THE UNIVERSITY OF ALBERTA

LRRL(k) GRAMMARS:

A LEFT TO RIGHT PARSING TECHNIQUE WITH REDUCED LOOKAHEADS

by

RAHMAN NOZOHOOR-FARSHI

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA
FALL 1986

ISBN 0-315-32461-9

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Rahman Nozohoor-Farshi

TITLE OF THESIS: LRRL(k) Grammars: A Left to Right Parsing
Technique with Reduced Lookaheads

DEGREE: Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1986

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

R. Nozohoor-Farshi
....
(Student's signature)

Permanent address:

1950 Cedar Village Cres. #102
North Vancouver, B.C. V7J 3M5

Date: October 10th, 1986

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **LRRL(k) Grammars: A Left to Right Parsing Technique with Reduced Lookaheads** submitted by **Rahman Nozohoor-Farshi** in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

....
Supervisor

Robert
....

....

Matthew
....

....

Date: September 8th, 1986

ABSTRACT

Since the emergence of Marcus' parser, the AI community has shown a growing interest in deterministic parsing of (subsets of) natural languages. However, these parsers are generally described in an informal and intuitive manner, and the grammar for the language is embedded in the code for a parser. LRRL(k) context-free grammars are defined to formalize such parsers.

The parsers for these grammars employ non-terminal as well as terminal symbols in a generalized lookahead strategy. For a fixed value of the parameter k, it is shown that this class of grammars is the largest known class that generalizes the concepts of LR(k) parsing while retaining the decidability of the membership problem of an arbitrary context-free grammar in the class.

The LRRL grammars, when augmented with attributes or features, generate a class of languages that includes the subsets of English which are parsable by a Marcus type parser. Thus introduction of LRRL grammars provides a capability for automatic generation of Marcus style parsers from a context-free underlying grammar plus the information about the feature set, their propagation and matching rules, and a limited number of transformation rules.

LRRL(k) grammars not only have a significant impact on natural language processing, but they also have many interesting applications in design and implementation of programming languages. In particular, they are very useful in production of one-pass compilers and robust error correction methods.

ACKNOWLEGEMENTS

I would like to express my sincere thanks to Dr. Len Schubert for [illegible] me computational linguistics, and for supervising this thesis. His advice and his encouragement have been invaluable to me.

I would like to thank Dr. Paul Sorenson for his helpful suggestions and his careful reading of the thesis.

I also would like to thank the other members of my supervisory committee, Drs. Jeff Pelletier, Matthew Dryer and Robert Reckhow for their interest in my work.

Finally, I would like to express my deepest appreciation to my family. Without their support and understanding, my efforts would not have been equal to this task.

This research has received financial support from the Natural Sciences and Engineering Research Council of Canada Operating Grant A8818.

TABLE OF CONTENTS

Chapter Page

APPENDIX

LIST OF FIGURES

Figure **Page**

LIST OF DIAGRAMS

Diagram Page

LIST OF TABLES

LIST OF DEFINITIONS

Definition **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

Recent years have witnessed a shift in the interest of the AI community in linguistic computation from parsers that capture the *competence* model of language to those that deal with a more practical *performance* model. A notable research area has been deterministic parsing of natural languages, advocated by M.P. Marcus [Marc 75,76,78,80] and enthusiastically explored by others among them R.C. Berwick [BeWe 82, Berw 83], E. Charniak [Char 83], K.W. Church [Chur 80], G.D. Ritchie [Ritc 83] and G. Sampson [Samp 83].

The value of such a linear parsing technique is evident when applied as a 'first attempt' for syntactic recognition in a general language processing system. Initially a deterministic parser can be employed to parse the input sentence and when it becomes apparent that the parsing is beyond the capability of the deterministic parser, the task can be passed to a more powerful but expensive parser. Also, it has been argued that in situations where parsing of erroneous sentences is desired, a deterministic parser is the most appropriate choice [Char 83].

The parser developed by Marcus incorporated some interesting design features but the lack of formalism in describing the mechanism made it difficult to see exactly what class of languages can be parsed by his technique. The parser employs non-terminal as well as terminal symbols as lookaheads to resolve local ambiguities. It combines top-down prediction with bottom-up recognition. The parser has two data structures: a pushdown stack which holds the constructs yet to be completed, and a finite size buffer which holds the lookahead symbols. A window is defined on the buffer and an operation "attention shift" moves the window on the buffer. This allows non-terminals to be parsed, to be put in the buffer and then to act as lookaheads. In fact this technique implements the "wait and see policy" of the Marcus parser, that is, the analysis of a constituent may be delayed until some neighbouring constructs are parsed.

The grammar is given in the form of packets of rules. Each rule consists of a pattern and an action. The packets of rules are activated and deactivated as the parse of the input sentence proceeds. The patterns in the active packets are compared against the top node of the stack and the content of the buffer. Upon a match, the corresponding action rule is invoked, resulting in a new state of the parser.

As mentioned, in this type of parser, the grammar for the language is embedded in the rules which direct the parsing. Viewed in another way, the rules are a set of procedures written in the very high level language **PIDGIN**, that can be interpreted by **PARSIFAL**.

From a theoretical point of view, such a parsing method suffers from at least three shortcomings:

- The language accepted by the parser is not well defined. Marcus relies on examples to show what kinds of sentences can be parsed by his parser.

- Given an arbitrary language, it is not obvious how to determine whether the method can be used to parse the language and how to build a parser.

- Verification of the correctness and the completeness of these parsers is unmanageable. A close scrutiny of the Marcus parser reveals that there is a context-free phrase structure grammar hidden in the packets of rules. The grammar is also augmented with features or attributes as they are better known in formal languages. Furthermore, some limited transformational capability is added to the grammar. Therefore the parsing strategy, in general, may be viewed as a modification of the Floyd-Evans Productions [Floy 61, AhUl 72a], in the sense that it provides a specification of a parsing algorithm for a certain language with a finite state control influencing decisions.

Despite this connection, it remains difficult to write parsers for arbitrary languages in this fashion. Moreover, any proof of the correctness of these parsers should rely on some sort of program verification methods.

This thesis formally defines a family of classes of context-free grammars; denoted by **LRRL** grammars (left to right parsable grammars with reduced lookaheads), and provides automatic generation of (precompiled) table-driven parsers for deterministic parsing of these grammars. LRRL grammars when augmented with attributes will generate a class of languages that includes the subsets of English which are parsable by a Marcus type parser.

The generator accepts a context-free grammar possibly augmented with attributes as input and produces a parsing table if the grammar satisfies certain conditions. Apart from attributes, the syntactic recognition of such languages as exemplified by the Marcus parser requires more parsing power than that the family of LR(k) parsers provide. Consequently, some generalization of LR(k) parsers is called for. The generalized parser takes a more complex right context or lookahead into consideration when deciding to reduce a phrase.

During this research, it was noted that Berwick [Berw 81] believed that the Marcus parser is some generalization of LR(k) parsers. In 1983, Berwick [Berw 83] independently observed that the stripped down version of the

Marcus parser 'may be formally characterized by Szymanski's LR(k,t) grammars [Szym 73, SzWi 76]. It will be shown that Berwick's conjecture in principle is untrue.'

One may pause here to observe that a generalization of LR(k) grammars implicitly indicates a bottom-up parsing method, but the Marcus parser is not strictly a bottom-up parser. In fact it proceeds in a top-down manner and when need arises it continues in a bottom-up fashion. However, the use of top-down prediction in such a parser does not affect its basic bottom-up completion of constructs. In fact the inclusion of grammars accepted by Marcus type parsers in the more general class of LRRL(k) grammars is analogous to the inclusion of LL(k) grammars in the class of LR(k) grammars. And, a subclass of LRRL grammars that can be handled in a mixed strategy similar to that of the Marcus parser will be defined.

Although the introduction of LRRL grammars is influenced by the Marcus parser, this research is not confined to the formalization of that parser. In the course of this thesis, theoretical aspects of these grammars will be investigated. As a method of parsing synthetic and

' Recently Berwick [Berw 84] remarked that Williams' bounded context parsable BCP(m,n) grammars [Will 75, SzWi 76] are adequate to formalize the Marcus parser. However, it is trivial to show that BCP grammars are unsuitable for such a purpose. A BCP-parser ignores the information obtainable from the left context (except the last m symbols). Whereas in the Marcus parser, the use of that information is the compelling reason for deployment of the packeting mechanism. In fact there are numerous simple LR(k) grammars that are not BCP, but are parsed by the Marcus parser.

natural languages, their impact on the general theory of language design / implementation will be discussed. The application of the method to such areas of programming languages as compiler construction, resolution of overloaded symbols in polymorphic languages and error correction (repair) will be demonstrated.

As for computational linguistics, this research introduces a formal method for automatic generation of powerful deterministic parsers that supersede the Marcus parser. It should be emphasised that no attempt is made here to build a new parser for natural language processing, in which, by adding ad-hoc devices one tries to capture as many linguistic facts and principles as possible. Rather, this research provides a tool to investigate what subsets of a natural language can be parsed in this particular deterministic fashion. Of course, such a subset will be defined by a context-free base grammar that is augmented with attributes. As case studies, Marcus parsing and Gazdar-style context-free parsing [Gazd 82] could be investigated.

1.2 An overview of the thesis

Chapter 2 examines the objectives of a Marcus-type parser in employing complex lookaheads. It is shown that these objectives and indeed Marcus' parser cannot be described in terms of the LR technique or existing generalizations of it. Some approaches are also rejected because of inherently undecidable problems in them.

The remainder of the thesis can be considered as two parts. Chapters 3, 4 and 5 are concerned solely with context-free grammars. Chapter 6 deals with the context-sensitivity of languages in the form of attributed grammars. Chapter 3 describes the development of the simplest class of grammars in the LRRL family, i.e., basic type I grammars. The definition of these grammars closely follows the objectives of the Marcus parser in a purely bottom-up parsing technique. Also in this chapter, basic type II grammars which are slightly more general than the previous class are described.

Chapter 4 is devoted to the properties of the basic LRRL(k) grammars and the implications of LRRL parsing. The idea behind the LRRL grammars is presented in a more abstract form in this chapter. A polynomial time algorithm for LRRL(k) testing is also developed in Chapter 4.

Chapter 5 considers the generalizations of basic LRRL(k) grammars. It is shown that the GLRRL grammars, i.e., the most general class of grammars in the LRRL family, with a suitable parameter, properly include LR(k,t) grammars. Thus, this thesis proposes the largest known class of context-free grammars that generalizes the concepts of LR(k) parsing while retaining the property that for a fixed parameter the membership problem in the class is still decidable. Subclasses of the LRRL grammars such as Marcus parsable grammars, and also LRRL grammars with ϵ-rules are discussed in this chapter.

The order in the presentation of the context-free LRRL grammars roughly follows the 'almost standard' way of introducing grammar classes found in [AhUl 72a], i.e., development, implications and generalization. However, it also reflects the history of the progress of this research, starting with rather simple grammars and intuitive methods and leading to more abstraction in Chapter 4 and generalizations in Chapter 5. Probably, if one has started with the abstract recursive definition which is now given in Chapter 4, then the report would have been shorter.

Chapter 6 examines some strategies for parsing attributed LRRL grammars. The chapter ends with the applications of the wait and see policy in the presence of attributes to such areas as overload resolution and repairing of syntactic errors.

Chapter 7 concludes the thesis by outlining the contributions of the research and suggestions for future study.

Finally, Appendix I formally proves that the string set accepted by Marcus' parser amounts to a context-free language. This provides a justification for assuming a context-free underlying grammar in automatic generation of

Marcus-type parsers.

CHAPTER 2

BACKGROUND

In this chapter, the reason for having complex lookaheads in deterministic parsing of some languages, including deterministic subsets of natural languages, is discussed. The Marcus parser will be used to illustrate the point. Section 2.2 will describe Marcus' parser in some detail. Then, the evolution of more formal research relevant to this topic, i.e., existing methods of generalizing LR(k) parsers will be studied in Section 2.3. Finally in Section 2.4, the inappropriateness of the cover theory as a practical means for replacing the lookahead strategy in this kind of parsing will be discussed.

2.1 Complex lookaheads in the Marcus parser

Marcus [Marc 80], in describing his determinism hypothesis, rightly pointed out that most of the natural language parsers operate by simulating non-deterministic machines, either by using backtracking or by pseudo-parallelism. However, he claimed that there are subsets of English excluding garden paths that can be parsed in a deterministic way with limited backtracking that uses lookaheads of a more complex nature than mere lexical words.

To illustrate the point, consider one of his early examples:

Sentence 1: *Have the students who missed the exam take the make up today.*

Sentence 2: *Have the students who missed the exam taken the make up today?*

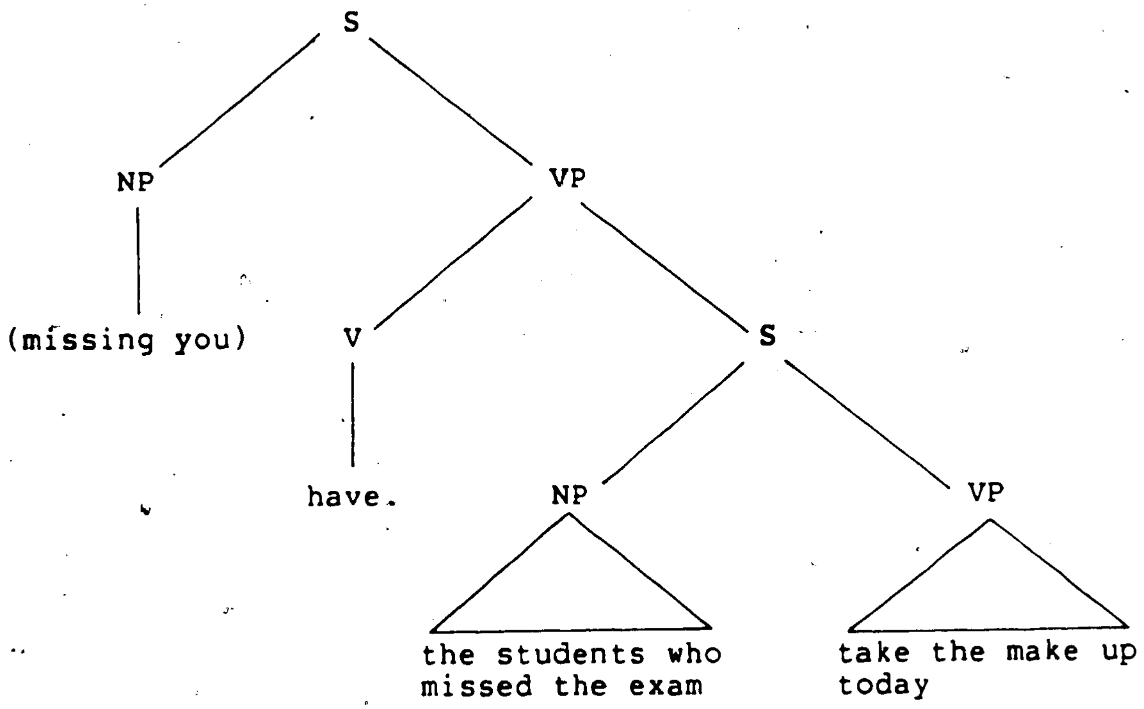


Fig. 2.1
Parse tree of imperative sentence

In the sentence 1 "have" is the main verb of an imperative sentence, while in the sentence 2 it serves as an auxiliary of the main verb "taken". The structure of the imperative sentence may be given by the parse tree in Figure 2.1, while the structure of the question sentence is shown by Figure 2.2.

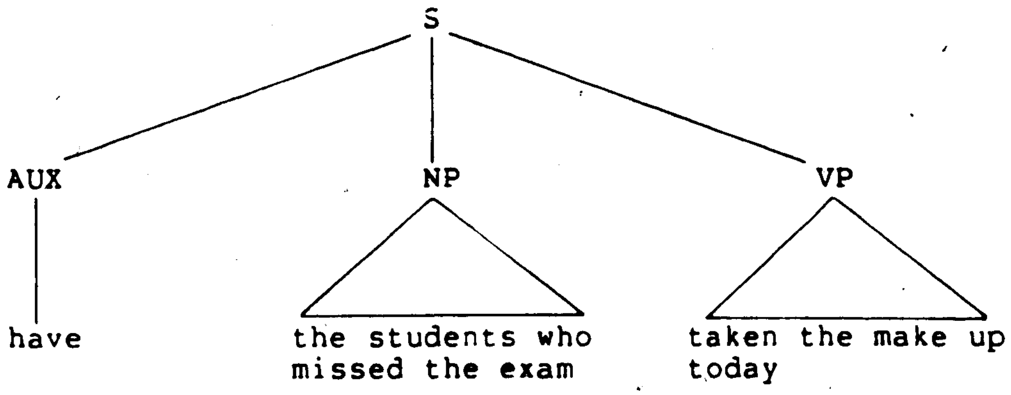


Fig. 2.2
Parse tree of yes/no question sentence

The first seven words of both sentences are identical, and on the surface it seems necessary to use some sort of non-deterministic process to analyze the sentences, since the first clue, i.e., the form of "take", could appear arbitrarily far into the sentence. However if the analysis of "have" is delayed until the intervening noun phrase is parsed, then the reduced phrases "[$_{NP}$ the students who missed the exam] take" and "[$_{NP}$ the students who missed the exam] taken", acting as lookaheads, can uniquely determine the role of "have". It may be observed that no traditional deterministic left to right parser such as LR(k) could handle this situation, since lookahead phrases in general can be arbitrary long strings of terminals.

An example similar to Marcus' "wait and see policy" was also given by Aho and Ullman [AhUl 72a,p486], where they considered the grammar G:

$$S \rightarrow 0\ A\ B\ b \mid 0\ a\ B\ c$$

$$A \rightarrow a$$

$$B \rightarrow B\ 1 \mid 1$$

L(G) is the regular set $0a1^*(b+c)$, but G is not LR. However, one can parse G bottom-up if one defers the decision of whether *a* is a phrase in a sentential form until the last input symbol is scanned. That is to say an input string of the form $Oa1^n$ can be reduced to *OaB* independently of whether it is followed by *b* or *c*. In the former case , *OaBb* is first reduced to *OABb* and then to *S*. In the latter case, *OaBc* is reduced directly to *S*. Of course, they do not give any characterization of such grammars or any formal algorithm for parsing them.

Situations like the above example normally do not occur in context-free description of programming languages. Language designers are usually careful to come up with constructs that can be described in a context-free grammar which is parsable by well known methods such as LR(1), though one might cite at least two occurrences in the context-free syntax of Ada [Ledg 81] where lookaheads of more than one symbol are required [Weth 81]. However, if a compiler designer wants his parser to achieve more than just parsing of context-free syntax of correct programs, then methods like LR(k) are hardly adequate for his goal.

The following three examples, related to error correction and context-sensitive syntax, illustrate the point.

Example I [error correction]:

A very common error in Pascal, made particularly by people who switch back and forth from other languages, is that they leave out the parentheses around relational expressions. Thus, one often sees the following incorrect usage:

IF A=B OR C=D THEN ...

The problem is due to Pascal's unconventional operator precedence levels. In particular, relational operators are applied last. Therefore *A=B OR C* is parsed as *A=[B OR C]*. Further, relational operators do not associate, so *A=B=C* is syntactically incorrect as is *A=B OR C=D* which parses as *A=[B OR C]=D*.

It is very desirable to have a Pascal parser that corrects such a common mistake. But any LR-based error corrector parser (using insertion/deletion recovery methods) either will delete *=D*, or will insert a *THEN* after *C* resulting in a cascade of new errors. In any case if *B* or *C* is not boolean, an additional confusing type error also will be generated in the semantic routines. Some compilers, e.g., Berkeley's Pascal, try to replace the second = with a +, with no success at repairing the error.

Error productions can be added to the Pascal specification to handle this case satisfactorily. However, the resulting grammar cannot be parsed with an LR(k) parser. When a parser that can postpone some reductions reaches *OR*, it will have the option of either reducing [*A*=*B*] as a relational expression, or shifting *OR* as part of expression[*B* *OR* ...]. If the reduction [*A*=*B*] is deferred, then the parser will be able to resolve the conflict upon seeing =*D*, and in this case the statement will be satisfactorily repaired. Notice that *C* could be a compound expression or a qualified variable, so no LR-based parser can handle this situation.

Some LR-based parsers in Pascal compilers add error productions that associate relational operators. However their capability is limited to signaling an error. By the time they have discovered the error it is too late to undo the parsing, but they can continue with an incorrect parse.

Example II [context-sensitive syntax] :

Consider the type declaration section in Pascal programs. Niklaus Wirth, in reporting Pascal [JeWi 74], never suggested that the declaration of a type used in defining another type must lexically precede the latter. If such a position is taken by compilers, it must be considered as a modification to the Pascal definition by complier writers, specially the first few implementors of the language. An honest implementation of a language should follow the language definition precisely, which here would mean that any type declaration in the current block is visible in the whole block [WeSH 77]. Almost all Pascal implementations deviate from this original rule, and recently so with the blessing of 'standardization' [Ledg 84]! But, when it comes to the 'chicken and egg' problem of declaring a pair of a dynamic data type and a pointer type to it , they do violate their own modified rule. In declaring:

```
Type
   Pointer = ↑ Node;
   Node = Record
            Field1 : Type1;
                 . . .
               [illegible] : Pointer
        end;
```

they allow [illegible] referencing. In Ada, the issue of forward referencing is adequately addressed by means of incomplete definitions [Ichb 79]. But one may assume that Pascal, although it was designed and meant to be one-pass compilable, allowed forward referencing in type declarations without introducing additional notation [WeSH 77]. Actually, Wirth could have taken the same approach as Ada, as he does so in the case of forward procedures. The incomplete declaration of procedures were needed to make Pascal one-pass compilable. Anyhow, the above exception allowed in so-called 'standard Pascal' alone suffices in the following argument.

Considering the segment of a Pascal program depicted in Figure 2.3, one realizes that any one-pass compiler would be at a loss at the statement (1); i.e., in deciding whether "*Node*" is the type defined in the procedure "*P*" or the one declared at the next statement in the procedure "*Q*". Very few compilers simplistically take the wrong choice (i.e., the type in "*P*") which would lead to a bizarre error later on in the program. But most others come to the correct decision by a special symbol table mechanism, or by very ad-hoc means in the semantic action routines. The use of ad-hoc and expensive routines, in which the parser does not play a part, has prohibited the extension of forward referencing to other constructed data types. Notice that programmers favouring a top-down design approach would have welcomed such an extension.

Some people incorrectly regard the above scoping problem as a semantic one. Regardless of being syntax or semantics, what really is needed is that the syntax-directed compiler should sense the potential local ambiguity in statement (1), and defer its analysis until the remaining part of the declaration section is fully parsed. Issues such as this one may be viewed as due to faulty language design. But, they can also be viewed as due to a deficiency of current parsers, or more accurately, due to inadequacies in the combination of grammars, parser generators and parsing algorithms that are used in construction of compilers, in dealing with situations like this.

```
...
Procedure P;
  Type
    Node = Record
             Field1 : real;
             Field2 : integer
           end;
  ...
  Procedure Q;
    Type
      Pointer = ↑Node;                              (1)
      Node = Record
               Field : boolean;
               Link : Pointer
             end;
    ...
  end; (* of proc Q *)
  ...
end; (* of proc  P *)
...
```

Fig. 2.3
Fragment of Pascal program

Another example of the "wait and see" policy may be observed in the realm of FORTRAN or some sort of block structured FORTRAN. Consider the situation depicted in Figure 2.4. Assuming that there is no label declaration, but only label definitions, there is an ambiguity in the statement *m goto n* as to which statement *n* refers to. The resolution does not come about till the end of outer block.

```
Block 1 |
        |
        |
        |  ...
        | n ...
        |  ...
        | Block 2 |
        |         |  ...
        |         |  ...
        |         | m goto n
        |         |  ...
        |         | n ...
        |         |  ...
        |  ...
        |  ...
```

Fig. 2.4
Forward referencing goto

Any one-pass compiler, or any language editor will need a parser with a capability of deferring the analysis of the role of constructs such as "goto n" in order to process the program texts smoothly.

In fact a one-pass compiler for any language with the 'goto' feature, needs to postpone the code generation for forward referencing 'goto' statements. Currently available compilers solve the problem by ad-hoc backpatching methods that involve expensive bookkeeping. In a one-pass compiler that uses a parser with deferral capability such deferrals can be handled automatically by the parser at no extra cost, provided that the referring statements do not lie in deeply nested constructs.

The last example almost touched on semantic analysis, but not quite so since the discussion of code generation and similar activities only concerned the interfaces between context-free parsing and semantic routines. The following example will clarify the role of such interfaces.

Example III [semantic directed parsing]:

In computational linguistics, the notion of semantic parsing is apt to suggest the works of Roger Schank and his associates, in which, loosely speaking, syntactic analysis has a lesser role (if any) in language processing. However, semantic directed parsing in programming languages has a different implication; namely, the context-free parsing process is allowed to be influenced by semantic analysis, formally, by the means of attributes that may convey totally semantic information.

Returning to natural language, a favourite example of ambiguous sentences among computational linguists is the following:

I saw a man with a pair of binoculars.

The above sentence is not only syntactically but also semantically ambiguous. The phrase "with a pair of binoculars" can be taken as a prepositional phrase that modifies the NP "a man" or the VP "saw a man" as illustrated in the figures 2.5 and 2.6.

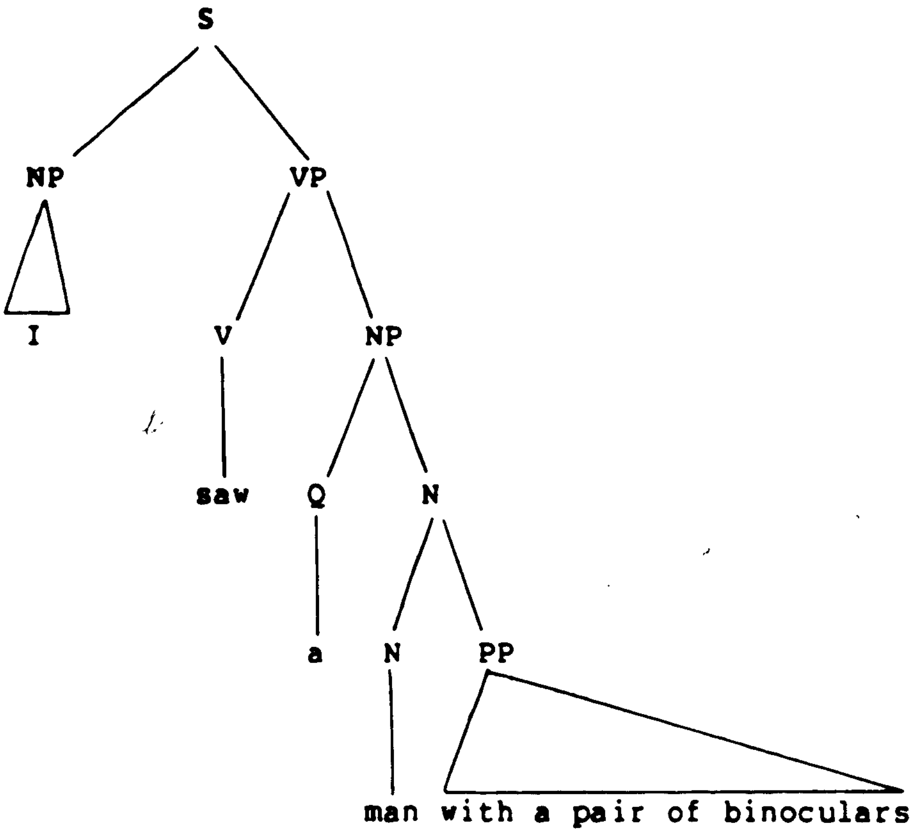


Fig. 2.5
Attaching PP to a noun

However there are examples of such constructs that their semantics make them unambiguous, e.g.,

1. I saw a bird with a pair of binoculars.

2. I saw a bird with a pair of ugly legs.

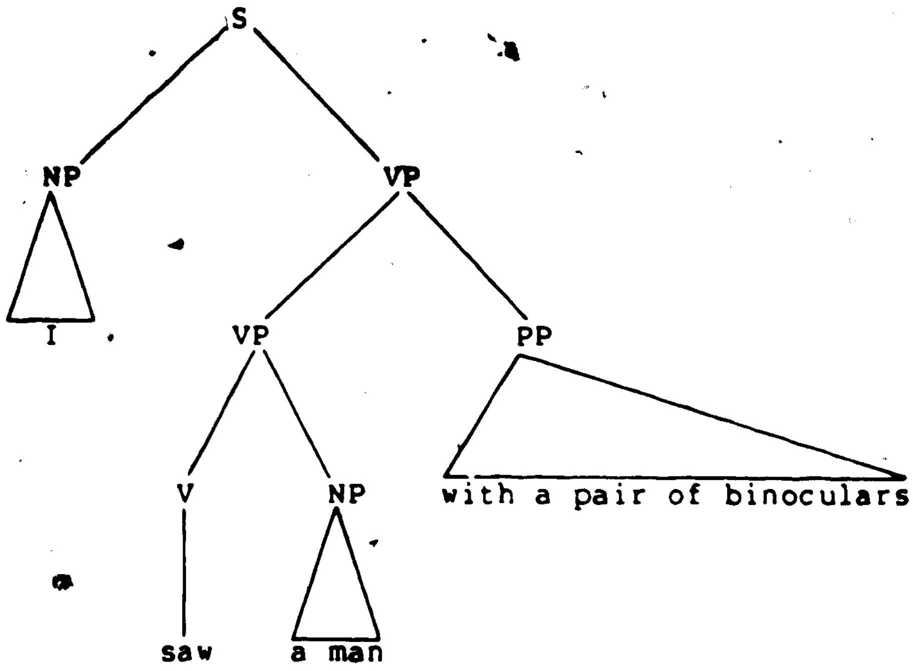


Fig. 2.6
Attaching PP to VP

The context-free grammar describing these sentences is ambiguous. An LR-type parser would have a conflict at the word "with" whether to reduce the rule: NP →Q N, or to continue with parsing of the rule: N →N PP by shifting the word "with". On the other hand a Marcus-type parser could delay the decision until the PP is completely parsed. Then upon having done some semantic analysis, the parser can decide whether the PP is a part of VP or a part of N. Such a process can be achieved by means of some form of feature or attribute propagation and compatibility testing.

