is empty if p is either the initial state or if it corresponds to a final LR(k) item. Otherwise p must correspond to a non-final item of the form $[A \rightarrow \theta_1.X\theta_2, u]$ and in this case CF-OUTSYM(p) = $\{ Y \in V \mid X \xrightarrow{+}_{c} Y \}$. But CFM shares the same states as M and so p may also be regarded as a state of M. Regarding it thus, we may speak of the OUTSYM of p (recall Definition 2.31) and this will be φ (i.e undefined) if p is either the initial state or if it corresponds a final LR(k) item. Otherwise p must correspond to a non-final LR(k) item of the form $[A \rightarrow \theta_1.X\theta_2, u]$ and

in this case OUTSYM(p) = X. We have just proved :

LEMMA 3.55

Let p be a state in $CFM_k^{(G,C)}$. Then it is also a state of M_k^G and

(a) CF-OUTSYM(p) = $\emptyset$ if and only if OUTSYM(p)=φ, and

(b) if CF-OUTSYM(p) $\neq \emptyset$ then

(i) CF-OUTSYM(p) = $\{X \in V \mid \text{OUTSYM}(p) \overset{*}{\underset{c}{\rightarrow}} X\}$ and so

(ii) OUTSYM(p) $\in$ CF-OUTSYM(p). □

We now define an equivalence relation on the vocabulary of G.

DEFINITION 3.56

Let C be a chain set for the grammar G. Define the equivalence relation $\underset{c}{\leftrightarrow}$ on V by $X \underset{c}{\leftrightarrow} Y$ if and only if there exists $Z \in V$ such that both $X \overset{*}{\underset{c}{\rightarrow}} Z$ and $Y \overset{*}{\underset{c}{\rightarrow}} Z$. □

From this definition and the preceeding lemma we immediately deduce the following result :

LEMMA 3.57

Let p and q be states in $CFM_k^{(G,C)}$.

CF-OUTSYM(p) $\cap$ CF-OUTSYM(q) $\neq \emptyset$ if and only if both

(a) OUTSYM(p) $\neq \varphi$ and OUTSYM(q) $\neq \varphi$, and

(b) OUTSYM(p) $\underset{c}{\leftrightarrow}$ OUTSYM(q). □

If a tabulation of the relation $\overset{c}{\leftrightarrow}$ is assumed to be available, then Lemma 3.57 indicates how the emptiness of the intersection CF-OUTSYM(p) $\cap$ CF-OUTSYM(q) can be decided at fixed cost. If this intersection is non-empty, then part b (ii) of Lemma 3.55 shows that the representative symbols X and Y from CF-OUTSYM(p) and CF-OUTSYM(q) can be provided by taking X = OUTSYM(p) and Y = OUTSYM(q).

Thus step 5 of Algorithm 3.54 can now be replaced by the two steps :

set X = OUTSYM(p) and Y = OUTSYM(q);
<u>if</u> $X \neq \varphi$ and $Y \neq \varphi$ and $X \overset{c}{\leftrightarrow} Y$ <u>then</u>
INSERT($\delta(p,X)$, $\delta(q,Y)$)
<u>endif</u>

Both these steps have fixed cost (provided a tabulation of $\overset{c}{\leftrightarrow}$ is available) which is what we require. But observe also an additional advantage conferred by this reformulation of step 5 : the only non-Λ transitions now involved are those of the form $\delta(p,X)$ where X = OUTSYM(p) and transitions of this type are identical in both CFM and M. Thus, in this modified form, Algorithm 3.54 will use only those transitions which are common to both CFM and M. This means that the algorithm now has the rather remarkable property that it evaluates STATE-PAIRS(CFM) directly from M. For completeness we state the modified algorithm in full.

ALGORITHM 3.58 (cf. Algorithms 2.32 and 3.54)

Evaluation of STATE-PAIRS($CFM_k^{(G,C)}$).

Input : The ENFA $M_k^G = (Q, q_0, F, I, \delta)$.

Output : The set STATE-PAIRS($CFM_k^{(G,C)}$).

Method : The data structures and the procedure INSERT are retained unchanged from Algorithm 2.32. The output is represented by the bit matrix PAIRS.

```
begin
      INSERT (q0,q0);
1.    while STACK is not empty do
2.          pop (p,q) from STACK;
3.          for each q' ∈ δ(p, ν) do INSERT(p,q') endfor;
4.          for each p' ∈ δ(q, ν) do INSERT(p',q) endfor;
5.          set X = OUTSYM(p) and Y = OUTSYM(q);
6.          if X ≠ φ and Y ≠ φ and X ≒ Y then
                        INSERT(δ(p,X), δ(q,Y))
            endif
      endwhile
end. □
```

Observe that this algorithm differs rrom Algorithm 2.32 only in the detail of its 6th step. Algorithm 2.32 has

```
if X ≠ φ and Y ≠ φ and X = Y then
                INSERT(δ(p,X), δ(q,Y))
endif
```

and Algorithm 3.58 differs only in that 'X = Y' is replaced by 'X ≒ Y'. Therefore, whenever both algorithms are applied to the same automaton, their complexities will be within a constant factor of each other. Now the rast LR(k) tests of Hunt et al.(1974 and 1975) reduce the cost of

Algorithm 2.32 by applying it to a variant of M (or, in the case of the later reference, to a series of variants) rather than to M itself. Since Algorithm 3.58 will work just as well when applied to these variants, its cost, too, may thereby be reduced to only $O(n^{2k+2})$ or $O(n^{k+2})$ as desired.

In its complete form, the basic $O(n^{3k+3})$ test for the CFLR(k) property is the following :

ALGORITHM 3.59 (cf. the third proof of Theorem 2.8)

Testing for the CFLR(k) property.

Input : The cs-grammar (G,C) to be tested and a value for k.

Output: 'Yes' if (G,C) is CFLR(k), otherwise 'no'.

Method: Tabulate the relation $\overset{c}{\leftarrow}$ and construct the automaton M_k^G. Then Use Algorithm 3.58 to evaluate the set STATE-PAIRS($CFM_k^{(G,C)}$) and use Lemma 3.52 to extract the set $CF\text{-}PAIRS_k^{(G,C)}$.

Test each member of this set for adequacy and output 'yes' if no inadequacies are found,'no' otherwise. □

This algorithm is similar in every way to the LR(k) test indicated in the third proof of Theorem 2.8 and, ignoring for the moment the cost of tabulating the relation $\overset{c}{\leftarrow}$, its complexity is therefore dominated by that of Algorithm 3.58. Since this can be reduced to $O(n^{k+2})$ by using the methods of Hunt et al.(1975) it follows that CFLR(0) testing can be performed in time $O(n^2)$. Hence it is necessary that the tabulation of the relation be performed in time $O(n^2)$ if this is not to dominate

the cost of CFLR(0) testing.

Now Definition 3.56 defines $\overset{*}{\underset{c}{\leftrightarrow}}$ in terms of the relation $\overset{*}{\underset{c}{\rightarrow}}$ and so it seems that tabulating $\overset{*}{\underset{c}{\leftrightarrow}}$ will be at least as expensive as tabulating the transitive closure of $\rightarrow$. This seems rather unpromising since all known algorithms for computing the transitive closures of arbitrary relations incur cost greater than $O(n^2)$ in the worst case. Fortunately however, the relation $\overset{*}{\underset{c}{\leftrightarrow}}$ has special properties which do allow it to be tabulated within the $O(n^2)$ time bound. In order to establish this fact we appeal to the following theorem which is due to Hunt et al.(1974, Theorem 6).

THEOREM 3.60

Let δ be an expression whose operands are relations having graphs with at most v vertices and e edges each and whose operators are chosen from composition, transitive closure, reflexive transitive closure, union and inverse. Then the relation denoted by δ can be computed in O(ve) steps. □

Now observe that the chain set C is really a relation on V and that the graph for C has $|V|$ (i.e. O(n)) vertices and as many edges as there are productions in C (i.e. at most O(n)). It follows that the product of the number of vertices and the number of edges in the graph of C is bounded by $O(n^2)$. The following identity is evident from Definition 3.57 :

$$\overset{*}{\underset{c}{\leftrightarrow}} = C^*.(C^{-1})^*$$

from which it follows by Theorem 3.60 that $\overset{*}{\underset{c}{\leftrightarrow}}$ can indeed be computed in time $O(n^2)$, where n is the size of the grammar concerned.

3.7. Chain Free Parsing the CFLR(k) Grammars.

In Section 3.3. we showed that a cf-parser for a CFLR(k) cs-grammar can be constructed by simply building the ordinary LR(k) parser for the corresponding cover grammar. The disadvantage of this indirect approach is that the LR(k) parser for the cover grammar can be very much larger than that for the basic grammar. For example, the LR(1) parser for G3 has but 22 states while that for COVER(G3,C3) has 73 states. Using this indirect approach, the speed benefits of cf-parsing are bought at the expense of inordinately large parsing tables.

We now show how cf-parsers for the CFLR(k) cs-grammars may be constructed directly. Recall that an ordinary LR(k) parser is formed by taking the basic table driven parser of Algorithm 1.4 and driving it with a set of LR(k)parsing tables. CFLR(k) cf-parsers are produced by an exactly analogous procedure: Algorithm 1.4 is retained but driven by a set of 'CFLR(k) parsing tables'. These are constructed from CFLR(k) statesets in just the same way as ordinary LR(k) tables are constructed from LR(k) statesets. Their formal definition is given by the following construction.

CONSTRUCTION 3.61 (cf. Construction 2.35)

The CFLR(k) parsing tables for (G,C) are denoted $CFT_k^{(G,C)}$ and are given by $CFT_k^{(G,C)} = (Q, s_o, g, f)$ where

(a) $Q = NAMES(CFS_k^{(G,C)})$,

(b) $s_o = NAMEOF(CFV(\not\Lambda))$,

(c) for all $\Delta \in CFS_k^{(G,C)}$ and $X \in V$.

$g(NAMEOF(\Delta), X) = NAMEOF(CF\text{-}GOTO(\Delta, X))$, and

(d) for all $\Delta \in CFS_k^{(G,C)}$ and $u \in V_T^{*k}$,

$$f(\text{NAMEOF}(\Delta), u) = \text{ACTION}(\Delta, u).$$

(The function ACTION was defined in Construction 2.35). □

Note that, just like the ordinary LR(k) case, the action function f in a set of CFLR(k) tables for (G,C) will be single-valued if and only if (G,C) is CFLR(k).

We now examine the theoretical properties of the CFLR(k) parsing algorithm, For the rest of this section we assume that we are concerned with the CFLR(k) parser for a cs-grammar (G,C) which is supposed to be CFLR(k) for the value of k concerned. As we should expect, the algorithm performs correctly when presented with valid input.

THEOREM 3.62 (cf. Theorem 2.36)

The CFLR(k) cf-parsing algorithm produces correct chain free parses for all inputs in L(G).

PROOF. The conditions which a set of cf-parsing tables must satisfy in order to drive Algorithm 1.4 correctly are given as part of the formal definition of such tables (Definition 3.13). These conditions are simply the chain free generalisations of those which ensure the correctness of ordinary LR(k) tables. Similarly, the CFLR(k) tables are themselves the natural chain free generalisation of LR(k) tables. Consequently, the proof of the correctness of the ordinary LR(k) parsing algorithm (see Theorem 2.36) may be adapted straightforwardly to the present case. We omit the details. □

The following result shows that the algorithm runs in linear time and space.

THEOREM 3.63 (cf. Theorem 2.37)

The number of moves made by the CFLR(k) cf-parsing algorithm in processing an input sentence of length n is O(n).

PROOF. It is, unfortunately, not sufficient to merely note that the CFLR(k) parser for (G,C) makes no more moves than the LR(k) parser for G and then appeal to the linear time result for LR(k) parsers. This is because there is no guarantee that G is LR(k). (Indeed there exist CFLR(k) cs-grammars (G,C) where G possesses sentences with (ordinary) parses of infinite length.) Instead we appeal to Theorems 3.25 and 3.26 and note that if (G,C) is CFLR(k) then COVER(G,C) is LR(k) and generates exactly the language L(G). Thus an LR(k) parser exists for COVER(G,C) and it is clear that its moves are in one-to-one correspondence with those of the CFLR(k) cf-parser for (G,C). The present theorem than follows directly from Theorem 2.37 which guarantees the linear time bound of the LR(k) parser for COVER(G,C). □

The final theorem shows that the CFLR(k) parsing algorithm retains the excellent error detection properties of the LR(k) algorithm.

THEOREM 3.64 (cf. Theorem 2.40)

Let $k > 0$ and let $x \in V_T^*$ be a string not in $L(G)$. Then the CFLR(k) parsing algorithm rejects x on :

(a) its first move if $EP(x) \leqslant k$, and

(b) the move following the $EP(x)-k$ 'th shift move if $EP(x) > k$.

PROOF. This result may be proved in a similar manner to the previous one - by appealing to the performance of the LR(k) parser for COVER(G,C). □ .

We end this section by displaying in Figure 3.8 the CFLR(1) parsing tables for (G3,C3) - these are obtained by applying Construction 3.61 to the CFLR(1) stateset for this cs-grammar which is shown in Figure 3.6.

STATE	CF-ACTION FUNCTION						CF-GOTO FUNCTION								
NO.	Λ	(	X	)	*	+	S	E	T	P	(	X	)	*	+
1		sh	sh					2	3	3	4	3			
2	1					sh									5
3	1				sh	sh								6	5
4		sh	sh					7	8	8	9	8			
5		sh	sh						10	10	4	10			
6		sh	sh							11	4	11			
7				sh		sh							12		13
8				sh	sh	sh							12	14	13
9		sh	sh					15	16	16	9	16			
10	2				sh	2								6	
11	4				4	4									
12	6				6	6									
13		sh	sh						17	17	9	17			
14		sh	sh							18	9	18			
15				sh		sh							19		13
16				sh	sh	sh							19	14	13
17				2	sh	2								14	
18				4	4	4									
19				6	6	6									

Figure 3.8 : $CFT_1^{(G3,C3)}$ - the CFLR(1) Parsing Tables for (G3,C3).

In Figure 3.9 we display the moves made by the CFLR(1) parsing algorithm for(G3,C3) - that is to say Algorithm 1.4 driven by the tables of Figure 3.8 - while processing the string X*(X+X). Figures 3.8 and 3.9 should be compared with their LR(1) counterparts which are shown in Figures 2.4 and 2.5 respectively. Observe that the CFLR(1) parser makes less than 60% of the moves made by the LR(1) parser when presented with the given input.

MOVE NO.	SYMBOL STACK CONTENTS	STATE STACK CONTENTS	UNCONSUMED INPUT	ACTION
1	↯	1	X*(X+X)	SHIFT
2	X	1,3	*(X+X)	SHIFT
3	X*	1,3,6	(X+X)	SHIFT
4	X*(	1,3,6,4	X+X)	SHIFT
5	X*(X	1,3,6,4,8	+X)	SHIFT
6	X*(X+	1,3,6,4,8,13	X)	SHIFT
7	X*(X+X	1,3,6,4,8,13,17	)	REDUCE E→E+T
8	X*(E	1,3,6,4,7	)	SHIFT
9	X*(E)	1,3,6,4,7,12	↯	REDUCE P→(E)
10	X*P	1,3,6,11	↯	REDUCE T→T*P
11	T	1,3	↯	REDUCE S→E and ACCEPT

Figure 3.9 : The Behaviour of the CFLR(1) cf-parser for (G3,C3) with Input X*(X+X).

3.8. Summary

This Chapter has introduced the idea of a chain specified grammar (a cs-grammar), that is a pair (G,C) where G is a context free grammar and C is a set of chain productions from G which are to be ignored during parsing. We have seen that, provided care is taken with the definitional framework, the basic ideas of parsing, and in particular the table driven bottom up parsing algorithm, can be generalised to accommodate the notion of chain free parsing (cf-parsing) - a form of parsing in which all chain productions are ignored. The major original contribution of this thesis lies in the introduction of the CFLR(k) cs-grammars. These are the largest class of cs-grammars which can be cf-parsed from left to right while looking k symbols ahead of the current point of the parse. The LR(k) grammars of Knuth are included as the special case in which the chain set C is empty.

The remainder of this chapter has been concerned with exploring the properties of the CFLR(k) cs-grammars. First we examined the relationship between the LR(k) and CFLR(k) properties and proved a significant result: if G is LR(k) and C is a chain set for G, then (G,C) is CFLR(k). Conversely, it was demonstrated that there exist CFLR(k) cs-grammars (G,C) in which the underlying grammar G is not LR(k), nor even unambiguous.

We then took a different tack and related the CFLR(k) property of a cs-grammar to the LR(k) property of a 'cover' grammar. In this way we were able to establish

that the LR(k) and CFLR(k) languages are co-extensive. This approach also provided theoretical solutions to the problems of testing for the CFLR(k) property and of constructing cf-parsers for the CFLR(k) cs-grammars. However, no practically feasible algorithms were provided by these techniques because the cover grammars were found to be very much larger than the cs-grammars which they covered.

In order to obtain practical CFLR(k) testing and parsing algorithms we then proceeded to generalise the corresponding LR(k) techniques. Each of the three methods of testing for the LR(k) property was succesfully generalised to test for the CFLR(k) property. It was shown that the greater generality of the CFLR(k) property need not increase the complexity of its decision procedures above those of the ordinary LR(k) case.

Finally, a method was presented for constructing table driven cf-parsers for the CFLR(k) cs-grammars. By virtue of their chain free nature, these parsers are faster than their LR(k) counterparts yet they preserve the same high quality of error detection. Their only disadvantage is that they may be rather larger than ordinary LR(k) parsers. In a later chapter (Chapter 5) we shall present an optimisation that removes this disadvantage.

We submit that the techniques presented here are the right and natural way to eliminate chain productions from LR(k) parsers. Unlike all other techniques (except that of Anderson (1972), which is fundamentally the same as our

own and, indeed, the source of its inspiration) ours places no restriction on the chain productions that may be eliminated (save only that none may have the goal symbol as its left part), nor does it add to the constraints upon the grammar: given an LR(k) grammar, any set of chain productions may be selected in the secure knowledge that a CFLR(k) chain free parser can be constructed.

It is also worth noting here that the similarity between the LR(k) and CFLR(k) testing and parser construction algorithms is such that, should an implementation of any of the LR(k) algorithms be available, then it may be converted into its CFLR(k) counterpart with only very modest effort.

CHAPTER 4.

CONVERTING LR(k) PARSERS INTO CHAIN FREE PARSERS

Section 3.7 described the standard method ror constructing CFLR(k) parsing tables. Nowwe shall reconsider this topic from another angle; we shall suppose that ordinary LR(k) parsing tables are already available and we shall seek to convert them directly into CFLR(k) tables.

We are concerned with this approach for two reasons : firstly because it could be useful in practice, and secondly because it will enable us (in Chapter 7) to relate our work to that of Aho and Ullman (1973b)and others.

It will be shown that tables constructed in this way are not always exactly the same as proper CFLR(k) parsing tables (in general they are much bigger) but their performance when used to drive Algorithm 1.4 is indistinguishable from that of CFLR(k) tables. We say that these new tables 'cover' the true CFLR(k) tables.

Later, we shall modify the technique in an attempt to reduce the size of tables produced. The modification takes the form of a simple, almost crude, optimisation, yet is is sufficient (in an important special case) to produce tables which are identical to true CFLR(k) parsing tables.

Much of the middle section of this chapter is concerned with elucidating the nature and properties or the special case alluded to above. This case concerns cs-grammars having

a property which we call 'Property A' and it is one which finds constant application throughout the remaining chapters.

4.1. The 'Post-Pass' Method for Constructing CFLR(k) Parsing Tables.

Although the goal of this chapter is to turn LR(k) parsing tables into CFLR(k) tables, we begin by discussing how the availability of LR(k) statesets can assist in the construction of CFLR(k) statesets. We do this because the technique is interesting in its own right, leads on naturally to the main topic, and enables us to explain some rather nice points that arise in Chapter 6.

The technique we have in mind depends upon associating with each CFLR(k) state, CFV(θ), a collection of LR(k) states given by :

$$\text{COLLECTION}(\theta) = \{V(\mu) \neq \emptyset \mid \mu \rightarrow_r^* \theta\}.$$

Now, by virtue of Theorem 3.46, we have

$$\text{CF-GOTO}(\text{CFV}(\theta),X) = \text{CFV}(\theta X) \tag{1}$$

and by virtue of Theorem 3.43 we have

$$\text{CFV}(\theta X) = \text{CF-STRIP}(\text{COLLECTION}(\theta X)) \tag{2}$$

while Theorem 2.20 gives

$$\text{COLLECTION}(\theta X) = \{\text{GOTO}(\Delta,Y) \neq \emptyset \mid \Delta \in \text{COLLECTION}(\theta),\ Y \rightarrow_r^* X\} \tag{3}$$

The identities (1), (2) and (3) effectively express CF-GOTO in terms of the functions CF-STRIP and GOTO. Now GOTO is only marginally easier to evaluate than CF-GOTO and so this result would be of little interest were it not for the fact that a tabulation of GOTO for all the arguments required in (3) is obtained as a by-product of the construction of the LR(k) stateset for G.

So if we have the LR(k) stateset and a tabulation of the GOTO function for our grammar G available we can reduce the evaluation of CF-GOTO to a series of table look-ups (for GOTO) and application of the (very simple) function CF-STRIP. In this way we can construct the CFLR(k) stateset for (G,C) and hence its CFLR(k) parsing tables.

Having introduced the basic idea, we must now give it a more practical realisation. In order to maintain uniformity with later constructions (which must manipulate LR(k) tables, not statesets) we shall associate with CFV(θ), not the set of actual LR(k) states COLLECTION(θ), but rather the set comprised of the names of these states. We shall call this set of names the "quasi CFLR(k) state for θ" and we shall define a function called ITEMS to enable LR(k) states to be recovered from their names. We shall also define a "quasi CF-GOTO function" to realise a version of the identity (3) above. Finally, instead of using these objects to construct the CFLR(k) stateset directly, we shall first construct a "quasi CFLR(k) stateset" composed of quasi CFLR(k) states and then convert this into the CFLR(k) stateset proper. We now give the formal definitions of these notions.

DEFINITION 4.1

Let $T_k^G = (Q, s_o, g, f)$ be the LR(k) parsing tables for G and let C be a chain set for G. We do not require that G be LR(k), nor that (G,C) be CFLR(k). For $\theta \in V^*$, define $QCFV_k^{(G,C)}(\theta)$, the quasi CFLR(k) state for θ, to be the subset of Q given by

$$QCFV_k^{(G,C)}(\theta) = \{NAMEOF(V(\mu)) \mid \mu \rightarrow_r^* \theta, V(\mu) \neq \emptyset\}.$$

As usual, we drop the sub and superscripts and write simply QCFV(θ) when it is safe to do so. We define $QCFS_k^{(G,C)}$ the quasi CFLR(k) stateset for (G,C), to be the set of all quasi states for (G,C) : $QCFS_k^{(G,C)} = \{QCFV(\theta) \neq \emptyset \mid \theta \in V^*\}$

and we define $QCF\text{-}GOTO_k^{(G,C)}$, the quasi CFLR(k) goto function for (G,C) as follows : when $M \subseteq Q$ and $X \in V$,

$$QCF\text{-}GOTO_k^{(G,C)}(M,X) = \{g(s,Y) \neq \varphi \mid s \in M, Y \rightarrow_r^* X\}$$

(recall that φ means 'undefined'). We use the term QCFLR(k) as an abbreviation for the phrase 'quasi CFLR(k)'. Finally, we define the function ITEMS to be the inverse of the NAMEOF function used in defining the set Q of LR(k) parsing states for G : for $s \in Q$, ITEMS(s) = $V(\theta)$ where $V(\theta)$ is the LR(k) state for G such that s = NAMEOF($V(\theta)$). It is convenient to extend the domain of ITEMS to sets of parsing states (such as quasi states) : when $M \subseteq Q$ define $ITEMS(M) = \bigcup_{s \in M} ITEMS(s)$. □

We require the following Lemma.

LEMMA 4.2

Let $\theta \in V^*$ and $X \in V$. Then

$$\text{QCF-GOTO}(\text{QCFV}(\theta),X) = \text{QCFV}(\theta X).$$

PROOF. We have

$$\begin{aligned}
\text{QCF-GOTO}(\text{QCFV}(\theta),X) &= \{g(s,Y) \neq \varphi \mid s \in \text{QCFV}(\theta), Y \xrightarrow{*}_{c} X\} \\
&= \{g(\text{NAMEOF}(V(\mu)),Y) \neq \varphi \mid \mu \xrightarrow{*}_{c} \theta, \\
&\qquad V(\mu) \neq \emptyset, Y \xrightarrow{*}_{c} X\} \\
&= \{\text{NAMEOF}(\text{GOTO}(V(\mu),Y)) \neq \varphi \mid \mu \xrightarrow{*}_{c} \theta, \\
&\qquad V(\mu) \neq \emptyset, Y \xrightarrow{*}_{c} X\} \\
&= \{\text{NAMEOF}(V(\mu Y)) \mid \mu \xrightarrow{*}_{c} \theta, Y \xrightarrow{*}_{c} X, \\
&\qquad V(\mu Y) \neq \emptyset\} \\
&= \text{QCFV}(\theta X). \ \square
\end{aligned}$$

This result ensures that QCF-GOTO has the property we require of a GOTO-type function and consequently the following algorithm may be used to construct QCFLR(k) statesets. Note that, as with all the other stateset construction algorithms, a tabulation of the goto function concerned (QCF-GOTO) may be produced during execution of this algorithm.

ALGORITHM 4.3

Construction of the quasi CFLR(k) stateset for (G,C).

Input : The cs-grammar (G,C), and $T_k^G = (Q, s_0, g, f)$ - the set of LR(k) parsing tables for (G,C).

Output : $QCFS_k^{(G,C)}$ - the QCFLR(k) stateset for (G,C).

Method : The method is similar to that of the earlier stateset construction algorithms. The quasi stateset is built up in the set-valued variable S; each quasi state has the usual marker flag attached to it and is unmarked when first added to S.

```
begin
    set S = {QCFV(λ)} ; (note QCFV(λ) = {s0})
    while S contains any unmarked quasi states do
        select an unmarked quasi state M from S
        and mark it;
        for each X ∈ V do
            compute N = QCF-GOTO(M,X);
            if N ≠ ∅ and N is not in S then
                add N to S endif
        endfor
    endwhile;
    set QCFS_k^(G,C) = S
end. □
```

To illustate this construction we show in Figure 4.1 the QCFLR(1) stateset and a tabulation of the QCF-GOTO function for the grammar (G3,C3). The numbers in the second column of this figure are the names of the LR(1) states comprising each quasi state. These names refer to the states of the LR(1) parsing tables for G3 given in Figure 2.4.

QUASI STATE No.	COMPONENT LR(1) PARSING STATES	QCF-GOTO								
		S	E	T	P	(	X	)	*	+
1	1		2	3	4	5	6			
2	2									7
3	2,3								8	7
4	2,3,4								8	7
5	5		9	10	11	12	13			
6	2,3,4,6								8	7
7	7			14	15	5	16			
8	8				17	5	18			
9	9							19		20
10	9,10							19	21	20
11	9,10,11							19	21	20
12	12		22	23	24	12	25			
13	9,10,11,13							19	21	20
14	14								8	
15	14,4								8	
16	14,4,6								8	
17	15									
18	15,6									
19	16									
20	17			26	27	12	28			
21	18				29	12	30			
22	19							31		20
23	19,10							31	21	20
24	19,10,11							31	21	20
25	19,10,11,13							31	21	20
26	20								21	
27	20,11								21	
28	20,11,13								21	
29	21									
30	21,13									
31	22									

Figure 4.1. The QCFLR(1) Stateset and QCF-GOTO Function for (G3,C3)

Observe from Figure 4.1 that the QCFLR(1) stateset for (G3,C3) contains 31 states whereas the true CFLR(1) stateset (Figure 3.6) contains only 19. Clearly, therefore, there is no simple one to one relationship between the quasi and the true CFLR(1) states for (G3,C3). However, the two sets of states <u>are</u> related: provided that the function ITEMS is available, the quasi stateset and QCF-GOTO function of Figure 4.1 can be converted into the true CFLR(1) stateset and CF-GOTO function of Figure 3.6. The CFLR(1) parsing tables for (G3,C3) can then be built using Construction 3.61 in the usual way. The proof of the following theorem indicates how this transformation may be performed in general.

THEOREM 4.4

Given the QCFLR(k) stateset and a tabulation of the QCF-GOTO function for (G,C), the functions ITEMS and CF-STRIP enable the corresponding CFLR(k) parsing tables to be constructed.

PROOF. In order to build CFLR(k) parsing tables using Construction 3.61 we need the CFLR(k) stateset and a tabulation of the CF-GOTO function. First we show how to convert the quasi CFLR(k) stateset for (G,C) into the true CFLR(k) stateset.

Let M be a set of LR(k) parsing states for G. Define the function ψ by $\psi(M) = \text{CF-STRIP}(\text{ITEMS}(M))$. Then we assert

Claim 1 : for each $\theta \in V^*$, $\psi(\text{QCFV}(\theta)) = \text{CFV}(\theta)$.

Proof of claim : Theorem 3.43 gives

$$\text{CFV}(\theta) = \text{CF-STRIP}(\{ V(\mu) \mid \mu \rightarrow_c^* \theta \})$$

and $\text{ITEMS}(\text{NAMEOF}(V(\mu))) = V(\mu)$ for all $\mu \in V^*$. Hence

$$\begin{aligned}\text{CFV}(\theta) &= \text{CF-STRIP}(\text{ITEMS}(\{ \text{NAMEOF}(V(\mu)) \mid \mu \rightarrow_c^* \theta, V(\mu) \neq \emptyset \})) \\ &= \text{CF-STRIP}(\text{ITEMS}(\text{QCFV}(\theta))) \\ &= \psi(\text{QCFV}(\theta))\end{aligned}$$

and the claim is proved.

The CFLR(k) stateset for (G,C) is given by

$$\text{CFS}_k^{(G,C)} = \{ \text{CFV}(\theta) \neq \emptyset \mid \theta \in V^* \}$$

and so, by the claim just proved,

$$\begin{aligned}\text{CFS}_k^{(G,C)} &= \{ \psi(\text{QCFV}(\theta)) \neq \emptyset \mid \theta \in V^* \} \\ &= \{ \psi(M) \neq \emptyset \mid M \in \text{QCFS}_k^{(G,C)} \}.\end{aligned}$$

This last identity reveals how the quasi CFLR(k) stateset for (G,C) can be simply converted into the true CFLR(k) stateset using just the function ψ (i.e. the composition of CF-STRIP and ITEMS).