Deterministic parsing of ambiguous grammars [AhJU 75] in which priorities are assigned to some rules, may be viewed as a very restrictive case of attribute directed parsing. It is assumed that priorities are constant attributes that are associated with the productions. A prime example of this technique is the use of LR(1) parser in the Algol 60 and Pascal's "dangling else" problem. The syntax of Pascal describes the "if statement" as:

If-stat → IF Condition THEN Statement
If-stat → IF Condition THEN Statement ELSE Statement.

To enforce the association of the *ELSE Statment* with the second *IF* in the following statement:

IF Condition THEN IF Condition THEN Statement ELSE Statement

i.e., to disambiguate the otherwise ambiguous grammar (equivalently right association in natural language), a higher priority is assigned to the second rule or the shift operation in the relevant state of the LR(1) parser.

Notice that this sort of capability of LR(1) parsers is very limited and cannot be used in the previous example since the prepositional phrase can be arbitrarily long. As a matter of interest, the "dangling else" problem can be solved by the use of "open" and "close" statements in an unambiguous LR(1) grammar without relying on priorities.

A proposal related to the use of priorities is the work of Shieber in natural language parsing [Shie 83]. In designing a parser based on LALR(1) parser, he uses priori syntactic preferences, e.g., right association and minimal attachment, to disambiguate sentences by a shift-reduce parsing technique. For example in the sentence:

Joe took the book that I bought for Mary.

his parser would attach the PP *for Mary* to the VP that dominates the verb *bought* rather than associating it with the first verb. However, Schubert [Schu 84] shows many problems with priori syntactic preferences, which indicates that such a scheme, by itself, cannot be used to obtain a correct analysis of every sentence. Schubert argues in favour of a system of syntactic, semantic and pragmatic preferences trade-offs in a multiple-path parser. In a single-path parser based on the LR(1) parsers, such interactions between syntactic and semantic components cannot take place unless they only deal with the information relating to the left context of a decision point. In Marcus' parsing or in LRRL parsing, this sort of interactions could involve the information about the right context of an ambivalent point. Thus LRRL parsing achieves the goal of preferences trade-offs in a limited form by delaying parsing decisions in a single-path parser.

Semantic directed or attribute directed parsing with LRRL grammars will be discussed in Chapter 6. The advantages of such a technique in resolving overloaded functions and operators in Ada-like languages will be illustrated.

2.2 Marcus Parsing

In this section, more details and a working example of the Marcus parser will be discussed. Then, from the point of view of the theory of parsing, some important results obtained from the study of the Marcus parser will be outlined.

2.2.1 Operation of the Marcus parser

The data structures used in the Marcus parser were outlined in section 1.1. The following describes the operation of the parser and shows an example grammar written in PIDGIN.

The Marcus parser has three basic operations:

(1) **Attach:** attaches a constituent in the buffer to the current active node (stack top). It is analogous to the shift operation in a shift-reduce parser.

(2) **Create (push):** creates a new active node, i.e., when the parser decides that the first constituent(s) in the buffer begin a new higher constituent, a new node of the specified type is created and pushed on the stack. However the create operation has a second mode in which the newly created node is first attached to the old active node, and then pushed on

the stack. Marcus indicates this by use of "attach a new node of 'type' to active node" in the grammar rules. Following Ritchie [Ritc 83], a shorter notation 'cattach' is used here for this second mode. One might see the create operation analogous to the predict operation in a shift-reduce parser.

(3) **Drop (pop)**: pops the top node of the stack (CAN). However if this node is not attached to a higher level node, it will be dropped in the first position of the window defined on the buffer.

Marcus uses different notations, namely "drop" and "drop into buffer", in the grammar to indicate the effect of drop operations. This suggests that a grammar writer must be aware of the attachment of the current active node. This thesis adheres to his provision about differentiating between these two modes of drop operations. However, it seems that there is no need for such a provision since PARSIFAL (the grammar interpreter) can take care of that by inserting an unattached node into the buffer, and the grammar can test the contents of the buffer to see if such insertion has taken place. The drop operation is similar to a reduction operation in other parsers.

The three basic operations plus "**attention shift**" and "**restore buffer**" (forward and backward window movements on the buffer) are sufficient for parsing some context-free grammars. In addition to these, the Marcus parser has

"label" operations that assign features to the nodes. Features will be dealt with in Chapter 6. Attention shifts are discussed in a detailed manner in Appendix I.

Now consider the context-free grammar G_1:

$S' \to S$

$S \to d$

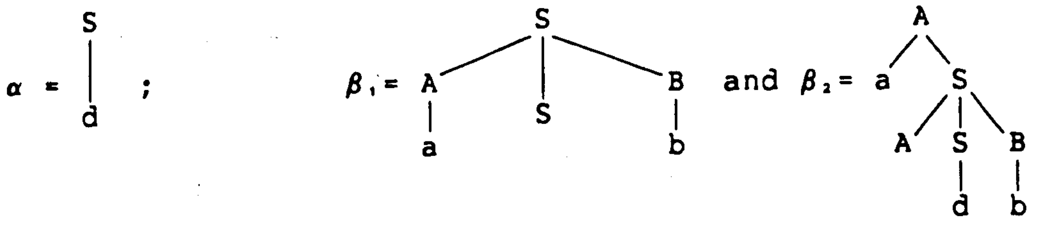
$S \to A\ S\ B$

$A \to a$

$A \to a\ S$

$B \to b$

In terms of Joshi's tree adjoining grammars [Josh 81], the language generated by G_1 can be characterized by the TAG $G_2=(I,A)$ where the set of initial trees $I=\{\alpha\}$ and the set of auxiliary trees $A=\{\beta_1,\beta_2\}$ and



In tree adjoining grammars a leaf nonterminal node in a tree can be replaced by a subtree having the same nonterminal as its root and no nonterminal leaf (e.g., the S-leaf in β_1 above can be replaced by α). An interior nonterminal can be replaced by a subtree having the same symbol as root and the unique nonterminal leaf (e.g., the A-node in β_1 above can be replaced by β_2; in this replacement the a-daughter of the A-node in β_1 becomes the daughter of the A-leaf in β_2). The set of trees with no nonterminal nodes represent sentences generated by the grammar. Such a representation may sometimes show more clearly the context of each construct.

The following gives a Marcus-style parser for $L(G_1)$, i.e., a grammar G_3 written in a PIDGIN-like language that can be interpreted by PARSIFAL. The symbols inside square brackets refer to the contents of buffer positions, except [CAN=] which indicates the current active node. The grammar has no attention shift rules.

G_3:

Packet 1 : Initial rule.

[a or d] create S'; activate 2.

Packet 2 : Create and attach an S node.

[true] deactivate 2; cattach S; activate 3 and 6.

Packet 3 : S-parsing.

[d] attach first; deactivate 3; activate 7.

[a] cattach A; activate 4.

[Sb] attach first; deactivate 3; cattach B;
activate 5.

Packet 4 : A-parsing.

[a] attach first; create S; activate 3.

[Sb] drop CAN.

[Sa or Sd] attach first; drop CAN; deactivate 3;
activate 2.

Packet 5 : B-parsing.

[b] attach first; drop CAN; activate 7.

Packet 6 : Completion of an attached S node.

[true] drop CAN; activate 8.

(with priority $p_1 <$ default priority)

Packet 7 : Completion of an unattached S node.

[true] drop CAN into buffer.

(with priority $p_2 < p_1$)

Packet 8 : B-prediction.

[CAN=S] [b] deactivate 8; cattach B; activate 5.

[CAN=S′] [empty] "Parse is finished".

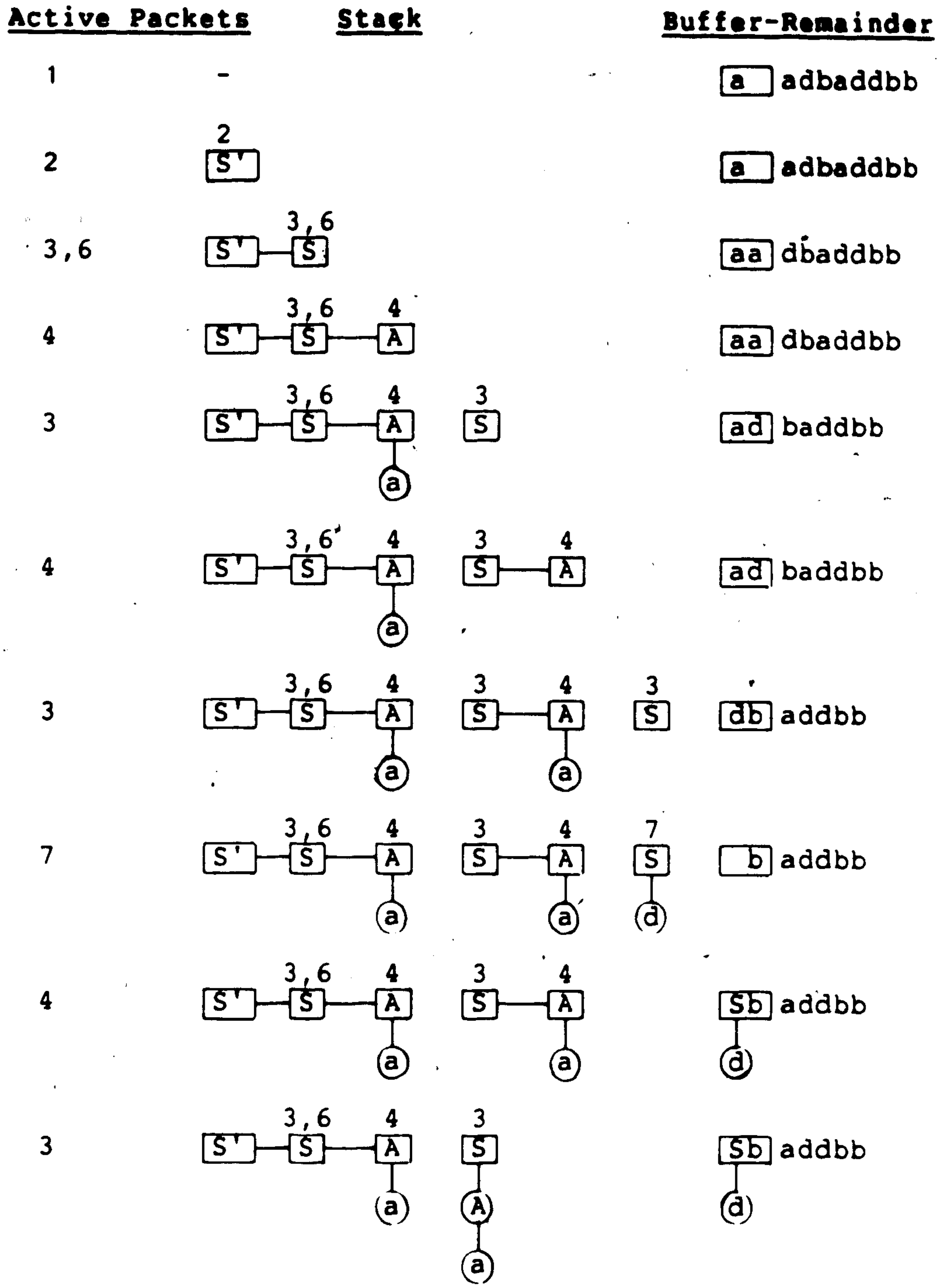
In the Marcus parser active packets are associated with the active node, that is, when a new node is created, some packets will usually be activated as well. When the node is no longer on top of the stack these packets become inactive. Unless a packet is deactivated explicitly this association remains with the node. So when a node on the stack becomes the active node again as a result of 'pop' operations, its associated packets will be reactivated. However there are also explicit operations for deactivation and reactivation of packets (by means of 'deactivate' and 'activate' commands). When a number of packets are active, the pattern segments of the rules in these packets are compared with the current active node and contents of the virtual buffer (the window). Then the action segment of a rule with the highest priority that matches is executed. The parse fails when no rule in the active packets can be matched (or in certain cases when no packet remains active).

No attempt will be made here to show formally the equivalence of G_1 and G_2, since there is no formal characterization of Marcus-style parsers yet. However one may, by going through examples, convince oneself that the parser given in PIDGIN parses $L(G_1)$. Such an example is illustrated in detail at end of this section. At this stage, any proof of the context-free grammar parsed by G_2 must rely on some form of program verification techniques. The proof will be specific to this example grammar.

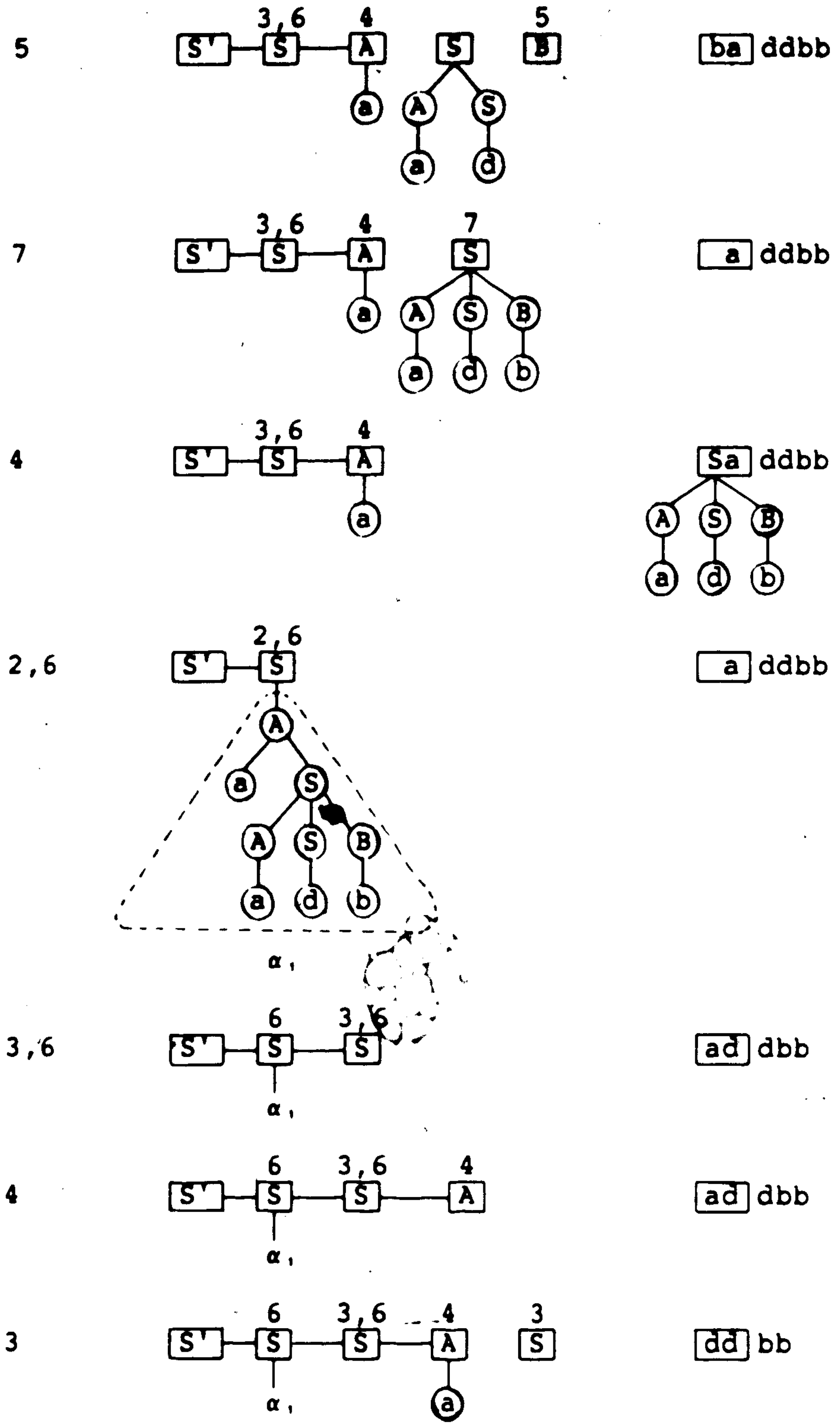
The position taken in this thesis is that given a grammar in PIDGIN (perhaps with some example sentences as Marcus does), one can hardly determine exactly what language is parsed by the parser. What one needs is an exact definition of the underlying base context-free phrase structure grammar, plus a precise definition of the feature set (including feature propagation and matching rules) and transformational rules. In the example grammar, without knowledge of the original context-free grammar G_1, the language described by G_3 is almost anybody's guess.

Example:

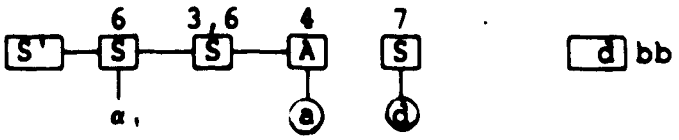
The diagram 2.1 illustrates the parsing of the sentence *aadbaddbb* $\in$ $L(G_1)$ by the parser described by G_3. The symbols inside the boxes are on the stack, and those inside the circles are already attached to a higher level symbol on the stack. The numbers above each stack node are the packets associated with that node. G_3 uses a buffer of size 2 shown on the right.



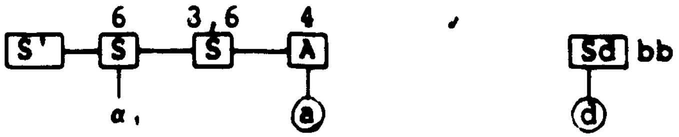
Diag. 2.1
Marcus parsing of the sentence aadbaddbb


5
3,6
4
5
S'
S
A
S
B
a
A
S
a
d
ba ddbb
7
3,6
4
7
S'
S
A
S
a
A
S
B
a
d
b
a ddbb
4
3,6
4
S'
S
A
a
Sa ddbb
A
S
B
a
d
b
2,6
2,6
S'
S
A
a
S
A
S
B
a
d
b
α₁
a ddbb
3,6
6
3,6
S'
S
S
α₁
ad dbb
4
6
3,6
4
S'
S
S
A
α₁
ad dbb
3
6
3,6
4
3
S'
S
S
A
S
α₁
a
dd bb

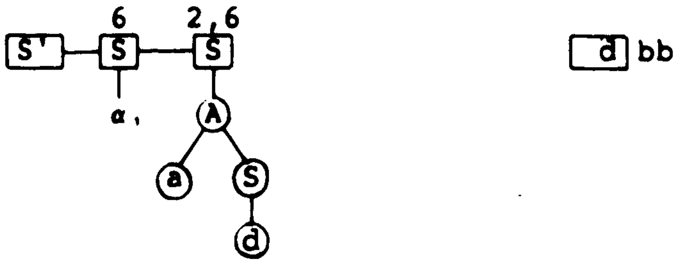
7



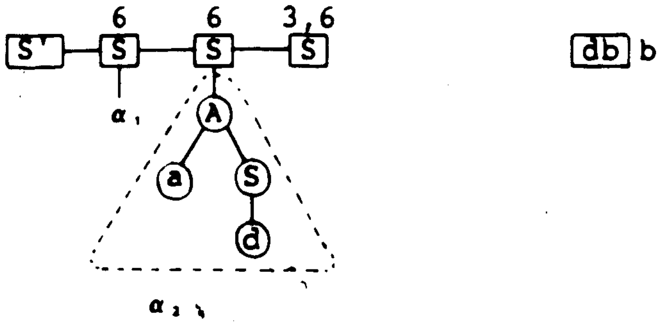
4



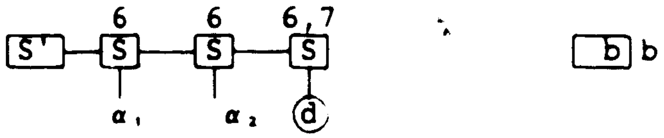
2,6



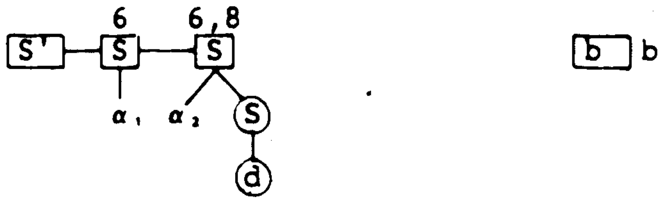
3,6



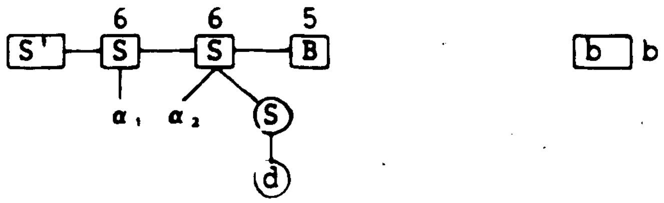
6,7



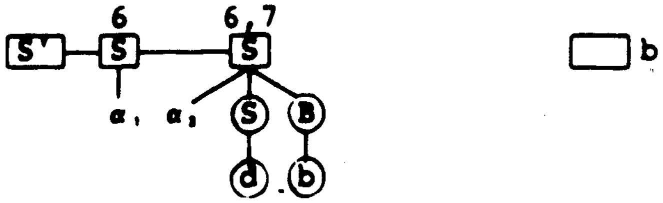
6,8



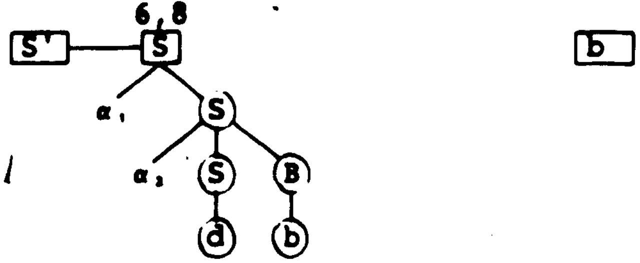
5



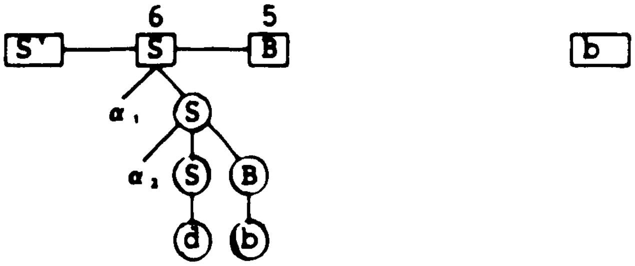
6,7



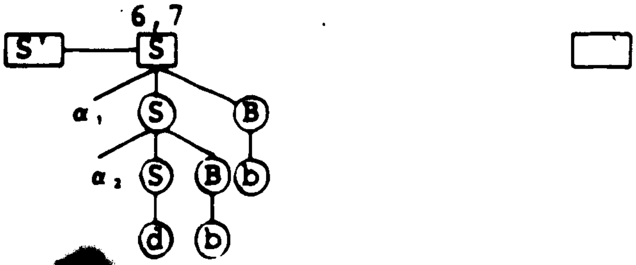
6,8



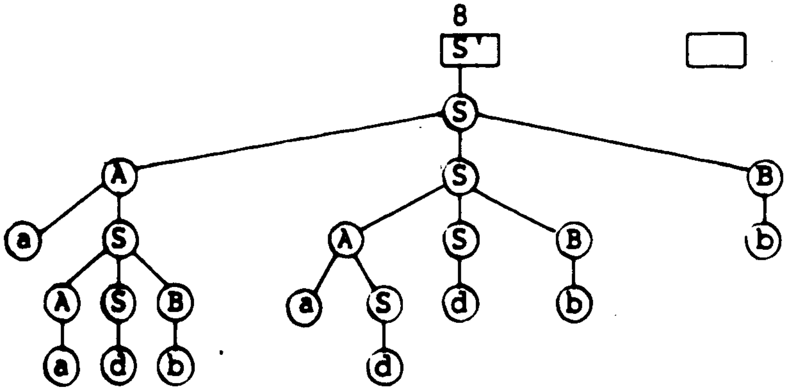
5



6,7



8



"Parse is finished."

This example shows the power of the Marcus parser in employing completed subtrees as lookaheads. The grammar G, is not LR(k) for any fixed k. Any a can be reduced to an A via production $A \rightarrow a$ or can be considered as a first symbol in production $A \rightarrow aS$ (i.e., a reduce/shift conflict in LR parser). For example, in the sentence $a^n db^n$, the initial n-1 a's must be reduced to A's, while this is not the case in the sentence $a^n (db)^n$. However, in the first case a is followed by an Sb, and in the latter by an Sd or an Sa. By postponing the parsing decisions about the completion of A's, Marcus' parser is able to produce the correct parse.

This seemingly unbounded (arbitrarily large) amount of lookahead in terms of terminal symbols in a deterministic parser has confused some people; e.g., G.R. Sampson [Samp 83] was led to make an incorrect analogy to flexible binoculars. What Marcus fails to assert clearly is this: though an arbitrary number of terminals may be consumed before deciding about the first undecided point in parsing, the whole terminal lookahead string is deterministically reduced in a recursive manner to a limited number of nonterminals. Also, some of the confusion comes from the fact that Marcus heavily relies on intuition to justify the deterministic nature of the language that his parser handles. Appendix I confirms the determinism of the language parsed by this parser in a formal way.

It will be shown later that the grammar G_1 is also not LR(k,t) [Szym 73].

2.2.2 Some conclusions about the Marcus parser

From the perspective of formal parsing theory, the following observations about the Marcus parser may be highlighted. A plus sign indicates a positive point, and a minus a negative attribute. Unmarked statements are points of theoretical interest that reflect neither favourably nor unfavourably on Marcus' parser.

(1) + Marcus establishes, perhaps for the first time, that more complex lookaheads, including nonterminals, have a practical use in parsing.

(2) + By using the above strategy he has been able to show that a significant subset of English and probably other natural languages can be parsed in a deterministic fashion.

(3) Marcus made a remark that the accounts of psychological and linguistic phenomena described in his thesis were not his original objective. The parser was originally constructed to demonstrate a computational point about the parsability of natural language. However, he later realized that the behaviour of his parser describes such phenomena. More specifically, Marcus asserts that his determinism hypothesis is consistent with the psychological claim that 'all sentences which people can parse without conscious

difficulty can be parsed strictly deterministically'. He argues that only those sentences that violate many of the constraints that Chomsky introduced in the mid-seventies, cannot be parsed by his method, e.g., garden path sentences. The reader should be reminded that there is a controversy among computational linguists (e.g. see [Bris 83]) as to the validity of this claim. The research reported here is not concerned with such psycholinguistic problems. Nevertheless, the basis it provides for automatic generation of Marcus-type parsers should be of interest to people attempting to verify Marcus' claims.

(4) The Marcus parser does not handle such constructs as comparatives or conjunctions. No explanation as to the difficulty of parsing these constructs is given.

(5) Marcus' parser operates in linear time, but no formal analysis is provided for this complexity.