To complete the proof we must show how a tabulation of CF-GOTO can be obtained from one of QCF-GOTO. Note that the function ψ is a surjective mapping from the set

$$\{ M \in QCFS_k^{(G,C)} \mid \psi(M) \neq \emptyset \}$$

to the true CFLR(k) stateset for (G,C). Consequently, we can find and tabulate a right inverse function ψ^{-1} for ψ; that is to say a function ψ^{-1} satisfying $\psi(\psi^{-1}(\Delta)) = \Delta$ for all CFLR(k) states Δ. (In general, ψ^{-1} will be neither unique nor a true inverse - but this is unimportant.) We now assert

Claim 2 : for each $\Delta \in CFS_k^{(G,C)}$ and each $X \in V$,

$$\psi(QCF\text{-}GOTO(\psi^{-1}(\Delta),X)) = CF\text{-}GOTO(\Delta,X).$$

Proof of Claim : let $\psi^{-1}(\Delta) = M$. Then $M \in QCFS_k^{(G,C)}$ and so there exists $\theta \in V^*$ such that $M = QCFV(\theta)$. Now by Lemma 4.2, $QCF\text{-}GOTO(QCFV(\theta),X) = QCFV(\theta X)$ and by claim 1 above, $\psi(QCFV(\theta X)) = CFV(\theta X)$. Hence

$$\psi(QCF\text{-}GOTO(\psi^{-1}(\Delta),X)) = CFV(\theta X)$$

and it only remains to show that $CFV(\theta X) = CF\text{-}GOTO(\Delta,X)$ or, equivalently, that $CFV(\theta) = \Delta$. But this is easy since Claim 1 gives $CFV(\theta) = \psi(QCFV(\theta))$ and, by construction, $QCFV(\theta) = M = \psi^{-1}(\Delta)$. Hence $CFV(\theta) = \psi(\psi^{-1}(\Delta)) = \Delta$ and the claim is proved.

Thus a tabulation of the CF-GOTO function can be produced from one for QCF-GOTO using only the function ψ and its right inverse. □

We call this approach to the construction of CFLR(k) parsers the "post pass" method. Clearly it is a rather roundabout process and we are certainly not proposing it as a practical method for building such tables. The important point is that we have shown that it can be done and that the tables produced must be the true CFLR(k) tables. Later (in Chapter 6) we shall encounter a situation in which cf-parsing tables constructed by the post-pass method are different to those constructed conventionally. Consequently, the more practical techniques to be introduced shortly will not generalise to that situation.

From a practical point of view, the unattractive feature of the post-pass method is the need to retain access (in the form of the function ITEMS) to the actual LR(k) states for which the parsing states are merely names. If we are to succeed in our goal of converting LR(k) tables directly into CFLR(k) tables then we must assume that LR(k) statesets are discarded once the tables have been built and we must therefore eschew the function ITEMS. This is the task of the next section.

4.2. Quasi CFLR(k) Parsing Tables.

Our original intention was to turn LR(k) tables directly into CFLR(k) tables. So far we have found how to use the LR(k) tables and stateset to ease the construction of the CFLR(k) stateset - from which the CFLR(k) tables may be constructed in the usual way. We now need to seek methods which bypass the need for an intermediate stage involving the CFLR(k) stateset.

Theorem 4.4 has shown that there is some relationship between quasi and true CFLR(k) statesets, and also between the corresponding CF-GOTO functions. It therefore seems plausible that a type of CFLR(k) parsing table could be built using these "quasi" objects: the names of the quasi states could furnish the parsing states while the QCF-GOTO function provides the goto function. The difficulty is to find some way of constructing an action function without referring to LR(k) items. The solution here is to realise that the ordinary LR(k) action function provides sufficient information to enable a "quasi CFLR(k) action function" to be constructed: each quasi state is a set of LR(k) parsing states and by combining the values of the action function for these component states, but excluding reduces by chain productions, we form a new action function of the required type. We define this construction formally as follows.

DEFINITION 4.5

Let $T_k^G = (Q,s_0,g,f)$ be the LR(k) parsing tables for G. Let $M \subseteq Q$, $u \in V_T^{*k}$ and let C be a chain set for G. Then define the value of QCF-ACTION(M,u) - the quasi cf-action function for (G,C) to be :

(a) SHIFT if f(s,u) = SHIFT for some s ∈ M,

(b) REDUCE q if q ∈ P\C and f(s,u) = REDUCE q for some s ∈ M

(c) ERROR if neither case (a) nor case(b) obtains. □

Thus equipped with a simple means for producing a quasi action function we can define the construction of quasi CFLR(k) parsing tables.

CONSTRUCTION 4.6 (cf. Construction 3.61)

The quasi CFLR(k) parsing tables for (G,C), denoted by $QCFT_k^{(G,C)} = (Q,s_0,g,f,)$, are constructed as follows :

(a) $Q = NAMES(QCFS_k^{(G,C)})$,

(b) $s_0 = NAMEOF(QCFV(\Lambda))$,

(c) for each $M \in QCFS_k^{(G,C)}$ and $X \in V$,
$g(NAMEOF(M),X) = NAMEOF(QCF\text{-}GOTO(M,X))$,

(d) for each $M \in QCFS_k^{(G,C)}$ and $u \in V_T^{*k}$,
$f(NAMEOF(M),u) = QCF\text{-}ACTION(M,u)$. □

An example of this construction is given in Figure 4.2. which displays the QCFLR(1) parsing tables for (G3,C3).

QUASI STATE No.	QCF-ACTION						QCF-GOTO								
	⋀	(	X	)	*	+	S	E	T	P	(	X	)	*	+
1		sh	sh					2	3	4	5	6			
2	1					sh									7
3	1				sh	sh								8	7
4	1				sh	sh								8	7
5		sh	sh					9	10	11	12	13			
6	1				sh	sh								8	7
7		sh	sh						14	15	5	16			
8		sh	sh							17	5	18			
9				sh		sh							19		20
10				sh	sh	sh							19	21	20
11				sh	sh	sh							19	21	20
12		sh	sh					22	23	24	12	25			
13				sh	sh	sh							19	21	20
14	2				sh									8	
15	2				sh									8	
16	2				sh									8	
17	4				4	4									
18	4				4	4									
19	6				6	6									
20		sh	sh						26	27	12	28			
21		sh	sh							29	12	30			
22				sh		sh							31		20
23				sh	sh	sh							31	21	20
24				sh	sh	sh							31	21	20
25				sh	sh	sh							31	21	20
26				2	sh	2								21	
27				2	sh	2								21	
28				2	sh	2								21	
29				4	4	4									
30				4	4	4									
31				6	6	6									

Figure 4.2 : $QCFT_1^{(G3,C3)}$ - the Quasi CFLR(1) Parsing Tables for (G3,C3).

These tables should be compared with the true CFLR(1) tables for (G3,C3) shown in Figure 3.8. It is clear that the two sets of tables are not the same; the quasi tables contain 31 states while the true ones contain only 19. We assert, however, that both sets of tables drive Algorithm 1.4 in exactly the same way. Formally we say that the quasi tables "cover" the true ones.

DEFINITION 4.7

Let $T = (Q,s_0,g,f)$ and $T' = (Q',s_0',g',f')$ be a pair of cf-parsing tables for the cs-grammar(G,C) using the same amount, k, of lookahead. Let H be a mapping $H: Q \to Q'$. Then we say that T *covers* T' *under* H if

(a) H is surjective,

(b) $H(s_0) = s_0'$,

(c) for each $s \in Q$ and $X \in V$, $H(g(s,X)) = g'(H(s),X)$, and

(d) for each $s \in Q$ and $u \in V_T^{*k}$, $f(s,u) = f'(H(s),u)$.

We say simply that T *covers* T' if some H exists such that T covers T' under H. If H is bijective then we say that T and T' are *equivalent*. □

If one set of tables T covers another set T' under a function H, then Definition 4.7 indicates that the role of any parsing state s in T is mirrored exactly by that of the state H(s) in T'. This means that by observing only the external behaviour of Algorithm 1.4 we will be unable to tell whether it is being driven by T or by T'; any difference between these sets of tables is to be found only in their internal structure. Since H is surjective, there may be more states in T than in T'. The tables T' are therefore more economical than T, and so may be considered more desirable, but are otherwise indistinguishable from them. Equivalent sets of tables are, to all intents and purposes, absolutely identical to each other.

We claim that the QCFLR(k) tables for any grammar (G,C) cover the corresponding true CFLR(k) tables. However, we shall not prove this fact as we prefer to postpone our proofs until the next section - in which we introduce a modified form of QCFLR(k) tables which contain rather fewer states.

4.3. Strong Quasi CFLR(k) Parsing Tables.

The QCFLR(k) parsing tables introduced in the previous section provide a solution to the problem of converting LR(k) parsing tables into chain free parsing tables. Unfortunately, this method suffers from the disadvantage that it produces tables which may be substantially larger than true CFLR(k) tables. For this reason QCFLR(k) tables are rather unattractive from a practical point of view. Now although it is possible to reduce the size of QCFLR(k) tables by using techniques akin to those for minimising finite state automata, it is much more interesting to enquire why these tables are so much larger than true CFLR(k) tables in the first place, and to ask whether anything can be done to mitigate this effect at its source.

The reason why QCFLR(k) tables contain more states than CFLR(k) tables is quite simple: it is due to the fact that $CFV(\theta) = CFV(\mu)$ does not imply $QCFV(\theta) = QCFV(\mu)$. Thus although the CFLR(k) states $CFV(\theta)$ and $CFV(\mu)$ are identified in the true CFLR(k) stateset, the quasi states $QCFV(\theta)$ and $QCFV(\mu)$ may be distinguished (unnecessarily) in the QCFLR(k) stateset. One particularly simple circumstance in which this can happen is when

$$QCFV(\theta) = QCFV(\mu) \cup M$$

where $M \neq \emptyset$ is some set of LR(k) parsing states such that $CF\text{-}STRIP(ITEMS(M)) = \emptyset$. We call such sets M "cf-useless". Now although this is not the only circumstance which can cause $QCFV(\theta)$ and $QCFV(\mu)$ to be distinguished

unnecessarily, it is a very common one. For example, in Figure 4.1 the quasi states numbered 14,15 and 16 are distinguished from one another ror just this reason.

It would seem a good idea to try and exclude cf-useless sets from quasi states. We have to ask :

(a) how, without using the function ITEMS can cf-useless sets be identified, and

(b) if they can be identified, does excluding them rrom quasi states do any harm?

The answer to (a) is straightforward: if M is any set of LR(k) parsing states, then M is cf-useless if and only if QCF-ACTION(M,u) = ERROR for all $u \in V_T^{*k}$. Furthermore, excluding cf-useless sets rrom quasi states cannot alter the value of the quasi action function, for if M is cf-useless and N is any other set of LR(k) parsing states then

$$\text{QCF-ACTION}(M \cup N,u) = \text{QCF-ACTION}(N,u)$$

for all $u \in V_T^{*k}$. Both of these results are trivial deductions from the definition of the function QCF-ACTION (Definition 4.5.).

We shall now exploit these observations and define a "strong " version of our QCFLR(k) constructions in which cf-useless sets are excluded from quasi states.

DEFINITION 4.8 (cf. Definition 4.1)

Let $T_k^G = (Q, s_o, g, f)$ be the LR(k) parsing tables for G and let C be a chain set for G. For each $\theta \in V^*$ define $SQCFV_k^{(G,C)}(\theta)$, the strong quasi CFLR(k) state for θ by :

$$SQCFV_k^{(G,C)}(\theta) = \{NAMEOF(V(\mu)) \mid \mu \overset{*}{\underset{c}{\rightarrow}} \theta, CF\text{-}STRIP(V(\mu)) \neq \emptyset\}.$$

Define $SQCFS_k^{(G,C)}$, the strong quasi CFLR(k) stateset for (G,C) as follows :

$$SQCFS_k^{(G,C)} = \{SQCFV(\theta) \neq \emptyset \mid \theta \in V^*\}.$$

Finally, define $SQCF\text{-}GOTO_k^{(G,C)}$, the strong quasi CFLR(k) goto function for (G,C) as follows.: when $M \subseteq Q$ and $X \in V$,

$$SQCF\text{-}GOTO_k^{(G,C)}(M,X) = \{g(s,Y) \mid s \in M,\ Y \overset{*}{\underset{c}{\rightarrow}} X, \text{ and } g(s,Y) \text{ is not cf-useless}\}.$$

We use the term SQCFLR(k) as an abbreviation for the phrase 'strong quasi CFLR(k)'. □

The results which we need in order to establish the new method and its correctness are provided by the following lemma.

LEMMA 4.9

Let $\theta \in V^*$ and $X \in V$. Then

(i) CFV(θ) = CF-STRIP(ITEMS(SQCFV(θ))),

(ii) CFV(θ) = $\emptyset$ if and only if SQCFV(θ) = $\emptyset$, and

(iii) SQCFV(θX) = SQCF-GOTO(SQCFV(θ),X).

PROOF. Part (i) : It is immediate from their definitions that QCFV(θ) and SQCFV(θ) differ only in that SQCFV(θ) contains no cf-useless states. Hence CF-STRIP(ITEMS(SQCFV(θ))) = CF-STRIP(ITEMS(QCFV(θ))). The result then follows from Claim 1 of the proof of Theorem 4.4.

Part (ii) : The result in the 'if' direction is immediate from part (i) of this lemma. For the 'only if' direction suppose that CFV(θ) = $\emptyset$. Then part (i) provides CF-STRIP(ITEMS(SQCFV(θ))) = $\emptyset$ and this can only be so if SQCFV(θ) is either empty or if it consists solely of cf-useless states. The latter possibility is excluded by the definition of strong quasi states and so the result follows.

Part (iii) : This may be proved by a similar argument to that used to establish the corresponding result (Lemma 4.2) for the ordinary quasi case. □

Part (iii) of this lemma indicates that SQCFLR(k) statesets may be constructed by a simple modification of the algorithm for constructing ordinary QCFLR(k) statesets.

ALGORITHM 4.10 (cf. Algorithm 4.3)

Construction of the strong quasi CFLR(k) stateset for (G,C)

Input : The cs-grammar (G,C), and $T_k^G = (Q,s_o,g,f)$ - the set of LR(k) parsing tables for (G,C).

Output: $SQCFS_k^{(G,C)}$ - the SQCFLR(k) stateset for (G,C).

Method : The method is just that of the ordinary QCFLR(k) case (Algorithm 4.3) but with the function QCF-GOTO replaced by its 'strong' counterpart : SQCF-GOTO. □

SQCFLR(k) parsing tables are also constructed in just the same way as ordinary QCFLR(k) tables.

CONSTRUCTION 4.11 (cf. Construction 4.6.)

The strong quasi CFLR(k) parsing tables for (G,C), denoted by $SQCFT_k^{(G,C)} = (Q,s_o,g,f)$ are constructed as follows :

(i) $Q = NAMES(SQCFS_k^{(G,C)})$,

(ii) $s_o = NAMEOF(SQCFV(\Lambda))$,

(iii) for each $M \in SQCFS_k^{(G,C)}$ and $X \in V$,
$g(NAMEOF(M),X) = NAMEOF(SQCF\text{-}GOTO(M,X))$,

(iv) for each $M \in SQCFS_k^{(G,C)}$ and $u \in V_T^{*k}$,
$f(NAMEOF(M),u) = QCF\text{-}ACTION(M,u)$. □

To illustrate this construction we show in Figure 4.3 the SQCFLR(1) stateset for (G3,C3). Observe that this stateset contains just 19 states - exactly the same number as in the true CFLR(1) stateset for this grammar shown in Figure 3.6. We do not illustrate the SQCFLR(1) parsing tables for (G3,C3) since these are exactly the same as the true CFLR(1) tables shown in Figure 3.8. Thus in this case our "strong"

STRONG QUASI STATE No	COMPONENT LR(1) PARSING STATES	SQCF-GOTO								
		S	E	T	P	(	X	)	*	+
1	1		2	3	3	4	3			
2	2									5
3	2,3								6	5
4	5		7	8	8	9	8			
5	7			10	10	4	10			
6	8				11	4	11			
7	9							12		13
8	9,10							12	14	13
9	12		15	16	16	9	16			
10	14								6	
11	15									
12	16									
13	17			17	17	9	17			
14	18				18	9	18			
15	19							19		13
16	19,10							19	14	13
17	20								14	
18	21							.		
19	22									

Figure 4.3 : The SQCFLR(1) Stateset and SQCF-GOTO Function for (G3,C3).

version of the "quasi" construction has succeeded in eliminating all the extra states introduced by the basic method. Unfortunately, it is not always so successful: in general, SQCFLR(k) tables are not equivalent to true CFLR(k) tables, they merely cover them. We now prove this fact. First we need two lemmas.

LEMMA 4.12

Let $\theta \in V^*$ and $u \in V_T^{*k}$. Then

$$\text{QCF-ACTION}(\text{SQCFV}(\theta),u) = \text{ACTION}(\text{CFV}(\theta),u).$$

PROOF. Suppose QCF-ACTION(SQCFV(θ),u) = SHIFT. Then, by Definition 4.5, SQCFV(θ) contains the name of some LR(k) state V(μ) such that ACTION(V(μ),u) = SHIFT. Now it can be shown by a straightforward, but rather tedious, argument that if ACTION(CFV(μ),u) = SHIFT, then ACTION(CF-STRIP(CFV(μ)),u) = SHIFT also. But if the name of V(μ) is a member of SQCFV(θ) it must be that $\mu \xrightarrow{*}_{\varepsilon} \theta$ and so, by Theorem 3.43, it follows that CF-STRIP(V(μ)) $\subseteq$ CFV(θ). Thus ACTION(CF-STRIP(V(μ)),u) = SHIFT implies ACTION(CFV(θ),u) = SHIFT and the result is proved for this case.

The result can be established for the remaining cases (i.e. REDUCE and ERROR actions) by similar arguments. □

LEMMA 4.13

Define the relation $\mathfrak{d}$ between the SQCFLR(k) and CFLR(k) statesets for (G,C) by :

$\mathfrak{d} = \{(M, \Delta) \mid M = \text{SQCFV}(\theta)$ and $\Delta = \text{CFV}(\theta)$ for some $\theta \in V^*\}$.

Then $\mathfrak{d}$ is a surjection.

PROOF. First we show that $\mathfrak{d}$ is indeed a function, that is to say it is single-valued.

Claim : $M \mathfrak{d} \Delta$ and $M \mathfrak{d} \Sigma$ implies $\Delta = \Sigma$.

Proof of claim. If $M \mathfrak{d} \Delta$ there exists $\theta \in V^*$ such that $M = \text{SQCFV}(\theta)$ and $\Delta = \text{CFV}(\theta)$. Hence, by part (i) of Lemma 4.9, $\Delta = \text{CF-STRIP}(\text{ITEMS}(M))$. Similarly, $M \mathfrak{d} \Sigma$ implies $\Sigma = \text{CF-STRIP}(\text{ITEMS}(M))$ and the conclusion $\Delta = \Sigma$ is immediate.

It remains to show that $\bar{\Phi}$ is a mapping - that is that its domain is the whole SQCFLR(k) stateset, and that it is surjective - that is its range is the whole CFLR(k) stateset. Both these properties are trivial consequences of part (ii) of Lemma 4.9. □

Using these result we can justify our claim that SQCFLR(k) parsing tables cover the corresponding CFLR(k) tables.

THEOREM 4.14

Let $SQCFT_k^{(G,C)} = (Q,s_0,g,f)$ and $CFT_k^{(G,C)} = (Q',s_0',g',f')$ be the SQCFLR(k) and CFLR(k) parsing tables respectively for (G,C).
Then $SQCFT_k^{(G,C)}$ covers $CFT_k^{(G,C)}$.

PROOF. First we need to construct a surjection H from Q to Q'. Now the previous lemma has provided a surjection $\bar{\alpha}$ between the statesets from which these sets of parsing states are constructed and so we can establish H as follows. Let s ∈ Q. Then s = NAMEOF(M) for some SQCFLR(k) state M. Define
H(s) = MAMEOF($\bar{\alpha}$(M)). Because $\bar{\alpha}$ is known to be a surjection and the NAMEOF functions are bijective, it follows that H is a surjection. We now prove that the SQCFLR(k) tables cover the CFLR(k) tables under this H.

Claim 1: $H(s_0) = s_0'$.

Proof of Claim. By Construction 4.11, s_0 = NAMEOF(SQCFV(λ)) and so $H(s_0)$ = NAMEOF($\bar{\alpha}$(SQCFV(λ))). But $\bar{\alpha}$(SQCFV(λ)) = CFV(λ) and, by Construction 3.61, NAMEOF(CFV(λ)) = s_0'. Hence $H(s_0) = s_0'$ and the claim is proved.

Claim 2 : $H(g(s,X)) = g'(H(s),X)$ for all s ∈ Q and X ∈ V.

Proof of Claim : If s e Q then s = NAMEOF(SQCFV(θ)) for some θ e V^* and so H(s) = NAMEOF(CFV(θ)). By Construction 4.11 we have g(s,X) = NAMEOF(SQCF-GOTO(SQCFV(θ),X)) and by part (iii) of Lemma 4.9 this gives g(s,X) = NAMEOF(SQCFV(θX)). Hence H(g(s,X)) = NAMEOF(E(SQCFV(θX)))

= NAMEOF(CFV(θX)). (1)

Now g'(H(s),X) = g'(NAMEOF(CFV(θ)),X) and so Construction 3.61 provides g'(H(s),X) = NAMEOF(CF-GOTO(CFV(θ),X)) which can be simplified using Theorem 3.46 to give

g'(H(s),X) = NAMEOF(CFV(θX)). (2)

The claim then follows from (1) and (2).

Claim 3 : f(s,u) = f'(H(s),u) for all s e Q and u e V_T^{*k}.

Proof of Claim. Again s e Q implies s = NAMEOF(SQCFV(θ)) and so, by Construction 4.11, f(s,u) = QCF-ACTION(SQCFV(θ),u). But, by Lemma 4.12, QCF-ACTION(SQCFV(θ),u) = ACTION(CFV(θ),u) and since Construction 3.61 gives f'(NAMEOF(CFV(θ)),u) = ACTION(CFV(θ),u), it follows that f(s,u) = f'(NAMEOF(CFV(θ)),u). The claim then follows because NAMEOF(CFV(θ)) = H(s).

All the conditions of Definition 4.7 have now been satisfied and so we conclude the theorem. □

This result shows that SQCFLR(k) tables are perfectly valid chain-free parsing tables, although they may be somewhat larger than true CFLR(k) tables. It is interesting to enquire whether any conditions can be found which ensure the complete equivalence of these two types of tables. Now it can be shown that this equivalence holds when G is LR(0) - but this result is too restrictive to be of any practical interest. From a practical point of view (and it is only from this point of view that the size of parsing tables is of any concern), the only important case is k = 1. Unfortunately, the requirement that G be LR(1) is not a sufficient condition for the result we seek. The following grammar demonstrates this point.

S	⟶	aAb |	(Grammar G8)
		aB |	
		DAb |	
		bB |	
A	⟶	C	
B	⟶	C	
C	⟶	Λ	
D	⟶	b	

This grammar is LR(1) but the SQCFLR(1) tables for(G8, $\{D \rightarrow b\}$) contain 13 states whereas the true CFLR(1) tables contain only 12.

In the next section we define a property called "Property A" which <u>is</u> sufficient to guarantee the equivalence of SQCFLR(k) and CFLR(k) parsing tables.We provide results which indicate that most LR(1) grammars may be expected to have this property.

4.4. 'Property A'.

We begin with a definition.

DEFINITION 4.15

The cs-grammar (G,C) has Property A if, whenever α and β are viable prefixes of G, the existence of $\rho \in V^*$ such that $\alpha \xrightarrow[c]{*} \rho$ and $\beta \xrightarrow[c]{*} \rho$ always implies that $\alpha = \theta X$ and $\beta = \theta Y$ for some $\theta \in V^*$ and $X,Y \in V$. (In other words, α and β may differ only in their final symbols.) □

This seemingly obscure property turns out to be extremely useful : in the next section we shall prove that if (G,C) has Property A then its CFLR(k) and SQCFLR(k) parsing tables are equivalent. This present section is concerned with the problem of testing for Property A and with determining how likely it is that a given grammar will possess the property. In fact we do not present algorithms for testing for Property A directly; instead we give a series of easily tested conditions which are sufficient to guarantee the property. These conditions suggest that Property A is possessed by all LR(1) grammars of the type likely to be encountered in practice. Since the property will only be invoked to prove results of purely practical interest, this restriction to the case k = 1 is perfectly acceptable.

The conditions we seek follow as corollaries to the next theorem. The following definition is needed during the proof and is also used in Chapter 6.

DEFINITION 4.16

Let (G,C) be a cs-grammar and let $M,N \subseteq V$. Then (M,N) is a maximally chained pair if each $X \in M$ and each $Y \in N$ satisfy $X \xrightarrow{*}_c Y$. A Symbol $W \in V$ is an intermediate for such a maximally chained pair if all $X \in M$ satisfy $X \xrightarrow{*}_c W$ and all $Y \in N$ satisfy $W \xrightarrow{*}_c Y$. An intermediate W is said to be a maximal intermediate if no other intermediate U satisfies $U \xrightarrow{+}_c W$. □

THEOREM 4.17

Let G be an LR(k) grammar where $k > 0$ and let C be a chain set for G. Let $\alpha = \theta X \delta$ and $\beta = \theta Y \delta$ be viable prefixes of G with $X \neq Y$ and such that for some $Z \in V$ and $\mu \in V^*$ both $X\delta \xrightarrow{*}_c Z\mu$ and $Y\delta \xrightarrow{*}_c Z\mu$. Then every $x \in V_T^*$ such that $\mu \xrightarrow{*} x$ satisfies $\text{len}(x) < k$.

PROOF. Since we have $X\delta \xrightarrow{*}_c Z\mu$ and $Y\delta \xrightarrow{*}_c Z\mu$, we must have $X \xrightarrow{*}_c Z$ and $Y \xrightarrow{*}_c Z$. We distinguish three cases according to whether or not X and Y chain derive each other.

Case 1: $Y \xrightarrow{*}_c X$. Because $X \neq Y$ we must have $Y \xrightarrow{+}_c X$ and so there exists $A \in V_N$ such that $Y \xrightarrow{*}_c A \rightarrow_c X$. Now $\beta = \theta Y \delta$ is a viable prefix and so there is some $\sigma \in V^*$ such that $S \xrightarrow{*}_R \theta Y \delta \sigma$. Let $y \in V_T^*$ be any string such that $\sigma \xrightarrow{*}_R y$ (such a string must exist because G is LR(k) and must therefore be reduced) and let $x \in V_T^*$ be such that $\mu \xrightarrow{*} x$. Since $\delta \xrightarrow{*}_c \mu$ and $\mu \xrightarrow{*} x$ it follows that $\delta \xrightarrow{*}_R x$ and so we have :

$S \xrightarrow{*}_R \theta Y \delta \sigma \xrightarrow{*}_R \theta Y \delta y \xrightarrow{*}_R \theta Y x y \xrightarrow{*}_R \theta A x y \rightarrow_R \theta X x y$.

Let the production $A \rightarrow X$ be called p and let $m = \text{len}(\theta X)$. Then it follows from this derivation that (p,m) is the handle of $\theta X x y$.

We now consider the string $\alpha = \theta X \gamma$. Since α is a viable prefix of G, there exists $\eta \in V^*$ such that $\alpha\eta$ is an rsf of G with a handle (q,n) satisfying $n \geq \text{len}(\alpha)$. Let $z \in V_T^*$ be any string such that $\eta \rightarrow^* z$. Since $\alpha = \theta X \gamma$, $\gamma \rightarrow_\varepsilon^* \mu$ and $\mu \rightarrow^* x$, G contains the following derivation for some $\psi \in V^*$:

$$S \rightarrow_R^* \psi \text{—}(q,n)\text{→}_R \theta X \gamma \eta \rightarrow_R^* \theta X \gamma z \rightarrow_R^* \theta X x z. \qquad (1)$$

We now distinguish two subcases according to the number of steps in the derivation $\gamma\eta \rightarrow_R^* xz$.

<u>Subcase (a)</u> : $\gamma\eta = xz$ (i.e. no steps at all). Then (q,n) is the handle of $\theta X x z$. Suppose $\text{len}(x) \geq k$. Then $(m+k):\theta X x y = (m+k):\theta X x z$ and certainly $m/\theta X x z \in V_T^*$. Since G is LR(k), these conditions imply that $(p,m) = (q,n)$ - but this is impossible because we have $n \geq \text{len}(\alpha)$, $\text{len}(\alpha) = m + \text{len}(\gamma)$, $\gamma \rightarrow^* x$, $\text{len}(x) \geq k > 0$, and so $\text{len}(\gamma) > 0$. We conclude that the supposition $\text{len}(x) \geq k$ is untenable.

<u>Subcase (b)</u> : $\gamma\eta \rightarrow_R^+ xz$ (i.e. at least one step). We can distinguish the last step of this derivation and write

$$\gamma\eta \rightarrow_R^* \pi \text{—}(q',n')\text{→}_R xz.$$

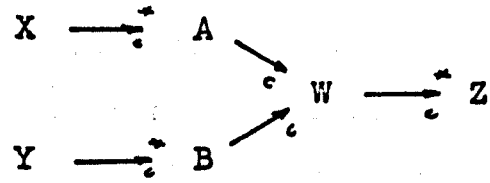
Note that if $n' = 0$ then $\deg(q') = 0$ also. The derivation (1) above then gives :

$$S \rightarrow_R^* \theta X \gamma \eta \rightarrow_R^* \theta X \pi \text{—}(q',n'+m)\text{→}_R \theta X x z$$

and so we see that $(q',n'+m)$ is the handle of $\theta X x z$. Now suppose that $\text{len}(x) \geq k$. As before the fact that G is LR(k) must imply that $(p,m) = (q',n'+m)$. This can only be so if $n' = 0$. But $n' = 0$ implies $\deg(q') = 0$ and we know that $\deg(p) = 1$ (remember that p is the production $A \rightarrow X$). Hence $(p,m) \neq (q',n'+m)$ and from this contradiction we again conclude that the supposition that $\text{len}(x) \geq k$ must be false.

Case 2 : $X \rightarrow_c^* Y$. The proof in this case is exactly analagous to the previous one.