(6) - Marcus' parser is partially top-down and it cannot handle some LR(k) grammars. For example, consider the LR(0) grammar G:

$S \rightarrow A$ $\quad$ $A \rightarrow cA$ $\quad$ $A \rightarrow a$

$S \rightarrow B$ $\quad$ $B \rightarrow cB$ $\quad$ $B \rightarrow b$

with any finite buffer, Marcus' parser will be flooded with *C*'s, before it can decide to put an *A* node or a *B* node on the stack. Therefore, being partially top-down greatly curtails the parsing power of the Marcus parser. If constructed purely bottom-up, it would have enjoyed broader coverage. In the Marcus parser incomplete nodes are put on the stack, while in a bottom-up parser completed nodes that seek a parent reside on the stack. No explanation is given for this choice in the Marcus parser. However, one may argue that due to the right branching nature of English constructs this is a more economical choice in terms of memory usage. But surely this is not a universal rule. A second reason may be that in this scheme some features can be propagated easily from dominating nodes, such as the last cyclic node, down to lower constructs.

(7) - As mentioned earlier total lack of formalism entails several serious disadvantages among them:

- It is not obvious exactly what language (which subset of English) is described by the outlined grammar, because the grammar itself is in fact the parsing algorithm. This kind of procedural approach to a language theory also found in the works of some other AI researchers such as Winograd [Wino 72] makes obtaining any clear conclusion about the theory impossible.

- The deterministic behaviour of the parser, which is explained in an intuitive manner, is not quite convincing. (e.g., see Brisco [Bris 83] and Sampson [Samp 83].)

- The lack of formalism makes it very hard to determine whether a given language can be parsed in this style and if so, how to construct a parser.

- The absence of formalism prohibits any analysis as to the correctness and completeness or the complexity of the parser to be carried out.

- Without a formalism, no parser of this type can be automatically generated. Anyone that wishes to write such a parser or modify an existing one must do so manually, with considerable time consumption. The writer must be aware of all the working details of the parser and must consider all the rule packets in order to modify a single rule. The advantages of an automatic parser generator are beyond any doubt. Any student in an introductory graduate level compiler course that uses a YACC-like system [John 75] for parser generation in a substantial compiler project would acknowledge that he has got a good working compiler without a great deal knowledge of the parsing theory. One of the goals of this research is to show that with a good formalism, Marcus-style parsers can in fact be generated automatically.

Having demonstrated that the LR(k) parsers are inadequate in handling certain languages like the one in Marcus parsing, the next two sections will examine two alternatives. Section 2.3 investigates the existing generalizations of the LR(k) methods, in particular the LR(k,t) method [Szym 73] which was suggested by Berwick [Berw 83] as a way of formalizing the Marcus parser. Section 2.4 will discuss the cover theory.

2.3 Previous generalizations of LR(k) grammars

Since the introduction of the two prominent classes of grammars with deterministic parsers, i.e., LL(k) and LR(k), according to a bibliography compiled by A. Nijholt [Nijh 83], over a thousand valuable research reports relating to the area have appeared in the literature in the past fifteen years. Due to the exponential nature of the number of states in an LR(k) parser and the weakness of LL(k) parsers, many researchers have attempted to fill in the gap between these two classes. One such class is the left corner grammars introduced by Rosenkrantz and Lewis [RoLe 70]. However, much effort has been put into optimization of LR(k) parsers, such as the works of Aho and Ullman [AhUl 72b, 73b], Demers [Deme 75], Pager [Page 77a, 77b] and Soisalon-Soininen [Sois 80, 82]. Also some research in sparse relations [HuSU 77] has contributed to the efficiency of LR(k) parser construction. Spector [Spec 81] and Ancona [AnDG 82] have discussed the efficient construction of LR(k) parsers with partial expansion of lookahead strings.

Such optimizations in terms of space and speed of construction, combined with the availability of cheap large memories, not only made the LR(k) parsers the most popular parsers, but also suggest that parsers for a more general class than LR(k), with a reasonable amount of overhead in terms of the number of states, are feasible and practical.

Several authors have attempted to extend the concepts of LR(k) parsing to grammars other than LR(k). Madsen [MaKr 76], Lalonde [Lalo 77, 79], Celentano [Cele 81] and Chapman [Chap 84] have developed LR-based parsers for extended context-free or regular right part grammars. These are the grammars that use regular expressions in the right hand side of productions. The extended LR(k) grammars, which are very similar to syntax charts, often provide a clearer and more natural way to describe the syntax of programming languages than pure BNF notation. But the power of these grammars is no greater than that of LR(k) grammars since there is a straightforward transformation of them into LR(k) grammars.

There are two classes of context-free grammars, which are the focus of discussion herein, namely LR-regular and LR(k,t) grammars (with the inclusion of LR(k,∞) and FSPA(k) grammars) that truly generalize the concepts of LR(k) parsing. The parsers for these grammars, in deciding to reduce a questionable phrase, much like parsers for LR(k) grammars, consider the whole left context. But unlike them, their right context lookahead is not limited to k terminal symbols.

The following subsections discuss these classes in detail.

2.3.1 LR-regular grammars

LR-regular grammars were defined by K. Čulik and R. Cohen [CuCo 73]. Basically, in LR-regular grammars arbitrarily long terminal lookahead is allowed before making a parsing decision, provided that the lookahead information can be represented by a finite number of regular sets. To illustrate the intuition behind such grammars, one may consider a fragment of the grammar given in [CuCo 73]. In a programming language which has set data types in addition to simple types, the syntax of a relation in an "IF relation THEN..." statement can be given as:

```
"relation"        → "arith exp" = "arith exp" |
                      "set exp" ≡ "set exp"
"arith exp"       → "arith exp" + "arith term" |
                      "arith exp" - "arith term" |
                      "arith term"
"arith term"      → "arith term" * "arith primary" |
                      "arith primary"
"arith primary"   → ( "arith exp" ) | identifier | constant
"set exp"         → "set exp" + "set term" | "set term"
```

```
"set term"        → "set term" * "set factor" | "set factor"
"set factor"      → "set factor" - "set primary" |
                        "set primary"
"set primary"     → ( "set exp" ) | identifier | constant .
```

There is a local ambiguity in parsing of relations, that is an identifier starting a relation can be reduced to either "set primary" or "arith primary". The problem can be stated in a better way in the more recent terminology of polymorphic languages, that is to say '+', '-' and '*' functions and possibly identifiers are overloaded. The only thing that differentiates set relations from arithmetic relations is the relation signs '=' and '≡'. Therefore the lookahead information can be given by two regular sets $(T-\{=\})^*\equiv T^*$ and $(T-\{\equiv\})^*=T^*$, where T is the set of terminal symbols. More formally, the LR-regular grammars can be defined in the following way.

Definition 1:

Let $\pi=\{R_1,\ldots,R_n\}$ denote a partition of T^* into a finite number of disjoint sets R_i. π is called a regular partition of T^* if all sets R_i are regular. If two strings x and y belong to the same set R_i then $x\equiv y(\mathrm{mod}\ \pi)$.

Definition 2:

Let $\pi=\{R_1,\ldots,R_n\}$ be any regular partition of T^*. A context-free grammar G=(N,T,P,S) is called LR(π) if given any two rightmost derivations of the form

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha_1 A_1 y_1 \underset{rm}{\Longrightarrow} \alpha_1 \gamma y_1$$

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha_2 A_2 y_3 \underset{rm}{\Longrightarrow} \alpha_1 \gamma y_2$$

where y_1, y_2 and y_3 are terminal strings and $y_1 \equiv y_2 (\mathrm{mod}\ \pi)$ then it may be concluded that $A_1 = A_2$, $\alpha_1 = \alpha_2$ and $y_2 = y_3$.

Definition 3:

A context-free grammar G is called LR-regular if G is LR(π) for some regular partition π of T^*.

Clearly every LR(k) grammar is LR-regular with respect to the regular partition $\pi_k = \{ \{u_1\}, \ldots, \{u_n\}, w_1 T^*, \ldots, w_m T^* \}$ where w_i, $1 \le i \le m$, are all the terminal strings of length k, and u_i, $1 \le i \le n$, are all the terminal strings of length less than k, including the empty string. It can be shown that the inclusion of LR(k) grammars in LR-regular grammars is a proper one.

The parsing algorithm for a LR(π) grammar essentially involves a right to left prescan of the input sentence that decides at any input symbol which regular set R_i the remainder of input belongs to.

The criterion for LR-regular grammars can effectively be reduced to the regular separability of two deterministic context-free languages as stated below.

Regular separability: Two languages L_1 and L_2 are said to be regularly separable if and only if there exists a regular set R such that $L_1 \subset R$ and $L_2 \cap R = \emptyset$.

The regular separability of two deterministic languages and thus the membership problem of an arbitrary context-free grammar in the class of LR-regular grammars, remained as open question at the time when the LR-regular grammars were defined. Later it was shown that the problem is undecidable.

2.3.2 LR(k,t), LR(k,∞) and FSPA(k) grammars

LR(k,t) grammars were originally proposed by D. Knuth in his landmark paper on parsing of LR(k) grammars [Knut 65], and later developed by T. Szymanski [Szym 73,SzWi 76]. Essentially the LR(k,t) parsing technique is a non-canonical extension of the LR(k) technique, in which instead of the reduction of the handle (the leftmost phrase) of a right sentential form, it is required that at least one of the t (a fixed number) leftmost phrases in any sentential form can be reduced. In other words, a grammar G is not LR(k,t) if in parsing of an input sentence the decision about reduction of t or more questionable phrases in a sentential form needs to be delayed. The reduction decision is reached by examining the whole left context and k symbols to the right of a phrase in a sentential form. This technique lends itself to construction of a finite number of sets of k right bounded regular parsing patterns and thus a finite state-automaton can be used in parsing of such grammars.

The LR(k,∞) grammars [Szym 73] may be viewed as the ultimate generalization of LR(k) grammars, in which it is required that at least one phrase in any sentential form can be reduced. Unfortunately the resulting parsing patterns in this technique, in general, are non-regular context-free sets and thus no finite automaton may be used to guide the parsing of such grammars.

The FSPA(k) (or FPRAP(k)) grammars [Szym 73,SzWi 76] are those LR(k,∞) grammars that yield regular parsing patterns that are k bounded on the right. Thus such grammars have a finite state parsing automaton which uses k symbols of lookahead.

In order to give a more formal algorithm for parsing of LR(k,t) grammars, and thus a formal definition of them, the following preliminaries are needed.

Definition 1: Phrase language

Let $G=(N,T,P,S)$ be a context-free grammar and let $\hat{N}=\{\hat{X}|X \in N\}$ and $B=\{\ ]_i\ |1\le i\le|P|\}$ be sets of new symbols. The phrase language of G, PL(G) is defined by the grammar:

$\hat{G}=(\hat{N},N\cup T\cup B,\hat{P},\hat{S})$ where

$\hat{P}=\{\hat{X}\to X_1,\ldots,X_{n_i}\ ]_i\ |X\to X_1,\ldots,X_{n_i}$ is the i th production of P}

$\{\hat{X}\to Y_1,\ldots,Y_{n_i}$ [illegible] $X\to$ [illegible] $.,X_{n_i}$ is the i th production of P, $Y_j=X_j$ or $\hat{X}_j$ for $1\le j\le n_i$ and at least one $Y_j=\hat{X}_j\}$.

Definition 2: Bracket erasing mapping

Bracket erasing mapping $m\colon (N\cup T\cup B)^* \to (N\cup T)^*$ is defined by

$$m(\alpha) = \begin{cases} Xm\ \text{[illegible]} & \text{if } \alpha=X\gamma \text{ and } X\in N\cup T \\ m\ \text{[illegible]} & \text{if } \alpha=X\gamma \text{ and } X\in B \\ \epsilon & \text{if } \alpha=\epsilon \end{cases}$$

Note that $m(PL(G)\cup\{S\})=SF(G)$, i.e., the set of sentential forms of G.

The LR(k,t) parsing algorithm which uses a stack and a bounded buffer of size kt is based on finding phrases in sentential forms. The phrase finding algorithm in turn is based on computation of valid LR(k,t) items as explained below.

Definition 3: LR(k,t) item

An LR(k,t) item is an ordered pair (γ,h) such that $\gamma\in(N\cup T\cup B)^*$ and contains at least k occurrences of symbols in $N\cup T$ and h is an integer such that $0\le h\le t$.

Definition 4: Valid LR(k,t) item for a string

An item $(\beta_1 Y_1 \ldots \beta_k Y_k \gamma, h)$ is a valid LR(k,t) item for the string $X_1 \ldots X_n Y_1 \ldots Y_k$ (where $\beta_i \in B^*$ and X_j, Y_i are elements of N∪T) iff there exist $\phi \in (N \cup T)^*$, $\alpha_1, \ldots, \alpha_n \in B^*$ such that

1) $\alpha_1 X_1 \ldots \alpha_n X_n \beta_1 Y_1 \ldots \beta_k Y_k \gamma\phi \in PL(G)$.
2) $|\alpha_1 \ldots \alpha_n| = h$.
3) $\gamma\phi$ contains no complete phrases.
4) γ contains at most kt symbols to the right of the rightmost bracket.

It can be shown that the set of all valid LR(k,t) items for a given grammar G and fixed values of k and t is a finite set.

Algorithm 1: Computation of $V_{k,t}(X_1 \ldots X_n Y_1 \ldots Y_k)$ the set of valid items for string $X_1 \ldots X_n Y_1 \ldots Y_k$.

It is assumed that input sentences for the parser will be appended by $\$^k$ on each side.

1) If $h=0$ set $V_1 = \{(\$^{k+1} S]_0 \$^k, 0)\}$ otherwise $V_1 = V_{k,t}(X_1 \ldots X_n Y_1 \ldots Y_{k-1})$.

2) Set $V_2 = \{(\beta_1 Z_1 \ldots \beta_k Z_k \gamma, h+|\alpha|) \mid$ there exists α: $(\alpha X_n \beta_1 Z_1 \ldots \beta_k Z_k \gamma, h) \in V_1\}$.

3) Repeat

add $(\beta_1 Z_1 \ldots \beta_k \alpha_i]_i \gamma', h)$ to V_2 such that

$(\beta_1 Z_1 \ldots \beta_k Z_k \gamma, h) \in V_2$

and $Z_k \to \alpha_i$ is the i th production in P

and γ'=first kt symbols of $m(\gamma)$.

Until no more items can be added.

4) Set $V_{k,t}(X_1 \ldots X_n Y_1 \ldots Y_k) = \{(\beta_1 Z_1 \ldots \beta_k Z_k \gamma, h) \in V_2 \mid Z_1 = Y_1, \ldots, Z_k = Y_k\}$.

5) If any item ends in a $\$$ but has fewer than kt symbols of right context append $\$$'s to the item until there are either k $\$$'s or kt symbols in the right context after the rightmost bracket.

Algorithm 2: phrase finding for LR(k,t) grammars.

Let the input string be $\Phi = X_1 \ldots X_m \in \$^k (N \cup T)^* \$^k$ (i.e., a sentential form)

1) Set n=0.

2) Compute $S = V_{k,t}(X_1 \ldots X_{n+k})$.

3) If $S = \emptyset$, then halt because $\Phi \notin SF(G)$.

4) If all items in S share $]_i$ as a common initial symbol then halt, indicating that reduction of the i-th production should be applied at position n of Φ.

5) If some item in S contains no brackets, then halt, G is not LR(k,t).

6) If some item (α, h) in S has $h \geq t$ then halt, G is not

LR(k,t).

7) If n+k=m, then halt, G is not LR(k,t).

8) Set n=n+1 and go to step 2.

It can be shown that the number of valid item sets is finite. One may observe that a deterministic finite automaton can be constructed in which each state represents a valid item set. The final states correspond to those sets which indicate a reduction. The constructed automaton takes a sentential form as an input and finds a reducible phrase in it. In fact Szymanski [Szym 73] uses such an automaton in the informal development of LR(k,t) parser and remarks that " To test whether (a grammar) G is LR(k,t) or not, simply compute the transitions for all possible states and see if any state containing an item whose count is equal to t or which contains no brackets is accessible." [Szym 73,page 100].

As noted, Szymanski does not provide a formal algorithm for generation of a parsing automaton. However, Algorithm 3 given here is designed for such a purpose. The construction of a phrase finding automaton is along lines similar to the characteristic parsing approach for LR(k) parsers [GeHa 77a, 77b, Heil 81], and thus provides a formal definition of LR(k,t) grammars in terms of construction of a characteristic finite state machine.

Algorithm 3: Construction of phrase finding automata for LR(k,t) grammars.

Given a context-free grammar $G=(N,T,P,S)$ this algorithm decides whether G is LR(k,t) for fixed numbers k and t. The algorithm will produce a phrase finding automaton if the answer is positive.

```
(1) Build the initial state $s_0$:
      Let the basis of $s_0$ be $\{(\$^{k+1} S]_0\$^{k},0)\}$.

(2) Repeat
      For a non-final state s whose successors are not yet
      determined do
      Begin
        If every item in s has the same bracket $]_i$ as
           its first symbol
        Then s is a final state indicating the reduction of
           the ith production.
        Elseif there exists an item $(\gamma,h)$ such that $\gamma$
              contains no brackets or $h=t$
        Then G is not LR(k,t); exit.
        Else
          Begin
            (a)- Close state s:
                 Repeat
                   For every item of the form
                   $(\beta_0 X\beta_1 Y_1 \beta_2 Y_2 \dots \beta_k Y_k \gamma,h)$ in s
```

```
                  ( where $\beta_j \in B^*$ ) and
                  production $Y_k \to \alpha_i$ in P add item
                  $(\beta_0 X \beta_1 Y_1 \beta_2 Y_2 \ldots \beta_k \alpha_i ] \gamma', h)$
                  to s (if it is not there already), where
                  $\gamma'$=first kt symbols of $m(\gamma)$.
               Until no more item can be added to s.
           (b)- Generate successors of s:
               For each string $Y_1 Y_2 \ldots Y_k \in (N \cup T)^k$
               create a state s' (successor of state s
               under lookahead string $Y_1 Y_2 \ldots Y_k$ )
               with basis items
               $\{(\beta_1 Y_1 \beta_2 Y_2 \ldots \beta_k Y_k \gamma, h+|\beta_0|) \mid$
               $(\beta_0 X \beta_1 Y_1 \beta_2 Y_2 \ldots \beta_k Y_k \gamma, h)$ in s$\}$
               If a state s" with the same basis
                  already exists,
               Then merge s' with s" fi.
        end
      fi
    end
  until no more states can be added.

3) Conclude G is LR(k,t).
```

Algorithm 3 terminates. So, a definition for LR(k,t) grammars can be given in the following way.

Definition 5: LR(k,t) grammars.

A context-free grammar is LR(k,t) if and only if a phrase finding automaton for it can be constructed according to the algorithm 3.

The parsing algorithm for LR(k,∞) grammars (which uses two stacks) is similar to LR(k,t) except that there is no bound on the number of brackets bypassed, and no bound on the length of right context in the items. Therefore a valid item for a string $X_1 \ldots X_n Y_1 \ldots Y_k$ defined in the following way.

Definition 6: LR(k,∞) valid item.

An item $\beta_1 Y_1 \ldots \beta_k Y_k \gamma$is valid for $X_1 \ldots X_n Y_1 \ldots Y_k$ iff there exist $\alpha_1, \ldots, \alpha_n \in B$ such that

1) $\alpha_1 X_1 \ldots \alpha_n X_n \beta_1 Y_1 \ldots \beta_k Y_k \gamma \in PL(G)$.

2) γ contains no complete phrases.

Note that in the computation of $V_{k,\infty}(X_1 \ldots X_n Y_1 \ldots Y_k)$ in step (3) γ' would be taken to be $m(\gamma)$ rather than the first kt symbols of it as it was the case in the LR(k,t) algorithm. But in the presence of left recursive rules in the grammar, to insure the termination of the algorithm one needs to represent such $m(\gamma)$ as a regular expression (in Szymanski's description of the algorithm such steps are not formalized).

One may observe that the number of valid LR(k,∞) item sets is not necessarily finite. In terms of the construction of an automaton (characteristic machine) or generation of a parser, this means that there may indeed not be such a finite automaton. In fact Szymanski [Szym 73,page 56] remarks that "Let us now attempt to generate a parser (for an LR(1,∞) grammar). We will do this by generating a PFA (phrase finding automaton) with a potentially infinite number of states."

As seen, LR(k,∞) grammars in general do not yield automatic table driven parsers. FSPA(k) grammars are the set of those LR(k,∞) grammars that yield a finite number of item sets.

The criterion for LR(k,∞) grammars with a fixed k is shown to be equivalent to determining the null intersection of LR(k) languages, which is an undecidable problem [Szym 73,SzWi 76]. For FSPA(k) grammars with a fixed k, the criterion effectively can be reduced to the regular separability of two deterministic context-free languages [Szym 73, SzWi 76].

2.3.3 Some conclusions about the LR-regular, LR(k,t),LR(k,∞) and FSPA(k) grammars

From the study of the grammar classes considered in the previous two sections, the following conclusions may be drawn.

- (1) Consider a grammar G that is LR(k,t), LR(k,∞) or FSPA(k) and a string in the phrase language of G, say

$$\Phi=\beta_1\alpha_{i_1}]_{i_1}\beta_2\alpha_{i_2}]_{i_2}\ldots\beta_m\alpha_{i_m}]_{i_m}\beta_{m+1}$$

such that $\beta_j,\alpha_{i_j}\in(N\cup T)^*$ for $1\le j\le m$, $\beta_{m+1}\in(N\cup T\cup B)^*$ and α_{i_j} is the right hand side of some production in P. Let the corresponding sentential form be $\Phi'=m(\Phi)=\gamma\alpha_{i_m}\gamma'$ where $\gamma=\beta_1\alpha_{i_1}\beta_2\alpha_{i_2}\ldots\beta_m$ and $\gamma'=m(\beta_{m+1})$. Suppose α_{i_m} is the phrase that the phrase finding algorithm decides on its reduction. Obviously $\alpha_{i_1},\ldots,\alpha_{i_{m-1}}$ are the bypassed phrases, and in particular $m\le t$ if G is a LR(k,t) grammar. Let γ'' be the first k symbols of γ'. $\gamma\alpha_{i_m}\#\gamma''$ in which # matches null string is a parsing pattern for the reduction of the production: $A_{i_m}\to\alpha_{i_m}$.

In case of LR(k,t) and FSPA(k) grammars such a pattern is accepted by a finite automaton, i.e., the collection of parsing patterns for reduction of a production is a regular set. In LR(k,∞) grammars, the sets of parsing patterns are in general context-free languages. The set of parsing patterns of a grammar constitutes a parsing scheme for the grammar.

Parsing patterns such as above which are k bounded on the right of # allow linear parsing with limited backtracking. When a phrase is reduced, one needs to backtrack only k symbols to continue with the parsing process.

• (2) Although all the grammar classes discussed in this section are linear parsable, there is no finite state phrase finding automaton for LR(k,∞) grammars. The fact that the constant factor in parsing of these grammars is so large makes them unattractive for practical purposes.

• (3) Efficient table driven parsers can be constructed for LR-regular, LR(k,t) and FSPA(k) grammars.

• (4) A serious problem with LR(k,∞), FSPA(k) and LR-regular grammars is that the problem of membership of an arbitrary context-free grammar in these classes is undecidable. The membership problem for the class of LR(k,∞) grammars is reducible to determining whether intersection of two LR(k) languages (deterministic languages) is empty.

For the FSPA(k) and LR-regular grammars, the problem can be reduced to the regular separability of two deterministic languages. Regular separability was posed by Culik and Cohen [CuCo 73] and Szymanski [Szym 73] as an open problem. However it seems that W. Ogden in a 1971 unpublished memorandum proved that the problem is undecidable. Sketches of the proof can be found in [SzWi 76] and [Hunt [illegible] can be shown that an instance of the halting [illegible] be reduced to the regular separability problem.

Problems similar to the above ones led H. Hunt [Hunt 82] to remark that "*any* attempt to generalize LL or LR parsing technique through the use of more complex right context must yield a grammar class with an undecidable membership problem". In the light of LR(k,t) grammars as well as LRRL(k) grammars, which use rather complex lookaheads, such remarks seem unwarranted.

Notice that for any free parameter k or t the membership problem for the class of LR(k,t) is undecidable as it is for the LR(k) grammars.

- (5) The above undecidable problems theoretically prohibit automatic generation of parsers for these classes except LR(k,t). Any attempt to generate parsers for LR-regular grammars must rely on some heuristics that may fail. For FSPA(k), it must depend on an algorithm whose termination is not guaranteed.

- (6) The class of LR(k,t) grammars does not include all context-free grammars that are parsable by a Marcus-type parser. The grammar G_1:

$S' \to S$
$S \to d$
$S \to ASB$
$A \to a$
$A \to aS$
$B \to b$

discussed in conjunction with the Marcus parser is not LR(k,t) for any finite numbers k and t. It can be shown that the construction algorithm 3 fails for this grammar with any k and t. For given k and t, $L(G_1)$ includes sentences with prefix a^n where $n>k+t$. In such sentences t initial a's have different interpretations depending on the other parts of the sentences. For example consider the two sentences:

(I) $a^n db^n$, $n>k+t$ (II) $a^n (db)^n$, $n>k+t$

In (I) all t initial a's must be reduced to A's, while in (II) none of them is a phrase (Figures 2.7 and 2.8). Therefore an LR(k,t) parser will need to delay reduction of more than t possible phrases in parsing of a sentence with a prefix a^n, $n>k+t$, and thus G_1 is not LR(k,t) for any given k and t.

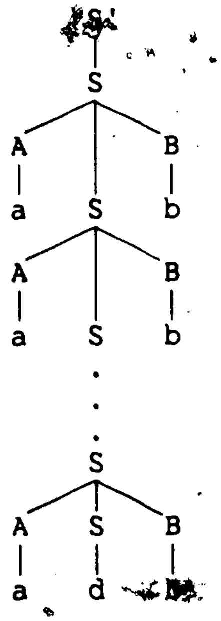


Fig. 2.7

Parse tree of $a^n db^n$

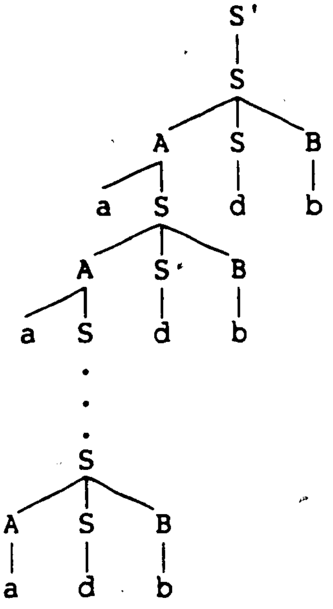


Fig. 2.8

Parse tree of $a^n (db)^n$

In fact, LR(k,t) parsers put a limit t on the number of delayed decisions at any time during the parsing. The basic LRRL(k) parser proposed in this research depending on the grammar will allow an unbounded number of decisions to be delayed. The basic class of LRRL(k) grammars that formally characterize Marcus style parsers has a non-empty intersection with the class of LR(k,t) grammars, but neither includes the other.

Several variations of LRRL(k) grammars will be discussed in this thesis. One variation, the class of generalized LRRL(k) grammars, includes LR(k',t) grammars ($k't \leq k$), and thus is proposed as the largest known class of non-canonical bottom-up parsable grammars that retains the decidability of membership in the class.

2.4 Covers

The theory of covers deals with similarity relations among grammars. Intuitively, a cover is a homomorphism from the set of parse trees of a grammar G to the set of parse trees of another grammar G'. Let $G_1=(N_1,T_1,P_1,S_1)$ and $G_2=(N_2,T_2,P_2,S_2)$ be context-free grammars, and $f_1 \subseteq T_1^* \times P_1^*$ and $f_2 \subseteq T_2^* \times P_2^*$ be parse relations. A cover homomorphism generally is a surjective homomorphism $h: f_1 \rightarrow f_2$ and is defined by two homomorphisms $\sigma: T_1^* \rightarrow T_2^*$ and $\delta: P_1^* \rightarrow P_2^*$ such that $(w,\pi) \in f_1$ implies $(\sigma(w),\delta(\pi)) \in f_2$ [Nijh 80]. Among different forms of covers, right and left covers are of interest in parsing.

Definition:

Let $G_1=(N_1,T,P_1,S_1)$ and $G_2=(N_2,T,P_2,S_2)$ be context-free grammars. G_1 right covers G_2 if there exists a homomorphism $\delta: P_1^* \to P_2^*$ such that

(a) if $S_1 \xrightarrow[G_1,rm]{\pi} w$, where $w \in T^*$, then $S_2 \xrightarrow[G_2,rm]{\delta(\pi)} w$, and

(b) for all derivations π_2 such that $S_2 \xrightarrow[G_2,rm]{\pi_2} w$, where $w \in T^*$,

there exists a derivation π_1 such that $S_1 \xrightarrow[G_1,rm]{\pi_1} w$ and $\delta(\pi_1)=\pi_2$.

Note that in this definition σ is an identity. Left covers can be defined in a similar way.

Suppose for a grammar G there is a simpler grammar G', say LR(k), that right covers G. One might parse the input according to G' and then find the corresponding parse tree in G. Notice that in programming languages one necessarily transforms the original grammar to, say some LR(k) cover, if the original is not already in that form. Usually, the parsing is only conducted in the transformed grammar even though the analysis may not correspond to a natural one as it is given in the original grammar. Addition of hooks in compiler construction also may be considered to constitute a cover for the original grammar.

Although finding of covers from special classes of grammars, say LR(1), for certain deterministic grammars, say LR(k) grammars, is relatively easy, determining existence of a cover from special classes, say LR(k) or LL(k), for an arbitrary context-free grammar is undecidable [Hunt 82, 84]. On the basis of this undecidable problem, in addition to the following reasons, it appears that covers do not provide a practical means for avoiding complex lookahead mechanisms.

- (1) Even assuming that for a grammar G an LR(k) right cover exists, determining the parse tree in the original grammar from the parse tree in the cover grammar is still a tedious and complex process.

- (2) Generally one has to parse the whole input sentence before the correct parse in the original grammar can be determined. That is to say computation of $\delta(\pi_1)$ cannot proceed sequentially as prefixes of π_1 become available. Therefore the method is not suitable for a one-pass compiler or translator unless one is content with the analysis being carried out only in the cover grammar.

- (3) Parsing in natural languages repeatedly interacts with semantic and pragmatic components of a general language processing system. Such interaction may direct the parsing or decide on unacceptability of an input on semantic and pragmatic grounds. Semantic rules are often closely associated with the original grammar defining the syntax, e.g., in [Gazd 82] and [ScPe 82]. Thus one needs to provide syntactic analysis according to the original grammar. Computational linguists are in general reluctant to opt for a cover grammar that arbitrarily distorts the natural form of the original grammar [Schu 83].

2.5 Conclusion

This chapter has introduced the notion of 'complex' lookaheads in the context of Marcus parsing. It was noted that despite the growing interest in Marcus' determinism hypothesis, it is almost impossible to draw a clear conclusion about his theory. The major reason for this is the procedural approach that he has taken for defining the language. Not only does the presence of parsing notions such as create, drop, etc. in a PIDGIN grammar make it difficult to determine exactly what subset of English is parsed by the grammar, but in the absence of formalism it is also very hard to determine whether a given arbitrary language can be parsed in this style. As an alternative, it was suggested that the languages parsed by Marcus-type parsers should be described in a declarative form by using context-free grammars plus the extension of feature assignment rules. Appendix I formally proves the context-freeness of the languages that are accepted by Marcus-type parsers, and thus provides a justification for the above suggestion.

It was shown that the context-free grammars parsable by Marcus' mechanism cannot be handled by the LR(k) parsers.

Furthermore, cover grammars and existing generalizations of LR(k) grammars were discussed and shown to be unsuitable for formalizing the concepts of Marcus parsing. Specifically, Berwick's conjecture regarding LR(k,t) grammars was proven to be inadequate. Another proposal by Berwick, with respect to BCP(m,n) grammars, is also rejected in [Nozo 86]. Therefore, it seems desirable that one should search for a new class of grammars which could provide a formal framework for Marcus' parser. The remainder of this thesis is devoted to the development of such grammars.

Chapters 3 through 5 deal with context-free grammars, or if one wishes the context-free component of a language. Chapter 6 considers the context-sensitive aspects of a language primarily in the form of attributed grammars.

CHAPTER 3

DEVELOPMENT OF LRRL(k) GRAMMARS

LRRL(k) parsing basically is a deterministic non-canonical bottom-up parsing technique which is influenced by the "wait and see" policy of the Marcus parser. LRRL(k) grammars denote a family of grammar classes that are parsed *l*eft to *r*ight with *k* *r*educed *l*ookaheads according to the above technique. The difference between these classes lies in the nature of lookaheads that they employ. Roughly, the class with more 'complex' lookaheads includes the class with 'simpler' lookaheads. In this chapter the basic LRRL(k) grammars without ϵ-rules are discussed. For convenience, sometimes these grammars will be referred to only as LRRL(k) grammars without the modifiers. Details about other classes will be given in Chapter 5.

3.1 Preliminary definitions

All the terminology follows that of LR(k) parsing as given in [AhUl 72a, 73a] with the exception of following additions or modifications. In all of the definitions only ϵ-free reduced context-free grammars are considered. Thus, for brevity, the premodifiers are dropped from here on.

Assume $G=(N,T,P,S)$ is a context-free grammar and $V=N\cup T$.

Definition 1: Prefixes of a string

Let $\alpha\in V^*$. $PF_k(\alpha)$ is the string of first k symbols in α if $|\alpha|\geq k$. Otherwise $PF_k(\alpha)=\alpha$.

Definition 2: Bounded concatenation

Let V_1 and V_2 be subsets of V^*. The k-bounded concatenation of V_1 with V_2 is

$V_1 \underset{k}{+} V_2=\{\omega \mid \text{for}\ \alpha\in V_1;\beta\in V_2;\ \omega=PF_k(\alpha\beta)\}$.

Definition 3: Relative prefixes

Let V_1 and V_2 be subsets of V^*. The set of relative prefixes of V_1 and V_2 is defined to be

$V_1\underline{\cap}V_2=V_2\underline{\cap}V_1=\{\omega \mid (\omega\in V_1\ \text{and}\ \omega\alpha\in V_2)\ \text{or}\ (\omega\in V_2\ \text{and}\ \omega\beta\in V_1)\}$.

Definition 4: Prefix free set

Let U be a subset of V^*. The prefix free set of U or Min(U) is defined to be

$Min(U)=U-UV^+=\{\omega\mid\ \omega\in U\ \text{and for no}\ k<|\omega|,\ PF_k(\omega)\in U\}$

Definition 5: Reduced right context

A k-symbol reduced right context of (a prefix of) a phrase in a parse tree is a sequence of k nodes to the immediate right of (the prefix of) the phrase such that none of them dominates any other, and the sequence contains no complete phrase. Formally, let $S\overset{*}{\Longrightarrow}\alpha A\gamma\Longrightarrow\alpha\beta_1\beta_2\gamma$ be a derivation in G. $PF_k(\beta_2\gamma)$ is a k-symbol reduced right context for β_1 if it does not contain a complete phrase.

Definition 6: Fully reduced right context

The k-symbol fully reduced right context of (a prefix of) a phrase in a parse tree is the sequence of k nodes to the immediate right of (the prefix of) the phrase which are daughters of the nodes dominating (the prefix of) the phrase. In other words these are the k symbols that follow (the prefix of) the phrase in the leftmost derivation of the tree. More formally, let $S \overset{*}{\underset{lm}{\Longrightarrow}} xA\gamma \underset{lm}{\Longrightarrow} x\beta_1\beta_2\gamma$ be a leftmost derivation in G. $PF_k(\beta_2\gamma)$ is the k-symbol fully reduced right context of β_1.

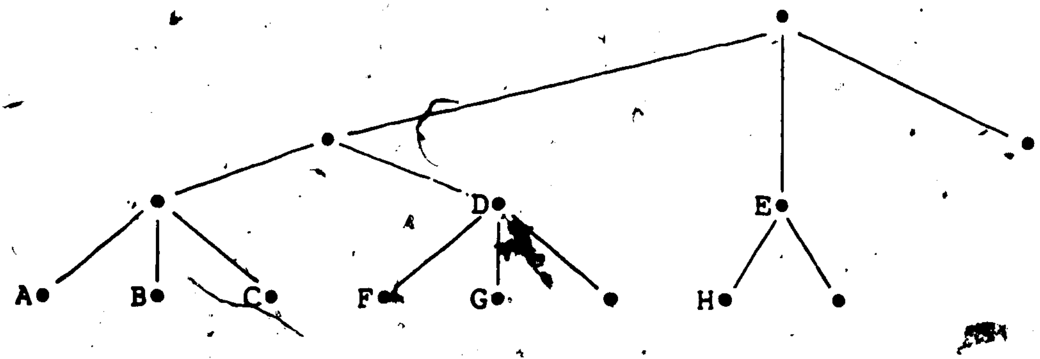


Fig 3.1
Reduced right contexts

As seen, the fully reduced context is the topmost reduced context and thus dominates any sequence of k subtrees to the immediate right of (the prefix of) the phrase. For example in Figure 3.1, "CDE", "CFG" and "CDH" are three-symbol reduced right contexts for "AB". While, "CDE" is the three-symbol fully reduced right context of "AB".

3.2 Basic LRRL(k) grammars

A basic LRRL(k) parser employs k-symbol fully reduced right contexts or lookaheads in making parsing decisions. This generalized lookahead policy implies that when a questionable handle in a right sentential form is reached, the decision to reduce it or not may be reached by parsing ahead a segment of the input that can be reduced to a relevant fully reduced right context of length k. For example, in parsing a sentence in $L(G_1)$, where G_1 is the grammar given in Chapter 2;

(1) $S \rightarrow d$
(2) $S \rightarrow A\ S\ B$
(3) $A \rightarrow a$
(4) $A \rightarrow a\ S$
(5) $B \rightarrow b$

after seeing the initial *a* there is a shift-reduce conflict as to whether one should reduce it according to the rule (3), or continue with the rule (4). However the 2-symbol fully reduced context for reduction is *SB*, and for the shift operation is *SS*, which indicates a possible resolution of conflict if one can parse the lookaheads. Therefore one postpones the reduction of this questionable phrase and adds two new auxiliary productions *SUBGOAL-RED(3)* → *SB* and *SUBGOAL-SHIFT* → *SS*, and continues with the parsing of these new constructs. Upon completion of one of these productions one will be able to resolve the conflicting situation. Furthermore, the same policy may be applied to the parsing of lookahead contexts themselves. This feature of LRR(k) parsing, i.e., the recursive application of the method to the lookahead information, is the one that differentiates this method from any other. The method is recursively applied whenever the need arises, i.e., at equivalent points during parsing. If one always insisted on parsing lookahead information before reducing a handle, then the resulting method would have been less powerful (paradoxically, it would have enjoyed more power in error detection because of early recognition of errors). However the corresponding class of grammars would have had a cleaner definition in terms of a special form of non-canonical derivations.

Note that the lookahead scheme does not allow one to examine any remaining segment of the input that is not a part of the lookahead context. The parsed context is put in a buffer of size k, and no reexamination of the segment of the input sentence that has been reduced to this right context is carried out. In addition, the right context which is k symbols or less does not contain a complete phrase, i.e., the symbols in the right context do not participate in any future reductions involving only these symbols. Rather, they participate only in reductions involving the questionable phrase or its dominators when the reduction of this phrase is concluded, or in the reduction of a phrase whose prefix was in question and its dominators. Szymanski [Szym 73, page 52] also makes a remark on such a fully reduced context:

> "The important thing to notice is that if the decision to reduce a phrase must be postponed, then it must be the case that in the corresponding sentential forms with all phrases to the right of the place in question reduced, the first k characters of fully reduced context tell us what decision to make."

But, neither informal intuition about LR(k,t) parsing nor its formal algorithm conforms with the above remarks.

3.2.1 Rationale for the choice of lookaheads

It seems to be a plausible linguistic assumption to base the reduction decision on the fully reduced context or the nodes in the parse tree that c-command the current construct. Roughly speaking, in the $\bar{X}$ theory [Jack 77, BeWe 82], node A c-commands (constituent commands) node B if and only if the immediate ancestor of A is an ancestor of B. There are grounds for such an assumption:

(1) There is precedence:

1.a LR(k) parsers use the entire left context in a fully reduced form.

1.b C-command is successfully used in natural languages in such areas as bindings, e.g. binding of pronouns by quantifiers [BeWe 82].

In (1.a) only left context is considered. In (1.b) due to the right branching of English, the nodes that c-command the current node are normally on its left side. However this cannot be a universal rule. If one decides to parse "English^R", i.e., English sentences in which the order of words is reversed, then c-command has to deal with the right context. This hypothetical case is not so much beyond reality in some languages in which the order of words in a sentence can be freely exchanged. Also in a sentence "He could not win the election," the president thought..." when doing surface analysis only (i.e. when no transformations are considered), the pronoun binding has to look for c-command in the right context.

(2) Most importantly, if the current phrase can be reduced on the basis of k nodes that are dominated by some nodes of the fully reduced right context, then surely the decision can also be reached on the basis of k fully reduced context. Stating it differently, if a reduction decision cannot be made using the k fully reduced right context, then no right context lookahead of size k will resolve the conflict. Otherwise, it can be concluded that the language under consideration is not context-free.

The crucial question in parsing is whether the right context can be parsed without looking beyond the segment of input that is derived from it. Allowing unrestricted lookahead beyond this segment results in undecidable problems similar to those in LR(k,∞) and LR-regular grammars. The requirement of parsability of the lookahead context is a constraint one has to live with in the basic LRRL(k) grammars. In fact, it will be shown that there are some subtle LR(k) grammars that are not in the basic LRRL(k) class for the same k. Relaxing the condition in a very restrictive way will provide a new class ELRRL(k) that includes all the LR(k) grammars with the same k.

Figure 3.2 together with Table 3.1 compare some parsing strategies against each other and the basic LRRL technique in terms of the amount of input stream read when the parser decides on the recognition of a production $X \to X_1 \ldots, X_n$, if the grammar falls within the corresponding class of the parser. The amount of extra storage besides the usual push-down stack is also given.

It is assumed that the length of the terminal string to the right of X to be greater than k.

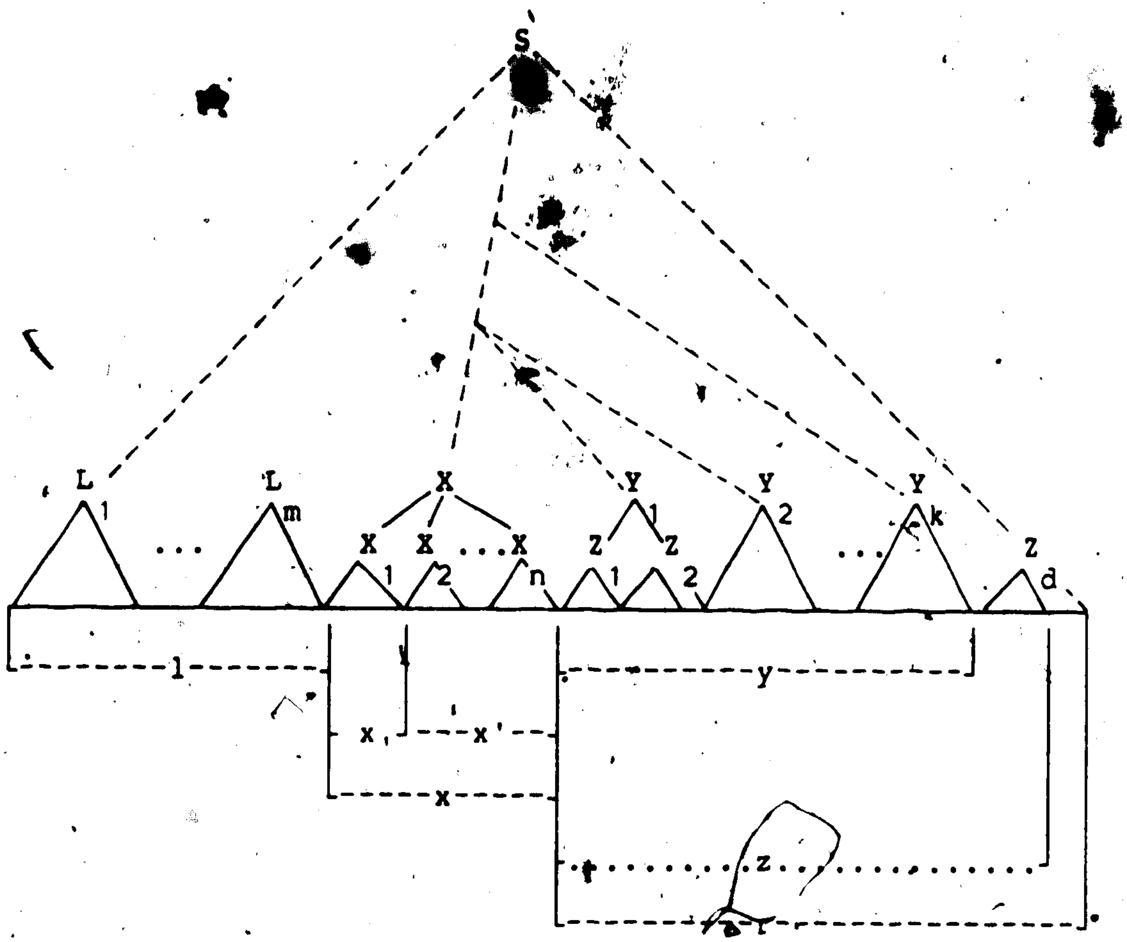


Fig. 3.2
Comparison of parsing strategies

grammar class	input read	complete subtrees constructed	type of storage
LL(k)	$1PF_k(xr)$	$L_1,\ldots,L_m$	buffer of size k
LC(k)	$1x, PF_k(x''r)$	$L_1,\ldots,L_m, X_1$	"
LR(k)	$1xPF_k(r)$	$L_1,\ldots,L_m, X_1,\ldots,X_n$	"
LR(k,t)	$1xz$ where z is a prefix of r s.t. it is reduced to d trees and $k \le d \le kt$, $k \le \|z\| \le \|r\|$	$L_1,\ldots,L_m, X_1,\ldots,X_n$ $Z_1,\ldots,Z_d$ $k \le d \le kt$ $Z_1,\ldots,Z_k$ may or may not be equal to $Y_1,\ldots,Y_k$.	buffer of size kt
LR(k,∞)	$1xz$ where z is a prefix of r s.t. it is reduced to d trees and $k \le d \le \|r\|$, $k \le \|z\| \le \|r\|$	$L_1,\ldots,L_m, X_1,\ldots,X_n$ $Z_1,\ldots,Z_d$ $k \le d \le \|r\|$ $Z_1,\ldots,Z_k$ may or may not be equal to $Y_1,\ldots,Y_k$.	stack
LR-regular	$1xr$	$L_1,\ldots,L_m, X_1,\ldots,X_n$	stack
Basic LRRL(k)	$1xy$	$L_1,\ldots,L_m, X_1,\ldots,X_n$ $Y_1,\ldots,Y_k$	buffer of size k

Table 3.1
Comparison of parsing strategies

3.2.2 Characteristic parsing automata approach to LRRL grammars

The approach to LR theory based on characteristic parsing automata, wherein a class of grammars is defined by construction of finite automata, has been in use since the early development of LR parsers, though only in a semi-formal way. Recently, the approach was completely formalized as a framework for defining the class of LR grammars [GeHa 77a,77b, Heil 81]. It is interesting to note that certain classes of LR grammars, e.g., LALR(k) grammars, can only be defined in terms of characteristic parsing. Szymanski [Szym 73] has also used the methodology in the form of phrase finding automata.

The parsing algorithm for, and definition of LRRL(k) grammars are based on construction of a Characteristic Finite State Machine. A CFSM is rather similar to the deterministic finite automaton that is used in LR(k) parsing for recognition of viable prefixes. However, there are three major differences:

(1) The nature of lookaheads. The lookaheads are fully reduced symbols as opposed to bare terminals in LR(k) parsing.

(2) Introduction of auxiliary productions.

(3) Partitioning of states which conceals conflicting items. Though the lookaheads are k symbol long, the decision for transition from one state to another is based on one symbol

rather than k symbols. Such a strategy is also advocated by some authors for efficient construction of full LR(k) parsers [Spec 81, AnDG 82].

The information extracted from this machine is in tabulated form that acts as the finite control for the parsing algorithm.

The following sections will describe an algorithm for the construction of a characteristic finite state machine for a grammar that is parsable by the basic LRRL(k) scheme.

3.2.3 The states of CFSM

Much as in LR(k) parsing, each state consists of one or more items or configurations of the form $A \rightarrow \alpha.\beta, L$; where L is a set of lookahead strings at most k symbols long. Unlike LR(k) items, lookahead strings may contain non-terminals as well as terminal symbols. When inadequacies arise as a result of reduce-reduce or reduce-shift conflicts in a state of CFSM, in order to enable the parser to postpone its decision, the basis of the state (i.e., the set of items $A \rightarrow \alpha X.\beta, L$ for which [illegible] corresponding item $A \rightarrow \alpha.X\beta, L$ in the predecessor state) is concealed. Next a new set of items of the form $SUBGOAL\text{-}RED(p) \rightarrow .\gamma, \{\epsilon\}$ / $SUBGOAL\text{-}SHIFT \rightarrow .\delta, \{\epsilon\}$ is added to the state, and then the closure of the new non-concealed basis is obtained. Thus in this way an inadequate state is partitioned into concealed and non-concealed items. The symbol SUBGOAL-RED(p) indicates the conflicting reduction of the production p in the state, and the symbol SUBGOAL-SHIFT shows a conflicting shift operation. The right-hand side of these items are obtained from the relevant lookahead sets that are associated with conflicting items. Note that such new items are added only when the lookaheads for each reduction operation are different from the lookaheads for any other reduction or shift operation. The productions SUBGOAL-RED(p) →γ and SUBGOAL-SHIFT →δ will be called auxiliary productions.

As outlined in the above, an inadequate state consists of a set of concealed and a set of non-concealed items. The concealed items are retained in the state so that the parser can take the appropriate action when it backtracks to this state after parsing the lookaheads. Consequently the successor states of a given state are built with respect to the two sets separately. Hence, existence of a global boolean variable FLAG is assumed that directs the parser as to which successor state to proceed. The successor state of the non-concealed set is obtained under the FLAG value OFF, while the successor state of the concealed set is given under the FLAG value ON. Notice that in the latter case only those items indicating a shift operation are taken into consideration.

One may pause here to observe that:

(1) Such a scheme is not the unique way to produce a CFSM

for a grammar that is parsable by the outlined method. In fact, it will be shown later that a CFSM can be constructed without use of a flag variable by a slightly different method.

(2) The scheme described above not only indicates the resolution in favour of a reduction at a conflicting point, but also is capable of showing resolutions in favour of shift operations by means of SUBGOAL-SHIFT productions. It will be shown shortly that the driver routine, so to speak the user, through the parse table will be aware of those shift operations that are in conflict with some reduction operations. One may do away with that, by concealing only those items that indicate a reduction in an inadequate state. In such a method, which defines a different class of grammars from the present one, the handle of a sentential form can be found if its right context could be parsed.

Therefore, the present class of grammars can be named as type I basic LRRL(k) grammars. Type II basic grammars that follow from the second scheme will be described in Section 3.2.11.

3.2.4 Closure operation

Recall that each item has the form $A \rightarrow \alpha.\beta, L$; where L is the set of lookaheads. To close the non-concealed basis of a state one takes the lookaheads into account. The following algorithm is used for this operation.

An important factor in this simulation has been the assumption that the buffer in a Marcus style parser is bounded. It is unlikely that all parsers with unbounded buffers written in this style can be simulated by deterministic pushdown automata. However, this does not mean that some Marcus-type parsers that use an unbounded buffer in a constrained way are not equivalent to pushdown automata. Shipman and Marcus [ShMa 79] consider a model of Marcus' parser in which the active node stack and buffer are combined to give a single data structure that holds both complete and incomplete subtrees. The original stack nodes and their lookaheads alternately reside on this structure. Letting an unlimited number of completed constructs and bare terminals reside on the new structure is equivalent to having an unbounded buffer in the original model. Given the restriction that attachments and drops are always limited to the k+1 rightmost nodes of this data structure, it is possible to show that a parser in this model with an unbounded buffer still can be simulated with an ordinary pushdown automaton. (The equivalent condition in the original model is to restrict the window to the k rightmost elements of the buffer. However simulation of the single structure parser is much more straightforward.)

```
Close(s: Set of items)
    Repeat
        For an item $A \rightarrow \alpha.B\beta, L$ in s; $B \in N$
        and for all $B \rightarrow \gamma \in P$
            add item $B \rightarrow .\gamma, L'=\{\beta\} +_k L$ to s
            if there is no item $B \rightarrow .\gamma, L''$ in s;
            else replace the latter with
            $B \rightarrow .\gamma, Min(L' \cup L'')$
    Until no change can be made in the set of items s.
```

3.2.5 Construction of a CFSM

Let $G=(N,T,P,S)$ be a (reduced ϵ-free) context-free grammar. Algorithm **A** decides whether the grammar G is basic LRRL(k) for the fixed number k, i.e., whether it can be parsed with k fully reduced lookaheads and a buffer of size k. The algorithm will produce a CFSM if the answer is positive.

Algorithm A :

```
(1) Add GOAL symbol to N and production
    0: GOAL $\rightarrow$ S to P.

(2) Build the initial state $s_0$:
        Let the basis of $s_0$ be { (GOAL $\rightarrow$ .S,{$}) }
        and its concealed set be $\emptyset$.
        Close non-concealed basis of $s_0$.
```

Let the set of states of CFSM, $\Omega=\{s_0\}$.

(3) <u>Repeat</u>

For a state t whose successors are not yet determined build its successor states under all applicable symbol X and FLAG values (OFF, ON) as described below.

(a)- Construction of a successor state s for the given state t under symbol X and FLAG=OFF:

```
(i) If there is no non-concealed item of the form
    $(A \rightarrow \alpha.X\beta,L)$ in t, then t has no successor
    under symbol X and FLAG=OFF.
    Else
        let basis of s=$\emptyset$.
        For each non-concealed
        item $(A \rightarrow \alpha.X\beta,L)$ in t add
        $(A \rightarrow \alpha X.\beta,L)$ to the basis of s.
```

(ii) Check for inadequacy in the basis of s:

If there are items of the form

$I_j : A_j \rightarrow \alpha_j X.\beta_j ,L_j ;$
$j=1,\ldots,m$ and $\beta_j \neq \epsilon$

and

$I_j : A_j \rightarrow \alpha_j X.,L_j ;$
$j=m+1,\ldots,n$

```
    where $m \geq 1$ and $n > m$
    (i.e. shift-reduce/reduce-reduce conflicts)
    or
    $m = 0$ and $n > 1$
    (i.e. only reduce-reduce conflicts)
Then

     let the shift lookaheads of s be
     $SHL(s) = Min \cup_{j=1}^{m} (\{\beta_j\} +_k L_j)$.
     If $L_j \cap SHL(s) = \emptyset$ for $j = m+1, \ldots, n$
        (i.e. possible shift-reduction
        resolution) and
        $L_i \cap L_j = \emptyset$ for $i, j = m+1, \ldots, n$, $i \neq j$
        (i.e. possible resolution of reduce-
        reduce conflicts)
     Then
        conceal the original conflicting basis
        items $I_1, \ldots, I_n$, and add new
        non-concealed basis, i.e,
        For $j = m+1, \ldots, n$
           add SUBGOAL-RED($p_j$) $\rightarrow . \gamma, \{\epsilon\}$
           to s for all $\gamma \in L_j$;
           where $p_j$ is the production $A_j \rightarrow \alpha_j X$.
        Also add SUBGOAL-SHIFT $\rightarrow . \gamma, \{\epsilon\}$ to s
        for all $\gamma \in SHL(s)$.
     Else
        conclude the grammar is not basic
```

LRRL(k) and exit.

(iii) Close the non-concealed basis of s.

(iv) Add state s to Ω if there is no state t' in Ω with the same items as s, or with items which match those of s apart from the lookahead sets, and for lookahead set L' in t' it is true that $L'=\mathrm{Min}(L \cup L')$ and $L'=L \cap L'$, where L is the corresponding lookahead set in s.
(i.e., each $\sigma' \in L'$ is a prefix of some $\sigma \in L$ and each $\sigma \in L$ has a prefix σ' in L'.
This latter condition is introduced as a means of optimization that reduces lengths of necessary lookaheads.)
Otherwise let t' be the successor of t.

(b)- Construction of a successor state s' for the given state t under symbol X and FLAG=ON :

(i) If there is no concealed item of the form

$A \to \alpha .' X\beta, L$ in t,

Then state t has no successor under symbol X and FLAG value ON.

Else

let non-concealed basis of s' be Ø.

For each concealed item $(A \rightarrow \alpha . X\beta, L)$ in t add $(A \rightarrow \alpha X . \beta, L)$ to the basis of s'.