Case 3 : $X \not\rightarrow_c^* Y$ and $Y \not\rightarrow_c^* X$. Since we have $X \rightarrow_c^* Z$ and $Y \rightarrow_c^* Z$, $(\{X, Y\}, \{Z\})$ is a maximally chained pair. Let W be the maximal intermediate for this pair. Note that W must exist and must be unique for otherwise G would be ambiguous and therefore could not be LR(k). Note also that $W \neq X$ and $W \neq Y$. (If W = X, for instance, then $Y \rightarrow_c^* W$ implies $Y \rightarrow_c^* X$ and this situation is excluded in the present case.) We therefore have $X \rightarrow_c^+ W$ and $Y \rightarrow_c^+ W$ and so there exist $A,B \in V_N$ such that $X \rightarrow_c^* A \rightarrow_c W$ and $Y \rightarrow_c^* B \rightarrow_c W$. Note that $A \neq B$ for $A = B$ implies that A is an intermediate which satisfies $A \rightarrow_c^+ W$ and this contradicts the requirement that W be the maximal intermediate. Pictorially we have :



Now let $x \in V_T^*$ be any string such that $\mu \rightarrow^* x$, let the production A → W be called p and let m = len (θX). Then because $\alpha = \theta X \delta$ is a viable prefix of G, we see by the argument used in case 1 that, for suitable $\sigma \in V^*$ and $y \in V_T^*$, G contains the derivation :

$S \rightarrow_R^* \theta X \gamma \sigma \rightarrow_R^* \theta X \delta y \rightarrow_R^* \theta X \mu y \rightarrow_R^* \theta X x y \rightarrow_R^* \theta A x y \xrightarrow{(p,m)}_R \theta W x y$. Thus (p,m) is the handle of the rsf θWxy. Now let the production B → W be called q. Then from the viable prefix $\beta = \theta Y \delta$ we deduce by a similar argument that (q,m) is the handle of the rsf θWxz for some $z \in V_T^*$. Suppose that $\text{len}(x) \geq k$.

Then $(m+k)\colon \theta Wxy = (m+k)\colon \theta Wxz$ and $m/\theta Wxz \in V_T^*$ and so, from the fact that G is LR(k), we deduce that $(p,m) = (q,n)$. But this is not so, because $A \neq B$ implies $p \neq q$. We again conclude that the supposition $len(x) \geq k$ is untenable and, since all cases have been considered, the theorem is proved. □

Now we can prove a series of increasingly powerful corollaries.

COROLLARY 4.18

Let G be an Λ-free LR(1) grammar and let C be a chain set for G. Then (G,C) has Property A.

PROOF. Let α and β be viable prefixes of G and let $\rho \in V^*$ be such that $\alpha \rightarrow_r^* \rho$ and $\beta \rightarrow_r^* \rho$. We need to show that $\alpha = \theta X$ and $\beta = \theta Y$ for some $\theta \in V^*$ and $X,Y \in V$. The result is trivial if $\alpha = \beta$, so suppose that $\alpha \neq \beta$ and let θ be the longest common prefix to both α and β. Certainly α, β and ρ all have the same length and so we can write them in the form :

$\alpha = \theta X\delta$, $\beta = \theta Y\delta$ and $\rho = \eta Z\mu$

where $X \neq Y$, $\theta \rightarrow_r^* \eta$, $X\delta \rightarrow_r^* Z\mu$ and $Y\delta \rightarrow_r^* Z\mu$.

Since G is LR(1), it follows from Theorem 4.17 that every $x \in V_T^*$ such that $\mu \rightarrow^* x$ satisfies $len(x) < 1$, that is $x = \Lambda$. But if G is Λ-free, this can only be so if $\mu = \gamma = \delta = \Lambda$. Thus $\alpha = \theta X$ and $\beta = \theta Y$ and the result is proved. □

Results which exclude Λ-rules are too restrictive to be useful in practice. The next result weakens this constraint a little.

COROLLARY 4.19

Let G be an LR(1) grammar in which every nonterminal A satisfies $A \Rightarrow^{+} x$ for some $x \in V_T^+$ and let C be a chain set for G. Then (G,C) has Property A.

PROOF. The proof of the previous result may be adapted straightforwardly to the present situation. □

We call nonterminals which generate only the empty terminal string 'null' nonterminals. Corollary 4.19 weakens its predecessor by replacing the constraint that G be Λ-free by the requirement that it contain no null nonterminals. While most programming language grammars are of this type, it is sometimes considered useful to introduce null nonterminals as 'hooks' upon which to hang semantic actions. The third and final corollary of this sequence indicates how grammars containing null nonterminals may be tested for Property A.

COROLLARY 4.20

Let $G = (V_N, V_T, P, S)$ be a grammar and let C be a chain set for G. Define a new grammar $G' = (V_N, V_T', P', S)$ where :

$$V_T' = V_T \cup \{ \divideontimes_A \mid A \text{ is a null nonterminal in } G \},$$

$$P' = P \cup \{ A \rightarrow \divideontimes_A \mid A \text{ is a null nonterminal in } G \}.$$

and each $\divideontimes_A$ is a new terminal symbol distinct from all others. Then (G,C) has Property A if G' is LR(1).

PROOF. By Construction, G' has no null nonterminals. Therefore by the previous result, (G',C) has Property A if G' is LR(1). Now it is clear that any subgrammar of a grammar with Property A also has that property. The result then follows because G is a subgrammar of G'. □

If G is LR(1), then it is surely a very strange grammar indeed if the grammar G' of Corollary 4.20 fails to be LR(1) (Grammar G8 is such a grammar). We conclude that, provided G is LR(1), (G,C) is very likely to possess Property A even if G contains null nonterminals.

Since one symbol lookahead (i.e. k = 1) is the only practical choice. Corollaries 4.18, 4.19 and 4.20 are sufficient to ensure that results which depend on Property A will generally be applicable in practice. The next section establishes one such result.

4.5. The Equivalence of SQCFLR(k) and CFLR(k) Parsing Tables.

In this section we shall prove that if (G,C) has Property A then its SQCFLR(k) and its CFLR(k) parsing tables are equivalent. The following lemma is the crux of the argument and is also the place where Property A is needed.

LEMMA 4.21

Let (G,C) have Property A and let θ and μ be cf-viable prefixes of (G,C) such that $CFV(\theta) = CFV(\mu)$. Let $\alpha,\beta \in V^*$ be such that $\alpha \xrightarrow[c]{*} \theta$, $\beta \xrightarrow[c]{*} \mu$ and $N(\alpha) \cap N(\beta) \neq \emptyset$. Then $V(\alpha) = V(\beta)$.

PROOF. If any of α,β, μ and θ are the empty string then all of them are, and the result is trivial in this case. So suppose $\alpha, \beta \neq \Lambda$. Since LR(k) states are uniquely determined by their nuclei, it is only necessary to prove that $N(\alpha) = N(\beta)$. We shall prove that $N(\alpha) \subseteq N(\beta)$. Symmetry will provide $N(\alpha) \supseteq N(\beta)$ and hence $N(\alpha) = N(\beta)$.

Because $\alpha \neq \Lambda$, we may write it in the form $\alpha = \alpha'X$ where $X \in V$. Then X is the associated symbol of all the LR(k) items in $N(\alpha)$ and since $N(\alpha) \cap N(\beta) \neq \emptyset$ it must also be the associated symbol of all items in $N(\beta)$. Thus β has the form $\beta = \beta'X$. Now let Δ be any LR(k) item in $N(\alpha)$. We must show that Δ is in $N(\beta)$ also. There are two cases to consider.

Case 1 : $\Delta \in$ CF-STRIP(N(α)). Because $\alpha \rightarrow_r^* \theta$, Theorem 3.43 provides $\Delta \in$ CFN(θ) in this case. Then because CFV(θ) = CFV(μ), it follows that $\Delta \in$ CFN(μ). Hence, again by Theorem 3.43, there exists $\gamma \in V^*$ such that $\gamma \rightarrow_r^* \mu$ and $\Delta \in N(\gamma)$. Note that X is the associated symbol of Δ and so γ has the form $\gamma = \gamma'X$. We also have $\gamma \rightarrow_r^* \mu$ and $\beta \rightarrow_r^* \mu$ and so, because (G,C) has Property A, if γ and β differ, it is only on their final symbols. But both have X as their final symbol and so we conclude that $\gamma = \beta$. Thus $\Delta \in N(\beta)$ as required.

Case 2 : $\Delta \notin$ CF-STRIP(N(α)). In this case, Δ must be a chain item. That is to say it is of the form Δ = [A → X., u] where A → X is a chain production. Clearly [A → .X,u] $\in V(\alpha')$ and, since this is an initial item, it must have been added to V(α') during the CLOSURE operation. That is, there must be some non-final item Σ = [B → γ.Cδ, v] $\in$ V(α') such that [A → .X,u] $\in$ CLOSURE($\{\Sigma\}$). Furthermore, Σ can be chosen to satisfy B → γCδ $\in$ P\C (i.e. Σ is not a chain item) and C $\rightarrow_r^*$ A. It follows that Σ' = [B → γC.δ, v] $\in$ CF-STRIP(V(α'C)). Now we have

$$\alpha'C \rightarrow_r^* \alpha'A \rightarrow_r \alpha'X = \alpha \rightarrow_r^* \theta$$

and so, by Theorem 3.43, we obtain $\Sigma' \in$ CFV(θ). But CFV(θ) = CFV(μ) and so $\Sigma' \in$ CFV(μ). Then, again by Theorem 3.43, it follows that $\Sigma' \in V(\sigma)$ for some $\sigma \rightarrow_r^* \mu$. Now σ must have the form $\sigma = \sigma'C$ and clearly $\Sigma' \in V(\sigma)$ implies $\Sigma \in V(\sigma')$. But Σ introduces [A → .X,u] during the CLOSURE operation and so [A → .X,u] $\in V(\sigma')$. Hence Δ = [A → X.,u] $\in N(\sigma'X)$. Now note that $\sigma'X \rightarrow_r^* \mu$ and $\beta \rightarrow_r^* \mu$. Therefore, because (G,C) has Property A, if

$\sigma'X$ and β differ, it is only on their final symbols. But both have X as their final symbol and so $\sigma'X = \beta$. Hence $\Delta \in N(\beta)$ as required to complete the proof. □

Using this lemma we can establish the next one.

LEMMA 4.22

Let (G,C) have Property A. Then the surjection Φ from the SQCFLR(k) to the true CFLR(k) stateset given in Lemma 4.13 is injective. (Hence it is a bijection).

PROOF. Let M and N be SQCFLR(k) states and let Δ be a CFLR(k) state such that $M \Phi \Delta$ and $N \Phi \Delta$. We need to prove that M = N. We will show that $M \subseteq N$. Symmetry will provide $M \supseteq N$ and hence $M = N$.

Now $M \Phi \Delta$ implies that $M = SQCFV(\theta)$ and $\Delta = CFV(\theta)$ for some $\theta \in V^*$. Similarly, $N \Phi \Delta$ implies $N = SQCFV(\mu)$ and $\Delta = CFV(\mu)$ for some $\mu \in V^*$. First we dispose of the case where θ (or, symmetrically μ) is the empty string. When $\theta = \Lambda$ we have $CFV(\mu) = \Delta = CFV(\theta) = CFV(\Lambda)$ and so $\mu = \Lambda$ also. Immediately this gives M = N and the proof is complete in this case.

We now suppose that $\theta \neq \Lambda$ and $\mu \neq \Lambda$. Remember that by Definition 4.8, M and N are sets of LR(k) parsing states for G. Let $s \in M$. The Lemma is proved if we can show that $s \in N$. Now if $s \in M$, we must have $s = NAMEOF(V(\alpha))$ where $\alpha \in V^*$ satisfies $\alpha \rightarrow_r^* \theta$ and $CF\text{-}STRIP(V(\alpha)) \neq \emptyset$. This last implies that $CF\text{-}STRIP(N(\alpha)) \neq \emptyset$ also, so let Σ be any LR(k) item in $CF\text{-}STRIP(N(\alpha))$. By virtue of Theorem 3.43, we then have $\Sigma \in CFN(\theta)$. But since $CFV(\theta) = CFV(\mu)$ we also have $CFN(\theta) = CFN(\mu)$ and so $\Sigma \in CFN(\mu)$. Then Theorem 3.43 implies that $\Sigma \in N(\rho)$ for some β such that

$\beta \rightarrow_c^* \mu$. It follows that CF-STRIP(N(β)) $\neq \emptyset$ and so, by Definition 4.8, we have NAMEOF(V(β)) $\in$ N. We also have $\alpha \rightarrow_c^* \theta, \beta \rightarrow_c^* \mu$, CFV($\theta$) = CFV($\mu$) and N($\alpha$) $\cap$ N(β) $\neq \emptyset$.

The previous lemma therefore provides V(α) = V(β). This means that NAMEOF(V(β)) = s and so s $\in$ N as required to complete the proof. □

Finally, we achieve the result we seek.

THEOREM 4.23

If (G,C) has property A then its SQCFLR(k) and its CFLR(k) parsing tables are equivalent.

PROOF. By virtue of Theorem 4.14,we know that the SQCFLR(k) tables for (G,C) cover its CFLR(k) tables under the mapping H constructed during the proof of that theorem. This mapping H is defined in terms of the mapping $\bar{\Phi}$ of Lemma 4.13. The previous lemma has established that $\bar{\Phi}$ is bijective when (G,C) has Property A and so it follows that H is also bijective in this case. The Theorem then follows from the definition of equivalent sets of parsing tables given in Definition 4.7. □

4.6. Summary.

Given the LR(k) parsing tables for a Grammar G, an object called the "quasi CFLR(k) stateset" may be constructed for the cs-grammar (G,C). There is a close correspondence between quasi and true CFLR(k) statesets: chain free parsing tables may be constructed for (G,C) using information contained in its quasi CFLR(k) stateset. Two methods for doing so were presented in this chapter. The first of these guarantees to produce the true CFLR(k) parsing tables for (G,C) but requires access to the actual LR(k) items associated with each LR(k) parsing state. This is called the "post pass" method of constructing CFLR(k) tables. The second method requires only the information contained in the LR(k) parsing tables for G and does not guarantee to deliver the true CFLR(k) tables for (G,C). Instead it produces "quasi" CFLR(k) tables which are often substantially larger than the true CFLR(k) tables, although their behaviour is the same when they are used to drive Algorithm 1.4.

A simple modification of the method leads to the generation of "strong quasi" CFLR(k) parsing tables. These are similar to ordinary quasi tables but contain fewer states : in many cases the strong quasi CFLR(k) tables are identical to the true CFLR(k) tables. A property of cs-grammars, called Property A, was introduced and shown to be a sufficient condition for guaranteeing this equivalence. Methods of testing for Property A were presented. These methods are specialised to those cs-grammars (G,C) where G is LR(1) and indicate

that only very.rarely will such grammars fail to possess the property.

The "strong quasi" constructions of this chapter provide a practical method for converting LR(k) parsers into chain free parsers. They should be useful in circumstances where an LR(k) parser generator is available but not amenable to conversion to a CFLR(k) generator. No disadvantage is likely to be incurred by adopting this approach in the important practical case k = 1.

CHAPTER 5.

OPTIMISING CFLR(k) PARSING TABLES

In Chapter 1 we claimed the LR(k) parsing algorithm to be one of the most attractive of all parsing methods because of its generality, speed and excellent error detection. We have seen that the CFLR(k) cf-parsing algorithm is even more widely applicable, and is substantially faster than the LR(k) algorithm, while affording the same high quality of error detection. Unfortunately however, just as it improves still further those features of the LR(k) parsing algorithm which are already excellent, so the CFLR(k) method exacerbates its major disadvantage - it usually makes the parsing tables even bigger.

This claim may surprise the reader who remembers that the LR(1) tables for G3 have 22 states while the CFLR(1) tables for (G3,C3) have only 19. However, had we taken the chain set $\{E \rightarrow T\}$ instead of C3 we should have found 23 states in the CFLR(1) tables. Now it can be shown that the CFLR(k) tables for (G,C) always contain fewer states than the LR(k) tables for G when G is LR(0), but in general it seems that CFLR(k) tables are usually larger than their LR(k) counterparts. Thus it may be that the speed benefits of CFLR(k) parsing are bought at the expense of excessively large parsing tables.

All is not lost, however, for in this chapter we present a simple technique for reducing the size of CFLR(k) tables which is so successful that it generally make these

tables substantially smaller than their LR(k) counterparts. This optimization technique exploits some redundancy that is always present in CFLR(k) tables and so it does not degrade the performance of the parser in any way. Thus our optimized CFLR(k) parsers have a double advantage over ordinary LR(k) parsers : they are not only much faster, but smaller too .

Although we shall introduce the optimisation technique as one to be applied to existing CFLR(k) tables, we will show later that it can also be applied during the construction of the tables and actually reduces the cost of their construction.

Before proceeding to describe the technique, we must point out that although we believe that it preserves the correctness of the CFLR(k) tables for any cs-grammar (G,C), given only that G is LR(k), we shall only prove this preservation of correctness in the case that (G,C) has Property A. Effectively, this restricts application of the technique to the case k = 1 - because our sufficient conditions for Property A (Corollaries 4.18, 4.19 and 4.20) are particular to this case. We make no apology for this restriction; our interest in reducing the size of CFLR(k) tables is motivated solely by practical necessity, and k = 1 is the only case of practical concern.

5.1. Inaccessible Entries in Parsing Tables.

We indicated earlier that our optimisation technique is based on the elimination of some redundancy from CFLR(k) parsing tables. The redundancy concerned is manifested by the presence of "inaccessible entries" within the tables. A parsing action or goto entry is said to be "inaccessible " if no input string whatsoever can cause the parsing algorithm to inspect the value of that entry. By virtue of their inaccessibility, the values possessed by such entries are irrelevant to the behaviour of the parser and so they may be changed in any way which proves convenient. By judicious manipulation of these entries it is sometimes possible to cause a group of parsing states to become so similar to one another that they may all be replaced by a single composite state. A detailed discussion of this general process, at least as it applies to ordinary LR(k) parsers, has been provided by Aho and Ullman (1972b).

The problem of exploiting inaccessible entries optimally (in the sense of reducing the number of states to a minimum) is similar to that of "minimising incompletely specified sequential machines" - a problem which Pfleeger (1973) has shown to be NP-complete in the general case. Thus optimal application of the technique may well be computationally intractable and so approximate solutions must be sought. Our technique for reducing the size of CFLR(k) tables is of this approximate type ; although very effective, it makes no claim to optimality. We speak of the technique as an optimisation in only an informal sense.

Now although our optimisation techique will turn out to be very simple, we must first complicate matters a little. It is here that we part company with Aho and Ullman (1972b) since their notion of inaccessibility is too blunt a tool for our purposes. At present the notion of inaccessibility is understood to have a global context: an entry is either inaccessible or it is not. We shall prefer a more local interpretation : certain entries may not be inaccessible in the former, global, sense, but they may be so when the states in which they occur are entered on some <u>particular</u> symbol. An example may help here.

Consider the following grammar :

1.	S	⟶	Xa |	(Grammar G9)
2.			Yb |	
3.			aXa |	
4.			aZb	
5.	X	⟶	x |	
6.			y |	
7.			z	
8.	Y	⟶	y	
9.	Z	⟶	x	

and take C9 = $\{X \rightarrow x, X \rightarrow y\}$ as the chain set. The CFLR(1) parsing tables for (G9,C9) are shown in Figure 5.1.

In Figure 5.1. the action f(8,b) is not inaccessible in the global sense because the input string ayb causes its value to be inspected. This entry <u>is</u> inaccessible, however, when state 8 is entered under a transition on the symbol X. This is because an X can only be

STATE	CF-ACTION						CF-GOTO								
NO.	↓	a	b	x	y	z	a	b	x	y	z	X	Y	Z	S
1		sh		sh	sh	sh	2		3	4	5	3	6		
2				sh	sh	sh			7	8	5	8		9	
3		sh					10								
4		sh	8				10								
5		7													
6			sh					11							
7		sh	9				12								
8		sh					12								
9			sh					13							
10	1														
11	2														
12	3														
13	4														

Figure 5.1 : The CFLR(1) Parsing Tables for (G9,C9).

produced by a reduction involving the production X → z and no such reductions are performed when b is the look-ahead string. Thus although the error entry f(8,b) cannot be changed, it is permissible to substitute state 7 for state 8 as the value of the goto entry g(2,X). This manipulation does not save any states in this particular instance, but it does cause the goto entries in the column for X to become identical to those in the column for x, thereby permitting a more economical representation of the tables in storage. This demonstrates another benefit of our optimisation technique : in general we are able to remove not only rows (i.e. states) from the parsing tables, but also columns (i.e. symbols).

We now define formally our notion of an inaccessible entry.

DEFINITION 5.1

Let $T = (Q, s_0, g, f)$ be a set of cf-parsing tables, using k symbol lookahead, for the cs-grammar (G,C). Let $p \in Q$, $X \in V$ and $u \in V_T^{*k}$. We say that the action $f(p,u)$ is inaccessible on X if, when Algorithm 1.4 is driven by the tables T, no input string whatsoever can cause it to inspect the value of $f(p,u)$ when X is the symbol on top of the parse stack. Similarly, when $p \in Q$ and $Y \in V$, we say that the goto entry $g(p,Y)$ is inaccessible if no input string whatsoever can cause the algorithm to inspect the value of $g(p,Y)$. □

Notice that this definition uses a "local" interpretation of inaccessibility for actions and a "global" one for goto's. (This distinction is only significant in the case of CFLR(k) tables. With LR(k) tables, our definition agrees with that of Aho and Ullman (1972b).)

Unlike Aho and Ullman (1972b), who use(their notion of) inaccessible entries to permit certain states to be completely replaced by others, we shall only seek to replace selected references to certain states by references to others. That is, we shall use the presence of inaccessible entries to permit us to change the values of certain goto entries (not only inaccessible ones). If we change the value of an accessible goto entry, then we must ensure that the new value is a "valid substitute" for the old.

DEFINITION 5.2

Let $T = (Q, s_0, g, f)$ be a set of cf-parsing tables for the cs-grammar (G,C) and let $p, r \in Q$ and $X \in V$. Then r is a valid substitute for $g(p,X)$ if changing the value of $g(p,X)$ to r makes no difference to the behaviour of the parser driven by these tables. □

Now we combine Definitions 5.1 and 5.2 and obtain the theorem which underlies all our subsequent developments.

THEOREM 5.3

Let $T = (Q, s_0, g, f)$ be a set of cf-parsing tables, using k symbol lookahead, for the cs-grammar (G,C). Let $p, r \in Q$ and $X \in V$ be such that either

(1) $g(p,X)$ is inaccessible, or

(2) $g(p,X) = q$ where $q \in Q$ and both

(i) for each $u \in V_T^{*k}$, either

(a) $f(q,u) = f(r,u)$, or

(b) $f(q,u)$ is inaccessible on X,

and (ii) for each $Z \in V$, either

(a) $g(q,Z) = g(r,Z)$, or

(b) $g(q,Z)$ is inaccessible.

Then r is a valid substitute for $g(p,X)$.

PROOF. The conditions satisfied by p,r and X are such that substitution of r for the original value of $g(p,X)$ is equivalent to (locally) changing the values of certain inaccessible action and goto entries. By virtue of the very definition of inaccessibility, such changes cannot alter the behaviour of the parser and the theorem follows. □

5.2. The Optimisation Technique.

Our task now is to determine the locations of (some of) the inaccessible entries within CFLR(k) parsing tables. The next two lemmas provide the results we need.

LEMMA.5.4

Let (G,C) be a CFLR(k) cs-grammar, where $k > 0$, and let $T = (Q, s_o, g, f)$ be its CFLR(k) cf-parsing tables. Let $q \in Q$ and $u \in V_T^{*k}$ be such that $f(q,u) = \text{ERROR}$. Then the action $f(q,u)$ is inaccessible on all nonterminals.

PROOF. Let $X \in V_N$ and suppose that Algorithm 1.4, while driven by the tables T, inspects the value of $f(q,u)$ when X is on top of its parse stack. Then because $f(q,u) = \text{ERROR}$, the parser's next action will be to declare ERROR and halt. Now since $X \in V_N$ lies on top of the parse stack, the previous move must have been a reduce move involving a production with X as its left part. But when $k > 0$, we know from Theorem 3.64 that all errors are declared immediately following a shift move. The conclusion follows. □

LEMMA 5.5

Let (G,C) be a CFLR(k) cs-grammar, where $k > 0$, and let $T = (Q, s_o, g, f)$ be its CFLR(k) cf-parsing tables. Let $q \in Q$ and $X \in V$ be such that $g(q,X)$ is undefined. Then $g(q,X)$ is inaccessible.

PROOF. When Algorithm 1.4 encounters an undefined goto entry it halts and declares ERROR. But when $k > 0$, we know from Theorem 3.64 (or more accurately, from the proof of Theorem 2.40 - which underlies that of Theorem 3.64) that all errors are detected during inspection of the action function. Thus undefined goto entries can never be examined. □

Next we need two lemmas which assist in the exploitation of Theorem 5.3.

LEMMA 5.6

Let (G,C) be a CFLR(k) cs-grammar and let $T = (Q, s_0, g, f)$ be its CFLR(k) cf-parsing tables. Let $X,Y \in V$ satisfy $X \xrightarrow[c]{*} Y$ and let $p,q,r \in Q$ be such that $q = g(p,X)$ and $r = g(p,Y)$. Then for each $u \in V_T^{*k}$ either :

(i) $f(q,u) = f(r,u)$ or

(ii) $f(q,u) = \text{ERROR}$.

PROOF. By Construction 3.61 we must have $p = \text{NAMEOF}(\text{CFV}(\theta))$ for some $\theta \in V^*$ and then $q = \text{NAMEOF}(\text{CFV}(\theta X))$ and $r = \text{NAMEOF}(\text{CFV}(\theta Y))$. Hence

$$f(q,u) = \text{ACTION}(\text{CFV}(\theta X), u) \text{ and}$$

$$f(r,u) = \text{ACTION}(\text{CFV}(\theta Y), u).$$

Since $X \xrightarrow[c]{*} Y$, Theorem 3.43 provides $\text{CFV}(\theta X) \leq \text{CFV}(\theta Y)$ and so any non-ERROR value of $f(q,u)$ must also be a value of $f(r,u)$. But $f(r,u)$ must be single-valued (for otherwise $\text{CFV}(\theta Y)$ is inadequate) and so if the value of $f(q,u)$ is not ERROR, then it must be the same as $f(r,u)$. □

It is in the next lemma that this development first requires Property A.

LEMMA 5.7

Let (G,C) be a cs-grammar with Property A and let $\alpha,\beta \in V^*$ be such that $\alpha \rightarrow_c^* \beta$. Then when $X \in V$ either

(i) $CFV(\alpha X) = CFV(\beta X)$, or

(ii) $CFV(\alpha X) = \emptyset$.

PROOF. Suppose $CFV(\alpha X) \neq CFV(\beta X)$ and $CFV(\alpha X) \neq \emptyset$. Because $\alpha \rightarrow_c^* \beta$ it follows from Theorem 3.43 that $CFV(\alpha X) \subseteq CFV(\beta X)$. Therefore, if these two states are different, there is some CFLR(k) item Δ in $CFV(\beta X)$ which is absent from $CFV(\alpha X)$. Theorem 3.43 then implies that $\Delta \in V(\psi)$ for some $\psi \in V^*$ such that $\psi \rightarrow_c^* \beta X$ and that $\psi \not\rightarrow_c^* \alpha X$. But because $CFV(\alpha X) \neq \emptyset$, it must contain some item Σ and so Theorem 3.43 gives $\Sigma \in V(\theta)$ for some $\theta \in V^*$ such that $\theta \rightarrow_c^* \alpha X$. Since $\alpha \rightarrow_c^* \beta$ it follows that $\theta \rightarrow_c^* \beta X$. Now we also have $\psi \rightarrow_c^* \beta X$, and because (G,C) has Property A it then follows that ψ and θ can only differ on their final symbols. They may therefore be written in the form $\psi = \mu Y$ and $\theta = \mu Z$ where $\mu \rightarrow_c^* \alpha \rightarrow_c^+ \beta$, $Y \rightarrow_c^* X$, and $Z \rightarrow_c^* X$. But then $\psi = \mu Y \rightarrow_c^* \alpha X$. This contradicts $\psi \not\rightarrow_c^* \alpha X$, and so we conclude the lemma. □

We may now combine Lemmas 5.4. to 5.7 with Theorem 5.3. and so obtain the crucial result.

THEOREM 5.8

Let $k > 0$ and let (G,C) be a CFLR(k) cs-grammar with Property A. Let $T = (Q, s_0, g, f)$ be the CFLR(k) cf-parsing tables for (G,C) and let $p \in Q$ and $X,Y \in V$. Then the value of $g(p,Y)$ is a valid substitute for $g(p,X)$ whenever $X \xrightarrow{*}_{t} Y$.

PROOF. If $g(p,X)$ is undefined then, by Lemma 5.5, it is inaccessible and the result follows from condition (1) of Theorem 5.3. So now suppose that $g(p,X)$ is defined and that $X \xrightarrow{*}_{t} Y$. Let $q = g(p,X)$. Then by Construction 3.61 we will have $p = \text{NAMEOF}(\text{CFV}(\theta))$ and $q = \text{NAMEOF}(\text{CFV}(\theta X))$ for some $\theta \in V^*$. We must have $\text{CFV}(\theta X) \neq \emptyset$ (for otherwise $g(p,X)$ would be undefined) and since Theorem 3.43 gives $\text{CFV}(\theta X) \subseteq \text{CFV}(\theta Y)$ it follows that $\text{CFV}(\theta Y) \neq \emptyset$. Because $g(p,Y) = \text{NAMEOF}(\text{CFV}(\theta Y))$ it then follows that $g(p,Y)$ is defined. Let $r = g(p,Y)$ and $u \in V_T^{*k}$ and Lemma 5.6 then provides either $f(q,u) = f(r,u)$ or $f(q,u) = \text{ERROR}$. In the former case, condition 2(i)(a) of Theorem 5.3 is satisfied. In the latter case, since $X \xrightarrow{*}_{t} Y$ must imply $X \in V_N$ (or $X = Y$, in which case the Theorem is trivial), it follows from Lemma 5.4 that $f(q,u)$ is inaccessible on X. Thus condition 2(i)(b) of Theorem 5.3. is satisfied in this case.

Now let $Z \in V$. Then $g(q,Z) = \text{NAMEOF}(\text{CFV}(\theta XZ))$ and $g(r,Z) = \text{NAMEOF}(\text{CFV}(\theta YZ))$ and so, by Lemma 5.7, either $\text{CFV}(\theta XZ) = \text{CFV}(\theta YZ)$ or $\text{CFV}(\theta XZ) = \emptyset$. In the former case we will have $g(q,Z) = g(r,Z)$ - which satisfies condition 2(ii)(a) of Theorem 5.3. In the latter case, $g(r,Z)$ will be undefined and therefore, by

Lemma 5.5, inaccessible. This will satisfy condition 2(ii)(b) of Theorem 5.3.

Thus all appropriate conditions of Theorem 5.3. are satisfied in all cases and the result is proved. □

Our method of optimisation should now be clear. We shall simply replace some or all goto entries g(p,X) by the value some g(p,Y) where $X \stackrel{\cdot\cdot}{\div} Y$. Note that since this process really only involves manipulation of inaccessible entries, the changing of any particular g(p,X) in this way cannot affect the assumptions which ensure the validity of any subsequent changes. Thus the identities of the particular substitutions made, and the order of their application, are immaterial to the preservation of the correctness of the tables. The issue that remains is to exploit this technique systematically and to maximum advantage.