(ii)-(iv) Repeat steps (ii),(iii) and (iv) of part (a) for state s'.

Until no more state can be added to Ω.

(4) Conclude that the grammar is basic LRRL(k).

Note:

The repeat statement of the step (3) can be implemented with the aid of a stack or a queue.

Theorem 3.1

The CFSM construction algorithm A terminates.

Proof: The proof is based on the finiteness of the number of the items. This gives an upper bound on the number of states that can be generated. First, one should note that each state is uniquely characterized by its original basis. Whether this basis is concealed or not is immaterial. Secondly, the dot appears after the first symbol in the right hand side of the core of a basis item (with the exception of the initial item in s_0). For counting purposes, one could consider an item like $(A \rightarrow \alpha.\beta,L)$ as a set of items $\{(A \rightarrow \alpha.\beta,\sigma)|\sigma \in L\}$. The number of distinct items $(A \rightarrow \alpha.\beta,\sigma)$, where $A \in V$ and $|\alpha| \geq 1$, is $m_1=(|G|-|P|)(1+|V|+...+|V|^k)<(k+1)|G|^{k+1}$, and the number of distinct items $(A \rightarrow \alpha.\beta,\epsilon)$, where A is some subgoal symbol and $|\alpha| \geq 1$, is $m_2=(|P|+1)(|V|+2|V|^2+...+k|V|^k)<k^2|G|^{k+1}$. Thus, the number of states is less than 2^m, where $m=(k+1)^2|G|^{k+1}$. □

Definition 1

A context-free grammar G=(N,T,P,S) is type I basic LRRL(k) iff a CFSM for it can be constructed according to the algorithm **A**.

Corollary 3.1

It is decidable whether an arbitrary (reduced ϵ-free) context-free grammar is LRRL(k) for a fixed number k.

It is obvious that the question of whether an arbitrary context-free grammar is LRRL for some k is undecidable (as in the case of LR grammars).

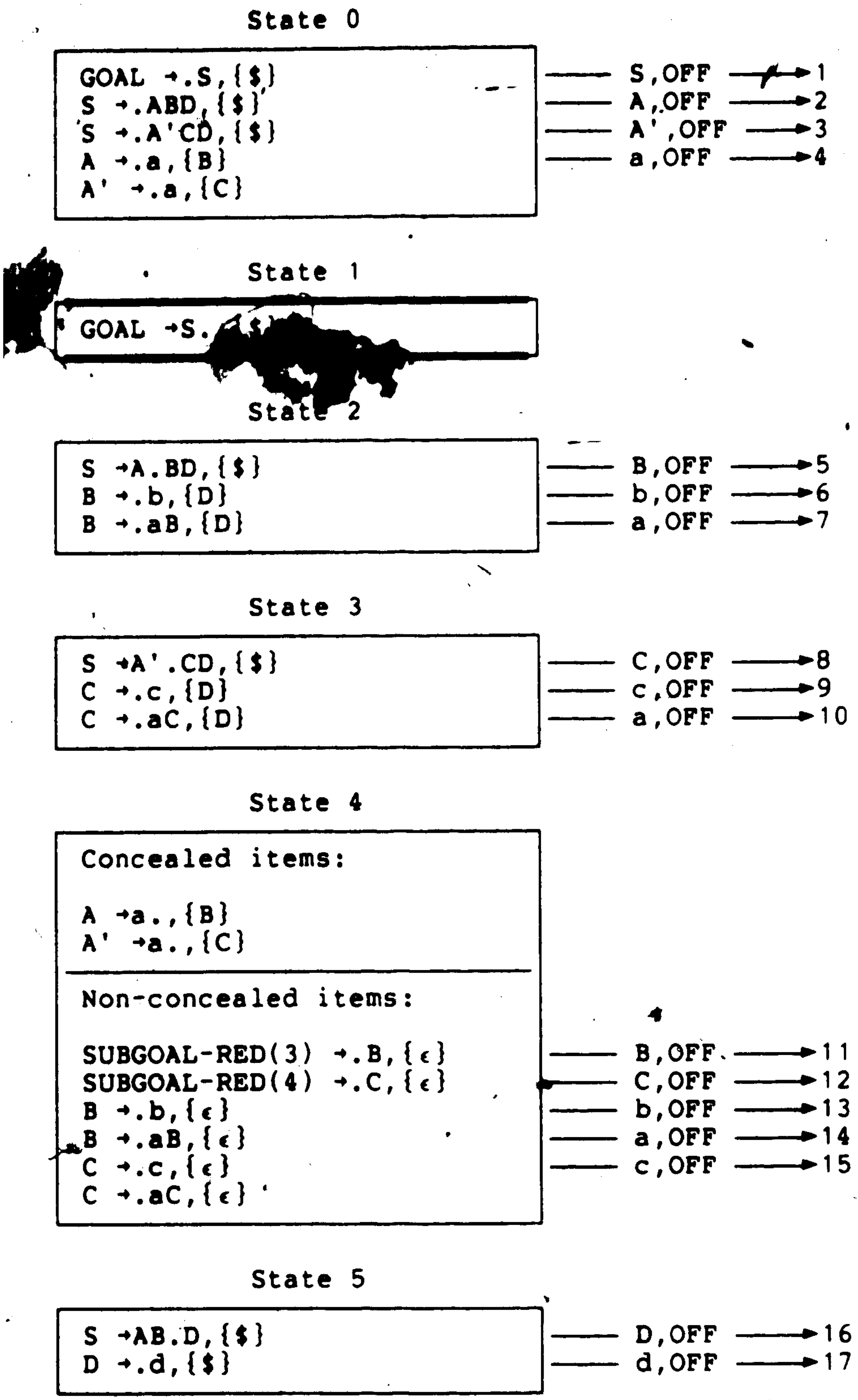
Example 1:

Consider grammar G which is not LR(k) for any k.

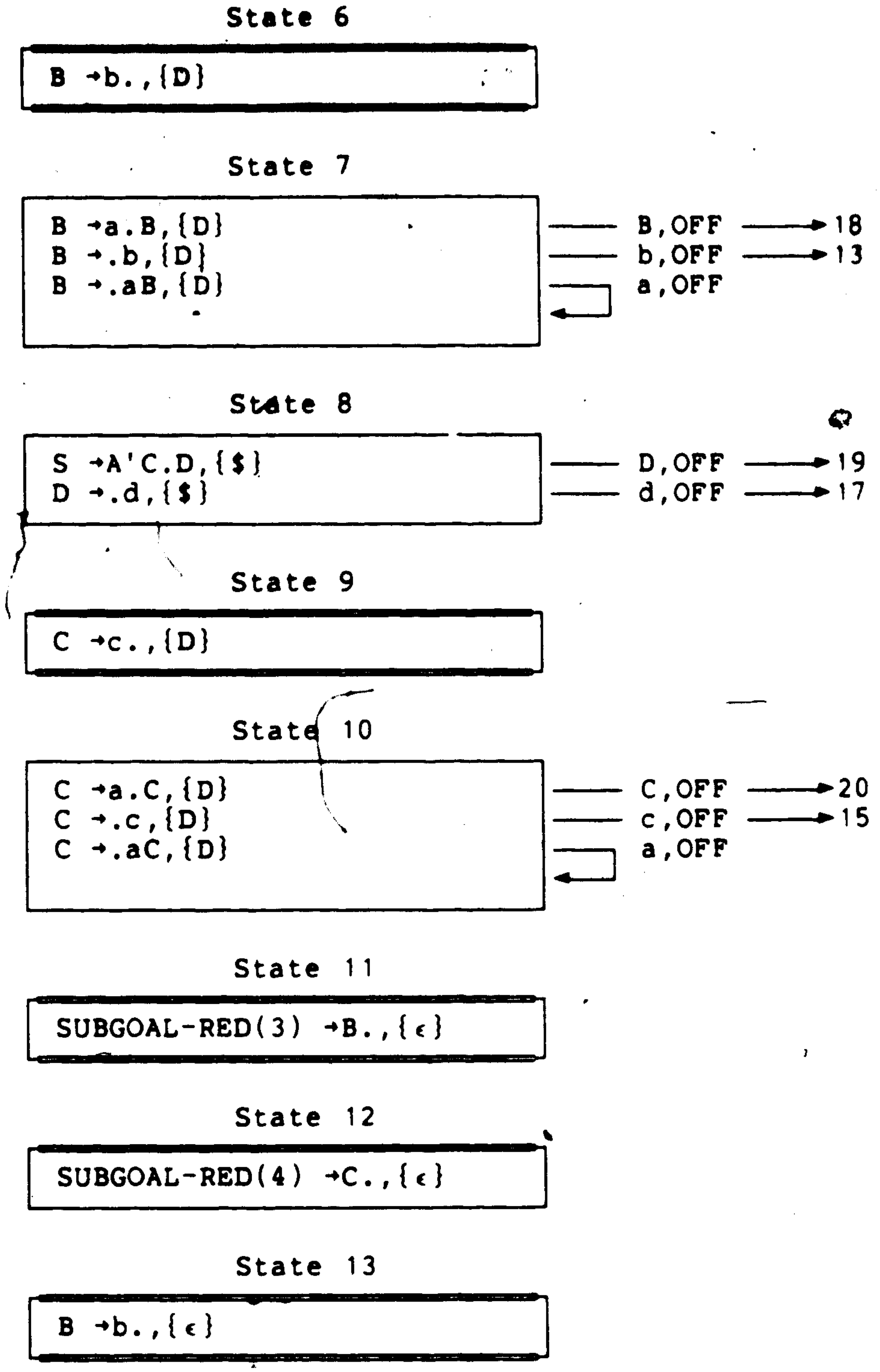
(1) S →A B D
(2) S →A' C D
(3) A →a
(4) A' →a
(5) B →b

(6) B →a B
(7) C →c
(8) C →a C
(9) D →d

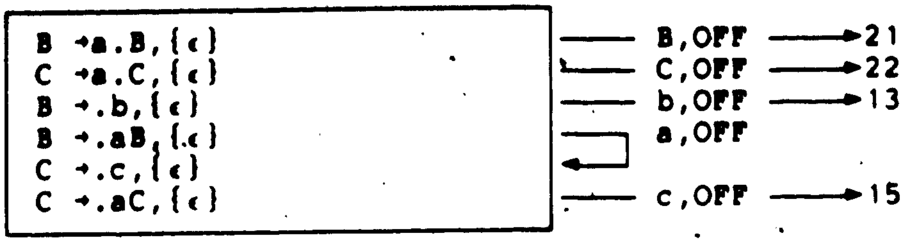
The following diagram is a CFSM for G with k=1, i.e. G is LRRL(1). Notice that G is also LR(1,2) and LR-regular.



Diag. 3.1
The CFSM for G


State 6
B →b.,{D}
State 7
B →a.B,{D}
B →.b,{D}
B →.aB,{D}
B,OFF
18
b,OFF
13
a,OFF
State 8
S →A'C.D,{$}
D →.d,{$}
D,OFF
19
d,OFF
17
State 9
C →c.,{D}
State 10
C →a.C,{D}
C →.c,{D}
C →.aC,{D}
C,OFF
20
c,OFF
15
a,OFF
State 11
SUBGOAL-RED(3) →B.,{ε}
State 12
SUBGOAL-RED(4) →C.,{ε}
State 13
B →b.,{ε}

State 14



State 15

C →c.,{ε}

State 16

S →ABD.,{$}

State 17

D →d.,{$}

State 18

B →aB.,{D}

State 19

S →A'CD.,{$}

State 20

C →aC.,{D}

State 21

B →aB.,{ε}

State 22

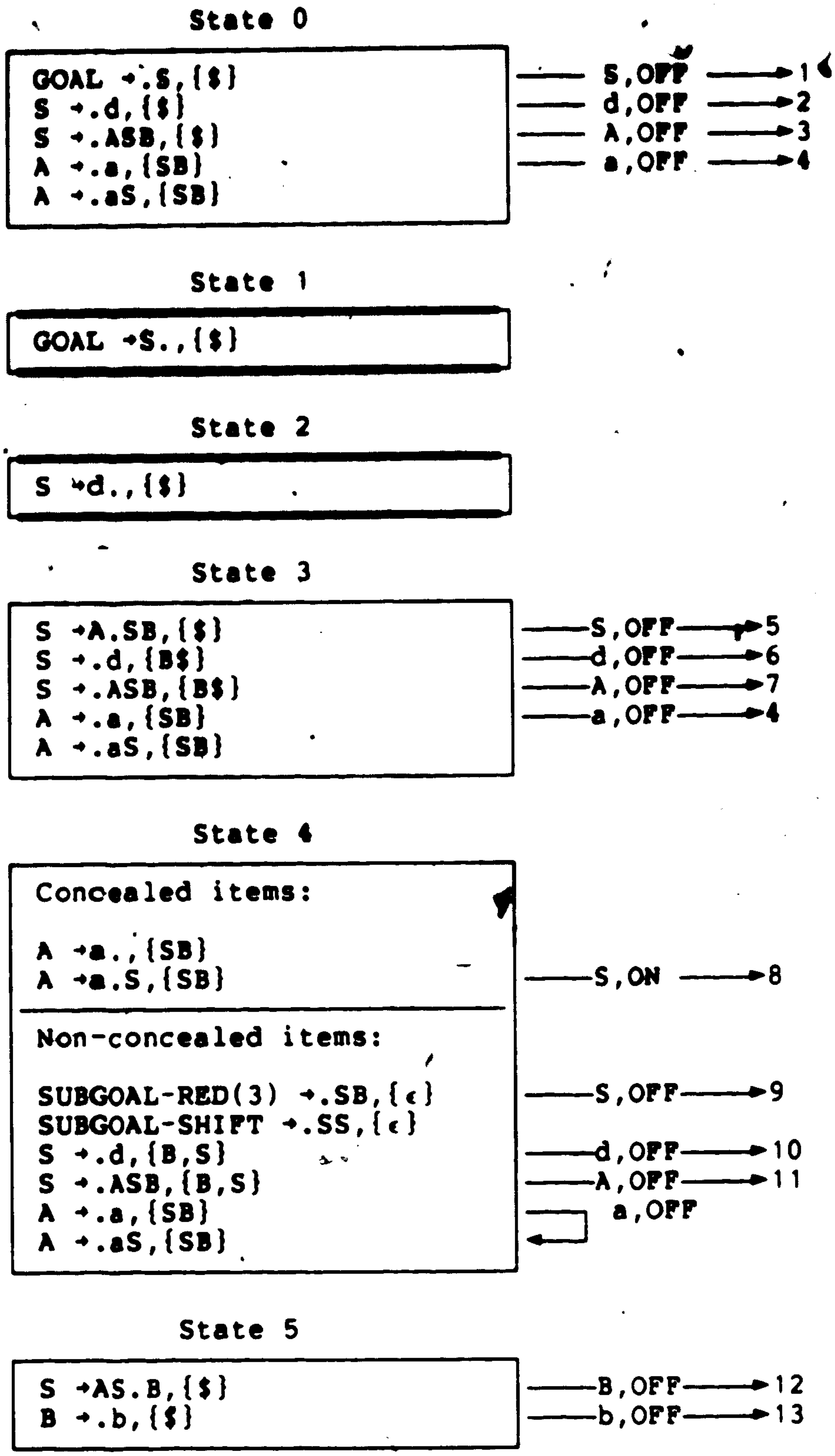
C →aC.,{ε}

Example 2:

Consider grammar G_1 of Chapter 2:

(1) $S \to d$
(2) $S \to A\ S\ B$
(3) $A \to a$
(4) $A \to a\ S$
(5) $B \to b$

The following diagram shows the CFSM for G_1 constructed according to the algorithm A with k=2. Therefore the grammar is LRRL(2), however earlier it was proved that_it is not LR(k,t) for any finite k and t. It is also believed that G_1 is not LR-regular.



Diag. 3.2
The CFSM for G_1

State 6

```
S →d.,{B$}
```

State 7

```
S →A.SB,{B$}        ——S,OFF——→14
S →.d,{BB}          ——d,OFF——→15
S →.ASB,{BB}        ——A,OFF——→16
A →.a,{SB}          ——a,OFF——→4
A →.aS,{SB}
```

State 8

```
A →aS.,{SB}
```

State 9

```
SUBGOAL-RED(3) →S.B,{ε}     ——B,OFF——→17
SUBGOAL-SHIFT →S.S,{ε}      ——S,OFF——→18
B →.b,{ε}                   ——b,OFF——→19
S →.d,{ε}                   ——d,OFF——→20
S →.ASB,{ε}                 ——A,OFF——→21
A →.a,{SB}                  ——a,OFF——→4
A →.aS,{SB}
```

State 10

```
S →d.,{B,S}
```

State 11

```
S →A.SB,{B,S}       ——S,OFF——→22
S →.d,{BB,BS}       ——d,OFF——→20
S →.ASB,{BB,BS}     ——A,OFF——→21
A →.a,{SB}          ——a,OFF——→4
A →.aS,{SB}
```

State 12

```
S →ASB.,{$}
```

State 13

```
B →b.,{$}
```

State 14

```
S →AS.B,{B$}        ——B,OFF——→23
B →.b,{B$}          ——b,OFF——→19
```

State 15

```
S →d.,{BB}
```

State 16

```
S →A.SB,{BB}        ——S,OFF——→24
S →.d,{BB}          ——d,OFF——→20
S →.ASB,{BB}          A,OFF
A →.a,{SB}
A →.aS,{SB}         ——a,OFF——→4
```

State 17

```
SUBGOAL-RED(3) →SB.,{ϵ}
```

State 18

```
SUBGOAL-SHIFT →SS.,{ϵ}
```

State 19

```
B →b.,{ϵ}
```

State 20

```
S →d.,{ϵ}
```

State 21

```
S →A.SB,{ϵ}         ——S,OFF——→25
S →.d,{B}           ——d,OFF——→20
S →.ASB,{B}           A,OFF
A →.a,{SB}
A →.aS,{SB}         ——a,OFF——→4
```

State 22

```
S →AS.B,{B,S}        ——B,OFF——→26
B →.b,{B,S}          ——b,OFF——→19
```

State 23

```
S →ASB.,{B$}
```

State 24

```
S →AS.B,{BB}         ——B,OFF——→27
B →.b,{BB}           ——b,OFF——→19
```

State 25

```
S →AS.B,{ε}          ——B,OFF——→28
B →.b,{ε}            ——b,OFF——→19
```

State 26

```
S →ASB.,{B,S}
```

State 27

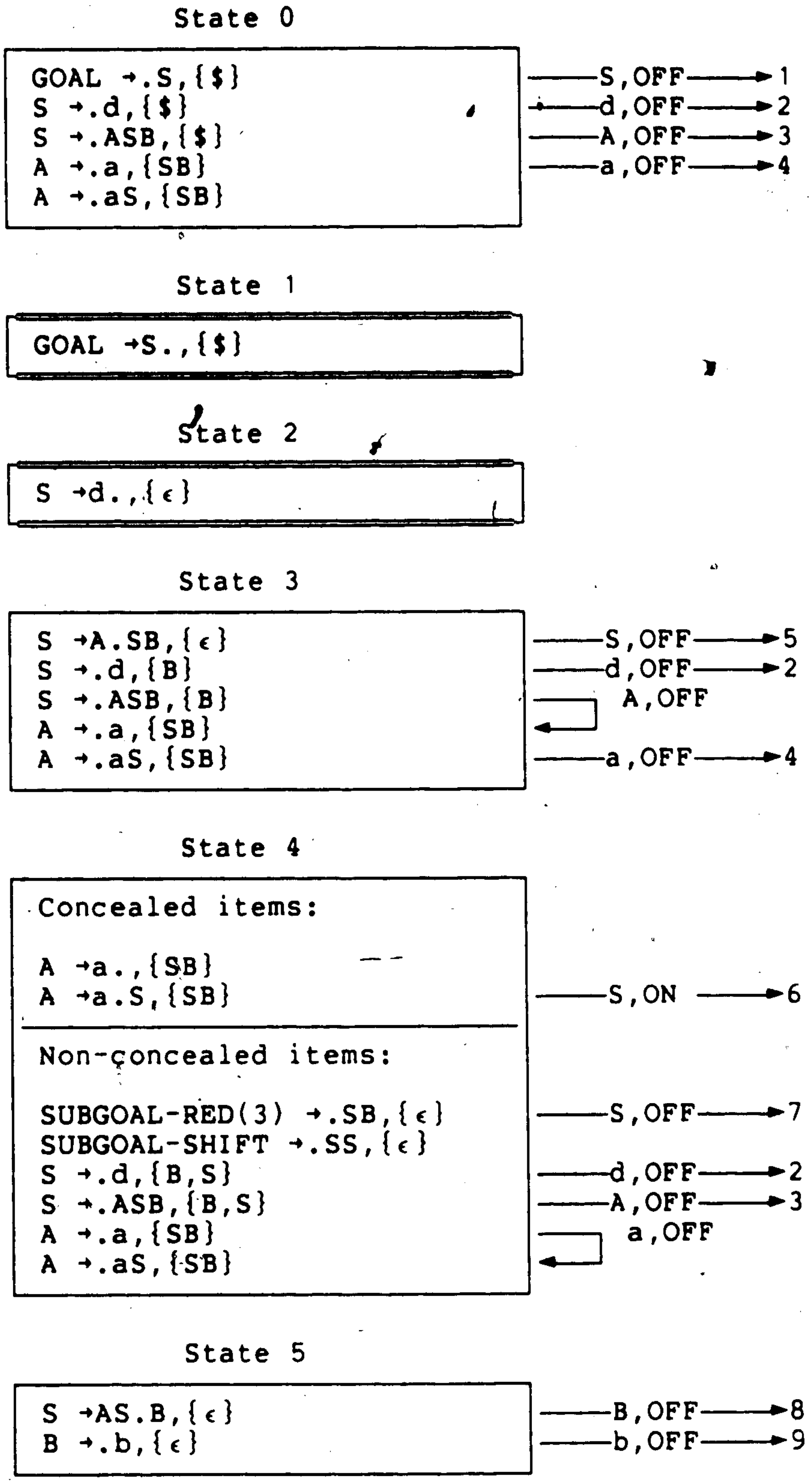
```
S →ASB.,{BB}
```

State 28

```
S →ASB.,{ε}
```

3.2.6 Post optimization of a CFSM

The output of the algorithm **A** in some cases, e.g., the CFSM produced for the grammar G, (example 2), is virtually more than what is needed to parse the grammar. By a suitable algorithm based on the intuition mentioned in the step (3)-a-(iv) of the algorithm **A**, one may collapse the states with longer lookahead strings onto the states with shorter lookahead strings. That is to say a state like $\{(A_i \rightarrow \alpha_i .\beta_i ,L_i)|i=1,...,n\}$ can be collapsed on a state like $\{(A_i \rightarrow \alpha_i .\beta_i ,L'_i)|i=1,...,n\}$, provided that $L'_i =\{\sigma'_{ij} |j=1,...,m\}$ and $L_i =\{\sigma'_{ij} \sigma_{ijl} |j=1,...,m$ and $l=1,...,m_{ij} \}$, or briefly, $L'_i =Min(L_i \cup L'_i)$ and L'_i is a prefix set of L, i.e., $L'_i =L\underline{\cap}L'_i$, for i=1,...,n. Such post optimization in no way changes the behaviour of the parser. The rationale behind this collapsing optimization is that if shorter lookahead strings are adequate in one state, then they are sufficient in the other state too. In the example 2, states 2,6,10 and 15 are collapsed on the state 20 giving new state 2. Similarly states {3,7,11,16}, {13} and {12,23,26,27} are collapsed on states 21,19 and 28 respectively resulting in the new states 3,9 and 8. The optimized CFSM is given in the next diagram.



Diag. 3.3
The optimized CFSM for G_1

State 6

```
A →aS.,{SB}
```

State 7

```
SUBGOAL-RED(3) →S.B,{ϵ}        ——B,OFF——→10
SUBGOAL-SHIFT →S.S,{ϵ}         ——S,OFF——→11
B →.b,{ϵ}                      ——b,OFF——→9
S →.d,{ϵ}                      ——d,OFF——→2
S →.ASB,{ϵ}                    ——A,OFF——→3
A →.a,{SB}                     ——a,OFF——→4
A →.aS,{SB}
```

State 8

```
S →ASB.,{ϵ}
```

State 9

```
B →b.,{ϵ}
```

State 10

```
SUBGOAL-RED(3) →SB.,{ϵ}
```

State 11

```
SUBGOAL-SHIFT →SS.,{ϵ}
```

Further to the previous optimization, final states 1,2,6,8,9,10 and 11 which indicate a reduction of a production may be removed from the CFSM, leaving the CFSM with only 5 transient states. Similar optimization regarding the final states (inessential states!) also can be found in LR parsers [Sois 82].

A second kind of optimization in the form of identification of states may be applied. Namely, one could collapse state $s=\{(A \rightarrow \alpha_i .\beta_i ,L_i)\}$ onto state $s'=\{(A \rightarrow \alpha_i .\beta_i ,L'_i)\}$ where $L'_i = \mathrm{Min}(L_i \cup L'_i)$ for $i=1,\ldots,|s|$, i.e., an item like $(A \rightarrow \alpha.\beta,\{X,X_2\})$ may be collapsed on item like $(A \rightarrow \alpha.\beta,\{X_1,Y_1\})$.

All the outlined optimizations do not change the character of the parsing mechanism or the class of grammars, except the last optimization technique may end up with a poor error detecting power, i.e., errors may be detected later than with CFSM that has no optimization of this kind. Such post optimizations could be embodied in the algorithm **A** with some elaboration of the algorithm.

A totally different merge optimization may also be carried out in the algorithm **A**, that is states $s=\{(A_i \rightarrow \alpha_i .\beta_i ,L_i)|i=1,\ldots,n\}$ and $s'=\{(A_i \rightarrow \alpha_i .\beta_i ,L'_i)|i=1,\ldots,n\}$ can be merged together giving state $s''=\{(A_i \rightarrow \alpha_i .\beta_i ,L''_i)|i=1,\ldots,n\}$ where $L''_i = \mathrm{Min}(L_i \cup L'_i)$ provided that there is no conflict in lookaheads of s", i.e.,

$(\{\beta_i \}+_k L''_i) \cap (\{\beta_j \}+_k L''_j)=\emptyset$ for $i,j=1,\ldots,n$ and $i \neq j$. Such mergings will require an elaborate use of back pointers to backtrack to an ancestor state if merging of the ancestor states results in conflicts in the successor states.

Such a method is similar in nature to the construction of LALR(k) parsers from LR(k) parsers. However, one may end up in a state with conflicting items such that adding auxiliary productions will not do any good since the lookaheads cannot be parsed. This latter optimization may also be included in the algorithm A in a way that is similar to the construction of an LALR(k) parser which is derived from LR(0)-CFSM. The corresponding class of grammars with this optimization will be more restrictive than LRRL(k) class as in this kind of parsing left contexts are collapsed together. These grammars will be analogous to LALR(k) grammars in canonical parsing.

3.2.7 Derivation of a parse table from the CFSM

The derivation of a parse table can be included in the construction of the CFSM, i.e. the algorithm A. However for clarity, here it has been given as a separate algorithm.

By observing the construction algorithm, it can be concluded that the non-concealed set of items in a CFSM state consists of either:

(a) one or more items of the form $A \rightarrow \alpha . X\beta, L$

or

(b) a single item of the form $A \rightarrow \alpha ., L$.

The case in (a) indicates a <u>shift-goto</u> operation on the FLAG value OFF and the appropriate input symbol X. The latter case (b) indicates the <u>reduction</u> of the production $A \rightarrow \alpha$ upon any input symbol and FLAG value OFF, except for the final state $f{:}[GOAL \rightarrow S., \{\$\}]$ where the input is required to be a \$. Notice that in other reduction states besides f, testing whether the input symbol is dominated by a symbol Y such that $Y=PF_1(\sigma)$ for $\sigma \in L \neq \{\epsilon\}$ would have given an advantage of early error detection in the parser.

In the case of a concealed set of items, only concealed items of the form $A \rightarrow \alpha.X\beta,L$ will determine a shift-goto operation on the appropriate input symbol X and FLAG value ON. The reduction operations indicated by a concealed item such as $A \rightarrow \alpha.,L$ are handled in a special way without referring to the state containing the concealed item as described below.

Observe that the states of the CFSM will not contain any item of the form SUBGOAL-RED(p) $\rightarrow \alpha.,L$ where p itself is an auxiliary production. So when a reduction of an auxiliary production say, SUBGOAL-RED(p) $\rightarrow \alpha$ in a state s under the FLAG value OFF, is encountered, the reduction of the original production p at an earlier state s' will automatically be concluded. The reduction of auxiliary productions cause their right hand sides, i.e., the parsed lookahead contexts to be transferred to the buffer.