The first point to note is that when $X \stackrel{\cdot\cdot}{\div} Y$ and $X \neq Y$, we have the option of either changing the value of g(p,X) to that of g(p,Y) or of leaving it alone. Clearly, we should change it and should do so in every state $p \in Q$ because in this way, and only in this way, will we cause all states which were formerly referenced only on X (that is all states q such that g(p,Z) = q implies Z= X) to become unreferenced. These states then become totally redundant and may be removed from the parsing tables.

This deletion of states is the major benefit of the technique and the source of its motivation. However, another valuable benefit is available, provided that substitutions are performed suitably. Suppose we have $X \xrightarrow[c]{*} Y$, $X \xrightarrow[c]{*} Z$ and $X \neq Y \neq Z$. Then in any state p we can change the value of g(p,X) either to that of g(p,Y) or to that of g(p,Z). For the purpose of being able to delete states, it is immaterial which of these substitutions is chosen. Indeed, we could choose one of the alternatives in certain states and the other in the remainder. But if the substitutions are performed consistently, for instance if g(p,Y) is substituted for g(p,X) in <u>every</u> state p, then (and only then) will the columns of the goto table corresponding to the symbols X and Y become identical. Only one of them need be represented explicitly and so we obtain the double benefit of deleting from the tables not only rows (i.e. states) but also columns (i.e. symbols).

Accordingly we define our optimisation technique in terms of an "optimising function" $F : V \rightarrow V$ which is used as follows : for each symbol $X \in V$ and in each state $p \in Q$, change the value of g(p,X) to that of g(p,F(X)). In order that only valid substitutions are performed by this process we must require that $X \xrightarrow[c]{*} F(X)$ for each $X \in V$. If any symbol $X \in V$ is not in the range of F, then all states accessed only on X become inaccessible and can be deleted, as also can the column of the goto table corresponding to X. Obviously the maximum benefit is obtained when the optimising function has the smallest range possible - since this will permit the deletion of the largest number of rows and columns.

We now define some terminology that allows precise specification of this type of optimising function.

DEFINITION 5.9

An optimising function for the cs-grammar (G,C) is a mapping $F : V \rightarrow V$ such that $X \rightarrow_c^* F(X)$ for each $X \in V$. For technical reasons we require that if $X \neq S$ then $F(X) \neq S$. (S is the goal symbol of G.) The alphabet which is the range of F is denoted R_F. That is : $R_F = \{ F(X) \mid X \in V \}$. We would expect (but do not require) that if $X \in R_F$, then $F(X) = X$.

A symbol $X \in V$ is called a leaf of (G,C) if $X \rightarrow_c^* Y$ implies $Y = X$ or $Y = S$. (Obviously all terminal symbols are leaves, and so is S.) Now if X if a leaf, any optimising function F must satisfy $F(X) = X$. Thus no optimising function can have a range smaller than the set of leaves. Conversely, whenever $A \in V_N$ is a non-leaf, a leaf X may always be found such that $A \rightarrow_c^+ X$, (Provided G is reduced and no symbol satisfies $A \rightarrow_c^+ A$). Consequently, an optimising function can always be found whose range is exactly the leaves of (G,C). Since this is the smallest range possible, we call such a function a full optimising function for (G,C). □

When we optimise a set of CFLR(k) tables by applying an optimising function F, we do so by replacing the value of $g(p,X)$ by that of $g(p,F(X))$ for each $p \in Q$ and $X \in V$. As mentioned earlier, this operation clearly renders inaccessible all states which were formerly referenced only by symbols outside the range of F. We now prove that all and only such states become inaccessible.

THEOREM 5.10

Let (G,C) have Property A and let $T = (Q,s_0,g,f)$ be a set of CFLR(k) tables for (G,C). Let F be an optimising function for (G,C) and define a new set of tables $T' = (Q,s_0,g',f)$ where $g'(p,X) = g(p,F(X))$ for each $p \in Q$ and $X \in V$. Then a state $p \in Q$ is accessible from s_0 in T' if and only if $p = g(q,X)$ for some $q \in Q$ and $X \in R_F$.

PROOF. First extend the domains of the goto functions g and g' from $Q \times V$ to $Q \times V^*$ in the usual way (i.e. define $g(p,\Lambda) = p$ and $g(p,\alpha X) = g(g(p,\alpha),X)$)) and extend the domain of F from V to V^* by the definitions $F(\Lambda) = \Lambda$ and $F(\alpha X) = F(\alpha)F(X)$. Then by (implied) definition, p is accessible from s_0 in T' if and only if $p = g'(s_0,\theta)$ for some $\theta \in V^*$. Now note that the construction of T' ensures that, for any $\theta \in V^*$, we have $g'(s_0,\theta) = g'(s_0, F(\theta)) = g(s_0,F(\theta))$. The "only if " direction of the present theorem follows straightforwardly from these observations and so we now consider the "if" direction. Suppose that $p = g(q,X)$ where $q \in Q$ and $X \in R_F$. We must have $q = \text{NAMEOF}(\text{CFV}(\theta))$ for some $\theta \in V^*$ and so $p = \text{NAMEOF}(\text{CFV}(\theta X))$. Let $\mu = F(\theta)$. By the definition of an optimising function we must have $\theta \xrightarrow{*}_{L} \mu$ and so by

virtue of Lemma 5.7, we have either

$$CFV(\theta X) = CFV(\mu X), \text{ or}$$

$$CFV(\theta X) = \emptyset.$$

Now because NAMEOF(CFV(θX)) is a state in Q (it is the state p) we cannot have $CFV(\theta X) = \emptyset$ and so we must have $CFV(\theta X) = CFV(\mu X)$. This implies that p = NAMEOF(CFV(μX)), which in turn implies that $p = g(s_o, \mu X)$. Now $\mu = F(\theta)$ and $X \in R_F$. Therefore $\mu X = F(\theta Z)$ for some $Z \in V$. Hence $p = g(s_o, F(\theta Z)) = g'(s_o, \theta Z)$. Thus p is accessible in T' and the theorem is proved. □

Supported by this result we may now safely define the construction of optimised CFLR(k) tables in the manner which was informally indicated earlier.

CONSTRUCTION 5.11

Let (G,C) be a CFLR(k) cs-grammar with Property A and let F be an optimising function for (G,C). Let the CFLR(k) cf-parsing tables for (G,C) be $T = (Q, s_0, g, f)$. Then the optimised CFLR(k) tables for (G,C) with respect to F are given by $OCFT_k^{(G,C),F} = (Q', s_0, g', f')$ where

(i) $Q' = \{s_0\} \cup \{g(p,X) \in Q \mid p \in Q,\ X \in R_F\}$,
(ii) g' is the restriction of g to domain $Q' \times R_F$, and
(iii) f' is the restriction of f to domain $Q' \times V_T^{*k}$.

Note that the definition of these tables depends not on F directly, but only on its range. Consequently, the tables corresponding to a full optimising function are independent of the particular function employed : they are, in short, unique. We call them the fully optimised CFLR(k) tables for (G,C) and denote them by $FOCFT_k^{(G,C)}$. For brevity we usually write OCFLR(k) instead of "optimised CFLR(k)" and FOCFLR(k) instead of "fully optimised CFLR(k)". □

Since a full optimising function can be found mechanically for any CFLR(k) cs-grammar, there seems little point in constructing any less than fully optimised CFLR(k) tables. In the present situation this is indeed the case. However, in the next chapter we shall encounter circumstances in which it may be necessary to consider optimising tables to less than the full extent.

In order to use OCFLR(k) tables, Algorithm 1.4 needs to be modified slightly so that it takes account of the absence of all goto columns corresponding to symbols outside the range of the optimising function concerned. The only references made to the goto table by Algorithm 1.4 occur in its steps 3 (a) (iii) and 3 (b) (vi). These are references to g(s,a) and g(r,A) respectively, where a $\in V_T$ and A is the "left part of production q". These steps must be changed so that they reference g(s,F(a)) and g(r,F(A)) instead and it seems that the need to apply the optimising function during these steps might slow them down. Fortunately, this is not so. All optimising functions are identities when applied to arguments in V_T - so step 3(a) (iii) need not be altered at all. The effect of altering step 3 (b) (vi) can be accomplished more neatly and rapidly by replacing the identity of the symbol A directly. That is, modify step 3 (b)(iii) of Algorithm 1.4 so that it reads "set A equal to the <u>image under F</u> of the left part of production q". (Note that this cannot disturb the test " A = S" performed in step 3 (b)(iv) because of the requirement that optimising functions satisfy F(A) = S only if A= S.) In practice, Algorithm 1.4 will determine left parts by a table look-up and so the effect of the required change can be achieved by simply replacing the table of left parts by its image under F. In this way the speed of the parser will be unimpaired.

To illustrate the construction we show the FOCFLR(1) tables for (G3,C3) in Figure 5.2. There is only one full optimising function for this grammar : it takes E,T and P to X and is an identity elsewhere. Thus the modified table

of left parts which is required by Algorithm 1.4 will record S as the left part of production 1 and X as the left part of all others. Observe that these FOCFLR(1) tables contain only 16 states, as opposed to 19 in the unoptimised CFLR(1) tables and 22 in the ordinary LR(1) tables. (See Figures 3.8 and 2.4 respectively).

STATE No.	CF-ACTION FUNCTION						CF-GOTO FUNCTION					
	∧	(	X	)	*	+	S	(	X	)	*	+
1		sh	sh					3	2			
2	1				sh	sh					5	4
3		sh	sh					7	6			
4		sh	sh					3	8			
5		sh	sh					3	9			
6				sh	sh	sh				10	12	11
7		sh	sh					7	13			
8	2				sh	2					5	
9	4				4	4						
10	6				6	6						
11		sh	sh					7	14			
12		sh	sh					7	15			
13				sh	sh	sh				16	12	11
14				2	sh	2					12	
15				4	4	4						
16				6	6	6						

Figure 5.2. $FOCFT_1^{(G3,C3)}$ - The FOCFLR(1) cf-Parsing Tables for (G3,C3).

5.3. Constructing Optimised CFLR(k) Parsing Tables Directly.

Building OCFLR(k) tables using Construction 5.11 involves first constructing the ordinary CFLR(k) tables and then modifying them. This is wasteful and unattractive and so we now consider methods for constructing optimised tables directly. We concentrate first on modifying the standard method for building CFLR(k) tables given in Construction 3.61.

The states in a set of optimised CFLR(k) tables are just those states from the ordinary CFLR(k) tables which are accessible on symbols in the range of the optimising function F. As noted in the proof of Theorem 5.10, there are the states q such that $q = g(s_o,\theta)$ for some $\theta \in R_F^*$. In Construction 3.61 the parsing states correspond to the names of CFLR(k) states and the goto function is based on the function CF-GOTO. It is therefore easy to see that $q = g(s_o,\theta)$ if and only if $q = NAMEOF(CFV(\theta))$. Consequently, the parsing states that remain in the OCFLR(k) tables are the names of the members of the set $\{ CFV(\theta) \neq \emptyset \mid \theta \in R_F^* \}$. We call this set the "optimised CFLR(k) stateset for (G,C) with respect to F". It is easily formed by modifying the usual CFLR(k) stateset construction algorithm (Algorithm 3.47) so that a state $\Sigma = CF\text{-}GOTO(\Delta,X)$ is added to the stateset only if $X \in R_F$. The OCFLR(k) tables may then be formed from this optimised stateset in just the same way as ordinary CFLR(k) tables are formed from ordinary statesets. Thus we obtain the following algorithm.

ALGORITHM 5.12

Direct construction of optimised CFLR(k) cf-parsing tables.

Input : The CFLR(k) cs-grammar (G,C), which must have Property A, and an optimising function F.

Output: $OCFT_k^{(G,C),F}$ - the optimised CFLR(k) tables for (G,C) with respect to F.

Method:

1. First construct the optimised CFLR(k) stateset for (G,C) with respect to F by using Algorithm 3.47, modified so that the loop "for X ∈ V do" becomes "for X ∈ R_F do".

2. Then build the optimised tables by applying Construction 3.61 to the optimised, rather than to the ordinary, CFLR(k) stateset for (G,C) and taking the domain of the second argument of the goto function as R_F rather than V. □

Similarly straightforward modifications may be applied to the constructions of Chapter 4. By replacing the loop "for X ∈ V do" in Algorithm 4.3 by one which reads "for X ∈ R_F do", that algorithm may be caused to construct an "optimised quasi CFLR(k) stateset" directly from a set of LR(k) parsing tables. This stateset may then be used to construct the OCFLR(k) parsing tables (assuming that the function ITEMS is available) by using the technique indicated in the proof of Theorem 4.4. Alternatively, it may be used as the basis for constructing the "optimised quasi CFLR(k) tables for (G,C)". These are formed by adapting Construction 4.6 in just the same way as Construction 3.61

is adapted in Step 2 of Algorithm 5.12. These optimised quasi tables will perform the same as the true OCFLR(k) tables but may be rather larger (that is to say, they will cover the true OCFLR(k) tables).

More interesting are the corresponding "strong" constructions based on the techniques of Section 4.3. In order to form optimised CFLR(k) tables with confidence in their correctness, the grammar (G,C) must have Property A - and this is precisely the property which ensures that SQCFLR(k) tables are equivalent to true CFLR(k) tables. Consequently the following algorithm converts LR(k) parsing tables directly into OCFLR(k) tables.

ALGORITHM 5.13

Direct conversion of LR(k) tables into optimised CFLR(k) tables.

Input : The CFLR(k) cs-grammar (G,C), which must have Property A, an optimising function F for (G,C), and the LR(k) parsing tables for G.

Output : $OCFT_k^{(G,C),F}$ - the optimised CFLR(k) tables for (G,C) with respect to F.

Method :

1. First construct the optimised strong quasi CFLR(k) stateset for (G,C) with respect to F by using Algorithm 4.10, modified so that the loop "for X ∈ V do" becomes "for X ∈ R_F do".

2. Then build the optimised tables by applying Construction 4.11 to the optimised, rather than to the ordinary, strong quasi CFLR(k) stateset for (G,C) and taking the domain of the second agument of the goto function as R_F rather than V. □

Note that in all these cases, causing an algorithm to produce optimised, rather than ordinary, statesets or tables actually reduces the amount of work performed by the algorithm.

5.4. The Value of Optimising CFLR(k) Tables.

The benfits of (full) optimisation are considerable: in practice it seems that fully optimised CFLR(1) tables always contain fewer states than ordinary LR(1) tables. However, this reduction in the number of states does not always occur, as the following grammar shows.

S ⟶	aA |	aB |	aC |	
	bA |	bB |	bF |	(Grammar G10)
	cA |	cE |	cC |	
	dA |	dE |	dF |	
	eD |	eB |	cC |	
	fD |	fB |	fF |	
	gD |	gE |	gC |	
	hD |	hE |	hF	

A ⟶ Xa
D ⟶ Xa
B ⟶ Yb
E ⟶ Yb
C ⟶ Zc
F ⟶ Zc
X ⟶ Y | a
Y ⟶ Z | b
Z ⟶ c

Grammar G10 is LR(1) and has 48 states in its LR(1) parsing tables. The cs-grammar (G10, $\{X \rightarrow Y, Y \rightarrow Z\}$) has Property A (since G10 is LR(1) and Λ-free) and is CFLR(1). Its FOCFLR(1) tables contain 50 states - two more than the LR(1) tables (the unoptimised CFLR(1) tables contain 56 states). Note that even here, since the optimised tables contain two less columns (those corresponding to X and Y) than the ordinary tables, the space required to represent the FOCFLR(1) tables is probably less than that required for the LR(1) tables in spite of the two extra states.

Grammar G10 is complex and highly contrived, and yet it is the smallest and simplest grammar we have been able to find where the FOCFLR(1) tables contain more states than the LR(1) tables. We suspect that all grammars which exhibit this behaviour will contain subgrammars similar to G10. For this reason we are convinced that, for the grammars likely to be encountered in practice, FOCFLR(1) tables will always be smaller than their LR(1) counterparts. Unfortunately, we have been unable to find an attractive characterization of the grammars for which this behaviour can be guaranteed.

5.5. Summary.

Redundancy present in CFLR(k) parsers may be exploited in order to reduce the size of the parsing tables. The optimisation technique is very straightforward : it simply removes those columns of the goto table which correspond to (some or all of) the symbols that appear as the left parts of chain productions, and then deletes any states which thereby become unreferenced. Furthermore, the algorithms for constructing CFLR(k) parsing tables are easily modified to produce optimized tables directly.

Algorithm 1.4 must be changed slightly in order to use these optimised tables, but these changes in no way impair the speed or error detection of the parser.

The correctness of these techniques is only guaranteed for those cs-grammars with Property A and this effectively constrains their application to the case k = 1. The space savings conferred by the technique are considerable : optimised CFLR(1) tables for programming language grammars are actually smaller than ordinary LR(1) tables.

CHAPTER 6.

APPROXIMATE CFLR(1) PARSING TABLES

The theory developed so far shows that the CFLR(k) chain free parsers surpass the ordinary LR(k) parsers in generality and speed and yet often occupy less space. Unfortunately, however, it is impractical to use CFLR(k) parsers in compilers for programming languages because their parsing tables are intolerably large when useful values of k (namely k = 1) are chosen. The same is true of ordinary LR(k) parsers but certain techniques have been developed which successfully overcome the problem in this case. Our goal now is to extend the application of these techniques to the CFLR(k) parsers and thereby develop truly practical chain free parsing algorithms.

In order to reduce the size of LR(k) parsing tables some of the benefits of the method have to be relinquished - notably generality and the immediacy of error detection. Several techniques for doing so have been proposed by various authors, for example Korenjak (1969), Pager (1970), DeRemer (1969,1971) and Anderson (1972). The most important of these methods, both theoretically and in practice, are the SLR and LALR methods developed independently by DeRemer and Anderson. The acronyms stand for Simple LR and Look Ahead LR respectively. Both of these methods use one symbol lookahead and are applicable to a large subset of the LR(1) grammars and yet yield only LR(0) size parsing tables. For a grammar similar in size to that of Algol this means tables with about 400 states as opposed to over 4000 in the LR(1) tables.

Aho and Ullman (1972b) have shown that the SLR and LALR techniques, and also some others, can be described in terms of two manipulations that may be performed on LR(1) parsing tables. These are the "postponement of error detection" and the "merging of compatible states". The first of these exploits the fact that the quality of error detection afforded by an LR(1) parser is so excellent that it may be degraded slightly and yet remain acceptably good. The technique is to replace certain ERROR actions by REDUCE actions. When a true LR(1) parser would halt and declare "error", a parser whose tables have been modified in this way may continue to make reduce moves but matters are so arranged that it too will halt and declare "error" before consuming another input symbol.

The benefit conferred by the postponement of error detection is that the actions in certain states of the parser may be caused to become sufficiently similar that whole groups of states may be replaced by a single composite state. This phase of the process is known as "merging compatible states".

We shall be concerned to extend these ideas to the context of CFLR(1) parsing tables. First we need to consider the postponement of error detection in a little more detail.

Aho and Ullman (1972b) define a "postponement" to be a triple : $(q,u,A \rightarrow \alpha)$ where q is a parsing state, u is a lookahead string, and $A \rightarrow \alpha$ is a production. This postponement is interpreted to mean that the action

f(q,u) should be changed to REDUCE $A \rightarrow \alpha$. Apart from certain constraints which are necessary to preserve the correctness of the modified tables, postponements of this type may be chosen freely. For our purposes this notion is rather too general : we shall only use a postponement $(q,u, A \rightarrow \alpha)$ when REDUCE $A \rightarrow \alpha$ is already present as the action on some other lookahead string in the state q. We call postponements of this type "weak" postponements and formally define them as follows.

DEFINITION 6.1

Let $T = (Q, s_0, g, f)$ be a set of cf-parsing tables for (G,C). (Note that throughout this chapter we assume that all parsing tables use one symbol lookahead). A weak postponement for T is a triple $(q,u,A \rightarrow \alpha)$ where $q \in Q$, $u \in V_T^{*1}$ and $A \rightarrow \alpha \in P$ are such that $f(q,v) =$ REDUCE $A \rightarrow \alpha$ for some $v \in V_T^{*1}$. Since we shall be concerned solely with weak postponements, we often refer to them as simply "postponements". A postponement $(q,u,A \rightarrow \alpha)$ is sometimes called a "postponement with A". The application of the postponement $(q,u, A \rightarrow \alpha)$ to T causes the action f(q,u) to become REDUCE $A \rightarrow \alpha$. A set of postponements for T is called a postponement set for T. □

We must ensure that the correctness of parsing tables is preserved under application of postponement sets. In particular we must ensure that only ERROR actions are changed and that all errors will eventually be detected. Accordingly we define "valid" postponement sets as follows.

DEFINITION 6.2

Let $T = (Q, s_0, g, f)$ be a set of chain free parsing tables for (G,C). Extend the domain of the goto function g from $Q \times V$ to $Q \times V^*$ by means of the definitions :

$$g(p, \Lambda) = p \quad \text{and}$$

$$g(p, \alpha X) = g(g(p,\alpha), X)$$

where $p \in Q$, $\alpha \in V^*$ and $X \in V$.

A postponement set R is valid for the tables T if all postponements $(q,u,A \to \alpha)$ in R satisfy

(i) $A \neq S$,

(ii) $f(q,u) = \text{ERROR}$ and

(iii) whenever $p \in Q$ is such that $g(p,\alpha) = q$

then (a) $g(p,A)$ is defined and

(b) $f(g(p,A),u) = \text{ERROR}$. □

Clearly, an algorithm may be constructed to test whether a given postponement set is valid for a particular set of tables. Matters are simplified when the tables are proper CFLR(1) tables.

THEOREM 6.3

Let T be the CFLR(1) tables for (G,C). Then any postponement set for T which satisfies conditions (i) and (ii) of Definition 6.2 also satisfies condition (iii) of that definition.

PROOF. This result is an elementary consequence of the definition of CFLR(k) tables (Construction 3.61) and the properties of CFLR(k) states. Note that the result is also true of any tables which cover the true CFLR(1) tables (such as SQCFLR(1) tables). □

The application of a valid postponement set preserves the correctness of the parsing tables.

THEOREM 6.4

Let T be a set of cf-parsing tables for (G,C) and let T' be the set of tables produced by application of a valid postponement set to T. Then

(i) all sentences in L(G) are cf-parsed correctly when Algorithm 1.4 is driven by T',

(ii) all erroneous strings $x \notin L(G)$ are rejected when Algorithm 1.4 is driven by T' and the number of symbols consumed prior to rejection will be the same as when Algorithm 1.4 is driven by T,

(iii) if T drives Algorithm 1.4. so that all errors are detected by encountering an ERROR action, then T'also has this property.

PROOF. Part (i) is immediate from the fact that valid postponement sets only alter ERROR actions - and these cannot be encountered while parsing valid sentences. Part (ii) follows from the proof of a similar result provided by Aho and Ullman (1972b, Theorem 4). Note that our notion of a valid postponement set is a special case of theirs, and of course they are concerned only with ordinary, not chain free, parsers. However, these points do not affect the argument used in the proof. Part (iii) is an elementary consequence of the constraints placed upon valid weak postponements. □

Figure 6.1. illustrates the valid weak postponement of error detection in the CFLR(1) tables for (G3,C3) which appeared in Figure 3.8. We do not indicate the postponement set used explicitly, but do so implicity by putting a circle around the REDUCE actions introduced by the postponement set.

STATE	CF-ACTION FUNCTION						CF-GOTO FUNCTION								
NO.	Λ	(	X	)	*	+	S	E	T	P	(	X	)	*	+
1		sh	sh					2	3	3	4	3			
2	1					sh									5
3	1				sh	sh								6	5
4		sh	sh					7	8	8	9	8			
5		sh	sh						10	10	4	10			
6		sh	sh							11	4	11			
7				sh		sh							12		13
8				sh	sh	sh							12	14	13
9		sh	sh					15	16	16	9	16			
10	2			②	sh	2								6	
11	4			④	4	4									
12	6			⑥	6	6									
13		sh	sh						17	17	9	17			
14		sh	sh							18	9	18			
15				sh		sh							19		13
16				sh	sh	sh							19	14	13
17	②			2	sh	2								14	
18	④			4	4	4									
19	⑥			6	6	6									

Figure 6.1 : The CFLR(1) Tables for (G3,C3) after Application of a valid Postponement Set.

The reader may care to check that the tables of both Figures 3.8 and 6.1 reject the invalid string "X*X)" after consuming the three symbols "X*X" but that the tables of Figure 6.1 make a reduce move after this point before rejecting the input.

Next we define the merging of compatible states. Our definition is a restriction of the one used by Aho and Ullman (1972b).

DEFINITION 6.5

Let $T = (Q, s_o, g, f)$ be a set of cf-parsing tables for (G,C). A partition π on Q is said to be compatible if, whenever p and q are in the same block of π, then

(i) for each $u \in V_T^{*1}$, $f(p,u) = f(q,u)$, and

(ii) for each $X \in V$, either $g(p,X)$ and $g(q,X)$ are both undefined or both are in the same block of π.

When π is a compatible partition on Q, the tables formed by applying π to T are denoted by $T_\pi = (Q', s_o', g', f')$ and defined as follows. Let $[q]$ denote the block of π to which the state $q \in Q$ belongs. Then

(i) $Q' = \{[q] \mid q \in Q\}$,

(ii) $s' = [s_o]$,

(iii) for each $q \in Q$ and $X \in V$

$$g'([q],X) = \begin{cases} \varphi \text{ if } g(q,X) = \varphi \text{ and} \\ [g(q,X)] \text{ otherwise,} \end{cases}$$

(iv) for each $q \in Q$ and $u \in V_T^{*1}$,

$f'([q], u) = f(q,u)$.

Tables formed by applying first a weak postponement set and then a compatible partition to T are called approximations to T. If the postponement set concerned is valid, then the approximation is said to be valid also. Tables formed in this way from the CFLR(1) tables for (G,C) are called approximate CFLR(1) tables for (G,C). □

Figure 6.2. illustrate the valid approximate CFLR(1) tables for (G3,C3) that are produced by applying the compatible partition {1,2,3,(4,9),(5,13),(6,14),(7,15),(8,16),(10,17),(11,18),(12,19)} to the tables of Figure 6.1. Observe the substantial reduction in the number of states obtained by this process.

STATE NO.	CF-ACTION FUNCTION						CF-GOTO FUNCTION								
	$\wedge$	(	X	)	*	+	S	E	T	P	(	X	)	*	+
1		sh	sh					2	3	3	4	3			
2	1					sh									5
3	1				sh	sh								6	5
4		sh	sh					7	8	8	4	8			
5		sh	sh						9	9	4	9			
6		sh	sh							10	4	10			
7				sh		sh							11		5
8				sh	sh	sh							11	6	5
9	2			2	sh	2								6	
10	4			4	4	4									
11	6			6	6	6									

Figure 6.2: A Set of Valid Approximate CFLR(1) Tables for (G3,C3).

Applying a compatible partition to a set of tables merely reduces their size, it does not affect their performance in any way. Consequently, it follows from Theorem 6.4. that valid approximate CFLR(1) tables are perfectly good cf-parsing tables although their error detection is slightly inferior to that of the CFLR(1) tables. We know from Theorem 3.64 that the CFLR(1) parser for (G,C) will reject an invalid input string x on the move immediately following the EP(x)-1'st shift move. A valid approximate

CFLR(1) parser will certainly make no more shift moves after this point, but may make some reduce moves before it rejects the input. This quality of error detection is superior to that of all other practical bottom up parsing schemes and is sufficient to allow the automatic generation of good error diagnostics and effective automatic error recovery (see Wynn (1973)). Furthermore, the technique of Eve (1973) allows full CFLR(1) quality error detection to be regained from approximate CFLR(1) tables at the cost of a modest decrease in parsing speed.

An attractive feature of true CFLR(1) parsers is that they detect all errors during inspection of the action function and so it is unnecessary to check for error conditions during steps 3(a) and 3(b) of Algorithm 1.4. This is beneficial to the speed of the parser and is essential to the optimisation technique of Chapter 5. Part (iii) of Theorem 6.4. ensures that valid approximate CFLR(1) tables preserve this property.

Two basically different classes of methods for producing approximate CFLR(1) tables may be identified. Methods in the first class take care to apply only valid postponement sets and then seek a compatible partition containing as few blocks as possible. These methods guarantee to produce valid approximate CFLR(1) tables but are complex and usually require access to the true CFLR(1) tables. Since these tables may be very large, methods in this class are generally considered impractical.

Methods in the second class (and for the ordinary LR(1) case these include the SLR and LALR methods) work rather differently. In effect, these methods seek to apply a particular partition and so they generate a postponement set which, if valid, will render that partition compatible. Clearly, the disadvantage of these methods is that they may generate invalid postponement sets. The utility of each such method depends upon the extent of the class of grammars for which it does generate valid postponement sets. The great advantage of these methods is that they are able to produce their approximate CFLR(1) tables directly, thereby removing the necessity to construct the CFLR(1) tables first.

We shall be solely concerned with methods in the second class. They may be characterized in terms of an "approximation function". The following definition defines this concept and shows how approximate CFLR(1) tables may be constructed directly.

DEFINITION 6.6

Let Δ be a set of CFLR(1) items for (G,C) and let $\theta \in V^*$. Then Δ is an approximate CFLR(1) state for θ if

(i) $\Delta \supseteq CFV_1(\theta)$, and

(ii) whenever $[B \rightarrow \beta_1 . \beta_2 , v] \in \Delta$ then

(a) if $B = S$, then $[B \rightarrow \beta_1 . \beta_2 , v] \in CFV_1(\theta)$

(b) if $B \neq S$, then $[B \rightarrow \beta_1 . \beta_2 , u] \in CFV_1(\theta)$ for some $u \in V_T^{*1}$.

We say that a function $\bar{\Phi}$ from the CFLR(1) states for (G,C) to sets of CFLR(1) items for (G,C) is an approximation function for (G,C) if $\bar{\Phi}(CFV_1(\theta))$ is an approximate CFLR(1) state for θ, for each $\theta \in V^*$. When $\bar{\Phi}$ is such an approximation function, we define the $\bar{\Phi}$ - approximate CFLR(1) stateset for (G,C) to be

$$CFS_{\bar{\Phi}}^{(G,C)} = \{\bar{\Phi}(CFV_1(\theta)) \neq \emptyset \mid \theta \in V^*\}$$

and we define the $\bar{\Phi}$ - approximate CF-GOTO function by :

$$CF\text{-}GOTO_{\bar{\Phi}}^{(G,C)} (\bar{\Phi}(CFV_1(\theta)), X) = \bar{\Phi}(CFV_1(\theta X)).$$

The $\bar{\Phi}$ - approximate CFLR(1) tables for (G,C) are given by simply substituting the $\bar{\Phi}$ - approximate stateset and CF-GOTO function for the normal stateset and CF-GOTO function in Construction 3.61. □

It is necessary to be able to check whether $\bar{\Phi}$ - approximate CFLR(1) tables are valid approximations or not. The next theorem provides the necessary result.