Algorithm B1:

Computes a parse table from a given CFSM.

```
For i:=0,...,|Ω|-1 do with state $s_i$
for every symbol $X \in (V \cup \{\$\})$ do
(1) If the state $s_i$ contains a single nonconcealed
        item $A \rightarrow \alpha.,L$
    Then
```

```
Case A member of
  {GOAL}: If $X=\$$ Then TABLE($s_i$,X,OFF):=<reduce 0>
            (i.e., accept the input sentence.)
  N-{GOAL}: TABLE($s_i$,X,OFF):=<reduce p> where
            p: $A \rightarrow \alpha.$
            (for optimization and early error
            detection one may test if X is reachable
            from the first symbols in L.)
  {SUBGOAL-SHIFT}: (i.e., when the single item is of
            the form SUBGOAL-SHIFT $\rightarrow \alpha.,\{\epsilon\}.$)
            TABLE($s_i$,X,OFF):=<transfer $|\alpha|$,ON>
            (i.e., transfer $|\alpha|$symbols from top of
            the stack to the buffer and switch FLAG
            to ON. For optimization of the parse
            table one needs only to consider
            symbols $X\epsilon(T\cup\{\$\})$.)
  {SUBGOAL-RED($p_j$)|j=1,...,|P|}:
            (i.e., when the single item is of the
            form SUBGOAL-RED(p) $\rightarrow \alpha.,\{\epsilon\}.$)
            TABLE($s_i$,X,OFF):=<transfer $|\alpha|$,reduce p>
            (i.e., transfer $|\alpha|$ symbols from the top
            of the stack to the buffer and then
            reduce production p.)
 end case.
Else
 (i.e., when the state $s_i$ contains one or more
 non-concealed items $A \rightarrow \alpha.\beta$,L.)
```

```
        If the successor state of $s_i$ under FLAG value
        OFF and symbol X is $s_j$
        Then TABLE($s_i$,X,OFF):=<shift,goto $s_j$>.
    ([illegible]) If the state $s_i$ contains concealed items and its
        successor under symbol X and FLAG value ON is state $s'_j$
        Then
            TABLE($s_i$,X,ON):=<shift,OFF,goto $s'_j$>.
            (i.e., shift, switch FLAG to OFF and goto state $s_j$.)
end do
end do.
```

All the undefined entries are assumed to indicate the rejection of the input sentence.

The parse table can be represented as a three dimensional array for fast accessing. However to optimize the memory usage it may be kept as a collection of lists of non-null (i.e. defined) entries; one list for each state or one list for each symbol. Further in practice, the parse table entries can be encoded. Also note that FLAG values can be omitted from the table entries, provided that one sets FLAG to OFF after every shift operation, and switches it to ON —on a transfer operation but not on a combined transfer-reduce operation.

Example:

The following gives the parse table obtained from the optimized CFSM for the grammar G_1.

States	Symbols						
	a	b	d	S	A	B	$
0	s4		s2	s1	s3		
1							r0
2	r1	r1	r1	r1	r1	r1	r1
3	s4		s2	s5	s3		
4	s4		s2	s7 s6 OFF	s3		
5		s9				s8	
6	r4	r4	r4	r4	r4	r4	r4
7	s4	s9	s2	s11	s3	s10	
8	r2	r2	r2	r2	r2	r2	r2
9	r5	r5	r5	r5	r5	r5	r5
10	t2 r3	t2 r3	t2 r3	t2 r3	t2 r3	t2 r3	t2 r3
11	t2 ON	t2 ON	t2 ON	t2 ON	t2 ON	t2 ON	t2 ON

Note: The first line in each entry refers to the corresponding value under the FLAG value OFF and second line to the value under the FLAG value ON.

Table 3.2
First parse table for G,

As indicated the algorithm B1 does not take advantage of early error detection. If one wishes, one can do so by testing whether the input symbol is reachable from the first symbols of the lookahead strings in the states with a reduction item of the form $A \to \alpha.,L$ where $L \neq \{\epsilon\}$. In the states with single reduction item $A \to \alpha.,\{\epsilon\}$ the input symbol must be element of $T \cup \{\$\}$. Such a scheme will decrease the number of non-null entries in the parse table. For example the previous table will be reduced to Table [illegible].3.

States	Symbols						
	a	b	d	S	A	B	$
0	s4		s2	s1	s3		
1							r0
2	r1	r1	r1				r1
3	s4		s2	s5	s3		
4	s4		s2	s7 s6 OFF	s3		
5		s9				s8	
6	r4		r4	r4	r4		
7	s4	s9	s2	s11	s3	s10	
8	r2	r2	r2				r2
9	r5	r5	r5				r5
0	t2 r3	t2 r3	t2 r3				t2 r3
1	t2 ON	t2 ON	t2 ON				t2 ON

Table 3.3
Second parse table for G,

3.2.8 Optimized parse tables

In the construction of the CFSM it was mentioned that if one does not wish to have errors detected as early as possible, then final states except f:[GOAL →S.,{$}] can be omitted and replaced with an indication that corresponding productions are to be reduced. Algorithm B2 gives an optimized table according to the above scheme. It is important to note that parsing algorithms which use the optimized parse tables are slightly different from those that employ unoptimized tables.

The optimized tables assume that shifting of the input symbol onto the stack always takes place before any reduction or goto action. Therefore the entries in the table are of the forms: goto n, goto n-OFF, reduce n, reduce n-OFF, transfer n-reduce m, transfer n-ON or transfer n-reduce m-OFF.

Algorithm B2 :

Computes an optimized parse table from a given CFSM. Some errors may not be detected at the earliest possible moment during the parsing process, i.e., a number of reductions might take place before discovering that the next input symbol is erroneous.

```
For i:=0,...,|Ω|-1 do
   If $s_i$ consists of a single reduction item
      of the form $A \to \alpha.,L$
```

```
Then
   If $s_i = [\mathrm{GOAL} \to \mathbf{S}., \{\$\}]$
   Then TABLE($s_i$,$\$$,OFF):=<reduce 0>
   fi;
Else
   For FLAG-value:=OFF,ON do
      For every $X \in V$ do
         If $s_i$ has a successor $s_j$ under
            symbol X and FLAG-value
         Then
            If $s_j$ contains a single reduction
               item $A \to \alpha.,L$
            Then
             Case A member of
             {GOAL}: TABLE($s_i$,X,FLAG-value):=<goto $s_j$>.
             (N-{GOAL}):
                     TABLE($s_i$,X,FLAG-value):=
                     If FLAG-value=OFF Then <reduce p>
                        Else <reduce p,OFF>
                     fi; where p: $A \to \alpha$.
             {SUBGOAL-SHIFT}:
                     (i.e., when the single item
                     of $s_j$ is of the form
                     SUBGOAL-SHIFT $\to \alpha X.,\{\epsilon\}$.)
                     TABLE($s_i$,X,FLAG-value):=
                     <transfer $|\alpha X|$,ON>.
             {SUBGOAL-RED($p_j$ |j=1,...,|P|}:
                  (i.e., when the single item
                  of $s_j$ is of the form
                  SUBGOAL-RED(p) $\rightarrow \alpha . X, \{\epsilon\}.$)
                  TABLE($s_i$,X,FLAG-value):=
                  If FLAG-value = OFF Then
                      <transfer $|\alpha X|$,reduce p> Else
                      <transfer $|\alpha X|$,reduce p,OFF>
                  fi.
             end case.
            Else
             TABLE($s_i$,X,FLAG-value):=
               If FLAG-value=OFF Then <goto $s_j$>
                   Else <goto $s_j$,OFF>
               fi.
            fi
          fi
       end do
     end do
   fi
end do.
```

Table 3.4 is the optimized parse table for G, obtained according to the algorithm B2.

States	a	b	d	S	A	B	[illegible]
0	g4		r1	g1	g3		
1							r0
3	g4		r1	g5	g3		
4	g4		r1	g7 r4 OFF	g3		
5		r5				r2	
7	g4	r5	r1	t2 ON	g3	t2 r3	

Table 3.4
Optimized parse table for G_1

3.2.9 Parsing algorithm

The parsing algorithm varies according to the kind of parse table that it uses. The following is a parsing algorithm that employs an optimized parse table.

Algorithm C:

There are three data structures in addition to the parse table. Two parallel stacks; SS: Parser states stack and PS: Parsed symbols stack or parse stack. The states stack is used to store the past states of the parser, while the parse stack holds those nodes that are seeking a parent.

In LR theory, these stacks are combined together but in practical cases they are usually treated as two separate stacks. There is a buffer of size k that acts as a finite depth stack. The scanner routine obtains the next symbol from the input sentence. The input routine provides the next input symbol to the parser either from the buffer if it is not empty or else by invoking the scanner routine.

```
Variables:
  PS: Stack of symbols;
  SS: Stack of state numbers;
  Buffer: Stack of size k of symbols;
  FLAG: (OFF,ON)
  Current symbol: Symbol;
  Current state: State number;
  p: Production number;
  s: State number;

Initialize:
  FLAG:=OFF;
  PS:=Ø;
  SS:=Ø;
  Push (0) on SS;
  Buffer:=Ø;
  Current symbol:= input();
Loop:
  Current state:= Top(SS);
  Push (Current symbol) on PS;
```

```
If
 TABLE(Current state,Current symbol,FLAG) has
 the pattern <goto s>
Then
 Push (s) on SS; Current symbol:= input();
Elseif
 TABLE(Current state,Current symbol,FLAG) has
 the pattern <goto s,OFF>
Then
 Push (s) on SS; Current symbol:= input();
 FLAG:= OFF;
Elseif
 TABLE(Current state, Current symbol, FLAG) has
 the pattern <reduce p>
Then
 If p=0 Then accept; exit Loop;
 Else  Pop $|\alpha|-1$ states from SS;
       Pop $|\alpha|$ symbols from PS (i.e. reduce p)
       where $\alpha$ is the right hand side of the
       production p;
       Current symbol:= left hand side of production p;
 fi
Elseif
 TABLE(Current state,Current symbol,FLAG) has
 the pattern <reduce p,OFF>
Then
 Pop $|\alpha|-1$ states from SS;
 Pop |α| symbols from PS; (i.e. reduce p)
 where α is the right hand side of the production p;
 Current symbol:= left hand side symbol of p;
 FLAG:=OFF;
Elseif
 TABLE(Current state,Current symbol,FLAG) has
 the pattern <transfer l, ON>
Then
 For i:=1,...,l do Push(Pop(PS)) on Buffer;
 (i.e., transfer l top elements of the parse
  stack to the buffer.)
 Pop l-1 states from SS;
 FLAG:=ON;
 Current symbol:= Input();
Elseif
 TABLE(Current state,Current symbol,FLAG) has
 the pattern <transfer l, reduce p>
Then
 For i:=1,...,l do Push(Pop(PS)) on Buffer;
 Pop |α|+l-1 states from SS;
 Pop |α| symbols from PS; (i.e. reduce p.)
 where α is the right hand side of the production p;
 Current symbol:= left hand side symbol of p;
Elseif
 TABLE(Current state,Current symbol,FLAG) has
 the pattern <transfer l, reduce p, OFF>
Then
```

```
      For i:=1,...,l do Push(Pop(PS)) on Buffer;
      Pop $|\alpha|$+l-1 states from SS;
      Pop $|\alpha|$ symbols from PS; (i.e reduce p.)
      where $\alpha$ is the r.h.s. of p;
      Current symbol:= l.h.s. of the production p;
      FLAG:=OFF;
     Else
      (i.e., when TABLE(Current state,Current symbol,FLAG) is
       null.)
      reject the input sentence; exit loop;
     fi
end Loop;

Routine Input():Symbol;
 If Buffer ≠ Ø Then return(Pop(Buffer))
 Else return(Scanner);
end Input.

Routine Scanner():Symbol;
 If remainder of input sentence ≠ Ø Then
 return( CAR (remainder)); remainder:=CDR(remainder);
 Else reject the input sentence; exit;
end Scanner.
```

Example:

The following illustrates parsing of the sentence aadbaddbb ∈ L(G). The derivation tree of the sentence is also shown in the next figure.

State stack	Parse stack		Current symbol	Flag	Buff	Rem. of input
-	-		-	-	-	aadbaddbb$
0	-		a	OFF	-	adbaddbb$
0,4	a		a	"	-	dbaddbb$
0,4,4	aa		d	"	-	baddbb$
0,4,4	aad	reduce 1	S	"	-	baddbb$
0,4,4,7	aaS		b	"	-	addbb$
0,4,4,7	aaSb	reduce 5	B	"	-	addbb$
0,4,4,7	aaSB	transfer 2,reduce 3	A	"	SB	addbb$
0,4,3	aA		S	"	B	addbb$
0,4,3,5	aAS		B	"	-	addbb$
0,4,3,5	aASB	reduce 2	S	"	-	addbb$
0,4,7	aS		a	"	-	ddbb$
0,4,7,4	aSa		d	"	-	dbb$
0,4,7,4	aSad	reduce 1	S	"	-	dbb$
0,4,7,4,7	aSaS		d	"	-	bb$
0,4,7,4,7	aSaSd	reduce 1	S	"	-	bb$
0,4,7,4,7	aSaSS	transfer 2,ON	S	ON	S	bb$
0,4,7,4	aSaS	reduce 4,OFF	A	OFF	S	bb$
0,4,7,3	aSA		S	"	-	bb$
0,4,7,3,5	aSAS		b	"	-	b$
0,4,7,3,5	aSASb	reduce 5	B	"	-	b$
0,4,7,3,5	aSASB	reduce 2	S	"	-	b$
0,4,7	aSS	transfer 2,ON	S	ON	S	b$
0,4	aS	reduce 4,OFF	A	OFF	S	b$
0,3	A		S	"	-	b$
0,3,5	AS		b	"	-	$
0,3,5	ASb	reduce 5	B	"	-	$
0,3,5	ASB	reduce 2	S	"	-	$
0,1	S		$	"	-	-
0	reduce 0: accept					

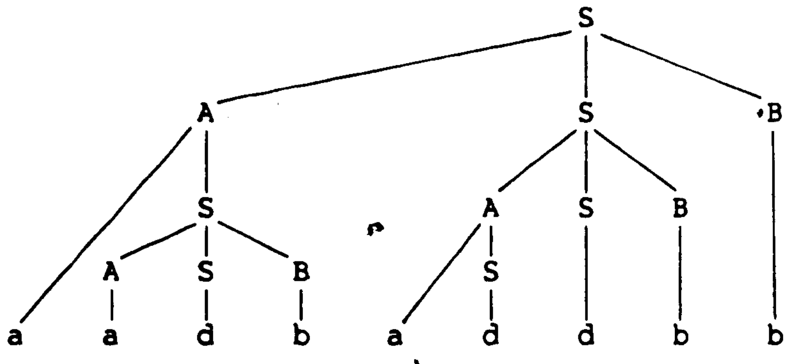


Fig 3.3
Parsing and derivation tree of $a^2dbad^2b^2$

3.2.10 Correctness of the algorithms

This section will prove that algorithms **A** through **C** correctly parse a grammar G which is LRRL(k). To prove the correctness of the construction algorithm **A**, intuitively it suffices to show that given a string $X_1 \ldots X_n \in V^*$, which is a prefix of some sentential form in SF(G), if one reaches the state s of the CFSM, then all the next moves indicated by this state are legitimate. Moreover, if there is a legitimate next move, it is included in the state s. In order to give a formal proof, one needs to introduce the concept of valid LRRL(k) items. Here again, items will be considered to be of the form $(A \rightarrow \alpha.\beta,\sigma)$, where σ is a string (as it was the case in the proof of the theorem 3.1). In fact, $(A \rightarrow \alpha.\beta,L)$ should be considered as a short notation for $\{(A \rightarrow \alpha.\beta,\sigma)|\sigma \in L\}$.

Definition: Valid LRRL(k) items for $X_1 \ldots X_n$

Suppose there exists a derivation:

$$GOAL\$ \overset{*}{\Longrightarrow} X_1 \ldots X_m A\gamma \Longrightarrow X_1 \ldots X_m \alpha\gamma'$$

such that γ does not contain a complete phrase of G and $X_1 \ldots X_m \alpha = X_1 \ldots X_n$ (not excluding $\alpha=\epsilon$ and $n=m$), i.e., rewriting of A produces the rightmost phrase in the sentential form $X_1 \ldots X_m \alpha\gamma'$. Either

(1) $|\gamma'| \geq |\gamma|$, i.e., $\gamma' = \beta\gamma$ and $A \rightarrow \alpha\beta$. Then $(A \rightarrow \alpha.\beta,\sigma)$, where σ is some prefix of γ with $|\sigma| \leq k$, or (SUBGOAL-SHIFT $\rightarrow \alpha'.\beta',\epsilon$), where α' is a suffix α and β' is a prefix of $\beta\gamma$ with $|\alpha'\beta'| \leq k$, is a valid LRRL(k) item for

$X_1 \ldots X_n$.

Or

(2) $|\gamma'|<|\gamma|$, i.e., $A \rightarrow \alpha'$, where $\alpha' = X_{m+1} \ldots X_{m+j-1}$ is a prefix of α and $m+j \leq n$. Then ($\text{SUBGOAL-RED}(A \rightarrow \alpha') \rightarrow X_{m+j} \ldots X_n . \beta', \epsilon$), where β' is a prefix of γ' and $|X_{m+j} \ldots X_n \beta'| \leq k$, or ($\text{SUBGOAL-SHIFT} \rightarrow X_{m+i} \ldots X_n . \beta', \epsilon$), where $i>j$, is a valid LRRL(k) item for $X_1 \ldots X_n$.

A valid item ($A \rightarrow \alpha . \beta, \sigma$) for $X_1 \ldots X_n$ is called a valid non-auxiliary item for $X_1 \ldots X_n$ if $A \in N$. Otherwise, it is a valid auxiliary item. If $\alpha \neq \epsilon$ (i.e., α includes X_n), then the item is a valid (original) basis item.

Theorem 3.2:

If state s is a successor of the initial state s, under the string $X_1 \ldots X_n$, then all the items in s are valid for $X_1 \ldots X_n$.

Proof: The proof is by induction on n. However, one first should note that the validity of all the items in a state follows from the validity of the original basis items in that state. Suppose a basis item ($A \rightarrow X_{m+j} \ldots X_n . \beta, \sigma$) is valid for $X_1 \ldots X_n$. Then either

(1) it is a non-auxiliary item, i.e., there exists a derivation

$$\text{GOAL}\$ \overset{*}{\Longrightarrow} X_1 \ldots X_m A\gamma \Longrightarrow X_1 \ldots X_m \ldots X_n \beta\gamma$$

such that γ does not contain a complete phrase, σ is a prefix of γ and $j=1$. Now, if as a result of conflict in the state, an item of the form (SUBGOAL-RED($A \rightarrow X_{m+j} \ldots X_n$) $\rightarrow .\sigma,\epsilon$) (when $\beta=\epsilon$), or (SUBGOAL-SHIFT $\rightarrow .\sigma',\epsilon$) (where σ' is a prefix of $\beta\sigma$) is added to the state, then the new item will be valid for $X_1 \ldots X_n$.

Or (2) it is an auxiliary item, i.e., A is some subgoal symbol and there exists a derivation

$$GOAL\$ \overset{*}{\Longrightarrow} X_1 \ldots X_m B\gamma \Longrightarrow X_1 \ldots X_m \ldots X_n \gamma',$$

$\sigma=\epsilon$ and β is a prefix of γ'. Again, if as a result of conflict in the state, the item (SUBGOAL-SHIFT $\rightarrow .\beta,\epsilon$) is added to the state, this item will be valid for $X_1 \ldots X_n$.

Furthermore, suppose an item ($A \rightarrow \alpha .B\beta,\sigma$) is valid for $X_1 \ldots X_n$, i.e., there exists a derivation

$$GOAL\$ \overset{*}{\Longrightarrow} X_1 \ldots X_m C\gamma \Longrightarrow X_1 \ldots X_m \alpha'\gamma'$$

such that γ does not contain a complete phrase, $X_1 \ldots X_m \alpha' = X_1 \ldots X_n$, α is a suffix of α' and $B\beta\sigma$ is a prefix of γ'. Then, if as a result of closure operation the item ($B \rightarrow .\beta_1,\sigma'$), where σ' is a prefix of $\beta\sigma$ with $|\sigma'| \leq k$, is added to the state, this new item will be valid for $X_1 \ldots X_n$, because there is a derivation

$$GOAL\$ \overset{*}{\Longrightarrow} X_1 \ldots X_m C\gamma \Longrightarrow X_1 \ldots X_m \alpha'\gamma' = X_1 \ldots X_n B\beta\gamma'' \Longrightarrow X_1 \ldots X_n B\beta_1\beta\gamma''$$

and rewriting of $B \rightarrow \beta_1$ produces the rightmost phrase in the sentential form.

Thus, validity of the added auxiliary items and those items that are produced as the result of closing operation, follows from the validity of the original basis items. Now, the inductive proof can be stated.

For n=0, state s_0 contains (GOAL →.S,\$) which is trivially valid for ϵ. All the other items in that state are introduced by closure operation on this item. Therefore all the items in s_0 are valid for ϵ. Suppose the induction hypothesis is true for $X_1 \ldots X_{n-1}$ and the state t predecessor of s. Since the basis items of s are obtained from valid items $(A \to \alpha.X_n \beta,\sigma)$ for $X_1 \ldots X_{n-1}$ in t, by shifting the dot to the right of X_n, these items are valid for $X_1 \ldots X_n$. Moreover, any other item that is added to s, either due to conflict in s or due to closure operation, is valid for $X_1 \ldots X_n$. □

Theorem 3.3:

Let state s be a successor of the initial state s_0 under the string $X_1 \ldots X_n$. The following statements are true.

(1) If there exists a derivation

$$GOAL\$ \overset{*}{\Longrightarrow} X_1 \ldots X_m A\gamma \Longrightarrow X_1 \ldots X_m X_{m+1} \ldots X_n \gamma',$$

such that γ does not contain a complete phrase of G, then

(i)- if $\gamma'=\beta\gamma$ (i.e., $|\gamma'|\geq|\gamma|$ and $A \to X_{m+1} \ldots X_n \beta$) then s includes a valid basis item (for $X_1 \ldots X_n$) either of the form $(A \to X_{m+1} \ldots X_n .\beta,\sigma)$, where σ is some prefix of γ and $|\sigma|\leq k$, or of the form $(SUBGOAL \to X_{m+j} \ldots X_n .\beta',\epsilon)$, where β' is a prefix of $\beta\gamma$,

(ii)- else (i.e., when $|\gamma'|<|\gamma|$ and $A \to X_{m+1} \ldots X_{m+j-1}$ with $m+j\leq n$), s contains a valid basis item (for $X_1 \ldots X_n$) either of the form

(SUBGOAL-RED($A \rightarrow X_{m+1} \ldots X_{m+j-1}$) $\rightarrow X_{m+j} \ldots X_{n} .\beta,\epsilon$), where β is a prefix of γ', or of the form (SUBGOAL-SHIFT $\rightarrow X_{m+i} \ldots X_{n} .\beta,\epsilon$), where $i>j$.

(2) If s is not a final state and there exists a derivation

$$GOAL\$ \overset{*}{\Longrightarrow} X_{1} \ldots X_{n} A\gamma \Longrightarrow X_{1} \ldots X_{n} \beta\gamma$$

such that γ does not include a complete phrase, then s includes a valid non-basis item for $X_{1} \ldots X_{n}$ of the form $(A \rightarrow .\beta,\sigma)$, where σ is a prefix of γ with $|\sigma| \leq k$.

Proof: The proof is by induction on n. For n=0, s_0 contains (GOAL $\rightarrow .S,\$$), and by closure operation includes any valid non-basis item for ϵ. Thus (1) and (2) are vacuously true for s_0 and ϵ. Suppose (1) and (2) are true for state t, predecessor of s, and $X_{1} \ldots X_{n-1}$. To show that the induction hypothesis is true for s and $X_{1} \ldots X_{n}$, a proof may be carried out in two steps for (1) and (2).

Step 1: Suppose there exists a derivation

$$GOAL\$ \overset{*}{\Longrightarrow} X_{1} \ldots X_{m} A\gamma \Longrightarrow X_{1} \ldots X_{m} \ldots X_{n} \gamma'$$

such that γ does not contain a complete phrase. Consider two cases:

Case (i): $|\gamma'| \geq |\gamma|$, i.e., $\gamma' = \beta\gamma$. Two subcases may be distinguished.

Case (i-1): $m+1=n$, i.e., $A \rightarrow X_{n} \beta$. Since t is not a final state, by the truth of (2) for t and $X_{1} \ldots X_{n-1}$, t must contain a valid non-basis item $(A \rightarrow .X_{n} \beta,\sigma)$ for $X_{1} \ldots X_{n-1}$, where σ is a prefix of γ. Therefore s includes the valid basis item $(A \rightarrow X_{n} .\beta,\sigma)$.

Case(i-2): m+1<n, i.e.,

$A \to X_{m+1} \ldots X_{n-1} X_n \beta$. Since t is not a final state, by the truth of (1), t contains a valid basis item for $X_1 \ldots X_{n-1}$, either of the form $(A \to X_{m+1} \ldots X_{n-1} . X_n \beta, \sigma)$, where σ is a prefix of γ, or of the form $(\text{SUBGOAL-SHIFT} \to X_{m+j} \ldots X_{n-1} . X_n \beta', \epsilon)$, where β' is a prefix of $\beta\gamma$ and j>1. Now, if there is no conflict in t or s is successor of t under flag value ON, then s must include $(A \to X_{m+1} \ldots X_{n-1} X_n . \beta, \sigma)$ or $(\text{SUBGOAL-SHIFT} \to X_{m+j} \ldots X_{n-1} X_n . \beta', \epsilon)$. Else, there was a conflict and s is the successor of t under flag value OFF, then an item $(\text{SUBGOAL-SHIFT} \to . X_n \beta'', \epsilon)$, where β'' is β' or a prefix of $\beta\sigma$, must have been added to t. So, s includes valid basis item $(\text{SUBGOAL-SHIFT} \to X_n . \beta'', \epsilon)$ for $X_1 \ldots X_n$.

Therefore (1) is true for s and $X_1 \ldots X_n$ when $|\gamma'|>|\gamma|$.

Case (ii): $|\gamma|'<|\gamma|$, i.e., $A \to X_{m+1} \ldots X_{m+j-1}$, (m+j≤n). Again two subcases may be distinguished.

Case (ii-1): m+j<n. Since t is not a final state, by the truth of (1)(case ii) for t and $X_1 \ldots X_{n-1}$, it contains a valid basis item for $X_1 \ldots X_{n-1}$, either of the form $(\text{SUBGOAL-RED}(A \to X_{m+1} \ldots X_{m+j-1}) \to X_{m+j} \ldots X_{n-1} . X_n \beta, \epsilon)$ or $(\text{SUBGOAL-SHIFT} \to X_{m+i} \ldots X_{n-1} . X_n \beta, \epsilon)$, where β is a prefix of γ' and i>j. If there is no conflict in t or s is the successor of t under flag value ON, then s includes the valid basis item $(\text{SUBGOAL-RED}(A \to X_{m+1} \ldots X_{m+j-1}) \to X_{m+j} \ldots X_{n-1} X_n . \beta, \epsilon)$ or $(\text{SUBGOAL-SHIFT} \to X_{m+i} \ldots X_{n-1} X_n . \beta, \epsilon)$ for $X_1 \ldots X_n$. Else, there is a conflict in t and s is the successor of t under flag value OFF, then the item (SUBGOAL-SHIFT $\rightarrow .X_n\ \beta,\epsilon$) is added to t. So s includes the valid basis item (SUBGOAL-SHIFT $\rightarrow X_n\ .\beta,\epsilon$).

Case (ii-2): m+j=n. Since t is not a final state, by the truth of (1) (case i) for t and $X_1 \ldots X_{n-1}$, t contains a valid basis item for $X_1 \ldots X_{n-1}$, either of the form ($A \rightarrow X_{m+1} \ldots X_{n-1}.,X_n\ \beta$), or of the form (SUBGOAL-SHIFT $\rightarrow X_{m+i} \ldots X_{n-1}\ .X_n\ \beta,\epsilon$), where β is a prefix γ' and i>1. If it was of the first form, since t is not a final state the item (SUBGOAL-RED($A \rightarrow X_{m+1} \ldots X_{n-1}$) $\rightarrow .X_n\ \beta,\epsilon$) is added to t. Then, if s is the successor of t under flag value OFF, it includes the valid basis item (SUBGOAL-RED($A \rightarrow X_{m+1} \ldots X_{n-1}$) $\rightarrow X_n\ .\beta,\epsilon$). Otherwise (when the item in t is of the second form), if there is no conflict in t or s is the successor of t under flag value ON, then s contains the valid basis item (SUBGOAL-SHIFT $\rightarrow X_{m+i} \ldots X_{n-1}\ X_n\ .\beta,\epsilon$). Else, there is a conflict in t and s is the successor of t under flag value OFF. Then item (SUBGOAL-SHIFT $\rightarrow .X_n\ \beta,\epsilon$) is added to t. So, s includes the valid basis item (SUBGOAL-SHIFT $\rightarrow X_n\ .\beta,\epsilon$) for $X_1 \ldots X_n$.

Therefore, (1) is true for s and $X_1 \ldots X_n$ when $|\gamma'|<|\gamma|$.

Step 2: Suppose there exists a derivation

$$\text{GOAL\$} \overset{*}{\Longrightarrow} X_1 \ldots X_n\ A\gamma \Longrightarrow X_1 \ldots X_n\ \beta\gamma$$

such that γ does not contain a complete phrase of G. Let $D=D_m \rightarrow D_{m-1}\ \beta_m$, ..., $D_1 \rightarrow D_0\ \beta_1 = A\beta$ (possibly m=0) be the productions that are used in rewriting of sentential form to introduce A such that

$$GOAL\$ \overset{*}{\Longrightarrow} X_1 \ldots X_n D_m\gamma \overset{*}{\Longrightarrow} X_1 \ldots X_n A\gamma,$$

and $D_m\gamma$ does not contain a complete phrase of G. Let C be the non-terminal that by rewriting it, the rightmost phrase in $X_1 \ldots X_n D_m\gamma$ is introduced, i.e.,

$$GOAL\$ \overset{*}{\Longrightarrow} X_1 \ldots X_j C_{m+1}\gamma \Longrightarrow X_1 \ldots X_j \ldots X_n D_m\gamma .$$

If the right hand side of the C-production includes D, i.e., $C \to X_{j+1} \ldots X_n D\beta_{m+1}$, then by step 1 (case i) and the fact that s is not a final state, this state includes a valid basis item of the form $(C \to X_{j+1} \ldots X_n .D\beta_{m+1}, \sigma)$ or $(SUBGOAL\text{-}SHIFT \to X_{j+i} \ldots X_n .D\beta', \epsilon)$, where i>1. Otherwise, $C \to X_j \ldots X_i$, where i≤n. Then by step 1 (case ii), s includes an item of the form (**subgoal** $\to \alpha.D\beta', \epsilon$), where α is a suffix of $X_{i+1} \ldots X_n$ and **subgoal** is either SUBGOAL-SHIFT or SUBGOAL-RED($C \to X_j \ldots X_i$). Thus, in any case s includes an item with dot to the left of D. Now, by closure operation s contains non-basis items $(D \to .D_{m-1}\beta_m, \sigma_m), \ldots, (D_1 \to . \to A\beta_1, \sigma_1)$ and $(A \to .\beta, \sigma)$. Therefore (2) is true for non-final state s and string $X_1 \ldots X_n$. □

Theorems 3.2 and 3.3 insure that given the string $X_1 \ldots X_n$, if one reaches the state s of the CFSM, then this string must be a prefix of some sentential form in SF(G). Furthermore, if $X_1 \ldots X_n X_{n+1} \ldots X_{n+m}$ is a sentential form in SF(G) and s is a successor of s_0 under $X_1 \ldots X_n$. Then, by considering the derivation

$$GOAL\$ \overset{*}{\Longrightarrow} X_1 \ldots X_j A\gamma \Longrightarrow X_1 \ldots X_n \gamma' \overset{*}{\Longrightarrow} X_1 \ldots X_n X_{n+1} \ldots X_{n+m} \$,$$ where γ does not contain a complete phrase, one concludes that

either (1) s is a final state implying reduction of the production $A \rightarrow X_{j+1} \ldots X_{n}$ or the right context for a deferred decision, or (2) s is a transient state containing a valid item with dot to the left of X_{n+1}, i.e., a valid next move on this symbol. Therefore the result of combining theorems 3.2 and 3.3 can be stated as:

Corollary 3.2: The construction algorithm A is correct.

One may note that there is a selective subset of sentential forms that have prefixes under which there is a successor state of s_0 in CFSM. Obviously, this subset includes all the sentences in L(G) and any sentential form which is produced by the parsing algorithm C. In LR(k) parsing, this selective set includes the right sentential forms, and their prefixes under which there is a successor state are the viable prefixes.

Algorithms B1 and B2 only reproduce the information contained in a CFSM, in a tabular form. Therefore their correctness is trivial. So, the correctness of the parsing algorithm is stated next.

Theorem 3.4:

Given a valid LRRL(k)-CFSM for G, Algorithm C correctly parses all and only the sentences in L(G).

Proof: The correctness of the reductions made by the algorithm is guaranteed by the construction algorithm.

Therefore to prove that sentences in L(G) are patsed correctly, it suffices to show that:

(i) Given a sentence in L(G), Algorithm C finds a phrase in it and reduces this phrase.

(ii) Having made a reduction, there is a valid next move.

(iii) The algorithm terminates.

(i)-

Suppose $a_1 \ldots a_m \$$ is a sentence in L(G)\$. Algorithm C by scanning an initial segment of the input, say $a_1 \ldots a_n$, reaches a final state. Three cases are possible.

(1) It reduces an original phrase of G, say $A \rightarrow a_i \ldots a_n$. In this way the sentential form $a_1 \ldots a_{i-1} A a_{n+1} \ldots a_m \$$ will be produced.

(2) It recognizes the right context of a phrase, say $A \rightarrow a_i \ldots a_j$, where $n-k \leq j < n$. After storing the lookahead string in buffer, the algorithm reduces this phrase. In this case the sentential form $a_1 \ldots a_{i-1} A a_{j+1} \ldots a_n \ldots a_m \$$ will be obtained.

(3) It recognizes the right context for a deferred shift operation at an earlier symbol say a_j, where $n-k \leq j < n$. In this case the algorithm retracts to the symbol a_{j+1}, and by switching the flag to ON takes a different path. Eventually the algorithm will find a phrase, because there is no state in which a shift beyond \$ symbol is possible.

(ii)-

Suppose the algorithm decides to reduce the phrase $X_i \ldots X_j$ in the sentential form $X_1 \ldots X_i \ldots X_j \ldots X_m \$$ to the non-terminal A. In doing so, it will pop $X_i \ldots X_j$ off the symbol stack. Also, a number of states will be dropped from the state stack, such that state s, where X_{i-1} was just shifted, becomes the top node on the stack. Since state s is not a final state and there exists a derivation

$$\mathrm{GOAL}\$ \overset{*}{\Longrightarrow} X_1 \ldots X_{i-1} \; A\gamma \overset{*}{\Longrightarrow} X_1 \ldots X_{i-1} \; X_i \ldots X_j \; \gamma \Longrightarrow$$
$$X_1 \ldots X_{i-1} \; X_i \ldots X_j \; X_{j+1} \ldots X_m \; ,$$

(where γ does not contain a complete phrase), state s must contain a valid item for $X_1 \ldots X_{i-1}$ in which the dot appears to the left of symbol A. Therefore there is a valid next move on A in the form of a shift operation.

(iii)-

To show the termination of the algorithm in a finite number of steps, one should note that a segment of a sentential form can be reduced only a finite number of times by itself. This is true by the unambiguity of the grammar, a property of LRRL(k) grammars which is to be proved in Chapter 4, that rules out existence of chain productions in the grammar. So, if the segment $X_i \ldots X_j$ is first reduced [illegible] D_1, then D_1 is reduced to D_2,..., and D_{m-1} is reduced to D_m, there can be no repetition of symbols in $D_1 \ldots D_m$, and thus $m \leq |N|$. Therefore, after a finite number of reductions the length of the sentential form is reduced at least by one, and eventually the sentential form will be reduced to S$.

To see that Algorithm C rejects those inputs which are not valid sentences in L(G)$, consider the string $a_1 \ldots a_n b_1 \ldots b_m \$ \in T^*$, in which $a_1 \ldots a_n$ is the longest correct prefix. By the assumption b_1 is the first erroneous symbol in the input, and there does not exist a derivation $GOAL\$ \overset{*}{\Longrightarrow} a_1 \ldots a_n b_1 \delta$. Let $X_1 \ldots X_i$ be the string that is produced by reductions on $a_1 \ldots a_n$ such that no more reductions on it by the algorithm are possible. At this point, the algorithm reaches a non-final state s which is a successor of the state s_0 under the string $X_1 \ldots X_i$. This state does not contain a valid item according to which the symbol b_1 can be shifted. Otherwise, there is a derivation:

$$GOAL\$ \overset{*}{\Longrightarrow} X_1 \ldots X_j A\gamma \Longrightarrow X_1 \ldots X_i b_1 \gamma' \overset{*}{\Longrightarrow} a_1 \ldots a_n b_1 \gamma',$$

(where γ does not contain a complete phrase, and $j \le i$), which contradicts the assumption. Therefore, there is no valid next move on b_1 in the state s and the algorithm rejects the input. □

Now the results of the theorems 3.2 to 3.4 can be summed up in the following corollary.

Corollary 3.3

Algorithms A through C correctly parse a grammar G if and only if it is LRRL(k).

3.2.11 Type II basic LRRL(k) grammars

The development of the type I basic LRRL(k) grammars closely followed the design philosophy of Marcus' parser in a purely bottom-up parsing technique. In Marcus parsing, when a shift operation (i.e. an attachment) is deferred, and after parsing the right context it was discovered that this operation was appropriate, the parser retracts to the deferred shift point. This is in line with the general view in that parser that the role of a construct is determined by three constructs to its right. Furthermore, Marcus' parser is partially top-down and most of the time it creates a parent node before having a complete set of its daughters available. Therefore, if there is a reduce-shift conflict on productions $A \rightarrow \alpha$ and $A \rightarrow \alpha\beta$ after attaching α to the parent node A, it continues with the parsing of the lookaheads without attaching any new node to A. If the right context resolves the conflict in favour of the longer production, the parser backtracks to the first symbol of β and continues with the attachment of nodes to the parent node A.

As was seen, the parsers for the type I LRRL(k) grammars, like the Marcus parser, backtrack over the reduced right context to a point where a shift operation was in question. However, despite some early semantic checking advantages, one may argue that for syntactic recognition such backtracking in a bottom-up parser is unnecessary. In fact, it introduces some inefficiency in the parsing algorithm. This point was mentioned in the beginning of Section 3.2, and it was observed that a CFSM can be constructed without the use of **FLAG** variable and **SUBGOAL-SHIFT** auxiliary productions.

The following algorithm describes the construction of a CFSM according to above scheme. In this scheme, the final states correspond to the recognition of either original rules or the right contexts for them. The resulting class of grammars will be called type II (basic) LRRL(k) grammars and they will be shown are slightly more general than type I grammars.

Algorithm A-II:

(1) Add **GOAL** symbol to N and production
    0: GOAL →S.to P.

(2) Build the initial state s_0:
        Let the basis of s_0 be { (GOAL →.S,{$}) }.
        Close the basis of s_0.
        Let the set of states of CFSM, $\Omega=\{s_0\}$.

(3) <u>Repeat</u>
        For a state t whose successors are not yet
        determined build its successor states under
        all applicable symbol X.

Construction of a successor state s for the given state t under symbol X:

(i) If there is no item of the form $(A \to \alpha.X\beta,L)$ in t, then t has no successor under symbol X.

Else

let basis of $s=\emptyset$.

For each item $(A \to \alpha.X\beta,L)$ in t add $(A \to \alpha X.\beta,L)$ to the basis of s.

(ii) Check for inadequacy in the basis of s:

If there are items of the form

$I_j : A_j \to \alpha_j X.\beta_j ,L_j ;$
$j=1,...,m$ and $\beta_j \neq \epsilon$

and

$I_j : A_j \to \alpha_j X.,L_j ;$
$j=m+1,...,n$

where $m\geq 1$ and $n>m$

(i.e. shift-reduce/reduce-reduce conflicts)

or

$m=0$ and $n>1$

(i.e. only reduce-reduce conflicts)

Then

let the shift lookaheads of s be

$SHL(s)=\text{Min} \bigcup_{j=1}^{m} (\{\beta_j\} +_k L_j).$

If $L_j \cap SHL(s) = \emptyset$ for $j=m+1,...,n$

```
        (i.e. possible shift-reduction resolution)
          and
          $L_i \cap L_j = \emptyset$ for $i,j=m+1,\ldots,n$ , $i \neq j$
        (i.e. possible resolution of reduce-reduce
        conflicts)
      Then
          conceal (delete) the conflicting basis
          items $I_{m+1},\ldots,I_n$ and
          add new basis items corresponding
          to their lookaheads, i.e.,
          For $j=m+1,\ldots,n$
              delete $I_j : A_j \to \alpha_j X_j.,L_j$ and
              add SUBGOAL-RED($p_j$) $\to .\gamma,\{\epsilon\}$
              to s for all $\gamma \in L_j$ ;
              where $p_j$ is the production $A_j \to \alpha_j X_j$.
      Else
          conclude the grammar is not type II
          basic LRRL(k) and exit.

(iii) Close the basis of s.

(iv)  Add state s to $\Omega$ if there is no state t'
      in $\Omega$ with the same items as s, or with
      items which match those of s apart from the
      lookahead sets, and for lookahead set L'
      in t' it is true that $L'=Min(L \cup L')$ and
      $L'=L \cap L'$,  where L is the corresponding
```

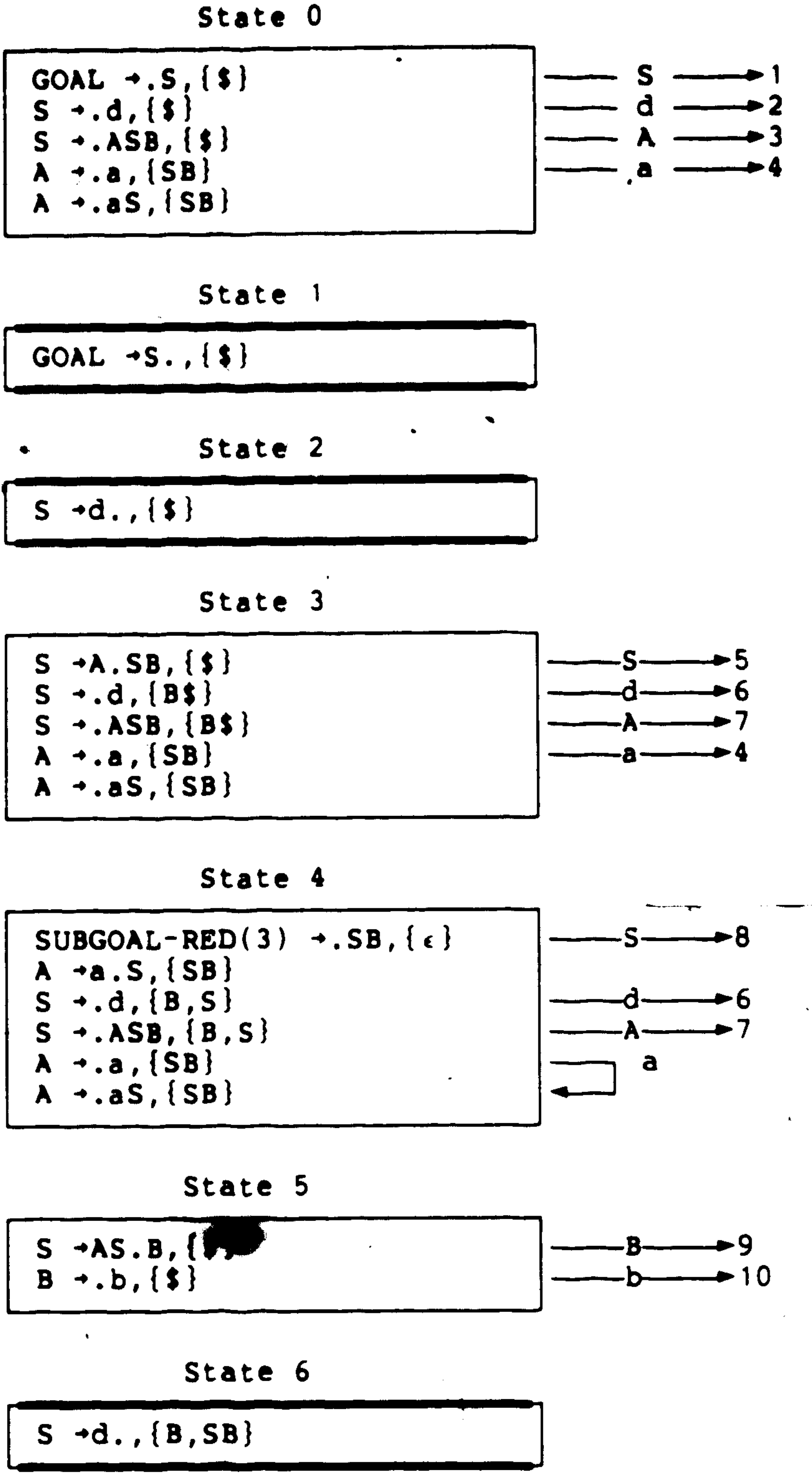
```
        lookahead set in s.
        Otherwise let t' be the successor of t.

  Until no more state can be added to Ω.
```

(4) Conclude that the grammar is type II basic LRRL(k).

Definition A-II

A context-free grammar G=(N,T,P,S) is type II basic LRRL(k) iff a CFSM for it can be constructed according to the algorithm A-II.

The following diagram shows the CFSM for G, constructed according to the algorithm A-II with k=2, after collapsing optimizations of first and second kinds are applied to it.



State 0

```
GOAL →.S,{$}
S →.d,{$}
S →.ASB,{$}
A →.a,{SB}
A →.aS,{SB}
```

State 1

```
GOAL →S.,{$}
```

State 2

```
S →d.,{$}
```

State 3

```
S →A.SB,{$}
S →.d,{B$}
S →.ASB,{B$}
A →.a,{SB}
A →.aS,{SB}
```

State 4

```
SUBGOAL-RED(3) →.SB,{ε}
A →a.S,{SB}
S →.d,{B,S}
S →.ASB,{B,S}
A →.a,{SB}
A →.aS,{SB}
```

State 5

```
S →AS.B,{[illegible]
B →.b,{$}
```

State 6

```
S →d.,{B,SB}
```

Diag. 3.4
The type II CFSM for G_1

State 7

```
S →A.SB,{B,$B}       ——S——→11
S →.d,{BB,BS}        ——d——→6
S →.ASB,{BB,BS}      ——┐   A
A →.a,{SB}           ←—┘
A →.aS,{SB}          ——a——→4
```

State 8

```
SUBGOAL-RED(3) →S.B,{ε}     ——B——→12
SUBGOAL-RED(4) →.SB,{ε}     ——S——→13
B →.b,{ε}                   ——b——→10
S →.d,{B}                   ——d——→6
S →.ASB,{B}                 ——A——→7
A →.a,{SB}                  ——a——→4
A →.aS,{SB}
```

State 9

```
S →ASB.,{$}
```

State 10

```
B →b.,{ε}
```

State 11

```
S →AS.B,{B,SB}       ——B——→14
B →.b,{B,SB}         ——b——→10
```

State 12

```
SUBGOAL-RED(3) →SB.,{ε}
```

State 13

```
SUBGOAL-RED(4) →S.B,{ε}     ——B——→15
B →.b,{ε}                   ——b——→10
```

State 14

```
S →ASB.,{B,SB}
```

State 15

SUBGOAL-RED(4) →SB.,{ε}

Table construction and parsing algorithm for type II grammars are similar to those in type I grammars, except that no flag variable and backtracking to a bypassed questionable shift point are involved here. Table 3.5 is the parse table obtained from the CFSM in Diagram 3.4 by omitting the final states except s_1. Figure 3.4 illustrates the parsing of the sentence aadbaddbb$\in G_1$, when G_1 is considered as a type II basic LRRL(2) grammar.

	Symbols						
States	a	b	d	S	A	B	$
	g4		r1	g1	g3		
1							r0
3	g4		r1	g5	g7		
4	g4		r1	g8	g7		
5		r5				r2	
7	g4		r1	g11	g7		
8	g4	r5	r1	g13	g7	t2 r3	
11		r5				r2	
13		r5				t2 r4	

Table 3.5
Optimized parse table for G_1

State stack	Parse stack		Current symbol	Buff	Remainder of input
-	-		-	-	aadbaddbb$
0	-		a	-	adbaddbb$
0,4	a		a	-	dbaddbb$
0,4,4	aa		d	-	baddbb$
0,4,4	aad	reduce 1	S	-	baddbb$
0,4,4,8	aaS		b	-	addbb$
0,4,4,8	aaSb	reduce 5	B	-	addbb$
0,4,4,8	aaSB	transfer 2,reduce 3	A	SB	addbb$
0,4,7	aA		S	B	addbb$
0,4,7,11	aAS		B	-	addbb$
0,4,7,11	aASB	reduce 2	S	-	addbb$
0,4,8	aS		a	-	ddbb$
0,4,8,4	aSa		d	-	dbb$
0,4,8,4	aSad	reduce 1	S	-	dbb$
0,4,8,4,8	aSaS		d	-	bb$
0,4,8,4,8	aSaSd	reduce 1	S	-	bb$
0,4,8,4,8,13	aSaSS		b	-	b$
0,4,8,4,8,13	aSaSSb	reduce 5	B	-	b$
0,4,8,4,8,13	aSaSSB	transfer 2,r[illegible]	A	SB	b$
0,4,8,7	aSA		S	B	b$
0,4,8,7,11	aSAS		B	-	b$
0,4,8,7,11	aSASB	reduce 2	S	-	b$
0,4,8,13	aSS		b	-	$
0,4,8,13	aSSb	reduce 5	B	-	$
0,4,8,13	aSSB	transfer 2,reduce 4	A	SB	$
0,3	A	S B	$		
0,3,5	AS		B	-	$
0,3,5	ASB	reduce 2	S	-	$
0,1	S	$ -	-		
0	reduce 0: accept				

Fig 3.4
Parsing of the sentence aadbaddbb
by the type II parser

Clearly the class of type II basic LRRL(k) grammars includes the type I grammars for the same parameter k. To show the inclusion is a proper one, consider the grammar G:

S →Ac	A →a
S →aBm	B →b
S →aDn	D →b

G is not type I basic LRRL(1). After seeing the terminal symbol 'a' there is a conflict whether it should be reduced to 'A' or one should continue with shifting symbols. So, three auxiliary productions SUBGOAL-RED(A →a) →c, SUBGOAL-SHIFT →B and SUBGOAL-SHIFT →D will be added. However the last two of these productions cannot be recognized from each other since both 'B' and 'D' derive a 'b'. One may note that G is type II LRRL(1) and also it is type I LRRL(2).

CHAPTER 4

IMPLICATIONS OF LRRL(k) PARSING

The first two sections of this chapter deal with some of the properties of the LRRL(k) grammars. Section 4.3 presents a novel methodology that leads to a recursive definition of LRRL(k) grammars. Section 4.4 develops a polynomial time algorithm for LRRL(k) testing. Finally, in Section 4.5, some informal comparisons are made between LRRL(k) parsing and some of the existing alternative techniques.

4.1 Unambiguity and linear parsability

From the previous chapter one may have observed that a CFSM provides a finite collection of parsing patterns for an LRRL(k) grammar in the form of regular sets. In fact, it will be shown shortly that these sets are obtained by taking the concatenation, union and closure of the left parts of the parsing patterns for a finite number of LR(k) grammars. The following two theorems are the immediate results of the existence of regular parsing patterns for a grammar [Szym 73]. However, here a direct proof is given.

Theorem 4.1:

If a context-free grammar G is LRRL(k), then it is unambiguous.

Proof for type II grammars: Suppose otherwise, i.e., there exists a sentence x in L(G) with two different derivation trees T_1 and T_2. Let $A=\{(\Phi_1,\Phi_2)\}$ be the set of pairs of the sentential forms in T_1 and T_2 such that $\Phi_1=\Phi_2$. Such a set is non-empty since $(x,x)\in A$. Let (Φ_1,Φ_2) be an element of A such that Φ_1 and Φ_2 do not contain a common phrase, i.e.,

$$S\$\overset{*}{\Longrightarrow}\gamma_1 A_1 \gamma_2 A_2 \ldots A_m \gamma_{m+1} \$ \Longrightarrow \gamma_1 \alpha_1 \gamma_2 A_2 \ldots A_m \gamma_{m+1} \$ \Longrightarrow \ldots$$

$$\Longrightarrow \gamma_1 \alpha_1 \gamma_2 \alpha_2 \ldots A_m \gamma_{m+1} \$ \Longrightarrow \gamma_1 \alpha_1 \gamma_2 \alpha_2 \ldots \alpha_m \gamma_{m+1} \$ = \Phi_1$$

and

$$S\$\overset{*}{\Longrightarrow}\delta_1 B_1 \delta_2 B_2 \ldots B_n \delta_{n+1} \$ \Longrightarrow \delta_1 \beta_1 \delta_2 B_2 \ldots B_n \delta_{n+1} \$ \Longrightarrow \ldots$$

$$\Longrightarrow \delta_1 \beta_1 \delta_2 \beta_2 \ldots B_n \delta_{n+1} \$ \Longrightarrow \delta_1 \beta_1 \delta_2 \beta_2 \ldots \beta_n \delta_{n+1} \$ = \Phi_2 = \Phi_1$$

where γ_i's and δ_j's do not contain complete phrases. Without loss of generality assume $|\gamma_{m+1}|\le|\delta_{n+1}|$. and let $\delta_{n+1}=\delta\gamma_{m+1}$. The CFSM for G cannot decide on reduction of any A_i or B_j phrase. Two cases are possible. Either before shifting the last symbol of α_m one reaches a state where there is a conflict and no more lookaheads are allowed, i.e., one of the lookahead sets is $\{\epsilon\}$ in that state, or the CFSM has a state s which is the successor of the initial state s_0 under the string $\gamma_1 \alpha_1 \gamma_2 \alpha_2 \ldots \alpha_m$. This state would contain the item $(A_m \to \alpha_m .,\gamma)$, where γ is a prefix of $\gamma_{m+1}\$$. Also, the state s should include another item either of the form $(B_n \rightarrow \beta_n ., \gamma')$, when $\delta = \epsilon$, or of the form $(\text{SUBGOAL-RED}(B_n \rightarrow \beta_n) \rightarrow \delta . \gamma", \epsilon)$, when $\delta \neq \epsilon$, where γ' and $\gamma"$ are some prefixes of $\gamma_{m+1}\$$. Now, the lookahead string for one of these items is a prefix of the lookahead for the other item. Therefore there is a conflict in state s that cannot be resolved by reduced lookaheads. Thus G is not LRRL(k) in either case, contradicting the assumption.

The proof for type I basic LRRL(k) grammars is immediate from the above, since type I grammars are a subset of type II grammars. □

Theorem 4.2:

LRRL(k) grammars are parsed in linear time and space.

Proof: The time complexity of the algorithm C can be analyzed in terms of the number of shift operations. In the proof of the termination of the algorithm it was noted that after at most |N| reductions the length of the sentential form is reduced at least by one. Therefore, the number of reductions is less than or equal to |N|n, where n is the length of the input sentence. For each reduction, the symbols on the right hand side of the production are pushed on the stack. Also, at most k lookaheads will be pushed on stack if a right context was necessary to reach for the reduction decision. These lookaheads then are transferred to the buffer. Therefore the number of shift operations for each reduction is at most $c=l+k$, where l is the length of the largest right hand side in the grammar. Thus the time complexity of the parsing algorithm for type II grammars is $O(c|N|n)$. For the type I grammars, however for each shift operation at most k additional symbols, i.e., right context, may be pushed on the stack which later are transferred to the buffer. Therefore, the time complexity of the parsing algorithm C for these grammars is $O(ck|N|n)$.