THEOREM 6.7

If the $\bar{\Phi}$ - approximate CFLR(1) stateset for (G,C) is adequate then

(i) (G,C) is CFLR(1) and

(ii) the $\bar{\Phi}$ - approximate CFLR(1) tables for (G,C) are valid.

PROOF. If (G,C) is not CFLR(1) then its CFLR(1) stateset will be inadequate and therefore, by virtue of part (i) of Definition 6.6, so will its $\bar{\Phi}$ - approximate CFLR(1) stateset.

Now suppose that (G,C) is CFLR(1). To each CFLR(1) parsing state NAMEOF(CFV(θ)) there corresponds a $\bar{\Phi}$ - approximate parsing state NAMEOF($\bar{\Phi}$(CFV(θ))). Now $\bar{\Phi}$(CFV(θ)) is formed by adding items to CFV(θ) and it is clear that the constraints on the items which may be so added ensure that if

ACTION(CFV(θ),u) ≠ ACTION($\bar{\Phi}$(CFV(θ)),u)

then ACTION($\bar{\Phi}$(CFV(θ)),u) = REDUCE q where, for some $v \in V_T^{*1}$, ACTION(CFV(θ),v) = REDUCE q also.

Thus the actions in the parsing state corresponding to $\bar{\Phi}$(CFV(θ)) are equivalent to those that may be obtained by applying a weak postponement set to the parsing state corresponding to CFV(θ). By virtue of Theorem 6.3 we need only to verify that conditions (i) and (ii) of Definition 6.2 are satisfied by this postponement. Condition (i) is clearly satisfied because of the

constraint (iia) of Definition 6.6. If condition (ii) is not satisfied, then there must be some item $[B \rightarrow \beta_1 . \beta_2 , v] \in CFV(\theta)$ with $u \in EFF_1(\beta_2 v)$ to which another, distinct item $[A \rightarrow \alpha . , u]$ is added when forming $\bar{\Phi}(CFV(\theta))$. Clearly this causes $\bar{\Phi}(CFV(\theta))$ to become inadequate and so we conclude that if the $\bar{\Phi}$ - approximate stateset is adequate, then the $\bar{\Phi}$ - approximate CFLR(1) tables are valid. □

Many different types of approximate CFLR(1) parsing tables may be defined by choosing suitable approximation functions. We shall concentrate initially on the chain free generalisation of the SLR method.

6.1. The CFSLR Method.

In this section we introduce a chain free generalization of the SLR method. We call this the "CFSLR" method and characterise it by an approximation function Φ_{SLR} as follows.

DEFINITION 6.8

Let $G = (V_N, V_T, P, S)$ be a grammar and $X \in V$. The followset of X in G is given by

$$\text{FOLLOW}^G(X) = \{1{:}x \mid S \Rightarrow^* \alpha X x,\ x \in V_T^*\}.$$

Clearly $\text{FOLLOW}(X) \subseteq V_T^{*1}$. Algorithmsmay easily be constructed for the evaluation of followsets. (See, for example, Anderson et al (1973).)

The CFSLR approximation method is defined by the approximation function Φ_{SLR} where, for each $\theta \in V^*$, we define

$$\Phi_{SLR}(CFV_1(\theta)) = \{ [B \to \beta_1 . \beta_2, v] \mid [B \to \beta_1 . \beta_2, u] \in CFV_1(\theta) \text{ and } v \in \text{FOLLOW}(B) \}$$

We write CFSLR rather than "Φ_{SLR} - approximate CFLR(1)"; CFSLR states are written as CFSLRV(θ) rather than $\Phi_{SLR}(CFV_1(\theta))$. We say that (G,C) is CFSLR if its CFSLR stateset is adequate; G is said to be CFSLR if there is a chain set C for G such that (G,C) is CFSLR. A language is CFSLR if it is generated by some CFSLR grammar. □

The CFSLR approximation function Φ_{SLR} certainly satisfies conditions (i) and (iib) of Definition 6.6. However, it may not always satisfy condition (iia) if the goal symbol appears in the right part of a production.(Consider the grammar $S \rightarrow aSa \mid b$.) This in turn may cause condition (i) of Definition 6.2 to be violated. The effect of this will be to force the reintroduction of error checking in part (vii) of step 3 (b) of Algorithm 1.4. (Consider the invalid string ba with respect to the grammar above). This is undesirable and may be avoided by augmenting the grammar with a production $S' \rightarrow S \perp$, where S' is a new goal symbol and $\perp$ is an (optional) endmarker.

When the chain set C is empty, the CFSLR method becomes the ordinary SLR method. We say that a grammar G is SLR if $(G,\emptyset)$ is CFSLR and that a language is SLR if it is generated by an SLR grammar. When the chain set is empty, we talk of SLR states, rather than CFSLR states, and write SLRV(θ) instead of CFSLRV(θ).

The CFSLR method generalizes the SLR method just as CFLR(k) generalises LR(k). We now ask similar questions of the CFSLR method to those considered for the CFLR(k) case in Chapters 3, 4 and 5. How extensive are the classes of CFSLR grammars and languages compared to those of SLR ? How can we test efficiently for the CFSLR property and how can we build CFSLR tables ? Can we build CFSLR tables from SLR tables by a post-pass construction, and can SLR tables be converted directly into CFSLR tables? Can CFSLR tables be optimised in the

sense of Chapter 5? We shall see that with respect to these questions the CFSLR method exhibits broadly similar properties to the CFLR(k) method. Not all approximations are so well behaved (as will be seen later when we consider the chain free generalization of the LALR method).

The first question we ask is whether the condition that G be SLR is sufficient to ensure that (G,C) is CFSLR for all choices of chain set C. Unlike the corresponding CFLR(k) result (Theorem 3.19) where the answer is always "yes", here we have only a qualified "yes" : We need an additional constraint upon the way in which Λ -rules may be used in G. Before introducing this result we need to establish some technical lemmas and theorems. The first of these expresses CFSLR states in terms of CFLR(0) states. (Definition 6.8 expresses them in terms of CFLR(1) states.)

LEMMA 6.9

Let $\theta \in V^*$. Then

$$\text{CFSLRV}(\theta) = \{[B \to \beta_1 . \beta_2, u] \mid [B \to \beta_1 . \beta_2, \Lambda] \in \text{CFV}_0(\theta) \text{ and } u \in \text{FOLLOW}(B)\}.$$

PROOF. The proof is trivial and we omit it. □

Next we prove the CFSLR analogue of Theorem 3.43 - a result which found constant application throughout Chapters 3,4 and 5.

THEOREM 6.10 (cf. Part (i) of Theorem 3.43)

Let $\theta \in V^*$. Then

$$\text{CFSLRV}(\theta) = \text{CF-STRIP}(\{\text{SLRV}(\mu) \mid \mu \rightarrow_c^* \theta\}).$$

PROOF. Suppose $[B \rightarrow \beta_1 . \beta_2, u] \in \text{SLRV}(\mu)$, $B \rightarrow \beta_1\beta_2 \in P \setminus C$ and $\mu \rightarrow_c^* \theta$. Then Lemma 6.9 provides $u \in \text{FOLLOW}(B)$ and $[B \rightarrow \beta_1 . \beta_2, \Lambda] \in V_0(\mu)$. Hence, by Theorem 3.43, $[B \rightarrow \beta_1 . \beta_2, \Lambda] \in \text{CFV}_0(\theta)$ and so, again by Lemma 6.9, $[B \rightarrow \beta_1 . \beta_2, u] \in \text{CFSLRV}(\theta)$. Thus

$$\text{CFSLRV}(\theta) \supseteq \text{CF-STRIP}(\{\text{SLRV}(\mu) \mid \mu \rightarrow_c^* \theta\}).$$

Containment in the other direction may be established equally straightforwardly and so we conclude the theorem. □

We now define a property of grammars which ensures that followsets are "well behaved" in the presence of Λ-rules. Grammars encountered in practice always seem to have this property.

DEFINITION 6.11

A grammar is <u>empty rule followset consistent</u> (<u>ERFC</u> for short) if, whenever $X \in V$ and $A \in V_N$ are such that $S \rightarrow^* \alpha X A \beta$ and $A \rightarrow^* \Lambda$, then $\text{FOLLOW}(X) \supseteq \text{FOLLOW}(A)$. □

The ERFC property is needed to establish the next result.

LEMMA 6.12

If G is an ERFC grammar and $\theta \in V^*$, $X \in V$ and $u \in V_T^{*1}$ are such that SLRV(θX) contains an item of the form $[B \to \beta_1 . \beta_2 , v]$ with $u \in EFF_1(\beta_2 v)$, then $u \in$ FOLLOW(X).

PROOF. If $[B \to \beta_1 . \beta_2 , v] \in$ SLRV(θX), then $v \in$ FOLLOW(B) and $[B \to \beta_1 . \beta_2 , \Lambda] \in V_0(\theta X)$. Therefore, G contains a derivation

$$S \xrightarrow[R]{*} \alpha B x \xrightarrow[R]{} \alpha \beta_1 \beta_2 x \qquad (1)$$

with $\theta X = \alpha \beta_1$. There are now two main cases to consider.

Case 1 : $\beta_2 \neq \Lambda$. Then $u \in EFF_1(\beta_2 v)$ implies $\beta_2 \xrightarrow{*} uy$ for some $y \in V_T^*$. (The possibility that $u \in EFF(\beta_2 v)$ because $\beta_2 v \xrightarrow{*} v$ and u=v is excluded by the requirement of an eff-derivation.) Hence (1) gives

$$S \xrightarrow[R]{*} \theta X \beta_2 x \xrightarrow{*} \theta X u y x$$

and the conclusion $u \in$ FOLLOW(X) is immediate.

Case 2 : $\beta_2 = \Lambda$. In this case, $u \in EFF_1(\beta_2 v)$ gives u=v, and so $u \in$ FOLLOW(B). Now if $\beta_1 = \Lambda$, the identity $\alpha \beta_1 = \theta X$ becomes $\alpha = \theta X$ and so (1) gives $S \to \theta X B x$. The conclusion $u \in$ FOLLOW(X) then follows from $u \in$ FOLLOW(B) by virtue of the ERFC property of G. If, on the other hand, $\beta_1 \neq \Lambda$, then the identity $\alpha \beta_1 = \theta X$ implies the production $B \to \beta_1$ has the form $B \to \mu X$ and the conclusion $u \in$ FOLLOW(X) follows directly from $u \in$ FOLLOW(B). □

We need one more technical lemma.

LEMMA 6.13

Let (G,C) be a cs-grammar and let $X, Y, Z \in V$ satisfy $X \xrightarrow{*}_c Y \rightarrow_c Z$. If $u \in FOLLOW(X)$ and $SLRV(\theta X) \neq \emptyset$ then $[Y \rightarrow Z., u] \in SLRV(\theta Z)$.

PROOF. We use induction on the number of steps in the c-derivation of Y from X. The basis is the case where there are no steps at all - that is $X = Y$. Since $SLRV(\theta X) \neq \emptyset$, $V_0(\theta X)$ must contain a non-initial item of the form $[B \rightarrow \beta_1 X . \beta_2, \mathcal{A}]$ and so $[B \rightarrow \beta_1 . X \beta_2, \mathcal{A}] \in V_0(\theta)$. It follows that $[Y \rightarrow . Z, \mathcal{A}] \in V_0(\theta)$ also and hence that $[Y \rightarrow Z., \mathcal{A}] \in V_0(\theta Z)$. Because $u \in FOLLOW(X)$ and $X = Y$ we then have $[Y \rightarrow Z., u] \in SLRV(\theta Z)$ and this completes the basis of the induction. For the inductive step, assume the result to be true whenever the c-derivation of Y from X contains n steps $(n \geq 0)$ and suppose $X \xrightarrow{n+1}_c Y$. Then we may distinguish the last step of this derivation and write $X \xrightarrow{n}_c W \rightarrow_c Y$. By the inductive hypothesis we deduce that $[W \rightarrow . Y., u] \in SLRV(\theta Y)$. Now $u \in FOLLOW(X)$ and $X \xrightarrow{*}_c Y$ imply $u \in FOLLOW(Y)$ and, since $SLRV(\theta Y) \neq \emptyset$, the basis of the induction then provides $[Y \rightarrow Z., u] \in SLRV(\theta Z)$ and so completes the inductive step and the proof of the lemma. □

We now have the technical apparatus required to establish sufficient conditions for (G,C) to be CFSLR, given that G is SLR. Because it will be required later, we present the crux of the argument in the following lemma.

LEMMA 6.14

Let G be an unambiguous ERFC grammar and let C be a chain set for G such that (G,C) has Property A. If CFSLRV(θ) is inadequate, then SLRV(μ) is inadequate for some $\mu \rightarrow_c^* \theta$.

PROOF. The case $\theta = \Lambda$ is trivial so assume $\theta \neq \Lambda$. If CFSLRV(θ) is inadequate, then it contains a pair of distinct items $\Delta = [B \rightarrow \beta_1 . \beta_2, v]$ and $\Sigma = [A \rightarrow \alpha . , u]$ such that $u \in EFF_1(\beta_2 v)$. By Theorem 6.10 we deduce that there exist $\gamma, \delta \in V^*$ such that $\Delta \in SLRV(\gamma)$, $\Sigma \in SLRV(\delta)$, $\gamma \rightarrow_c^* \theta$ and $\delta \rightarrow_c^* \theta$. The lemma is immediate if $\gamma = \delta$, so assume $\gamma \neq \delta$. Since $\gamma \rightarrow_c^* \theta$ and $\delta \rightarrow_c^* \theta$ and (G,C) has Property A it follows that γ and δ can differ on only their final symbols. Therefore γ, δ and θ may be written in the form $\gamma = \psi X$, $\delta = \psi Y$ and $\theta = \rho M$ where $\psi \rightarrow_c^* \rho$, $X \rightarrow_c^* M$, $Y \rightarrow_c^* M$ and $X \neq Y$. Note that Lemma 6.12 provides the results $u \in FOLLOW(X)$ and $u \in FOLLOW(Y)$. Now $(\{X,Y\},\{M\})$ is a maximally chained pair and since G is unambiguous there must be a unique maximal intermediate, say Q, for this pair. We then have $X \rightarrow_c^* Q$, $Y \rightarrow_c^* Q$ and $Q \rightarrow_c^* M$. There are three cases to consider.

Case 1 : $Q = X$. Because $X \neq Y$, $Q \rightarrow_c^* Y$ implies $X \rightarrow_c^+ Y$ in this case. Thus $X \rightarrow_c^* U \rightarrow Y$ for some $U \in V$. Now $u \in FOLLOW(X)$ and $SLRV(\psi X) \neq \emptyset$ and so Lemma 6.13 provides $[U \rightarrow Y., u] \in SLRV(\psi Y)$. But $SLRV(\psi Y)$ also contains the item Σ and this must be distinct from $[U \rightarrow Y., u]$ because it does not involve a chain production. (Since it comes from $CFSLRV(\theta)$.) The two items Σ and $[U \rightarrow Y., u]$ are clearly in conflict and so $SLRV(\psi Y)$ is inadequate and the lemma is proved in this case.

Case 2 : $Q = Y$. This case is exactly similar to the previous one.

Case 3 : $Q \neq X$, $Q \neq Y$. Since $X \rightarrow^* Q$ and $Y \rightarrow_c^* Q$ we must have $X \rightarrow_c^* U \rightarrow_c Q$ and $Y \rightarrow_c^* W \rightarrow_c Q$ in this case. Then because $u \in FOLLOW(X)$ and $SLRV(\psi X) \neq \emptyset$, Lemma 6.13 provides $[U \rightarrow Q., u] \in SLRV(\psi Q)$.
Similarly we obtain $[W \rightarrow Q., u] \in SLRV(\psi Q)$.
These two items must be distinct (for otherwise $U = W$, which contradicts the requirement that Q be a maximal intermediate) and are therefore in conflict. Thus $SLRV(\psi Q)$ is inadequate and since $\psi Q \rightarrow_c^* \theta$ we may conclude the lemma. □

Now we can prove the main result.

THEOREM 6.15 (cf. Theorem 3.19)

Let G be an ERFC, SLR grammar and let C be a chain set for G such that (G,C) has Property A. Then (G,C) is CFSLR.

PROOF. Suppose (G,C) is not CFSLR. Then CFSLRV(θ) is inadequate for some $\theta \in V^*$. Since G is SLR, it is certainly LR(1) and therefore unambiguous. Hence, by Lemma 6.14, SLRV(μ) is inadequate for some $\mu \overset{*}{\underset{r}{\Rightarrow}} \theta$ But this contradicts the assumption that G is SLR and so we conclude the theorem. □

The conditions that G be ERFC and that (G,C) has Property A are sufficient to guarantee that (G,C) is CFSLR when G is SLR but they are not necessary conditions. The following grammar demonstrates this point.

S	⟶	AB |	(Grammar G11)
		Cx |	
		Bx	
A	⟶	y	
C	⟶	y	
B	⟶	z |	
		Λ	

Grammar G11 is SLR but not ERFC (the followset of A is $\{\Lambda, z\}$ while that of B is $\{\Lambda, x\}$) and yet (G11, $\{ C \to y \}$) is CFSLR.

On the other hand, the simple condition that G be SLR is not sufficient on its own to guarantee that (G,C) is CFSLR. This point is also demonstrated by the grammar G11, for the cs-grammar (G11, $\{A \to y\}$) is not CFSLR despite the fact that G11 is SLR. Although this

example demonstrates that CFSLR cf-parsers cannot always be substituted for SLR parsers, no difficulty is likely to arise in practice because the conditions of Theorem 6.15 are very mild and almost certain to be satisfied by SLR programming language grammars. (Property A will be required anyway by certain of our later optimisations.)

Grammar G11 is a slight modification of one due to Anderson (1972) whose technique for eliminating chain productions from SLR parsers is the same as the CFSLR method. Anderson stated and proved a weaker form of Theorem 6.15 in his Ph.d thesis (Anderson (1972),Theorem A). Expressed in our terminology, Anderson's result becomes : "(G,C) is CFSLR if G is both SLR and Λ-free". Because (G,C) must have Property A if G is SLR and Λ-free (this follows from Corollary 4.18), Anderson's result follows as a corollary to our Theorem 6.15. Anderson also exhibited CFSLR grammars which are not SLR, nor even LR(k). Thus, just as in the LR(k) case, the CFSLR grammars are more extensive than the SLR grammars. However, also like the LR(k) case, the corresponding classes of languages are the same. To see this we note that results of Mickunas (1976) show that the SLR languages are precisely the deterministic languages. The CFSLR languages must include all SLR languages, but must themselves be included within the CFLR(1) languages - and these are just the deterministic languages again. It follows that the CFSLR languages are precisely the deterministic languages.

This result may also be deduced using the cover grammar approach of Section 3.3 : because followsets are preserved under the cover grammar construction (Construction 3.24) it is easy to adapt the proofs of Theorems 3.26 and 3.29 in order to establish that (G,C) is CFSLR if and only if COVER(G,C) is SLR. This observation also provides an indirect method of testing for the CFSLR property.

Direct methods of testing for the CFSLR property may be found by exploiting Lemma 6.9. The most straightforward technique is to construct the CFLR(0) stateset and then convert it into the CFSLR stateset by using Lemma 6.9. The CFSLR stateset may then be tested for adequacy in the usual way. Since the cardinality of CFLR(0) statesets can be exponential in the size of the grammar (recall the grammars EXP(n) of section 2.3) this method is of exponential complexity.

The most efficient method of testing for the CFSLR property is to construct the set $\text{CF-PAIRS}_0^{(G,C)}$ and then test each pair in this set for an "SLR conflict" where a pair of distinct LR(0) items are said to have an SLR conflict if they have the form $[B \rightarrow \beta_1 . \beta_2 , \lambda]$ and $[A \rightarrow \alpha . , \lambda]$ and satisfy either

(i) $\beta_2 = \lambda$ and $\text{FOLLOW}(A) \cap \text{FOLLOW}(B) \neq \emptyset$ or

(ii) $\beta_2 \neq \lambda$ and $\text{FOLLOW}(A) \cap \text{EFF}_1(\beta_2) \neq \emptyset$.

It is an elementary deduction from Lemma 6.9 that (G,C) is CFSLR if and only if no pair in $\text{CF-PAIRS}_0^{(G,C)}$ has

an SLR conflict. We have seen in Section 3.6 that the enumeration of the set $CF\text{-}PAIRS_0^{(G,C)}$ can be performed in time $O(n^2)$, where n is the size of G. Since the techniques of Hunt et al.(1974) allow a pair of LR(0) items to be tested for SLR conflict in fixed time, it follows that this method of testing for the CFSLR property has complexity $O(n^2)$.

Just as CFSLR testing is based on CFLR(0) constructions, so is the construction of CFSLR parsing tables. The CFSLR stateset may be formed from the CFLR(0) stateset by using Lemma 6.9 and the CFSLR parsing tables may then be constructed from this stateset and a tabulation of the CFLR(0) CF-GOTO function by adapting Construction 3.61 in the obvious manner. It is clear from this construction that CFSLR tables have the same number of states as CFLR(0) tables. It is more convenient in practice to build CFSLR parsing tables directly from CFLR(0) statesets rather than first convert these into CFSLR statesets. This is accomplished by replacing the function ACTION of Construction 3.61 with a more sophisticated function: SLR-ACTION. This is defined as follows :

When Δ is a set of CFLR(0) items for (G,C) and $u \in V_T^{*1}$ the value of SLR-ACTION (Δ , u) is :

(i) SHIFT if Δ contains an item $[B \to \beta_1 . \beta_2 , \lambda]$ with $\beta_2 \neq \Lambda$ and $u \in EFF(\beta_2)$,

(ii) REDUCE q if Δ contains the item $[A \to \alpha . , \lambda]$ where $A \to \alpha$ is production q and $u \in FOLLOW(A)$.

(iii) ERROR if neither case (i) nor case (ii) obtains.

It is clear that this construction is equivalent to the previous one. To illustrate the technique we display the CFLR(0) stateset for (G3,C3) in Figure 6.3. Note that we omit the lookahead string when writing CFLR(0) items, since it is always Λ.

STATE No.	CFLR(0) STATES NUCLEUS	COMPLETION	CF-GOTO S	E	T	P	(	X	)	*	+
1	[S → .E]	[E → .E+T] [T → .T*P] [P → .(E)]		2	3	3	4	3			
2	[S → E.] [E → E.+T]										5
3	[S → E.] [E → E.+T] [T → T.*P]									6	5
4	[P → (.E)]	[E → .E+T] [T → .T*P] [P → .(E)]		7	8	8	4	8			
5	[E → E+.T]	[T → .T*P] [P → .(E)]			9	9	4	9			
6	[T → T*.P]	[P → .(E)]				10	4	10			
7	[P → (E.)] [E → E.+T]								11		5
8	[P → (E.)] [E → E.+T] [T → T.*P]								11	6	5
9	[E → E+T.] [T → T.*P]									6	
10	[T → T*P.]										
11	[P → (E).]										

Figure 6.3: The CFLR(0) Stateset and CF-GOTO Function for (G3,C3).

The followsets of the nonterminals of G3 are easily computed and may be found to be as follows :

FOLLOW (S) = $\{\Lambda\}$,

FOLLOW (E) = $\{\Lambda, +,)\}$,

FOLLOW (T) = FOLLOW (P) = $\{\Lambda, +,), *\}$.

Using this information the CFSLR parsing tables for (G3,C3) may be constructed directly from Figure 6.3. We do not display these tables here since the tables

of Figure 6.2 are, in fact, the CFSLR tables for (G3,C3).

We now briefly consider alternative methods for producing CFSLR parsing tables, based upon the LR(k) to CFLR(k) table conversion methods of Chapter 4.

We saw in Chapter 4 how "quasi" CFLR(k) statesets and CF-GOTO functions can be computed from information contained in LR(k) parsing tables and Theorem 4.4 showed how these may be used to construct CFLR(k) parsing tables by a "post pass" method. It is elementary to show that similar constructions apply to the CFSLR case and that CFSLR tables produced by the post pass method are exactly the same as those of Definition 6.8.

Direct conversion of SLR into CFSLR tables may be achieved by obvious adaptions of the methods for producing QCFLR(k) and SQCFLR(k) tables (Constructions 4.6 and 4.11 respectively). Just as in the LR(k) case, it can be shown that these "QCFSLR" and "SQCFSLR" tables cover the true CFSLR tables. Furthermore, when the cs-grammar concerned has Property A, its SQCFSLR tables can be shown to be identical to its true CFSLR tables. Again, this is exactly similar to the CFLR(k) case.

The reason why these techniques and results apply to the CFSLR method exactly as they do in the CFLR(k) case is because Theorem 6.10 is an exact parallel for Theorem 3.43. We shall see later that in the case of the CFLALR method there is no parallel to Theorem 3.43 and that these post pass and table conversion methods behave rather differently in that case.

6.2. Optimising CFSLR Parsing Tables.

As we have already noted, CFSLR parsing tables contain the same number of states as CFLR(0) tables. For a typical programming language grammar this amounts to several hundreds of states, as opposed to the thousands of states in CFLR(1) tables. The SLR tables for an ALGOLW grammar, for example, contain 328 states whereas the LR(1) tables are an order of magnitude bigger - 4091 states.

Using a compact list representation, Anderson (1972) found that SLR parsing tables for typical programming language grammars could be encoded in two or three thousand bytes. Using a different representation, permitting rather faster access, Joliat (1973) used between one and a half and two times as much space as Anderson. These quantities are sufficiently small to be tolerable, at least on medium and large scale machines, but any reduction would be welcome. Conversely, any more than a modest increase might well be unacceptable in many applications.

Now just as in the LR(k) case, CFSLR tables generally have more states than SLR tables. With realistic programming language grammars the increases are often substantial. For example, an ALGOLW grammar with 13 chain productions has 328 states in its SLR tables and 519 in its CFSLR tables - an increase of 60%. Anderson (1972) found that the space required to

encode the tables grew even more dramatically than the number of states. His SLR tables for ALGOLW occupied 2145 bytes while his CFSLR tables required 5344 bytes - an increase of 150%. This was in spite of the fact that several of the encoding techniques used were especially designed with CFSLR tables in mind.

These observations strongly motivate an attempt to apply the optimisation technique of Chapter 5 to CFSLR tables. Although it is more difficult to establish than in the CFLR(k) case, CFSLR tables are sufficiently well behaved to allow full optimisation in the sense of Chapter 5. In order to prove this fact we need to repeat the argument of Lemmas 5.4 through 5.7 and Theorem 5.8 for the case of CFSLR tables. The first problem here is that the crucial Lemma 5.4 is not true of CFSLR tables. Fortunately, however, we can substitute the following result.

LEMMA 6.16 (cf. Lemma 5.4)

Let $T = (Q, s_0, g, f)$ be the CFSLR parsing tables for (G,C). Then when $q \in Q$ and $u \in V_T^{*1}$, the action $f(q,u)$ is inaccessible on any nonterminal A such that $u \notin$ FOLLOW(A).

PROOF. If the action f(q,u) is inspected when a nonterminal A is on top of the parse stack, then the previous move made by the parser must have been REDUCE q where q is some production with A as its left part. But such actions only occur when the look-ahead is in the followset of A. The result follows. □

Lemma 5.5 is, in fact, true of all approximations.

LEMMA. 6.17 (cf. Lemma 5.5)

Let $T = (Q, s_o, g, f)$ be a set of valid approximate CFLR(1) parsing tables for (G,C). Let $q \in Q$ and $X \in V$ be such that $g(q,x)$ is undefined. Then $g(q,x)$ is inaccessible.

PROOF. When Algorithm 1.4 encounters an undefined goto entry it halts and declares ERROR. When driven by CFLR(1) tables this circumstance cannot occur because all errors are detected during inspection of the action function. Part (iii) of Theorem 6.3 ensures that all valid approximate CFLR(1) tables share this property and so the result follows. □

The CFSLR version of Lemma 5.6 is rather more complex. We need not only that (G,C) is CFSLR, but also that it has Property A and that G is both SLR and ERFC.

LEMMA 6.18 (cf. Lemma 5.6)

Let G be SLR and ERFC and let C be a chain set for G such that (G,C) is CFSLR and has Property A. Let $T = (Q, s_0, g, f)$ be the CFSLR tables for (G,C), let $X, Y \in V$ satisfy $X \rightarrow_c^* Y$ and let $p, q, r \in Q$ be such that $q = g(p,X)$ and $r = g(p,Y)$.

Then for each $u \in V_T^{*1}$, either

(i) $f(q,u) = f(r,u)$ or

(ii) (a) $f(q,u) = \text{ERROR}$ and

(b) $u \notin \text{FOLLOW}(X)$.

PROOF. The argument used to prove Lemma 5.6 may be used to show that if (i) is false then (iia) is true. The hard part is to prove that (iib) is also true. To do this we suppose that $f(q,u) = \text{ERROR}$ and $f(r,u) \neq \text{ERROR}$. Now p is a CFSLR parsing state and so $p = \text{NAMEOF}(\text{CFSLRV}(\theta))$ for some $\theta \in V^*$. It follows that $q = \text{NAMEOF}(\text{CFSLRV}(\theta X))$ and $r = \text{NAMEOF}(\text{CFSLRV}(\theta Y))$. Because $f(r,u) \neq \text{ERROR}$, CFSLRV(θY) must contain an item $\Sigma = [B \rightarrow \beta_1 . \beta_2, v]$ with $u \in \text{EFF}_1(\beta_2 v)$, and because CFSLRV(θX) cannot be empty (because its name is a parsing state) there is some item $\Delta \in \text{CFSLRV}(\theta X)$. By Theorem 6.10 we deduce that there exist μ and γ such that $\Sigma \in \text{SLRV}(\mu)$ and $\Delta \in \text{SLRV}(\gamma)$ where $\mu \rightarrow_c^* \theta Y$ and $\gamma \rightarrow_c^* \theta X$. Since $X \rightarrow_c^* Y$ it follows that $\gamma \rightarrow_c^* \theta Y$ and then, because (G,C) has Property A, it follows that μ and γ can only differ on their final symbols. We may therefore write $\mu = \alpha U$ and $\gamma = \alpha W$ where $U \rightarrow_c^* Y$ and $W \rightarrow_c^* X \rightarrow_c^* Y$. Now $(\{U, X\}, \{Y\})$ is a maximally chained pair and because G is SLR (and therefore both reduced and unambiguous) there must be

some unique maximal intermediate, say Z, for this pair. We then have $U \rightarrow_\epsilon^* Z$, $X \rightarrow_\epsilon^* Z$ and $Z \rightarrow_\epsilon^* Y$. There are three cases to consider :

Case 1: $Z = X$. Here $U \rightarrow_\epsilon^* Z$ becomes $U \rightarrow_\epsilon^* X$ which implies that $\mu = \alpha U \rightarrow_\epsilon^* \theta X$ and then, by Theorem 6.10, $\Sigma \in SLRV(\mu)$ gives $\Sigma \in CFSLRV(\theta X)$. But this is impossible - for it would imply that $f(q,u) = f(r,u)$. We conclude that this case cannot occur.

Case 2 : $Z = U$. Here $X \rightarrow_\epsilon^* Z$ becomes $X \rightarrow_\epsilon^* U$ but since we cannot have $X = U$ (because the case above has shown $U \rightarrow_\epsilon^* X$ to be impossible), there must be some $A \in V$ such that $X \rightarrow_\epsilon^* A \rightarrow_\epsilon U$. Now $SLRV(\alpha X) \neq \emptyset$ and so if $u \in FOLLOW(X)$ it would follow from Lemma 6.13 that $[A \rightarrow U., u] \in SLRV(\alpha U)$. But this item conflicts with the non-chain item Σ, which is also a member of $SLRV(\alpha U)$, and this contradicts the premiss that G is SLR. We conclude that $u \notin FOLLOW(X)$.

Case 3 : $Z \neq U$, $Z \neq X$. In this case, $U \rightarrow_\epsilon^* Z$ and $X \rightarrow_\epsilon^* Z$ imply that $U \rightarrow_\epsilon^* A \rightarrow_\epsilon Z$ and $X \rightarrow_\epsilon^* D \rightarrow_\epsilon Z$ for some $A, D \in V_N$. Now because $\Sigma \in SLRV(\alpha U)$ and Σ has an action on u it follows by Lemma 6.12 that $u \in FOLLOW(U)$. Then since $SLRV(\alpha U) \neq \emptyset$ it follows from Lemma 6.13 that $[A \rightarrow Z., u] \in SLRV(\alpha Z)$. Similarly, if $u \in FOLLOW(X)$, it follows that $[D \rightarrow Z., u] \in SLRV(\alpha Z)$. These two items must be in conflict (or else they are the same - which implies that $A = D$ and this contradicts the choice that Z is a maximal intermediate) and this contradicts the premiss that G is SLR. We conclude that $u \notin FOLLOW(X)$ and the proof is complete. □

Lemma 5.7 extends directly to all $\bar{\Phi}$ - approximate CFLR(1) tables.

LEMMA 6.19 (cf. Lemma 5.7)

Let (G,C) have Property A and let T = (Q,s_0,g,f) be the $\bar{\Phi}$ - approximate CFLR(1) tables for (G,C) where $\bar{\Phi}$ is a CFLR(1) approximation function. Let $\alpha,\beta \in V^*$ be such that $\alpha \underset{c}{\rightarrow}^* \beta$. Then when $X \in V$, either

(i) $\bar{\Phi}(CFV_1(\alpha X)) = \bar{\Phi}(CFV_1(\beta X))$ or

(ii) $\bar{\Phi}(CFV_1(\alpha X)) = \emptyset$.

PROOF. This result is a trivial consequence of Lemma 5.7 and Definition 6.6. □

Finally, we achieve the result we seek.

THEOREM 6.20 (cf. Theorem 5.8)

Let G be SLR and ERFC and let C be a chain set for G such that (G,C) is CFSLR and has Property A. Let T = (Q,s_0,g,f) be the CFSLR tables for (G,C) and let $p \in Q$ and $X, Y \in V$. Then the value of $g(p,Y)$ is a valid substitute for $g(p,X)$ whenever $X \underset{c}{\rightarrow}^* Y$.

PROOF. This result may be proved by the argument used to establish Theorem 5.8, with Lemmas 6.16 through 6,19 replacing Lemmas 5.4. through 5.7 respectively. □

It follows that (under the mild conditions of Theorem 6.20) CFSLR tables can be optimised just like CFLR(k) tables - by selecting an optimising function F and employing the obvious modification of Construction 5.11. Optimised CFSLR tables (OCFSLR tables for short) can, of course, be constructed directly using straightforward adaptions of Algorithms 5.12 or 5.13. We illustrate the effect of optimisation by showing the fully optimised CFSLR tables (FOCFSLR tables for short) for (G3,C3) in Figure 6.4.

STATE No.	CF-ACTION FUNCTION						CF-GOTO FUNCTION					
	↓	(	x	)	*	+	S	(	X	)	*	+
1		sh	sh					3	2			
2	1				sh	sh					5	4
3		sh	sh					3	6			
4		sh	sh					3	7			
5		sh	sh					3	8			
6				sh	sh	sh				9	5	4
7	2			2	sh	2					5	
8	4			4	4	4						
9	6			6	6	6						

Figure 6.4 : The FOCFSLR Tables for (G3,C3)

The SLR tables for G3 and the CFSLR and FOCFSLR tables for (G3,C3) contain 12,11 and 9 states respectively. These savings conferred by optimisation are maintained with larger, more realistic, programming language grammars. For example, the ALGOLW grammar mentioned earlier has 328, 519 and 321 states in its SLR, CFSLR and FOCFSLR tables respectively.

Practical experience with programming language grammars indicates that FOCSLR tables always contain fewer states than ordinary SLR tables. However, as in the LR(k) case, Grammar G10 demonstrates that this reduction in the number of states is not universal. This grammar is SLR and has 48 states in its SLR tables whereas the FOCFSLR tables for (G10, $\{X \rightarrow Y , Y \rightarrow Z\}$) contain 50 states.

6.3. OCFSLR Parsing Tables and the Further Postponement of Error Detection.

Postponement of error detection is an inherent feature of the CFSLR method and is the price paid for obtaining parsing tables of an acceptable size. Now in order to achieve compact representations of CFSLR tables it is useful to use even more extensive postponement of error detection than is strictly required by the method. As an example, compare the CFSLR tables for (G3,C3) shown in Figure 6.2. with those shown in Figure 6.5, where extensive additional valid postponement of error detection has occurred. (The reduce entries introduced by the postponement are circled in Figure 6.5).

STATE NO.	CF-ACTION FUNCTION						CF-GOTO FUNCTION								
	↓	(	X	)	*	+	S	E	T	P	(	X	)	*	+
1		sh	sh					2	3	3	4	3			
2	1					sh									5
3	1				sh	sh								6	5
4		sh	sh					7	8	8	4	8			
5		sh	sh						9	9	4	9			
6		sh	sh							10	4	10			
7				sh		sh							11		5
8				sh	sh	sh							11	6	5
9	2	②	②	2	sh	2								6	
10	4	④	④	4	4	4									
11	6	⑥	⑥	6	6	6									

Figure 6.5: The CFSLR Tables for (G3,C3) after Application of a Valid Postponement Set.

The tables of Figure 6.5 are capable of more compact representation than those of Figure 6.2. In Figure 6.5, the action table entries in state 10 are all REDUCE 4 and using the representational techniques of Anderson (1972) or Joliat (1973) we can avoid the need to store the individual actions in this state at all; we just record the single fact that all the actions in this state are REDUCE 4. Similarly, in state 9 we explicitly record the fact that f(9,*) = SHIFT and then simply note that all other actions in that state are REDUCE 2 without needing to store them individually. The details of these encoding techniques do not concern us here, the important fact is that substantial savings in the space required to represent CFSLR (and other CFLR(k)-type) parsing tables are made possible by allowing additional postponement of error detection.

Now with ordinary SLR and CFSLR tables, this additional postponement of error detection can cause no problems: the weak postponement $(q,u,A \rightarrow \alpha)$ will be valid provided only that $A \neq S$ and $f(q,u) = \text{ERROR}$. The issue we need to consider here is that of possible interaction between the further postponement of error detection and the technique presented in the previous section for optimising CFSLR tables. The potential for interaction certainly exists because the further postponement of error detection may invalidate Lemma 6.16 - one of the results upon which the correctness of the optimisation technique depends. We demonstrate that interaction can actually occur by means of an example.

We use the following grammar.

1	S	→	Ea|	(Grammar G 12)
2			eba	
3	E	→	e|	
4			c	

This grammar is SLR and Λ-free. It must therefore be ERFC and furthermore, the cs-grammar (G12, {E → e}) must have Property A and be CFSLR. The FOCFSLR tables for (G12, {E → e}) are shown in Figure 6.6.

STATE	CF-ACTION FUNCTION					CF-GOTO FUNCTION				
No.	Λ	a	b	c	e	S	a	b	c	e
1				sh	sh				3	2
2		sh	sh				4	5		
3		4								
4	1									
5		sh					6			
6	2									

Figure 6.6. The FOCFSLR Tables for (G12, {E → e}).

If the weak postponement (3,b,E → c) is applied to these tables it will cause them to accept the invalid string cba. The reason for this is, of course, that the postponement (3,b, E → c) is not valid.

Thus, unlike the case of unoptimised CFSLR tables where it is only necessary to ensure that postponements satisfy conditions (i) and (ii) of Definition 6.2 in order to be valid, it seems that with optimised CFSLR tables the condition (iii) must be checked each time as well. There are two disadvantages to this technique. On the practical side, it is rather complex and time consuming; while on the theoretical side, it gives no indication of how much additional postponement of error detection is likely to be possible - we do not know whether we are likely to reap virtually all, or virtually none, of the benefits which the further postponement of error detection confers on unoptimised tables.

Similar objections may be raised to the converse scheme of first applying additional postponement of error detection to the unoptimised CFSLR tables and then checking that each substitution required by the optimisation technique is indeed valid.

In contrast to these "try it and see" techniques, we present a theoretical result which may be used à priori to guarantee the validity of certain combinations of optimisation and the additional postponement of error detection. This result suggests that with conventional programming language grammars <u>both</u> techniques may be exploited to the full.

THEOREM 6.21

Let $T = (Q, s_0, g, f)$ be the OCFSLR tables for (G,C) with respect to the optimising function F where the validity of the substitutions performed during optimisation are guaranteed by Theorem 6.20. The weak postponement $(q, u, A \rightarrow \alpha)$ is valid if :

(i) $A \neq S$,
(ii) $f(q,u) = \text{ERROR}$, and
(iii) $u \notin \text{FOLLOW}(F(A))$.

PROOF. We have to ensure that all three conditions of Definition 6.2 are satisfied. Certainly its conditions (i) and (ii) are satisfied since they are the same as those in the statement of the theorem. It remains to prove that condition (iii) above implies condition (iii) of Definition 6.2. That is, we must show that whenever p is such that $g(p, \alpha) = q$, then

(a) $g(p,A)$ is defined, and
(b) $f(g(p,A),u) = \text{ERROR}$.

Now by the definition of a weak postponement, there must be some $v \in V_T^{*1}$ such that $f(q,v) = \text{REDUCE } A \rightarrow \alpha$ and so (a) must always be true - for otherwise an undefined goto entry $g(p,A)$ could be encountered with the tables in their original form.

In order to see that (b) must be true, note that the state p must also be a state in the unoptimised CFSLR tables for (G,C). Let these unoptimised tables be $T' = (Q', s_0', g', f')$. Then $g(p,A) = g'(p,F(A))$. By combining Theorem 6.10 and Lemma 6.12 it is easy to prove that for any $p \in Q'$ and $X \in V$ we must have $f'(g'(p,X),u) = \text{ERROR}$ whenever $u \notin \text{FOLLOW}(X)$. Hence, if $u \notin \text{FOLLOW}(F(A))$

it follows that

$$f(g(p,A),u) = f'(g'(p,F(A)),u) = \text{ERROR}$$

and the proof is complete. □

This result is extremely powerful. Observe that there is no point in applying a weak postponement $(q,u,A \rightarrow \alpha)$ to CFSLR tables when $u \in FOLLOW(A)$ because we must have $f(q,u) = \text{REDUCE } A \rightarrow \alpha$ already. Thus it is only postponements where $u \notin FOLLOW(A)$ that are of interest. Theorem 6.21 applies to postponements where $u \notin FOLLOW(F(A))$ and since we must have $FOLLOW(A) \subseteq FOLLOW(F(A))$ it follows that the "dark area" in which Theorem 6.21 is of no help is given by $DARKAREA(A) = FOLLOW(F(A)) \setminus FOLLOW(A)$ The utility of Theorem 6.21 is due to the fact that such dark areas are usually very small.

In any realistic programming language grammar only a handful of nonterminals will take part in chain productions ; all others will satisfy $A = F(A)$ and therefore $DARKAREA(A) = \emptyset$ for these nonterminals. Even when $A \neq F(A)$, the cardinality of $DARKAREA(A)$ is likely to be quite small. (If the grammar admits of more than one full optimising function, then minimising the size of these dark areas provides a useful criterion for selecting among them). Furthermore, it is commonly the case that dark areas contain non-error actions and therefore no postponement is possible there anyway. In the FOCFSLR tables for (G3,C3) (Figure 6.4), for example, the only non-empty dark area is $DARKAREA(E) = \{*\}$. Thus

part (iii) of Theorem 6.21 gives no help in deciding the validity of the postponement (7,*,2) (production 2 is E → E + T). However, f(7,*) = SHIFT and so this postponement will be rejected by part (ii) of Theorem 6.21.

Hence it seems that postponements which satisfy conditions (i) and (ii) but not (iii) of Theorem 6.21 are very few. They may either be tested for validity by using the full form of Definition 6.2, or simply rejected outright. In abnormal circumstances, where few postponements are found to be valid, it may be desirable to use a less than fully optimising function in order to increase the amount of postponement which is possible. Experimentation would be needed to evaluate the trade-offs concerned.

We are confident that the full benefits of both optimisation and the further postponement of error detection are simultaneously available with CFSLR parsing tables, and that these tables may be represented in less space than ordinary SLR tables. Naturally, we should welcome empirical investigation of these claims.

We conclude that CFSLR parsers are viable for use in compilers. They are as widely applicable as the SLR parsers, provide the same excellent error detection, are much faster and probably smaller. Furthermore, the CFSLR grammars and parsers exhibit very similar theoretical properties to those of the CFLR(k) grammars and parsers :

most results concerning CFLR(k) grammars have natural counterparts in the CFSLR case. In the rest of this chapter we shall consider another class of approximate CFLR(1) parsers for which this similarity does not hold so well.

6.4. The CFLALR Method.

Like SLR, the LALR method is an LR(1) approximation which was first proposed by DeRemer (1969). Several formulations of the method exist, we shall use one due to Anderson (1972) and we define its chain free generalisation, which we call CFLALR as follows.

DEFINITION 6.22

The CFLALR approximation method is defined by the approximation function Φ_{LALR} where, for each $\theta \in V^*$, we define

$$\Phi_{LALR}(CFV_1(\theta)) = \bigcup_{CFV_0(\mu)=CFV_0(\theta)} CFV_1(\mu).$$

For convenience, we write CFLALR states as CFLALRV(θ) rather than Φ_{LALR} ($CFV_1(\theta)$). The notion of the "core" of a set of CFLR(1) states is often useful in discussion of the CFLALR method. If Δ is a set of CFLR(1) items, then its core, denoted by CORE(Δ) is theset of CFLR(0) items formed by changing to λ all the second (lookahead) components of the items in Δ. That is :

$$CORE(\Delta) = \{[B \rightarrow \beta_1.\beta_2, \lambda] \mid [B \rightarrow \beta_1.\beta_2, u] \in \Delta\}.$$

Note that the CFLALR state for θ is simply the union of all CFLR(1) states having the same core as $CFV_1(\theta)$. Note also that if $CORE(CFV_1(\theta)) = CORE(CFV_1(\mu))$ (or, equivalently, if $CFV_0(\theta) = CFV_0(\mu)$) then CFLALRV($\theta$) = CFLALRV($\mu$).

We say that (G,C) is CFLALR if its CFLALR stateset is adequate ; G is CFLALR if there is a chain set C for G such that (G,C) is CFLALR. A language is CFLALR if it is generated by some CFLALR grammar. □

When the chain set C is empty, the CFLALR method becomes the ordinary LALR method. We say that a grammar G is LALR if (G,∅)is CFLALR and that a language is LALR if it is generated by an LALR grammar. When the chain set is empty,we talk of LALR states, rather than CFLALR states and write LALRV(θ) instead of CFLALRV(θ).

As in the CFSLR case, the CFLALR approximation function always satisfies conditions (i) and (iib) of Definition 6.6 but may not satisfy condition (iia) if the goal symbol appears in the right part of a production. This difficulty may be circumvented by augmenting the grammar with a new production $S' \rightarrow S \perp$.

It is clear from Definition 6.22 that two cf-viable prefixes share the same CFLALR state if and only if their CFLR(0) states are the same. If follows that CFLALR states are in one-to-one correspondence with CFLR(0) states and therefore with CFSLR states also. Thus the CFLALR and CFSLR methods yield parsing tables of the same size as each other.

Of the two methods, CFLALR is the more powerful; in fact CFSLR tables can be formed by applying a weak postponement set to CFLALR tables. This result is established in the following theorem.

THEOREM 6.23

Let $\theta \in V^*$. Then

(i) $CFLALRV(\theta) \subseteq CFSLRV(\theta)$ and

(ii) if $[B \rightarrow \beta_1 . \beta_2, u] \in CFSLRV(\theta)$ then $[B \rightarrow \beta_1 . \beta_2, v] \in CFLALRV(\theta)$ for some $v \in V_T^{*1}$.

PROOF. Clearly, $[B \rightarrow \beta_1 . \beta_2, u] \in CFLALRV(\theta)$ implies $[B \rightarrow \beta_1 . \beta_2, \Lambda] \in CFV_0(\theta)$ and $u \in FOLLOW(B)$. Hence, by Lemma 6.9, $[B \rightarrow \beta_1 . \beta_2, u] \in CFSLRV(\theta)$. On the other hand, $[B \rightarrow \beta_1 . \beta_2, u] \in CFSLRV(\theta)$ implies $[B \rightarrow \beta_1 . \beta_2, \Lambda] \in CFV_0(\theta)$ and so $[B \rightarrow \beta_1 . \beta_2, v] \in CFV_1(\theta)$ for some $v \in V_T^{*1}$. It follows that $[B \rightarrow \beta_1 . \beta_2, v] \in CFLALRV(\theta)$. □

It is clear from this result that the classes of grammars acceptable to each method satisfy the following inclusions $CFLR(0) \subseteq CFSLR \subseteq CFLALR \subseteq CFLR(1)$. Examples may easily be constructed to show that each of these inclusions is strict.(See, for example, Anderson (1972), pp 36 - 48).

Since CFLALR is a more general method than CFSLR it might seem that it is to be preferred to CFSLR in all practical applications. This may not be so, however, for the CFLALR method does have its disadvantages. Principal among these are the complexity of the algorithms used to test for the property and to build its parsing tables. Whereas the CFSLR algorithms are comparable in complexity to those for CFLR(0), the CFLALR algorithms are more akin to those for CFLR(1). Furthermore, the mathematics of the CFLALR method is rather intractable and some results from the CFLR(k) theory extend less well to this method than

to the simpler CFSLR technique.

On the other hand the greater generality of the CFLALR method may sometimes be a practical asset. Although the natural grammars for many programming languages do seem to be CFSLR, there are some which are not but which are CFLALR. While it is always possible to convert any LR(1) grammar, and therefore any LALR grammar, into a similar SLR grammar for the same language (see Mickunas (1976)), this may be neither easy nor desirable.

We shall now examine some of the properties of the CFLALR method. The first question we ask is : given an LALR grammar G and a chain set C, can we be sure that (G,C) is CFLALR ? Recall that in the LR(k) case the answer is always "yes" (Theorem 3.19) while in the SLR case we had to require that G be ERFC and that (G,C) have Property A (Theorem 6.15). In the present case, the answer is again a qualified "yes"; we shall have to require that all chain sets are "simple" and define this property as follows.

DEFINITION 6.24

A chain set is simple if every maximally chained pair (recall Definition 4.16) has an intermediate. □

In simple chain sets all chains from one set of symbols to another must pass through a common intermediate symbol. This excludes chain sets of the form

$\{X \rightarrow A, X \rightarrow B, Y \rightarrow A, Y \rightarrow B\}$ because there is no intermediate for the maximally chained pair $(\{X, Y\}, \{A, B\})$.

Before proving the main result we need a minor, but interesting, lemma.

LEMMA 6.25

Let (G,C) be CFLR(1). Then any inadequacies in the CFLALR stateset for (G,C) are due to conflicts of the reduce/reduce variety.

PROOF. Suppose the CFLALR stateset for (G,C) contains an inadequacy due to a shift/reduce conflict. Then for some $\theta \in V^*$ we have

$$[B \rightarrow \beta_1 . \beta_2, v], \ [A \rightarrow \alpha ., u] \in CFLALRV(\theta)$$

where $\beta_2 \neq \Lambda$ and $u \in EFF_1(\beta_2 v)$. Then, by virtue of the definition of CFLALR states, there must exist some $\mu \in V^*$ and $w \in V_T^{*1}$ such that $CFV_0(\theta) = CFV_0(\mu)$ and $[B \rightarrow \beta_1 . \beta_2, w], \ [A \rightarrow \alpha ., u] \in CFV_1(\mu)$.
Now since $\beta_2 \neq \Lambda$ the value of $EFF_1(\beta_2 v)$ is independent of v and so we have $u \in EFF_1(\beta_2 w)$. But then $CFV_1(\mu)$ is inadequate and this contradicts the premise that (G,C) is CFLR(1). □

Now we can state and prove the main result.

THEOREM 6.26

Let G be an LALR grammar and let C be a simple chain set for G such that (G,C) has Property A. Then (G,C) is CFLALR.

PROOF. The proof is long and arduous. We suppose the theorem to be false and proceed to derive a contradiction.

Now if (G,C) is not CFLALR, there exists $\xi \in V^*$ such that CFLALRV(ξ) is inadequate. Because G is LALR it must be LR(1) and so, by Theorem 3.19, (G,C) is CFLR(1). Hence, by Lemma 6.25, the inadequacy in CFLALRV(ξ) must be due to a reduce/reduce conflict. Therefore CFLALRV(ξ) contains a pair of distinct final items $\Sigma = [A \to \alpha .,u]$ and $\Delta = [B \to \beta .,u]$. By the definition of CFLALR states, there must exist $\rho,\rho' \in V^*$ such that $\Sigma \in CFV_1(\rho)$, $\Delta \in CFV_1(\rho')$ and $CFV_0(\xi) = CFV_0(\rho) = CFV_0(\rho')$. Note that $\rho \neq \rho'$, for otherwise $CFV_1(\rho)$ would be inadequate, and that $\rho \neq \Lambda$ and $\rho' \neq \Lambda$. (This is because $CFV_0(\rho) = CFV_0(\Lambda)$ if and only if $\rho = \Lambda$.) We now establish a series of claims.

Claim 1 : There exist $\theta,\theta' \in V^*$ such that $\theta \xrightarrow{*}_r \rho$, $\theta' \xrightarrow{*}_r \rho'$, $V_0(\theta) = V_0(\theta')$, $\Sigma \in V_1(\theta)$ and $\Sigma' \in V_1(\theta')$ where Σ' is an item of the form $\Sigma' = [A \to \alpha .,v]$.

Proof of claim. Suppose first that Σ is a non-initial item (i.e. $\alpha \neq \Lambda$). Then $\Sigma \in CFN_1(\rho)$ and so, by Theorem 3.43, there exists $\theta \xrightarrow{*}_r \rho$ such that $\Sigma \in N_1(\theta)$ and hence $\Sigma \in V_1(\theta)$. Now $CFV_0(\rho) = CFV_0(\rho')$ and so $CFN_1(\rho')$ contains an item $\Sigma' = [A \to \alpha .,v]$. It follows that $\Sigma' \in N_1(\theta')$, and hence $\Sigma' \in V_1(\theta')$, for some $\theta' \to \rho'$. Now note that $[A \to \alpha .,\Lambda] \in N_0(\theta) \cap N_0(\theta')$. It

therefore follows by Lemma 4.21 that $V_0(\theta) = V_0(\theta')$ and the claim is proved for this case.

Now suppose that Σ is an initial item (i.e. $\alpha = \Lambda$). Then there must be some non-initial item $\Gamma = [D \rightarrow \sigma_1 . \sigma_2 , a]$ in $CFN_1(\rho)$ such that $\Sigma \in CLOSURE(\{\Gamma\})$. By the argument used above, it follows that there exist $\theta, \theta' \in V^*$ such that $\theta \rightarrow_r^* \rho$, $\theta' \rightarrow_r^* \rho'$, $V_0(\theta) = V_0(\theta')$, $\Gamma \in V_1(\theta)$ and $\Gamma' \in V_1(\theta')$ where Γ' is of the form $\Gamma' = [D \rightarrow \sigma_1 . \sigma_2 , b]$. Now because $\Sigma \in CLOSURE(\{\Gamma\})$ it follows that $\Sigma \in V_1(\theta)$ and that $\Sigma' \in CLOSURE(\{\Gamma'\})$ where Σ' is of the form $\Sigma' = [A \rightarrow \alpha . , v]$. Hence $\Sigma' \in V_1(\theta')$ and the proof of the claim is complete.

Claim 2 : There exist $\mu, \mu' \in V^*$ such that $\mu \rightarrow_r^* \rho$, $\mu' \rightarrow_r^* \rho'$, $V_0(\mu) = V_0(\mu')$, $\Delta \in V_1(\mu)$ and $\Delta' \in V_1(\mu')$ where Δ is an item of the form $\Delta = [B \rightarrow \beta . , w]$.

Proof of claim. The proof is identical to that of claim 1.

Claim 3 : The strings $\theta, \theta', \mu, \mu', \rho$ and ρ' have the form :

$$\theta = \gamma X \ , \ \theta' = \delta X,$$

$$\mu = \gamma Y \ , \ \mu' = \delta Y,$$

$$\rho = \sigma M \ , \ \rho' = \pi N$$

where $\gamma \rightarrow_r^* \sigma$, $\delta \rightarrow_r^* \pi$ and $(\{X, Y\} , \{M, N\})$ is a maximally chained pair.

Proof of claim. From claim 1 we have $V_0(\theta) = V_0(\theta')$ and therefore θ and θ' must both have the same final symbol. (Consider the associated symbol of any non-initial item in $V_0(\theta)$.) Since $\theta \rightarrow_r^* \rho$, and $\theta' \rightarrow_r^* \rho'$ we may therefore write $\theta = \gamma X$, $\theta' = \delta X$, $\rho = \sigma M$ and $\rho' = \pi N$ where $\gamma \rightarrow_r^* \sigma$, $\delta \rightarrow_r^* \pi$, $X \rightarrow_r^* M$ and $X \rightarrow_r^* N$. We also have both $\theta \rightarrow_r^* \rho$, and $\mu \rightarrow_r^* \rho$ and since (G,C)

has Property A it follows that θ and μ can differ on only their final symbols. Thus $\mu = \gamma Y$ where $Y \rightarrow_c^* M$. By the same argument it follows that θ' and μ' can differ on only their final symbols, and because $V_0(\mu) = V_0(\mu')$ it follows that μ and μ' share the same final symbol. Thus $\mu' = \delta Y$ and since $\mu' \rightarrow_c^* \rho'$ we deduce that $Y \rightarrow_c^* N$ and the claim is proved.

The remainder of the proof is a case analysis.

Case 1 : $X = Y$. In this case $\theta = \mu$ and $\theta' = \mu'$. It follows that $\Sigma \in V_1(\theta)$ and $\Delta' \in V_1(\theta')$. Since $V_0(\theta) = V_0(\theta')$ it then follows that $\Sigma, \Delta' \in$ LALRV(θ) and so LALRV(θ) is inadequate. This contradicts the premise that G is LALR.

Case 2 : $X \neq Y$. Since $(\{X,Y\},\{M,N\})$ is a maximally chained pair and C is constrained to be a simple chain set, there must be some Z which is an intermediate for this pair. Without loss of generality we may assume that Z is a maximal intermediate. There are three sub-cases to consider, depending on the identity of Z.

Sub-case (a) : $Z = X$. Because Z is an intermediate, we have $Y \rightarrow_c^* Z$. But if $X \neq Y$ and $Z = X$ this becomes $Y \rightarrow_c^+ Z$ and so there exists $W \in V$ such that $Y \rightarrow_c^* W \rightarrow_c Z$. Now $\Delta' = [B \rightarrow \beta\ .,u] \in V_1(\delta Y)$ and this implies (by a straightforward but tedious argument which we omit) that $[W \rightarrow Z.,u] \in V_1(\delta Z)$. Because $X = Z$, we have $\delta Z = \delta X = \theta'$ and so $[W \rightarrow Z.,u] \in V_1(\theta')$. Now $\Sigma \in V_1(\theta)$ and $V_0(\theta) = V_0(\theta')$. It follows that both Σ and $[W \rightarrow Z.,u]$ are members of LALRV(θ). These two items must be distinct (because one involves a chain production while the other does not) and so they are in

conflict. Thus LALRV(θ) is inadequate and this contradicts the premise that G is LALR.

Sub-case (b) : $Z = Y$. The analysis of this case proceeds exactly as in the case above.

Sub-case (c) : $Z \neq X$, $Z \neq Y$. In this case we have $X \rightarrow_c^+ Z$ and $Y \rightarrow_c^+ Z$ and so there exist $U, W \in V$ such that $X \rightarrow_c^* U \rightarrow_c Z$ and $Y \rightarrow_c^* W \rightarrow_c Z$. Now because $\Delta' = [B \rightarrow \beta\ .\ , u] \in V_1(\delta Y)$ it follows that $[W \rightarrow Z.\ , u] \in V(\delta Z)$. Similarly, because $\Sigma = [A \rightarrow \alpha\ .\ , u] \in V_1(\gamma X)$, it follows that $[U \rightarrow Z.\ , u] \in V(\gamma Z)$ and because $\Delta = [B \rightarrow \beta\ .\ , w] \in V_1(\gamma Y)$ it follows that $[W \rightarrow Z.\ , w] \in V_1(\gamma Z)$. Now we have $[W \rightarrow Z.\ , \Lambda] \in N_0(\delta Z) \cap N_0(\gamma Z)$ and $\delta Z \rightarrow_c^* \pi N = \rho'$, $\gamma Z \rightarrow_c^* \sigma M = \rho$ and $CFV_0(\rho) = CFV_0(\rho')$. Hence, by Lemma 4.21, it follows that $V_0(\delta Z) = V_0(\gamma Z)$. This implies that $[W \rightarrow Z.\ , u]$, $[U \rightarrow Z.\ , u] \in LALRV(\gamma Z)$ and since these items are distinct (for otherwise $U=W$ and Z would not be a maximal intermediate), they are in conflict. Thus LALRV(γZ) is inadequate and this contradicts the premise that G is LALR.

In all cases we have derived a contradiction and so we conclude the theorem. □

The conditions that (G,C) has Property A and that C is simple are sufficient to guarantee that (G,C) is CFLALR, given that G is LALR. However, they are not necessary conditions and examples may easily be constructed to demonstrate this point. Some constraints on both G and C are certainly necessary, though, as the following two grammars demonstrate.

(Grammar G13)

$$
\begin{aligned}
S &\longrightarrow a\ P\ u \mid \\
&\quad\ a\ Q\ v \mid \\
&\quad\ a\ B\ a \mid \\
&\quad\ b\ P\ w \mid \\
&\quad\ b\ Q\ u \mid \\
&\quad\ b\ A\ b \mid \\
P &\longrightarrow X \\
Q &\longrightarrow Y \\
X &\longrightarrow A \mid \\
&\quad\ B \\
Y &\longrightarrow A \mid \\
&\quad\ B \\
A &\longrightarrow a \\
B &\longrightarrow b
\end{aligned}
$$

This grammar is LALR. When the non-simple chain set $C13 = \{X \to A, X \to B, Y \to A, Y \to B\}$ is chosen we obtain a cs-grammar (G13,C13) which is not CFLALR.(Observe that G13 is Λ-free and therefore (G13,C13) has Property A.)

The next grammar demonstrates the necessity of some constraint similar to Property A.

(Grammar G14)

$$
\begin{aligned}
S &\longrightarrow a\ Y\ b \mid \\
&\quad\ a\ X \mid \\
&\quad\ B\ X\ b \mid \\
&\quad\ b\ Y \\
X &\longrightarrow C \\
Y &\longrightarrow C \\
C &\longrightarrow \Lambda \\
B &\longrightarrow b
\end{aligned}
$$

This grammar is LALR and the chain set $\{B \to b\}$ is certainly simple. However, the cs-grammar $(G14, \{B \to b\})$ does not have Property A and is not CFLALR.

We have argued previously that CFLR(1) cs-grammars without Property A are unusual, if not pathological, and unlikely to occur in practice. Non-simple chain sets may surely be regarded likewise and so Theorem 6.26 is sufficient to ensure that CFLALR parsers can be substituted successfully for LALR parsers in practice.

We note in passing that, as with all the other classes of grammars considered, there exist CFLALR grammars which are not LALR. However, the class of CFLALR and LALR languages are easily seen to be co-extensive.

We now consider methods of testing for the CFLALR property and for constructing CFLALR parsing tables. In theory, solutions to these problems are provided by Definition 6.22 itself for we can simply construct the CFLR(1) stateset in the usual way and then combine all CFLR(1) states having identical cores, thereby producing the CFLALR stateset. This can then be tested for adequacy, thereby providing a solution to the problem of testing for the CFLALR property. Given an adequate CFLALR stateset, the corresponding parsing tables can be constructed in the usual way.

The difficulty with this approach lies in its method for constructing the CFLALR stateset; since this involves constructing the CFLR(1) stateset first, the method runs straight into the difficulty which the approximate CFLR(1) parsers are designed to avoid - namely the impracticability of constructing such very large objects.

Fortunately, a modification of the CFLR(1) stateset construction algorithm permits CFLALR statesets to be built directly. Instead of first generating all the CFLR(1) states and only then combining those which have the same cores, the modified algorithm combines states with identical cores as they are generated. In doing so, it may enlarge a state whose successors have already been evaluated, and so these successors must be re-evaluated in case they too are to be enlarged. This means that the amount of work performed by this modified algorithm is likely to remain comparable to that involved in constructing CFLR(1) statesets, even though the stateset itself remains CFLR(0) sized throughout the computation. In practice as much as an order of magnitude more effort may be required to construct a CFLALR stateset rather than a CFSLR one. The modified algorithm for constructing CFLALR statesets is formally defined as follows.

ALGORITHM 6.27 (cf. Algorithm 3.47).

Evaluation of the CFLALR stateset for (G,C).

Input : The cs-grammar (G,C)

Output : The CFLALR stateset for (G,C).

Method : The stateset is built up in the set-valued variable S. States added to S have marker flags attached to them and are unmarked when first added. Unlike the algorithm for the CFLR(k) case (Algorithm 3.47), states can be changed after being added to S; when a state is changed, its mark is erased in order to force its successor states to be recomputed.

```
begin
    set S = {CFV1(Λ)};
    while S contains any unmarked states do
      select an unmarked state Δ in S and mark it;
      for each X ∈ V do
          compute Σ = CF-GOTO1(Δ,X);
          if there exists Γ ∈ S such that
             CORE(Γ) = CORE(Σ) then
             if Γ ⊉ Σ then replace Γ by
                  Γ ∪ Σ and erase its mark
             endif
          else add Σ to S endif
      endfor
    endwhile;
    S now contains the CFLALR stateset for (G,C)
end. □
```

Since it is not obvious, we now prove the correctness of Algorithm 6.27.

THEOREM 6.28

Algorithm 6.27 correctly computes the CFLALR stateset for (G,C).

Proof. First note that a straightforward deduction from Definition 6.22 yields

$$CFV_1(\theta X) \subseteq CF\text{-}GOTO_1(\Delta, X) \subseteq CFLALRV(\theta X)$$

whenever $X \in V$ and $CFV_1(\theta) \subseteq \Delta \subseteq CFLALRV(\theta)$. Then, by an induction on the total number of different states that have been placed in the variable S, it can be shown that any state Δ present in S at any instant must satisfy

$$CFV_1(\theta) \subseteq \Delta \subseteq CFLALRV(\theta)$$

for some $\theta \in V^*$. Since only a finite number of such states Δ exist, Algorithm 6.27 must terminate. For each cf-viable prefix θ, S must contain a state Δ_θ on termination such that $CFV_1(\theta) \subseteq \Delta_\theta \subseteq CFLALRV(\theta)$. (To suppose otherwise leads to an immediate contradiction - consider the shortest θ for which no Δ_θ exists). Now $CFV_0(\theta) = CFV_0(\mu)$ implies $\Delta_\theta = \Delta_\mu$ (because they have the same core) and so $\Delta_\theta \supseteq \bigcup_{CFV_0(\mu) = CFV_0(\theta)} CFV_1(\theta)$, that is $\Delta_\theta \supseteq CFLALRV(\theta)$.

Since we also have $\Delta_\theta \subseteq CFLALRV(\theta)$, it follows that $\Delta_\theta = CFLALRV(\theta)$ and the correctness of the algorithm is proved. □

In common with all algorithms for constructing LR(k) - type statesets, the worst-case complexity of Algorithm 6.27 is exponential in the size of the input grammar. This is an inescapable consequence of the fact that the cardinality of such statesets can be exponential in the size of the grammar. (Recall the family of grammars EXP(n) of Section 2.3.) Now in order to build parsing tables, we have to first construct the appropriate stateset and so there is no alternative but to suffer this exponential complexity. But when we wish only to test whether a grammar is in a certain class, we have seen for the CFLR(k) and CFSLR cases that algorithms of only polynomial complexity are available. In particular, we have seen that the CFLR(1) property can be tested in time $O(n^3)$, where n is the size of the grammar, while CFSLR can be decided in time $O(n^2)$. Since the CFLALR property is intermediate between CFSLR and CFLR(1), it might be expected that it too should be decidable in low order polynomial time. Surprisingly, it seems that it cannot. Certainly no such algorithms are known, and material in Hunt et al. (1975) suggests that LALR testing is a very hard problem indeed. An intuitive explanation of this phenomenom is that the CFLALR state CFLALRV(θ) depends, not just on θ itself, but upon all those μ whose CFLR(1) states have the same core as $CFV_1(\theta)$.

Further evidence that CFLALR is a rather complex, ill-behaved property is provided by its behaviour with respect to the post-pass and table conversion algorithms developed (for the CFLR(k) case) in Chapter 4. Remember

that in the CFLR(k) case, the post-pass construction of Section 4.1 generates precisely the same CRLR(k) tables as those constructed normally, while the QCFLR(k) and SQCFLR(k) constructions produce tables which cover the true CFLR(k) tables. Now although the CFSLR method exhibits the same behaviour as CFLR(k) with respect to these constructions, the CFLALR method does not. In general, the chain free parsing tables which can be formed from LALR tables are not the same as CFLALR tables constructed directly. The reason for this in that no result similar to Theorem 3.43 is available in the CFLALR case. That is to say, it is not necessarily true that

$$\text{CFLALRV}(\theta) = \text{CF-STRIP}\left(\{ \text{LALRV}(\mu) \mid \mu \rightarrow_{\ell}^{*} \theta \} \right).$$

In fact, not only may these sets be different to one another, but either may be adequate while the other is not. This may be demonstrated by examples.

Consider the following grammar.

(Grammar G15)

$$\begin{aligned} S &\longrightarrow a\ Z\ b \mid \\ &\quad\ X\ c \\ Z &\longrightarrow X \mid \\ &\quad\ d\ c \\ X &\longrightarrow Y \\ Y &\longrightarrow d \end{aligned}$$

and take $\{Y \rightarrow d\}$ as the chain set. For convenience, denote the set $\text{CF-STRIP}(\{\text{LALRV}(\mu) \mid \mu \rightarrow_{\ell}^{*} \theta\})$ by PPCFLALRV(θ) (for Post Pass CFLALR). Then for (G15, $\{Y \rightarrow d\}$) we have :

$$\text{CFLALRV}(ad) = \{ [X \rightarrow Y.,b],\ [Z \rightarrow d.c,b] \},$$

which is adequate, and

PPCFLALRV(ad) = CF-STRIP({LALRV(ad),LALRV(aY)})

= [X → Y.,b,c], [Z → d.c,b]

which is inadequate.

The reverse situation may be demonstrated using the grammar (G14,{B → b}) which was introduced earlier. Here we have :

CFLALRV(bC) = {[Y → C.,Λ,b],[X → C.,Λ,b]}

which is inadequate, and

PPCFLALRV(bC) = CF-STRIP({LALRV(bC),LALRV(BC)})

= {[Y → C.,Λ], [X → C.,b]}

which is adequate.

Now the post-pass method for constructing CFLR(k) tables depends upon the identity

CF-STRIP (ITEMS(QCFV(θ))) = CFV(θ)

which allows CFLR(k) states to be recovered from QCFLR(k) states. We can certainly define quasi CFLALR states in the same spirit as QCFLR(k) states, that is :

QCFLALRV(θ) = {NAMEOF(LALRV(μ)) | $\mu \xrightarrow{*}_{r} \theta$}

and it is easy to see that

CF-STRIP (ITEMS(QCFLALRV(θ))) = PPCFLALRV(θ).

But, as the examples above clearly demonstrate, PPCFLALRV(θ) does not necessarily equal CFLALRV(θ). Therefore, the post-pass method does not, in general, enable CFLALR tables to be constructed from LALR tables. However, although PPCFLALRV(θ) may not equal CFLALRV(θ), it is a perfectly good approximation to the CFLR(1) state $CFV_1(\theta)$. In fact, this is true of all approximation functions.

THEOREM 6.29

Let $\bar{\mathfrak{m}}$ be an LR(1) approximation function. Then for each $\theta \in V^*$, CF-STRIP($\{ \bar{\mathfrak{m}}(V_1(\mu)) \mid \mu \underset{c}{\rightarrow}^* \theta \}$) is an approximate CFLR(1) state for θ.

PROOF. This result follows straightforwardly from Definition 6.6 and Theorem 3.43. We omit the details. □

Thus, when applied to LALR tables, the post-pass method of Section 4.1 will produce a set of approximate CFLR(1) tables, although not necessarily the CFLALR tables. If the approximate tables so produced are adequate, then they are valid, but their adequacy is not related to that of the true CFLALR tables. For example, the post-pass method will deliver inadequate tables for the CFLALR grammar (G15,{Y → d}) adequate tables for the non - CFLALR grammar (G14,{B → b}).

The algorithms given in Chapter 4 for converting LR(k) tables directly into QCFLR(k) or SQCFLR(k) tables may be adapted straightforwardly to convert LALR tables into QCFLALR and SQCFLALR tables but it can be seen that these will cover, not the true CFLALR tables, but those produced by the post-pass method. (We conjecture that if (G,C) has Property A, then the SQCFLALR tables are the same as those produced by the post-pass method).

We conclude that although the techniques of Chapter 4 do provide methods for producing chain free parsing tables from LALR tables, their theoretical appeal is definitely less in this case than for the CFLR(k) and CFSLR cases.

6.5. Optimising CFLALR Parsing Tables.

We now consider the problem of applying the optimisation technique of Chapter 5 to CFLALR parsing tables. The method of optimising CFLR(k) tables depends upon a result (Theorem 5.8) which shows that whenever q is a parsing state and X and Y are symbols such that $X \xrightarrow{*}_{\varepsilon} Y$, then the value of $g(q,Y)$ is a valid substitute for that of $g(q,X)$. Section 6.2 showed that the same result holds for certain CFSLR tables and so the optimisation method extends straightforwardly to those tables. Unfortunately, no simple result of this type is available in the case of CFLALR tables. We demonstrate this by example. Consider the following grammar.

1	S	⟶	x A x |
2			z A y |
3			x a y |
4			z b z
5	A	⟶	a |
6			b |
7			c

(Grammar G16)

and take C16 = {A → a, A → b} as the chain set. The cs-grammar (G16, C16) is CFLALR and its CFLALR tables are shown in Figure 6.7.

STATE No.	CF-ACTION FUNCTION							CF-GOTO FUNCTION							
	↓	a	b	c	x	y	z	a	b	c	x	y	z	A	S
1					sh		sh				2		3		
2		sh	sh	sh				4	5	6				5	
3		sh	sh	sh				7	8	6				7	
4					sh	sh					9	10			
5					sh						9				
6					7	7									
7						sh						11			
8						sh	sh					11	12		
9	1														
10	3														
11	2														
12	4														

Figure 6.7: The CFLALR Tables for (G16,C16).

If we chose to optimise these tables using the optimising function which has F(A) = a and is the identity elsewhere, then we should substitute the value of g(2,a), that is 4, for that of g(2,A). But this substitution is invalid because the error action f(5,y) is accessible on A - making the substitution will allow the invalid string

xcy to be accepted. More complicated situations can be contrived where the action corresponding to f(5,y) is not even an error. We shall seek a subclass of the CFLALR grammars in which these / unfortunate circumstances do not arise and whose tables can be optimised successfully by the method of Chapter 5.

Since we know that CFSLR parsing tables can be optimised, and that CFLALR tables are only slightly different to CFSLR tables, we shall define a class of grammars which

have SLR-like behaviour in the "locality of symbols involved in chain productions" in the hope that this will provide a subclass of the CFLALR grammars with the properties we desire.

DEFINITION 6.30

When G is a grammar and X ∈ V, we say that G is <u>SLR in the locality of X</u> if SLRV(θX) is adequate for each θ ∈ V^*. When C is a chain set for G, we say that (G,C) is <u>SLR in the locality of all chains to Y</u> if G is SLR in the locality of all X such that $X \xrightarrow[c]{*} Y$. □

Before proceeding further we wish to demonstrate another complication that can arise when attempting to optimise CFLALR tables. In the CFLR(k) and CFSLR cases we know (Lemmas 5.6 and 6.18 respectively) that when X and Y are symbols such that $X \xrightarrow[c]{*} Y$ and p , q and r are parsing states such that q = g(p,X) and r = g(p,Y), then for each u ∈ V_T^{*1} either f(q,u) = f(r,u) or f(q,u) = ERROR. In the CFLALR case, however, this may not be so - as the following example shows:

(Grammar G17)

1	S ⟶	x A x |
2		y A y |
3		x a z
4	A ⟶	B
5	B ⟶	a |
6		b

Grammar G17 is LALR and (G17,{B → a}) is CFLALR. Its CFLALR tables are shown in Figure 6.8.

STATE	CF-ACTION FUNCTION						CF-GOTO FUNCTION							
No.	Λ	a	b	x	y	z	a	b	x	y	z	A	B	S
1				sh	sh				2	3				
2		sh	sh				4	5				6	7	
3		sh	sh				7	5				8	7	
4				4		sh					9			
5				6	6									
6				sh					10					
7				4	4									
8					sh					11				
9	3													
10	1													
11	2													

Figure 6.8 : The CFLALR Tables for (G17, {B → a}).

Optimisation of these tables requires the value of g(2,a), that is 4, to be substituted for the value of g(2,B), that is 7. Now f(7,y) = REDUCE A → B while f(4,y) = ERROR. After substitution, therefore, strings which originally caused the non-error action f(7,y) to be inspected will now cause an error to be declared. It might appear that this corrupts the behaviour of the parser, for valid strings might now be rejected. In fact this is not so. We will show that the original action (i.e. f(7,y)) is never encountered following a shift from state 2 during the parsing of valid strings. Consequently, the substitution of g(2,a) for g(2,B) cannot affect the parsing of valid strings, it merely causes certain invalid strings (for example xby) to be rejected earlier.

We may now prove the main lemma of this section. Notice that this result is similar to Lemma 6.18 except for

the introduction of case (iii) which is needed to cope with the situation illustrated above.

LEMMA 6.31 (cf. Lemmas 5.6 and 6.18)

Let G be an ERFC grammar and let C be a chain set for G such that (G,C) is CFLALR and has Property A. Let $X,Y \in V$ be such that $X \xrightarrow{*}_{c} Y$ and let G be SLR in the locality of all chains to Y. Let $T = (Q,s_0,g,f)$ be the CFLALR tables for (G,C) and let $p,q,r \in Q$ be such that $q = g(p,X)$ and $r = g(p,Y)$. Then for each $u \in V_T^{*1}$ either

(i) $f(q,u) = f(r,u)$ or

(ii) (a) $f(q,u) =$ ERROR and

(b) $u \notin$ FOLLOW(X) or

(iii) (a) $f(r,u) =$ ERROR and

(b) the value of $f(q,u)$ is never inspected following $g(p,X)$ during the parse of any sentence in L(G).

PROOF. If $X = Y$ the result is trivial, so assume that $X \neq Y$. Suppose $f(q,u) \neq f(r,u)$ and that neither has the value ERROR. We must have $p =$ NAMEOF(CFLALRV(θ)) for some $\theta \in V^*$ and so $q =$ NAMEOF(CFLALRV(θX)) and $r =$ NAMEOF(CFLALRV(θY)). We then have $f(q,u) =$ ACTION(CFLALRV(θX),u) and $f(r,u) =$ ACTION(CFLALRV(θY),u). Now from Theorem 6.23 we have

$$\text{CFLALRV}(\theta X) \subseteq \text{CFSLRV}(\theta X) \quad \text{and}$$
$$\text{CFLALRV}(\theta Y) \subseteq \text{CFSLRV}(\theta Y).$$

and from Theorem 6.10 we obtain

$$\text{CFSLRV}(\theta X) \subseteq \text{CFSLRV}(\theta Y).$$

It follows that the value of ACTION(CFSLRV(θY),u) must be that of both $f(q,u)$ and $f(r,u)$ and since, by hypothesis,

these are different, it follows that CFSLRV(θY) is inadequate. Hence, by Lemma 6.14, SLRV(μZ) is inadequate for some $\mu \underset{c}{\rightarrow}^{*} \theta$ and $Z \underset{c}{\rightarrow}^{*} Y$. This contradicts the premise that G is SLR in the locality of all chains to Y and so we conclude that one of f(q,u) and f(r,u) has the value ERROR. We consider the two cases separately.

Case 1 : f(q,u) = ERROR. Suppose that u ∈ FOLLOW(X). Then the same argument may be used as in the proof of Lemma 6.18 to show that SLRV(μZ) is inadequate for some $\mu \underset{c}{\rightarrow}^{*} \theta$ and $Z \underset{c}{\rightarrow}^{*} Y$. (We can use this argument even though G may not be SLR because the requirement that G be SLR in the locality of all chains to Y ensures that G has SLR-like behaviour in all those states which are considered by the proof of Lemma 6.18.) The inadequacy of SLRV(μZ) contradicts the premise that G is SLR in the locality of all chains to Y and so we conclude that $u \notin$ FOLLOW(X) in this case.

Case 2 : f(r,u) = ERROR. Suppose the value of f(q,u) is inspected following g(p,X) during the parse of some sentence x ∈ L(G). Then at the time when f(q,u) is inspected, the parse stack will contain θX and the unconsumed input will be z for some θ and z such that 1:z = u and $S \rightarrow^{*} \theta Xz \rightarrow^{*} x$. Since $X \underset{c}{\rightarrow}^{*} Y$ it follows that θYz is a sentential form of G. Now we must have NAMEOF(CFLALRV(θX)) = q and so NAMEOF(CFLALRV(θY)) = r. But f(r,u) = ERROR and so θYz cannot be a sentential form of G. From this contradiction we conclude that f(q,u) is never inspected under these circumstances. □

Finally we achieve the result which permits optimisation of certain CFLALR tables.

THEOREM 6.32 (cf. Theorems 6.20 and 5.8)

Let G be an ERFC grammar and let C be a chain set for G such that (G,C) is CFLALR and has Property A. Let T = (Q, s_0, g, f) be the CFLALR tables for (G,C) and let $p \in Q$ and $X,Y \in V$. Then provided G is SLR in the locality of all chains to Y, the value of g(p,Y) is a valid substitute for g(p,X) whenever $X \xrightarrow{*}_{c} Y$.

PROOF. Note that because of Theorem 6.23, Lemma 6.16 is true of CFLALR tables as well as CFSLR ones. Lemmas 6.17 and 6.19 are certainly true of CFLALR tables and Lemma 6.31 is very similar to Lemma 6.18. Therefore, the present result follows, just like Theorem 6.20, from the argument used to prove Theorem 5.8. A small amount of extra care is needed in order to take account of possibility (iii) in Lemma 6.31 but the details are obvious. □

For completeness, we restate Theorem 6.32 in terms of the conditions which must be satisfied if CFLALR tables are to be optimised with respect to an optimising function F.

COROLLARY 6.33

Let F be an optimising function for (G,C). Then the CFLALR tables for (G,C) may be optimised with respect to F provided :

(i) (G,C) is CFLALR,

(ii) (G,C) has Property A,

(iii) G is ERFC, and

(iv) G is SLR in the locality of all chains to F(X) whenever $F(X) \neq X$.

PROOF. Optimising the CFLALR tables for (G,C) with respect to F means substituting the value of $g(p,F(X))$ for that of $g(p,X)$ in each parsing state p and for each symbol X ∈ V. The conditions given in the statement of the corollary are just those necessary to ensure that Theorem 6.32 guarantees these substitutions to be valid. □

If we return to the grammar (G16,C16) whose CFLALR tables were shown in Figure 6.7 we see that G16 is SLR in the locality of all chains to b (though not in the locality of all chains to a - SLRV(xa) is inadequate). Thus, although these tables cannot be optimised with respect to the optimising function which has F(A) = a, Corollary 6.33 ensures that they can be optimised with respect to the full optimising function which has F(A)= b and is the identity elsewhere.

Because of the number and the strength of the conditions in its statement, Corollary 6.33 can hardly be considered an elegant result. However, this chapter is mainly concerned with practicalities, not with elegance (although it is a pity that we cannot have both) and Corollary 6.33 seems a reasonable compromise between simplicity and utility. Certainly all of its conditions may be tested easily, and all except (iv) may surely be expected of any civilised LALR grammar. Furthermore, because chain productions mainly occur in well-behaved portions of grammars (such as those which define arithmetic expressions) it is likely that even condition (iv) of the corollary will be satisfied by most programming language

grammars. We conclude that optimised CFLALR parsing tables (OCFLALR tables for short) may be constructed for most LALR programming language grammars and that these tables may be expected to contain fewer states than ordinary LALR tables.

Finally we note that, just as in the CFLR(k) and CFSLR cases, OCFLALR tables may be constructed directly by simply replacing the loop "for each X ∈ V do " in Algorithm 6.27 by one which reads "for each X ∈ R_F do".

6.6. OCFLALR Parsing Tables and the Further Postponement of Error Detection.

Just as with CFSLR tables, useful reductions in the amount of space needed to represent CFLALR tables are made possible by applying further weak postponement of error detection to the tables. When $T = (Q, s_0, g, f)$ are the unoptimised CFLALR tables for (G,C), the weak postponement $(q, u, A \rightarrow \alpha)$ is valid whenever $A \neq S$ and $f(q,u) = \text{ERROR}$. These are exactly the same conditions as those which apply to unoptimised CFSLR tables. Now also as in the CFSLR case, the further postponement of error detection interacts with the substitutions performed when optimising CFLALR tables and so we need results which guarantee the validity of certain combinations of optimisation and further postponement of error detection for the case of CFLALR tables. Throughout this section we suppose that the validity of the OCFLALR tables for (G,C) with respect to the optimising function F is guaranteed by Corollary 6.33. Our first result is exactly similar to one presented earlier for the CFSLR case.

THEOREM 6.34 (cf. Theorem 6.21)

Let $T = (Q, s_0, g, f)$ be the OCFLALR tables for (G,C) with respect to the optimising function F. The weak postponement $(q, u, A \rightarrow \alpha)$ is valid if :

(i) $A \neq S$,

(ii) $f(q,u) = \text{ERROR}$, and

(iii) $u \notin \text{FOLLOW}(F(A))$.

PROOF. Because the ERROR actions in CFLALR tables include all those of the corresponding CFSLR tables, this result may be proved by exactly the same argument as that used to prove the corresponding result for the CFSLR case (Theorem 6.21). □

In OCFSLR tables, weak postponements of the form $(q,u,A \rightarrow \alpha)$ where $u \in FOLLOW(A)$ are pointless because the value of $f(q,u)$ must be REDUCE $A \rightarrow \alpha$ already. This is not the case in OCFLALR tables and so these tables contain larger "dark areas" - areas in which postponements might usefully be performed but where their validity cannot be guaranteed by Theorem 6.34. For example, Figure 6.9 shows the fully optimised CFLALR tables (FOCFLALR tables for short) for $(G17, \{B \rightarrow a\})$. (The unoptimised CFLALR tables for this grammar were shown in Figure 6.8.)

STATE	CF-ACTION FUNCTION						CF-GOTO FUNCTION							
No.	Λ	a	b	x	y	z	a	b	x	y	z	A	B	S
1				sh	sh				2	3				
2		sh	sh				4	5				6	4	
3		sh	sh				7	5				8	7	
4				4		sh					9			
5				6	6									
6				sh					10					
7				4	4									
8					sh					11				
9	3													
10	1													
11	2													

Figure 6.9 : The FOCFLALR Tables for $(G17, \{B \rightarrow a\})$.

The full optimising function for this grammar has F(B) = a and is the identity elsewhere. Figure 6.10 shows the same tables after applying all the further postponements of error detection which Theorem 6.34 guarantees to be valid. (The reductions introduced by the postponement are shown circled in Figure 6.10).

STATE	CF-ACTION FUNCTION						CF-GOTO FUNCTION							
No.	Λ	a	b	x	y	z	a	b	x	y	z	A	B	S
1				sh	sh				2	3				
2		sh	sh				4	5				6	4	
3		sh	sh				7	5				8	7	
4	(4)	(4)	(4)	4		sh					9			
5	(6)	(6)	(6)	6	6									
6				sh					10					
7	(4)	(4)	(4)	4	4	(4)								
8					sh					11				
9	3													
10	1													
11	2													

Figure 6.10: The FOCFLALR Tables for(G17,{B → a})
after Application of a Valid Postponement Set.

It would be nice if we could apply the postponement (4,y, 4) to these tables, but unfortunately production 4 is A → B and y ∈ FOLLOW (F(A)) and so Theorem 6.34 does not ensure the validity of this postponement. However, its validity is ensured by the following result.

THEOREM 6.35

Let $T = (Q, s_0, g, f)$ be the OCFLALR tables for (G,C) with respect to the optimising function F. The weak postponement $(q, u, A \rightarrow \alpha)$ is valid if :

(i) $A \neq S$,
(ii) $f(q,u) = \text{ERROR}$, and
(iii) $u \in \text{FOLLOW}(A)$.

PROOF. We have to show that all three conditions of Definition 6.2 are satisfied. The argument used in the proof of Theorem 6.21 shows that conditions (i), (ii) and (iiia) of that definition are satisfied; we will only consider condition (iiib) here. That is, we must prove that whenever $p \in Q$ is such that $g(p, \alpha) = q$, then $f(g(p,A),u) = \text{ERROR}$. Let $T' = (Q', s_0', g', f')$ be the unoptimised CFLALR tables for (G,C) and suppose that $p \in Q$ is such that $g(p, \alpha) = q$. Then p is also a state in Q' and $g(p,A) = g'(p,F(A))$. Now certainly $f'(g',(p,A),u) = \text{ERROR}$ (for otherwise $f(q,u) \neq \text{ERROR}$) and so, by Lemma 6.31, either $f'(g'(p,F(A)),u) = \text{ERROR}$ also, or $u \notin \text{FOLLOW}(A)$. (Remember $A \overset{*}{\rightarrow} F(A)$.) Therefore, if $u \in \text{FOLLOW}(A)$ we must have $f'(g'(p,F(A)),u) = \text{ERROR}$ and because $g'(p,F(A)) = g(p,A)$ it follows that $f'(g(p,A),u) = f(g(p,A),u) = \text{ERROR}$ and the proof is complete. □

In combination, Theorems 6.34 and 6.35 allow the validity of the postponement $(q,u,A \rightarrow \alpha)$ to be determined whenever $u \notin FOLLOW(F(A))$ or $u \in FOLLOW(A)$. Thus the "dark area" where these results are of no help is given by $DARKAREA(A) = FOLLOW(F(A)) \setminus FOLLOW(A)$ and this is exactly the same as in the CFSLR case. Consequently, the remarks made in Section 6.3 to the effect that few useful postponements are likely to occur in dark areas apply here also and so we are confident that the full benefits of both optimisation and the further postponement of error detection are simultaneously available in the case of CFLAIR parsing tables.

To bring the technical content of this thesis to an end we note that the results of both this and the previous section can be extended without difficulty to apply, not just to the CFLALR method, but to all CFLR(1) approximations which are defined by an approximation function $\bar{\Phi}$ satisfying $\bar{\Phi}(CFV_1(\theta)) \subseteq CFSLRV(\theta)$ for all $\theta \in V^*$. This includes the PPCFLALR method, and also the chain free generalisations of the methods of Korenjak (1969) and Pager (1977).

6.7. Summary

The real aim of this thesis is to develop chain free parsing algorithms for use in compilers. The CFLR(k) parsers are unsuited to this task because the corresponding class of grammars is too restrictive when k = 0, while the parsing tables are too large when k > 0. The ordinary LR(k) parsers suffer from the same problem but modifications to the basic LR(1) method have been developed which yield small, fast parsers for most programming language grammars. In this chapter we have considered similar modifications to the CFLR(1) method.

The fundamental notion employed here is that of "approximate" CFLR(1) parsing tables. These are produced by first introducing extra REDUCE entries into the action function, so that the similarity between different states is increased, and then replacing groups of similar states by a single composite state. The error detection afforded by approximate CFLR(1) tables is slightly inferior to that of proper CFLR(1) tables but is still acceptably good and very much better than that of most other bottom up parsing methods.

Two methods for producing approximate CFLR(1) tables have been studied in detail. We call them the CFSLR and CFLALR methods and, as their names are intended to suggest, they are the natural chain free generalisations of the well known and successful SLR and LALR parsing methods respectively.

It was shown that if G is SLR and C is a chain set for G, then subject to very mild additional conditions, (G,C) is sure to be CFSLR. It seems that CFSLR tables are rather larger than ordinary SLR tables but this disadvantage is easily overcome because the optimisation technique of Chapter 5 extends to CFSLR tables (again, subject to mild conditions). Fully optimised CFSLR tables for programming language grammars are invariably smaller than their SLR counterparts. Considerable economies in the space required to represent SLR and unoptimised CFSLR tables are made possible by allowing selected REDUCE entries to replace certain ERROR entries in the action function. In the case of optimised CFSLR tables, this "further postponement of error detection" interacts with the substitutions performed during the optimisation and can cause the parsing tables to become invalid. A slight restriction on the postponements that may be applied is sufficient to prevent this occurrence while retaining most of the benefit of the technique.

The CFLALR method is more powerful than CFSLR but is more complicated both theoretically and practically. As in the CFSLR case, we are able to show that if G is LALR then, subject to mild conditions, (G,C) is sure to be CFLALR and its CFLALR parsing tables may be optimised and then subjected to the further postponement of error detection. The conditions which ensure that CFLALR tables may be optimised safely are a little less natural than those for the CFSLR case and slightly more difficult to check, but are probably still sufficiently mild to be useful in practice.

Although both CFSLR and CFLALR yield CFLR(0) size parsing tables, the effort required to construct CFLALR tables remains comparable to that for the CFLR(1) case whereas constructing CFSLR tables is only slightly more complicated than constructing CFLR(0) tables. Furthermore, the CFSLR property can be decided in time $O(n^2)$, where n is the size of the grammar, while it seems that the CFLALR property cannot be decided in less than exponential time. The techniques of Chapter 4 may be used to convert SLR parsing tables into (tables which cover) CFSLR ones but they perform rather differently in the LALR case, producing chain free tables which, though valid, are not necessarily related to the proper CFLALR tables.

Both the CFSLR and CFLALR methods yield parsers for programming languages which are suitable for practical exploitation. They are faster than their SLR and LALR counterparts while probably occupying less space. We believe that the advantages of the CFSLR and CFLALR parsers are such that they should substantially replace other methods used in current practice. The CFSLR method is the simpler and in many ways more attractive of the two but the greater generality of the CFLALR method may be an important practical asset.

CHAPTER 7.

CONCLUSION

In this final chapter we wish to summarize the main conclusions to be drawn from our work, to discuss its relationship to previous work, and to suggest topics for future research. The discussion will be rather informal and we hope this will assist the reader who has not studied the technical development of the previous chapters in detail.

The introduction of the CFLR(k) grammars is the foundation of our work. Since, by definition, these form the largest class of grammars which can be chain free parsed from left to right, it follows that no technique for eliminating chain productions from ordinary LR(k) parsers can achieve greater generality than the CFLR(k) chain free parsing algorithm. The secure theoretical basis of our constructions enables a thorough investigation of their properties. In particular, it enables us to prove that if the grammar G is LR(k), then for any chain set C in G, (G,C) is CFLR(k). Observe that no arbitrary restrictions are placed on the value of k, nor on the form of the grammar G, nor on the chain productions that may appear in C. Furthermore, the CFLR(k) parsers retain the excellent error detection of LR(k) parsers and only minor modifications are necessary to cause an LR(k) parser generator to produce CFLR(k) parsers.

A CFLR(k) parser for a typical programming language grammar may be as much as 2½ times faster than a conventional LR(k) parser but unfortunately the CFLR(k) parsing tables will usually be larger than the LR(k) tables. This is mostly due to the presence of redundancy within CFLR(k) parsing tables which may be removed by a simple optimisation. The optimisation removes not only states (i.e. rows) from the parsing tables, but also those symbols (i.e. columns) which appear as the left parts of chain productions.

Application of this optimisation does not impair the quality of the parser in any way and its effectiveness is such that it reduces the size of CFLR(k) tables to below that of their LR(k) counterparts in all except the most extreme of pathological cases.

Our proof of the validity of our optimisation technique requires that the class of grammars considered is restricted to those possessing "Property A". We do not prescribe direct methods of testing for this rather strange property, instead we give a series of easily tested conditions which are sufficient (though not necessary) to ensure the property. These conditions require that the grammar is LR(1) and so they effectively constrain application of the optimisation technique to the case k = 1. Although ungainly, this restriction is unlikely to cause practical difficulties because the case k = 1 is the only one suitable for practical exploitation and only very rarely will an LR(1) grammar fail to satisfy our sufficient conditions for Property A.

Truly practical chain free parsers may be obtained by modifying the CFLR(1) construction to yield what we term " approximate " CFLR(1) parsing tables. Two approximation techniques have been studied in detail; we call them the CFSLR and CFLALR methods. They are the natural chain free generalisations of the highly successful SLR and LALR parsing methods and are applicable to large subsets of the CFLR(1) grammars while yielding only CFLR(0) size parsing tables.

Because of the secure theoretical foundations underlying our constructions, we are able to prove that most of our results concerning CFLR(k) parsers extend to the CFSLR and CFLALR methods. Specifically we are able to prove that if G is SLR then, subject to certain mild conditions,we have :

(i) any chain set C in G may be chosen in the certain knowledge that (G,C) will be CFSLR,

(ii) the CFSLR parsing tables for (G,C) may be optimised, and

(iii) extra REDUCE entries may be introduced into the optimised CFSLR tables for (G,C) in order to permit application of established techniques for representing SLR tables in a highly compact form.

These results ensure that compact CFSLR parsers may be constructed for most SLR programming language grammars. The resulting parsers will be much faster than ordinary SLR parsers and probably smaller too. We suggest the following as a safe and systematic recipe for

constructing a CFSLR parser for a grammar G.

(1) Check that G is SLR. If it is not, then modify it.

(2) Check that G is ERFC (Definition 6.11). If it is not then modify it and return to step 1.

(3) If G contains null nonterminals (nonterminals generating only the empty terminal string) then form the grammar G' of Corollary 4.20 and check that G' is SLR (or LR(1)). If it is not, then modify G and return to step 1.

(4) Select a chain set C in G.((G,C) is sure to have Property A and to be CFSLR).

(5) Construct a (full) optimising function F for (G,C).

(6) Construct the optimised CFSLR tables for (G,C) with respect to F.

(7) Carefully (use Theorem 6.21) apply whatever further postponement of error detection can be usefully exploited by the chosen technique for representing the tables and encode the tables compactly.

The CFLALR method behaves similarly to CFSLR but its greater generality is offset by the complexity of the method for generating its parsing tables and by a slight complication in the conditions which ensure the validity of the optimisation technique in this case. The following recipe will safely deliver a CFLALR parser for the grammar G.

(1) Check that G is LALR. If it is not, then modify it.

(2) Check that G is ERFC (Definition 6.11). If it is not then modify it and return to step 1.

(3) If G contains null nonterminals then construct the grammar G' of Corollary 4.20 and check that G' is LALR (or LR(1)). If it is not, then modify G and return to step 1.

(4) Select a simple chain set C in G (Definition 6.24) ((G,C) is sure to have Property A and to be CFLALR).

(5) Construct an optimising function F for (G,C) and check that the conditions of Corollary 6.33 are satisfied. If they are not, then either modify F and repeat this step, or modify G and return to step 1.

(6) Construct the optimised CFLALR tables for (G,C) with respect to F.

(7) Carefully (use Theorems 6.34 and 6.35) apply whatever further postponement of error detection can be usefully exploited by the chosen technique for representing the tables and encode the tables compactly.

In conclusion, we believe that the techniques developed here provide methods for the automatic construction of parsers for programming languages which surpass all known competitors in all major respects.

7.1. Comparison with Previous Work.

The first published technique for eliminating chain productions from LR(k)-type parsers was that of Anderson (1972) which subsequently appeared, in a more accessible form, in Anderson et al.(1973). Because the research that led to this thesis grew out of an attempt to understand Anderson's techniques more fully, the relationship between his work and our own may be explained most clearly by giving a (somewhat idealized) historical account of our research. We hope the reader will forgive this indulgence (for such it seems) and that he will find it interesting.

Anderson's technique for eliminating chain productions is to modify the method of constructing the statesets from which LR parsers are produced.(We use LR as a blanket term to denote all LR(k) - like methods, including LR(k) itself, SLR and LALR). If Δ is an LR state for some string θ, then the LR state for θX is computed by applying a "GOTO" function to Δ and X : GOTO (Δ,X) will be the LR state for θX.The precise form of the GOTO function depends upon exactly which LR method is being considered but all such functions may be expressed as the composition of two simpler functions : NEXT and CLOSURE. NEXT computes the nucleus of the state for θX by taking all those items from Δ in which the dot precedes an X and moves the dot over the X :

$$\text{NEXT}(\Delta,X) = \{[B \rightarrow \beta_1 X.\beta_2, u] \mid [B \rightarrow \beta_1 .X\beta_2, u] \in \Delta\}.$$

CLOSURE adds to NEXT(Δ,X) all those initial items which are needed to complete the LR state for θX. Anderson's method replaces these conventional NEXT and CLOSURE functions by "chain free" variants :

CF-NEXT and CF-CLOSURE. CF-NEXT differs from NEXT in that it takes all those items from Δ in which the dot precedes a symbol which derives X by a sequence of chain productions and moves the dot over that symbol. Thus

$$\text{CF-NEXT}(\Delta,X) = \{[B \to \beta_1 Y . \beta_2, u] \mid [B \to \beta_1 . Y \beta_2, u] \in \Delta, Y \xrightarrow{*}_{c} X\}.$$

CF-CLOSURE is like ordinary CLOSURE except that it discards all those items which involve chain productions.Anderson's justification of these techniques was based upon informal arguments and the research which led to this thesis was originally undertaken with the primary aim of attempting to prove their correctness rigorously. Anderson showed that certain grammars could be parsed by his modified LR parsers, though not by conventional LR parsers. Conversely, he exhibited an SLR grammar which gave an inadequate stateset when certain chain productions were eliminated. Anderson proved that this latter circumstance could not arise with Λ-free grammars and he conjectured that it could not arise at all in the LR(k) and LALR cases. The secondary aim of our research was to investigate these conjectures and to enquire into the relationship between the grammars which could be parsed by a particular LR method, and those which could be parsed by the same method with chain productions eliminated.

During the course of this research it became clear that little progress was possible so long as Anderson's method was regarded as a modification of conventional LR parsing - it had to be regarded as an independent parsing technique, similar to LR, but not directly reducible to it nor derivable from it. The turning point

in our understanding of Anderson's method came with the realisation that it had a theory which ran parallel to that of LR but from a different starting point. The key to real understanding was to (attempt to) forget the finishing point (i.e. Anderson's method) and to find the starting point and then develop the theory. The search for the starting point caused us to investigate the notion of chain-free parsing and to introduce the CFLR(k) property. Having established this foundation, the development of the theory was conceptually straightforward (though at the time we found it technically difficult). The outcome of this development was the CFLR(k) parsing algorithm and its approximations, CFSLR and CFLALR (which we refer to collectively as CFLR), and the gratifying "discovery" that these are exactly the same as the parsers produced by Anderson's method.

Before proceeding to discuss the work of other authors, we first wish to summarize the attractive features of the basic CFLR methods (and therefore of Anderson's methods also) :

(i) The methods are completely general : to each LR method there corresponds a CFLR method. There is no restriction to only the LR(1) and SLR cases.

(ii) Apart from the requirement that the cs-grammar (G,C) has an adequate CFLR stateset, there is no restriction on either.

(a) the form of the grammar G (Λ-rules and null nonterminals are <u>not</u> excluded), nor

(b) the chain productions in the set C (In particular, chain productions may have terminal right parts and different chain productions may share the same left part.)

(iii) The application of the further postponement of error detection in the interest of achieving economies in the representation of the parsing tables may proceed exactly as in the ordinary LR methods.

The single (but substantial) disadvantage of these methods is that their parsing tables are usually considerably larger than those of the corresponding ordinary LR method.

All published work on the elimination of chain productions from LR parsers which appeared subsequent to that of Anderson has relinquished, to a lesser or (usually) greater extent, the attractive features listed above in order to reverse the growth in table size which the CFLR methods incur.

The technique of Aho and Ullman (1973b) is restricted to the case of LR(1) and SLR parsers and excludes grammars containing null nonterminals and chain productions with terminal right parts. Furthermore, full elimination of chain productions is only guaranteed when no pair of chain productions share the same left part and no techniques for safely applying the further postponement of error detection

are proposed. (Indeed, no mention is made of the fact that the further postponement of error detection can corrupt these parsers). Despite these drawbacks, Aho and Ullman's technique has one outstanding advantage over the CFLR methods - it invariably causes the parsing tables to become smaller when chain productions are eliminated. It was this observation which prompted us to investigate the differences and similarities between the CFLR methods and those of Aho and Ullman. The investigation led to the development of the optimisation technique for CFLR parsers which is presented in Chapter 5. The discovery that this optimisation technique interacted with the further postponement of error detection led to the development of the results presented in Sections 6.3 and 6.6 which are largely successful in resolving the difficulties caused by this interaction.

We contend that the difficulties and restrictions associated with Aho and Ullman's technique are due to its failure to distinguish the issue of eliminating chain productions from that of optimising the resulting parsers. It seems clear from our separate treatment of these issues that the restrictions associated with Aho and Ullman's method stem from the implicit optimisation which it performs, not from the elimination of chain productions itself. (It is interesting to note that whereas we require the ERFC property in order to optimise CFSLR tables, this property is apparently not required by Aho and Ullman's method for eliminating chain productions from SLR tables. However, their Lemma 7 is false unless the grammar is ERFC).

The gains that follow from our separation of these issues are :

(i) The restrictions on chain sets are removed.

(ii) Null nonterminals may be admitted.

(iii) The LALR case can be handled as well as LR(1) and SLR.

(iv) It is guaranteed that certain columns can be deleted from the GOTO table.

(v) The further postponement of error detection is handled satisfactorily.

Unlike the CFLR methods, which build their parsing tables directly, Aho and Ullman's technique is presented as a method for modifying existing LR(1) or SLR parsing tables. Demers (1975) showed (in the LR(1) case) that tables equivalent to those produced by Aho and Ullman's method can also be generated directly. Basically, Demers' method uses a modified NEXT function, which we may call D-NEXT (D for Demers), with the form :

$$\text{D-NEXT}(\Delta, X) = \{[B \to \beta_1 Y . \beta_2, u] \mid [B \to \beta_1 . Y \beta_2, u] \in \Delta, \; X \to_c^* Y \text{ or } Y \to_c^* X\}.$$

Essentially, this function treats alike all those symbols which are related by chain productions in any way. Our technique for optimising CFLR tables is rather similar except that it is a little more careful in the selection of the symbols which are to be treated alike. If we have $X \to_c^* Y$ and $X \to_c^* Z$, then our optimisation technique may treat X the same as <u>one of</u> Y or Z whereas D-NEXT will treat X similarly to <u>both</u> Y and Z. This lack of selectivity causes difficulties which Demers avoids by precluding different chain productions from

having the same left part. In this restricted case, Demers' method is basically equivalent to our method for constructing fully optimised CFLR(1) tables directly.

It was the relationship between Demers' method and that of Aho and Ullman which prompted the investigation into the relationship between CFLR tables constructed directly and those formed by modifying LR tables which is reported in Chapter 4.

Pager (1974) presents a method for eliminating chain productions from LR(1) parsers which is similar to that of Aho and Ullman except that it removes the constraint that no pair of chain productions share the sama left part. The method seems to be equivalent to our method for constructing fully optimised SQCFLR(1) tables.

Backhouse (1976) considers Pager's method from a different viewpoint while Lalonde (1976) presents a method for applying Pager's method during the construction of the parsing tables, rather than subsequently as in the original method. Lalonde's technique seems to be similar to our method for constructing FOCFLR(k) tables directly.

Soisalon - Soininen (1977) presents a method similar to Lalonde's formulation of Pager's method but observes, correctly, that difficulties arise in the LALR case and also when the further postponement of error detection is performed. He then modifies the construction to avoid such difficulties but in doing so may fail to completely

eliminate all references to chain productions. Thus, while Soisalon-Soininen's construction produces correct parsers, it may fail to achieve the full benefit of eliminating chain productions totally. Again, it seems to be a failure to separate the issue of chain elimination from that of optimisation which causes the deficiencies of the technique.

As Soisalon-Soininen (1977) observes, Joliat (1976) merely re-presents an elementary technique of Anderson (1972) for partially eliminating chain productions from LR parsers. This technique appeared in the open literature in Anderson et al.(1973) and is of limited appeal because it eliminates only some references to chain productions.

Clearly, our work is related to that of these earlier investigators and owes much to them. We believe, however, that our approach represents the first rigorous and reasonably complete treatment of the topic.

7.2. Suggestions for Future Research.

We suggest four areas where future research appears profitable. The first concerns our extensive use of "Property A". Because the requirement that (G,C) has Property A is only a very mild restriction in the cases of practical interest (namely, CFLR(1), CFSLR and CFLALR), we have used this property wherever it is convenient to do so. This requirement is a minor irritant from the practical viewpoint and inelegant theoretically. It is interesting to enquire to what extent Property A is necessary to those results which make use of it. We certainly believe Theorem 5.8 to be true in its absence, although Lemma 5.7 is not. By using more sophisticated versions of Theorem 5.3 and Lemma 5.7 we believe it possible to prove Theorem 5.8 given only that G is LR(k). We also believe that Property A can be dispensed with as a separate requirement in all results concerning CFSLR (notably in Theorems 6.15 and 6.20) since we suspect that (G,C) must have Property A if G is SLR and ERFC. If true,† this result might just enable Corollary 6.33 to dispense with Property A also. Because of the behaviour of Grammars G8 and G14, we believe that(something like) Property A really is necessary in Theorems 4.23 and 6.26.

---

† Footnote. This result is true - see Addendum on page 388.

Our second proposal is that it would be interesting to enquire whether our techniques can be applied to the development of chain free versions of other parsing strategies. However, there is little practical motivation for this since other bottom-up methods are not competitive with LR methods, while chain productions are less of an issue with top-down schemes.

The third topic we wish to propose for investigation is a little more speculative and concerns the work of Earley (1975) and Aho et al.(1975). Rather than eliminate chain productions from parsers, these authors propose techniques which avoid their introduction altogether. They suggest using an ambiguous grammar to specify the basic form of the language concerned, coupled with a set of rules which may be used to resolve the ambiguities. For example, the language generated by grammar G3 is more concisely specified by the ambiguous grammar :

$$
\begin{aligned}
S &\longrightarrow E \\
E &\longrightarrow E+E \mid \\
&\qquad\ E*E \mid \\
&\qquad\ (E) \mid \\
&\qquad\ X
\end{aligned}
$$

The ambiguities in this grammar may be resolved by the three rules :

(1) * is left associative,

(2) + is left associative, and

(3) * has higher precedence than +.

A deterministic parser may be constructed from this description by first building an LR parser for the ambiguous grammar (this parser will, of course, be inadequate) and then resolving the conflicts within the parsing tables by using the disambiguating rules. The advantages claimed for this approach are twofold. Firstly, it provides a more flexible and natural method for describing syntax than conventional grammars, and secondly, the resulting parsers are smaller and faster than conventional ones (because they contain no chain productions). We do not dispute that these advantages are worthwhile, but it seems to us that resolving the conflicts of an inadequate parser in this way is something of a "black art" involving ad-hoc manual intervention. Now it turns out that the parsing tables produced by this technique are exactly the same as those which result from first disambiguating the grammar by introducing appropriate chain productions and then constructing an optimised CFLR parser for this amended grammar.

We wish to propose that a more secure method of utilising the descriptive technique advocated by these authors would be to develop ways of converting their descriptions into conventional grammars. Since this approach involves transforming one description of a language into another, it should be easier to establish its correctness than that of a technique which involves modifying a parser for the language. Once a conventional grammar has been obtained, a small, fast parser may be produced using fully optimised CFLR constructions.

Provided the disambiguating rules do not alter the language generated by the ambiguous grammar (sometimes they do - consider the rule " + is not associative") then it seems to us that the correctness of applying optimisation and the further postponement of error detection to these parsers may be guaranteed independently of the usual conditions.

Our final suggestion for future work is to advocate the construction of a parser generator using our techniques and to investigate its performance and that of its parsers empirically. If these parsers perform as well as we predict, then we hope that they may be used in real compilers before too long.

References

Aho A.V., Johnson S.C. and Ullman J.D. (1975)
Deterministic Parsing of Ambiguous Grammars
CACM 18 p.441

Aho A.V. and Ullman J.D. (1972a)
The Theory of Parsing, Translation and
Compiling. Vol. 1 : Parsing
Prentice-Hall

Aho A.V. and Ullman J.D. (1972b)
Optimization of LR(k) Parsers
JCSS 6 p.573

Aho A.V. and Ullman J.D. (1973a)
The Theory of Parsing, Translation and
Compiling. Vol. 2 : Compiling
Prentice-Hall

Aho A.V. and Ullman J.D. (1973b)
A Technique for Speeding Up LR(k) Parsers
SIAM J. Comput. 2 p.106

Anderson T. (1972)
Syntactic Analysis of LR(k) Languages
Ph.D. Thesis, Department of Computing
Science, University of Newcastle
upon Tyne

Anderson T., Eve J. and Horning J.J. (1973)
Efficient LR(1) Parsers
Acta Informatica 2 p.12

Backhouse R.C. (1976)
An Alternative Approach to the Improvement
of LR(k) Parsers
Acta Informatica 6 p.277

Demers A.J. (1975)
Elimination of Single Productions and Merging Nonterminal Symbols of LR(1) Grammars
Computer Languages 1 p.105

DeRemer F.L. (1969)
Practical Translators for LR(k) Languages
Ph.D. Thesis, Department of Electrical Engineering, M.I.T.

DeRemer F.L. (1971)
Simple LR(k) Grammars
CACM 14 p.453

Earley J. (1968)
An Efficient Context-Free Parsing Algorithm
Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon University

Earley J. (1975)
Ambiguity and Precedence in Syntax Description
Acta Informatica 4 p.183

Eve J. (1973)
On the Recovery of LR(1) Parsers from Compacted Parsing Tables
Memo. MRM/68, Computing Laboratory, University of Newcastle upon Tyne

Floyd R.W. (1963)
Syntactic Analysis and Operator Precedence
JACM 10 p.316

Geller M.M. and Harrison M.A. (1973)
Characterizations of LR(0) Languages
IEEE Conf. Record, 14'th Annual Symp. on Switching and Automata Theory p.103

Gray J.N. and Harrison M.A. (1972)
On the Covering and Reduction Problems for Context-Free Grammars
JACM 19 p.675

Harrison M.A and Havel I.M (1973)
Strict Deterministic Grammars
JCSS 7 p.237

Harrison M.A. and Havel I.M. (1974)
On the Parsing of Deterministic Languages
JACM 21 p.525

Hopcroft J.E. and Ullman J.D. (1969)
Formal Languages and their Relation to Automata
Addison-Wesley

Hunt III H.B., Szymanski T.G. and Ullman J.D. (1974)
Operations on Sparse Relations and Efficient Algorithms for Grammar Problems
IEEE Conf. Record, 15'th Annual Symp. on Switching and Automata Theory p.127

Hunt III H.B., Szymanski T.G. and Ullman J.D. (1975)
On the Complexity of LR(k) Testing
CACM 18 p.707

Joliat M.L. (1973)
On the Reduced Matrix Representation of LR(k) Parser Tables
Ph.D. Thesis, Department of Electrical Engineering, University of Toronto

Joliat M.L. (1976)
A Simple Technique for Partial Elimination of Unit Productions from LR(k) Parsers
IEEE Trans. Comp. C-27 p.763

Knuth D.E. (1965)

On the Translation of Languages from Left to Right

Information and Control 8 p.607

Korenjak A.J. (1969)

A Practical Method for Constructing LR(k)

Processors

CACM 12 p.613

Lalonde W.R (1971)

An Efficient LALR Parser Generator

Technical Report CSRG-2, University

of Toronto

Lalonde W.R. (1976)

On Directly Constructing LR(k) Parsers

without Chain Productions

Conf. Record, 3'rd ACM SIGACT-SIGPLAN

Symp. on Principles of Programming

Languages p.127

Mickunas M.D. (1976)

On the Complete Covering Problem for LR(k)

Grammars

JACM 23 p.17

Pager D. (1970)

A Solution to an Open Problem by Knuth

Information and Control 17 p.462

Pager D. (1974)

On Eliminating Unit Productions from LR(k)

Parsers

Lecture Notes in Computer Science 14

(Springer-Verlag) p.242

Pager D. (1977)

A Practical General Method for Constructing LR(k) Parsers

Acta Informatica 7 p.249

Pfleeger C.P. (1973)

State Reduction in Incompletely Specified Finite-State Machines

IEEE Trans. Comp. C-22 p.1099

Purdom P. (1974)

The Size of LALR(1) Parsers

BIT 14 p.326

Salomaa A. (1973)

Formal Languages

Academic Press

Soisalon-Soininen E. (1977)

Elimination of Single Productions from LR Parsers in Conjunction with the use of Default Reductions

Conf. Record, 4'th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages p.183

Valiant L.G. (1975)

General Context-Free Recognition in Less than Cubic Time

JCSS 10 p. 308

Wirth N. and Weber H. (1966)

EULER : A Generalisation of ALGOL, and its Formal Definition

CACM 9 p.13

Wynn P. (1973)

Error Recovery in SLR Parsers

M.Sc. Dissertation, Department of Computing Science, University of Newcastle upon Tyne

ADDENDUM

We present here a useful theorem that was discovered too late to be included in the main body of the thesis.

THEOREM

Let G be ERFC and SLR and let C be a chain set for G. Then (G,C) has Property A.

PROOF. Let α and β be viable prefixes of G and let $\rho \in V^*$ be such that $\alpha \xrightarrow[c]{*} \rho$ and $\beta \xrightarrow[c]{*} \rho$. We need to prove that α and β may differ on only their final symbols. The result is trivial if $\alpha = \beta$ so suppose that $\alpha \neq \beta$ and let θ be the longest common prefix to both α and β. Then we can write α, β and ρ in the form $\alpha = \theta X \delta$, $\beta = \theta Y \delta$ $\rho = \eta Z \mu$ where $X \neq Y$, $X\gamma \xrightarrow[c]{*} Z\mu$ and $Y\delta \xrightarrow[c]{*} Z\mu$. Since G is SLR, it is certainly LR(1) and therefore, by Theorem 4.17, we must have $\mu \xrightarrow{*} \Lambda$. If $\mu = \Lambda$ there is nothing to prove so suppose $\mu \neq \Lambda$. Then there exists $Q \in V_N$ such that $\mu \xrightarrow[R]{*} Q \xrightarrow[R]{} \Lambda$. We now distinguish three cases.

Case 1 : $Y \xrightarrow[c]{*} X$. Because α is a viable prefix there must be some $x \in V_T^*$ such that $S \xrightarrow[R]{*} \alpha x$. Using the relations above, this derivation may be extended to give $S \xrightarrow[R]{*} \alpha x = \theta X \gamma x \xrightarrow[R]{*} \theta X \mu x \xrightarrow[R]{*} \theta X Q x \xrightarrow[R]{} \theta X x$. Thus $(Q \rightarrow \Lambda, \text{len}(\theta X))$ is a handle of $\theta X x$ and so $[Q \rightarrow \Lambda ., u] \in SLRV(\theta X)$ for all $u \in FOLLOW(Q)$. Now if $Y \xrightarrow[c]{*} X$ and $Y \neq X$ there must exist $W \in V$ such that $Y \xrightarrow[c]{*} W \xrightarrow[c]{} X$ and since β is a viable prefix there exists $y \in V_T^*$ such that $S \xrightarrow[R]{*} \beta y$. We then have

$S \xrightarrow[R]{*} \beta y = \theta Y \delta y \xrightarrow[R]{*} \theta Y \mu y \xrightarrow[R]{*} \theta Y Q y \xrightarrow[R]{} \theta Y y \xrightarrow[R]{*} \theta W y \xrightarrow[R]{} \theta X y$ and so $[W \rightarrow X., v] \in SLRV(\theta X)$ for all $v \in FOLLOW(W)$. But note that $S \xrightarrow[R]{*} \theta Y Q y \xrightarrow{*} \theta W Q y$ and therefore, since G is ERFC, $FOLLOW(W) \supseteq FOLLOW(Q)$. Hence, for any $w \in FOLLOW(Q)$ we have both $[Q \rightarrow \Lambda., w]$, $[W \rightarrow X., w] \in SLRV(\theta X)$. These items conflict and so contradict the hypothesis that G is SLR.

Case 2 : $X \xrightarrow{*}_{\tau} Y$. This case is exactly similar to the previous one.

Case 3 : $X \not\xrightarrow{*}_{\tau} Y, Y \not\xrightarrow{*}_{\tau} X$. Here $(\{X,Y\},\{Z\})$ is a maximally chained pair so let R be a maximal intermediate for this pair. Then there exist distinct U,W such that

$$X \xrightarrow{*}_{\tau} U \xrightarrow{}_{\tau} R \quad \text{and}$$
$$Y \xrightarrow{*}_{\tau} W \xrightarrow{}_{\tau} R .$$

Using the argument from Case 1 above, it is easy to see that $[U \rightarrow R., u]$, $[W \rightarrow R., v] \in SLRV(\theta R)$ for all $u \in FOLLOW(U)$ and $v \in FOLLOW(W)$. Also as in case 1, because G is ERFC, we may deduce that $FOLLOW(U) \supseteq FOLLOW(Q)$ and $FOLLOW(W) \supseteq FOLLOW(Q)$ and hence, for any $w \in FOLLOW(Q)$ we have both $[U \rightarrow R., w]$, $[W \rightarrow R., w] \in SLRV(\theta R)$ and since these items are distinct they are in conflict and so contradict the hypothesis that G is SLR.

Thus in all cases the supposition that $\mu \neq \Lambda$ has led to a contradiction and so we conclude that μ (and hence γ and δ) is the empty string and the theorem is proved. □

This result removes the need to demand Property A as a separate requirement in all our theorems concerning

the CFSLR property. This simplifies the statements of Theorems 6.15 and 6.20 and allows the deletion of step (3) of the recipe for constructing CFSLR parsers which was given on page 368.