The linear space is obvious from the fact that the size of the stack plus the buffer never grows to more than n+k. □

4.2 Problem with parsing some left recursive and self-embedding constructs

The LRRL(k) parsers cannot handle those cases where reductions of a number of left recursive or self-embedding constructs are in conflict at a questionable point. The CFSM construction will fail for such grammars. Intuitively, any finite number of reduced lookaheads still will shield the necessary context for the reduction of the proper construct. An example is the grammar G:

$$\begin{array}{ll} S \rightarrow A\ b & S \rightarrow B\ c \\ A \rightarrow a & A \rightarrow A\ D \\ B \rightarrow a & B \rightarrow B\ D \\ D \rightarrow d & \end{array}$$

where any number of D's will shield the necessary information (i.e., b or c) for reducing the initial a to A or B. (Alternatively, by using the approach outlined in the next section, one can show that $\hat{G} \in D(G,k)$=the LRRL(k) decision set of G, is not LR(k) for any k). LR(k,t) grammars also suffer from the same disadvantage. LR(k,∞) grammars, since they can lookahead up to the right end of an input sentence, are able to parse some left recursive or self-embedding constructs that their parse cannot be given by an LR(k,t) or LRRL(k) parser. An example grammar is G':

$$\begin{array}{ll}
S \to A\ b & S \to B\ c \\
A \to a & A \to A\ D' \\
B \to a & B \to B\ D'' \\
D' \to d & D'' \to d
\end{array}$$

which is also LR-regular.

The above problem may be taken as an argument that the Marcus parser cannot handle coordinated constructs. That is only true if the coordinated constructs are given by left recursive rules, and different coordinated constructs can compete for reduction at the same time.

4.3 A second look at the LRRL(k) grammar

In this section the reduced contexts language of a grammar and the grammar that generates this language are introduced. It is shown that using these devices one can develop a new methodology that allows classes of context-free grammars to be defined recursively. In fact the approach can be considered as a one that spearheads a new direction in the theory of formal languages.

4.3.1 Reduced contexts grammar and reduced contexts language

Consider a grammar G_0 say

$S \rightarrow A\ C\ E$
$S \rightarrow B\ C\ F$
$A \rightarrow a$
$B \rightarrow a$
$C \rightarrow b$
$C \rightarrow b\ C$
$E \rightarrow e$
$F \rightarrow f$

and the parse tree of a sentence in $L(G_0)$ say abbbe (Figure 4.1).

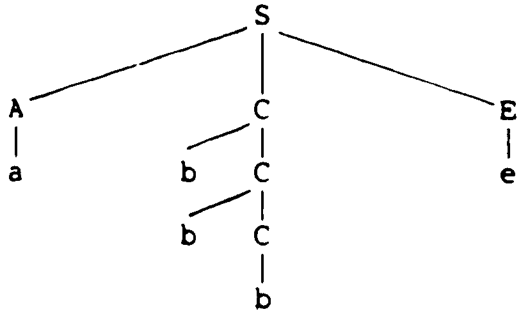


Fig. 4.1
Parse tree of abbbe

Suppose one is interested to know what is the fully reduced (left and right) context of a terminal node in the sentence say the second b. Such a reduced context can be shown by the leaf nodes of the tree in Figure 4.2. The reduced contexts for other symbols in the sentence abbbe are: $\bar{a}CE$, $A\bar{b}CE$, $Abb\bar{b}E$ and $AC\bar{e}$. Generally the reduced contexts for symbols in $L(G_0)$ are given by:

$$R = \{\bar{a}CE, \bar{a}CF, Ab^{*}\bar{b}CE, Bb^{*}\bar{b}CF, Ab^{*}\bar{b}E, Ab^{*}\bar{b}F, AC\bar{e}, BC\bar{f}\}.$$

One may observe that the role of a terminal symbol is determined by the symbols to its left and two symbols to the right of it in the set *R*, and in fact it is so because G_0 is LRRL(2). The following formalizes such a set *R*.

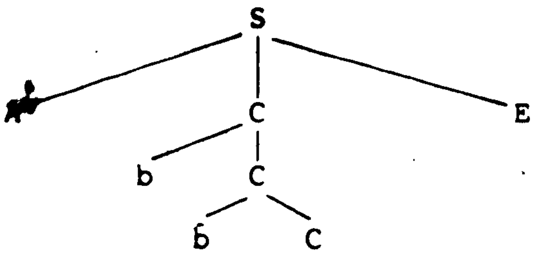


Fig. 4.2
The reduced context Ab$\bar{b}$CE

Definition: Reduced contexts language and grammar

Let $G=(N,T,P,S)$ be a context-free grammar. The reduced contexts language of G is $R(G)=L(\bar{G})$, where $\bar{G}=(\bar{N},T\cup N\cup\bar{T},\bar{P},\bar{S})$ and $\bar{N}=\{\bar{A}\mid A \in N\}$, $\bar{T}=\{\bar{a}\mid a \in T\}$ and $\bar{P}=\{\bar{A}\to X_1\ldots\bar{X}_i\ldots X_n \mid A\to X_1\ldots X_i\ldots X_n \in P \text{ and } 1\le i\le n\}$. $\bar{G}$ will be called the reduced contexts grammar of G.

Example:

The following shows the $\bar{G}_0$ for The grammar G_0.

$\bar{G}_0$:

$\bar{S}\to\bar{A}\ C\ E$
$\bar{S}\to A\ \bar{C}\ E$
$\bar{S}\to A\ C\ \bar{E}$
$\bar{S}\to\bar{B}\ C\ F$
$\bar{S}\to B\ \bar{C}\ F$
$\bar{S}\to B\ C\ \bar{F}$
$\bar{A}\to\bar{a}$
$\bar{B}\to\bar{a}$
$\bar{C}\to\bar{b}$

$$\begin{aligned}
&\check{C} \to \bar{b}\ C\\
&\check{C} \to b\ \check{C}\\
&\check{E} \to [illegible]\\
&\check{F} \to f
\end{aligned}$$

Figure 4.3 shows a sentence in $L(\check{G}_*)=R(G_*)$ and the corresponding parse tree.

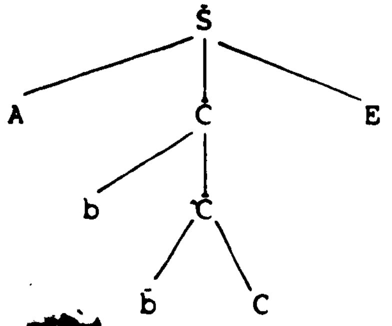


Fig. 4.3
Sentence $Ab\bar{b}CE \in L(\check{G}_*)=R(G_*)$

Suppose $G=(N,T,P,S)$ is context-free grammar and $\check{G}$ is its reduced contexts grammar. One can observe that the set of right sentential forms of $\check{G}$ is the same as the set of its left sentential forms and is equal to $SF(\check{G})$. This is evident from the fact that $\check{G}$ is a linear grammar by definition, and thus every sentential form in $SF(\check{G})$ has at most one non-terminal symbol in $\check{N}$. Now, by considering the set of sentential forms of $\check{G}$ the following theorems may be stated.

Theorem 4.3:

$\alpha\check{X}\beta$ is a sentential form of $\check{G}$ if and only if there exists $y \in T^*$ such that $\beta \xrightarrow{*} y$ and $\alpha X y$ is a right sentential form of G.

Proof: Suppose $\alpha\hat{X}\beta$ is a sentential form in SF($\hat{G}$), i.e., there exists a derivation

$$\hat{S} \xRightarrow[\hat{G}]{} \alpha_1 \hat{A}_1 \beta_1 \xRightarrow[\hat{G}]{} \alpha_1 \alpha_2 \hat{A}_2 \beta_2 \beta_1 \xRightarrow[\hat{G}]{} \ldots \xRightarrow[\hat{G}]{} \alpha_1 \alpha_2 \ldots \alpha_n \hat{X}\beta_n \ldots \beta_2 \beta_1 ,$$

where $\alpha_1 \alpha_2 \ldots \alpha_n = \alpha$ and $\beta_n \ldots \beta_2 \beta_1 = \beta$. Consider the productions $S \to \alpha_1 A_1 \beta_1$, $A_1 \to \alpha_2 \beta_2$, ..., and $A_{n-1} \to \alpha_n X \beta_n$ in G corresponding to the productions $\hat{S} \to \alpha_1 \hat{A}_1 \beta_1$, $\hat{A}_1 \to \alpha_2 \hat{A}_2 \beta_2$, ..., and $\hat{A}_{n-1} \to \alpha_n \hat{X}\beta_n$ in $\hat{G}$. There exists a rightmost derivation

$$S \xRightarrow[G,rm]{} \alpha_1 A_1 \beta_1 \overset{*}{\xRightarrow[G,rm]{}} \alpha_1 A_1 y_1 \xRightarrow[G,rm]{} \alpha_1 \alpha_2 A_2 \beta_2 y_1 \overset{*}{\xRightarrow[G,rm]{}} \alpha_1 \alpha_2 A_2 y_2 y_1 .$$

$$\xRightarrow[G,rm]{} \ldots \xRightarrow[G,rm]{} \alpha_1 \alpha_2 \ldots \alpha_n X\beta_n \ldots y_2 y_1 \overset{*}{\xRightarrow[G,rm]{}} \alpha_1 \alpha_2 \ldots \alpha_n Xy_n \ldots y_2 y_1$$

i,e., αXy, where $y = y_n \ldots y_2 y_1$, is a right sentential in SF(G).

Similarly, if αXy is a right sentential form in SF(G) then $\alpha\hat{X}\beta$, where β is the fully reduced form of y in G, is a sentential form of $\hat{G}$. □

Theorem 4.4:

$\alpha\hat{X}\beta$ is a sentential form of $\hat{G}$ if and only if there exists $y \in T^*$ such that $\alpha \overset{*}{\Longrightarrow} y$ and $yX\beta$ is a left sentential form of G.

Proof: is similar to the proof of Theorem 4.3.

Theorem 4.5:

If G is unambiguous then $\bar{G}$ is too.

Proof: Suppose otherwise, i.e., $\bar{G}$ is ambiguous. Then there must exist derivations

$$\bar{S} \xrightarrow[\bar{G}]{*} \gamma_1 \bar{A} \gamma_2 \xrightarrow[\bar{G}]{} \gamma_1 \alpha_1 X \alpha_2 \gamma_2 \quad \text{and}$$

$$\bar{S} \xrightarrow[\bar{G}]{*} \delta_1 \bar{B} \delta_2 \xrightarrow[\bar{G}]{} \delta_1 \beta_1 \bar{X} \beta_2 \delta_2 = \gamma_1 \alpha_1 \bar{X} \alpha_2 \gamma_2 ,$$

where $\gamma_1 \bar{A} \gamma_2 \neq \delta_1 \bar{B} \delta_2$. Let y be a terminal string that can be derived from $\alpha_2 \gamma_2$ in G. Then there exist rightmost derivations

$$S \xrightarrow[G,rm]{*} \gamma_1 Az \xrightarrow[G,rm]{} \gamma_1 \alpha_1 X \alpha_2 z \xrightarrow[G,rm]{*} \gamma_1 \alpha_1 Xy \quad \text{and}$$

$$S \xrightarrow[G,rm]{*} \delta_1 Bw \xrightarrow[G,rm]{} \delta_1 \beta_1 X \beta_2 w \xrightarrow[G,rm]{} \delta_1 \beta_1 Xy = \gamma_1 \alpha_1 Xy$$

in G, implying that there are two different derivations of the right sentential form $\gamma_1 \alpha_1 Xy$. Thus, G is ambiguous which contradicts the assumption. □

Suppose a grammar G is unambiguous. Then the role of a "dashed" symbol in a sentential form of $\bar{G}$ can be determined by the symbols on its both sides. However, it was noted earlier that the left context and only two symbols on the right of a dashed symbol in the grammar $\bar{G}$, are sufficient to determine its role. For example, in $\bar{a}CE$ two symbols CE indicate that $\bar{a}$ is to be reduced to $\bar{A}$. Similarly in $A\bar{b}CE$, $\bar{b}C$ must be reduced to a $\bar{C}$, but in $A\bar{b}E$, $\bar{b}$ should be reduced to a $\bar{C}$. More formally one could say that the grammar $\bar{G}$, is LR(2), and this is so because G_0 is LRRL(2). The following theorem proves this fact for the general case.

Theorem 4.6:

If G is an LRRL(k) grammar then the grammar $\bar G$ is LR(k).

Proof: Suppose $\bar G$ is not LR(k). Then there exist derivations

$$\bar S \overset{*}{\underset{\bar G,rm}{\Longrightarrow}} \gamma_1 \cdot \bar A \gamma_2 \underset{\bar G,rm}{\Longrightarrow} \gamma_1 \alpha \bar X \gamma_2 \quad \text{and}$$

$$\bar S \overset{*}{\underset{\bar G,rm}{\Longrightarrow}} \delta_1 \bar B \delta_2 \underset{\bar G,rm}{\Longrightarrow} \delta_1 \alpha' \bar X \beta \delta_2 = \gamma_1 \alpha \bar X \beta \delta_2 ,$$

such that $PF_k(\gamma_2) = PF_k(\beta\delta_2)$. That is there is an inadequate state in constructing an LR(k)-CFSM for $\bar G$ with conflicting items $[\bar A \to \alpha \bar X., PF_k(\gamma_2)]$ and $[\bar B \to \alpha' \bar X.\beta,\ PF_k(\delta_2)]$. Therefore in constructing an LRRL(k)-CFSM for G there is an inadequate state (the successor of the initial state under the string $\gamma_1 \alpha X$, a viable prefix of a right sentential form), with LRRL(k) items $(A \to \alpha X., PF_k(\gamma_2))$ and $(B \to \alpha' X.\beta, PF_k(\delta_2))$ such that $PF_k(\gamma_2) \cap (\beta + PF_k(\delta_2)) \neq \emptyset$. With this condition prevailing, the conflict in that state cannot be resolved. Thus G is not LRRL(k), which contradicts the assumption. □

4.3.2 LRRL(k) decision set of a grammar

In the previous section it was noted that $\bar G_0$ is an LR(2) grammar. Now consider the LR(2)-CFSM for $\bar G_0$ depicted in Diagram 4.1, and an incomplete LRRL(2)-CFSM for the grammar G_0 (Diag. 4.2) in which auxiliary productions are not processed further (i.e., ignore the states with dotted boundaries). In the LR(k)-CFSM, states 5,9 and 14 are LR(0) inadequate. However LR(2) lookaheads resolve the conflicts in these states. Notice that the lookaheads in the conflicting states of LRRL(2)-CFSM (states 3,5 and 7) are the same as the resolving LR(2) lookaheads in the LR(0) inadequate states of the LR(2)-CFSM for $\bar{G}_0$, though in the latter the symbols are considered to be terminals.

Next consider the grammars G_1, G_2 and G_3 where $G_i=(N\cup\{S'\},T,P_i,S')$, $P_1=P\cup\{S'\to CE,S'\to CF\}$, $P_2=P\cup\{S'\to E,S'\to CE\}$ and $P_3=P\cup\{S'\to F,S'\to CF\}$. These grammars are obtained for each LR(0) inadequate state of the LR(2)-CFSM of $\bar{G}_0$ by augmenting the grammar G_0 by productions $S'\to\alpha$, where α is an LR(2) lookahead string in that state and S' is a new start symbol. The reduced form of these grammars are:

G_1:
$S'\to CE$
$S'\to CF$
$C\to b$
$C\to bC$
$E\to e$
$F\to f$

G_2:
$S'\to E$
$S'\to CE$
$C\to b$
$C\to bC$
$E\to e$

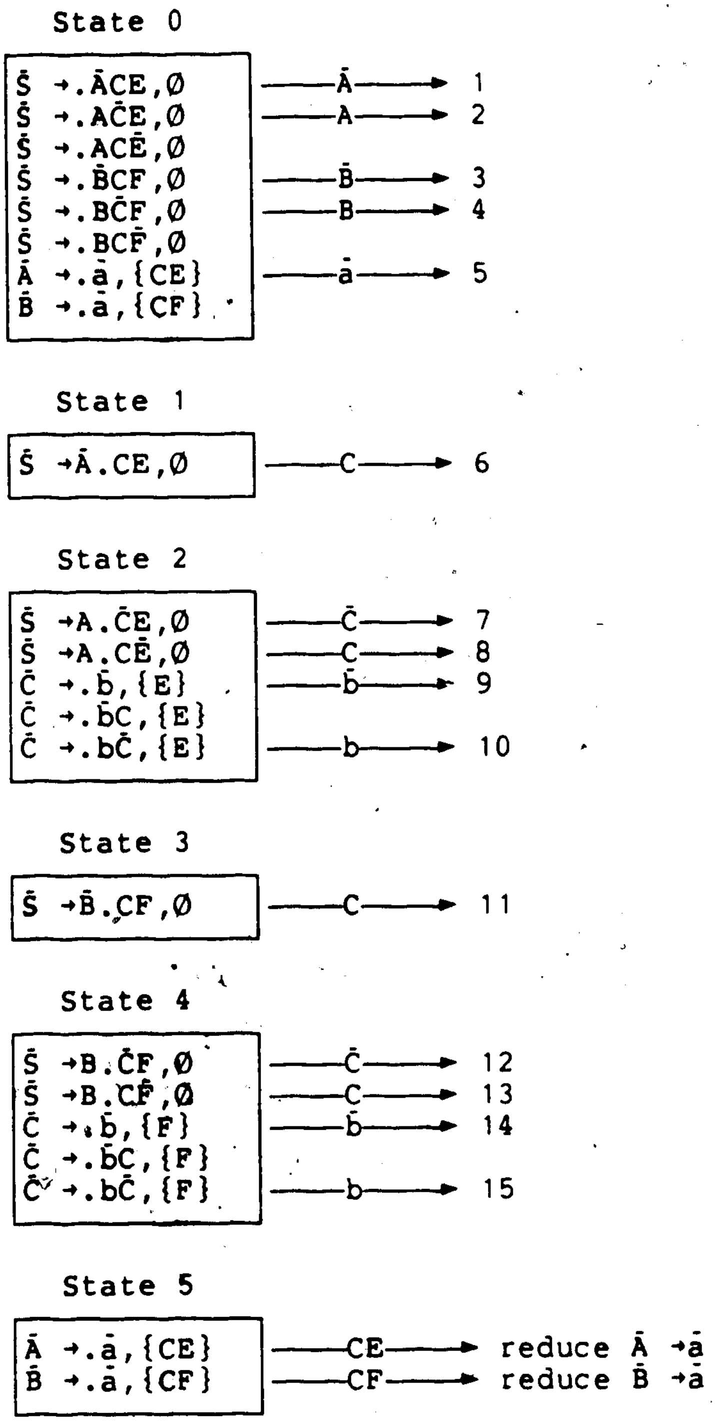
G_3:
$S'\to F$
$S'\to CF$
$C\to b$
$C\to bC$
$F\to f$

Other productions, i.e., $S\to ACE$ | BCF, $A\to a$, and $B\to a$ are deleted since they are useless.

Furthermore, consider the reduced contexts grammars of G_i, i=1,2,3 namely $\bar{G}_1$, $\bar{G}_2$ and $\bar{G}_3$ and LR(2)-CFSM's for these grammars which are shown in diagrams 4.3, 4.4 and 4.5. One can observe that the lookaheads in LR(0) inadequate states of these three CFSM's are equivalent to those in LRRL(2)-CFSM when the first set of auxiliary items are expanded (Diag 4.2). In a fashion that G_1, G_2 and G_3 were constructed, one can construct grammars G_4, G_5 and G_6 from the the above lookaheads. It turns out that $G_5=G_2$, $G_6=G_3$ and G_4 is given by the following productions.

G_4:
S' →E
S' →F
S' →CE
S' →CF
C →b
C →bC
E →e
F →f

The reduced contexts grammar of G_4 is the grammar $\bar{G}_4$. Diagram 4.6 shows $\bar{G}_4$ with its LR(2)-CFSM. One may observe that constructing a grammar from the LR(0) inadequate state 3 of this CFSM will result in a grammar $G_7=G_4$. Therefore intuitively one sees that there are five grammars $\bar{G}_i$, i=0,...,4 that are LR(2), and the LR(2) condition on these grammars is equivalent to the LRRL(2) condition on G_0.


State 0
$\bar{S} \to .\bar{A}CE,\emptyset$ — $\bar{A}$ → 1
$\bar{S} \to .A\bar{C}E,\emptyset$ — A → 2
$\bar{S} \to .AC\bar{E},\emptyset$
$\bar{S} \to .\bar{B}CF,\emptyset$ — $\bar{B}$ → 3
$\bar{S} \to .B\bar{C}F,\emptyset$ — B → 4
$\bar{S} \to .BC\bar{F},\emptyset$
$\bar{A} \to .\bar{a},\{CE\}$ — $\bar{a}$ → 5
$\bar{B} \to .\bar{a},\{CF\}$
State 1
$\bar{S} \to \bar{A}.CE,\emptyset$ — C → 6
State 2
$\bar{S} \to A.\bar{C}E,\emptyset$ — $\bar{C}$ → 7
$\bar{S} \to A.C\bar{E},\emptyset$ — C → 8
$\bar{C} \to .\bar{b},\{E\}$ — $\bar{b}$ → 9
$\bar{C} \to .\bar{b}C,\{E\}$
$\bar{C} \to .b\bar{C},\{E\}$ — b → 10
State 3
$\bar{S} \to \bar{B}.CF,\emptyset$ — C → 11
State 4
$\bar{S} \to B.\bar{C}F,\emptyset$ — $\bar{C}$ → 12
$\bar{S} \to B.C\bar{F},\emptyset$ — C → 13
$\bar{C} \to .\bar{b},\{F\}$ — $\bar{b}$ → 14
$\bar{C} \to .\bar{b}C,\{F\}$
$\bar{C} \to .b\bar{C},\{F\}$ — b → 15
State 5
$\bar{A} \to .\bar{a},\{CE\}$ — CE → reduce $\bar{A} \to \bar{a}$
$\bar{B} \to .\bar{a},\{CF\}$ — CF → reduce $\bar{B} \to \bar{a}$

Diag 4.1
LR(2)-CFSM for $\bar{G}_0$

State 6

$\bar{S} \to \bar{A}C.E,\emptyset \xrightarrow{E}$ reduce $\bar{S} \to \bar{A}CE$

State 7

$\bar{S} \to A\bar{C}.E,\emptyset \xrightarrow{E}$ reduce $\bar{S} \to A\bar{C}E$

State 8

$\bar{S} \to AC.\bar{E},\emptyset \xrightarrow{\bar{E}}$ reduce $\bar{S} \to AC\bar{E}$
$\bar{E} \to .\bar{e},\emptyset \xrightarrow{\bar{e}}$ reduce $\bar{E} \to \bar{e}$

State 9

$\bar{C} \to \bar{b}.,\{E\} \xrightarrow{E}$ reduce $\bar{C} \to \bar{b}$
$\bar{C} \to \bar{b}.C,\{E\} \xrightarrow{CE}$ 16

State 10

$\bar{C} \to b.\bar{C},\{E\} \xrightarrow{\bar{C}}$ 17
$\bar{C} \to .\bar{b},\{E\} \xrightarrow{\bar{b}}$ 9
$\bar{C} \to .\bar{b}C,\{E\}$
$\bar{C} \to .b\bar{C},\{E\}$
b

State 11

$\bar{S} \to \bar{B}C.F,\emptyset \xrightarrow{F}$ reduce $\bar{S} \to \bar{B}CF$

State 12

$\bar{S} \to B\bar{C}.F,\emptyset \xrightarrow{F}$ reduce $\bar{S} \to B\bar{C}F$

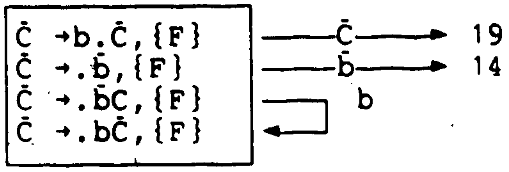
State 13

$\bar{S} \to BC.\bar{F},\emptyset \xrightarrow{\bar{F}}$ reduce $\bar{S} \to BC\bar{F}$
$\bar{F} \to .\bar{f},\emptyset \xrightarrow{\bar{f}}$ reduce $\bar{F} \to \bar{f}$

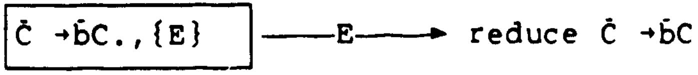
State 14

$\bar{C} \to \bar{b}.,\{F\} \xrightarrow{F}$ reduce $\bar{C} \to \bar{b}$
$\bar{C} \to \bar{b}.C,\{F\} \xrightarrow{CF}$ 18

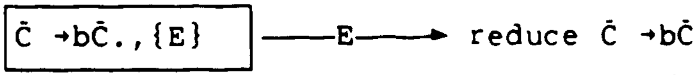
State 15



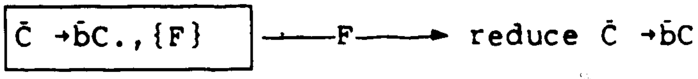
State 16



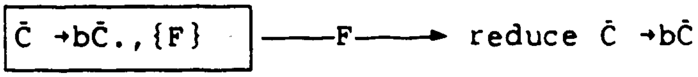
State 17



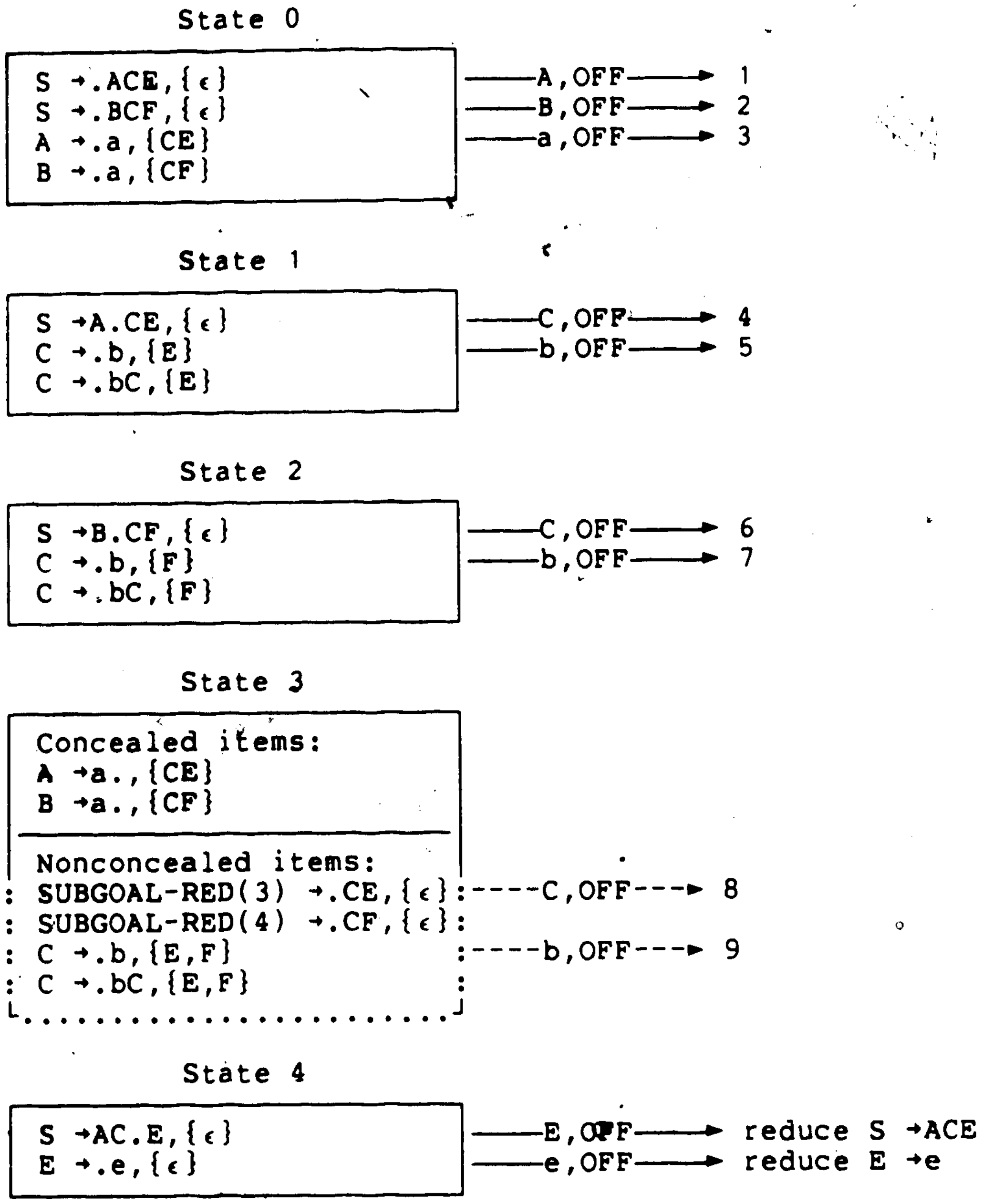
State 18



State 19



LRRL(2) CFSM for G



Diag. 4.2
LRRL(2)-CFSM for G_0

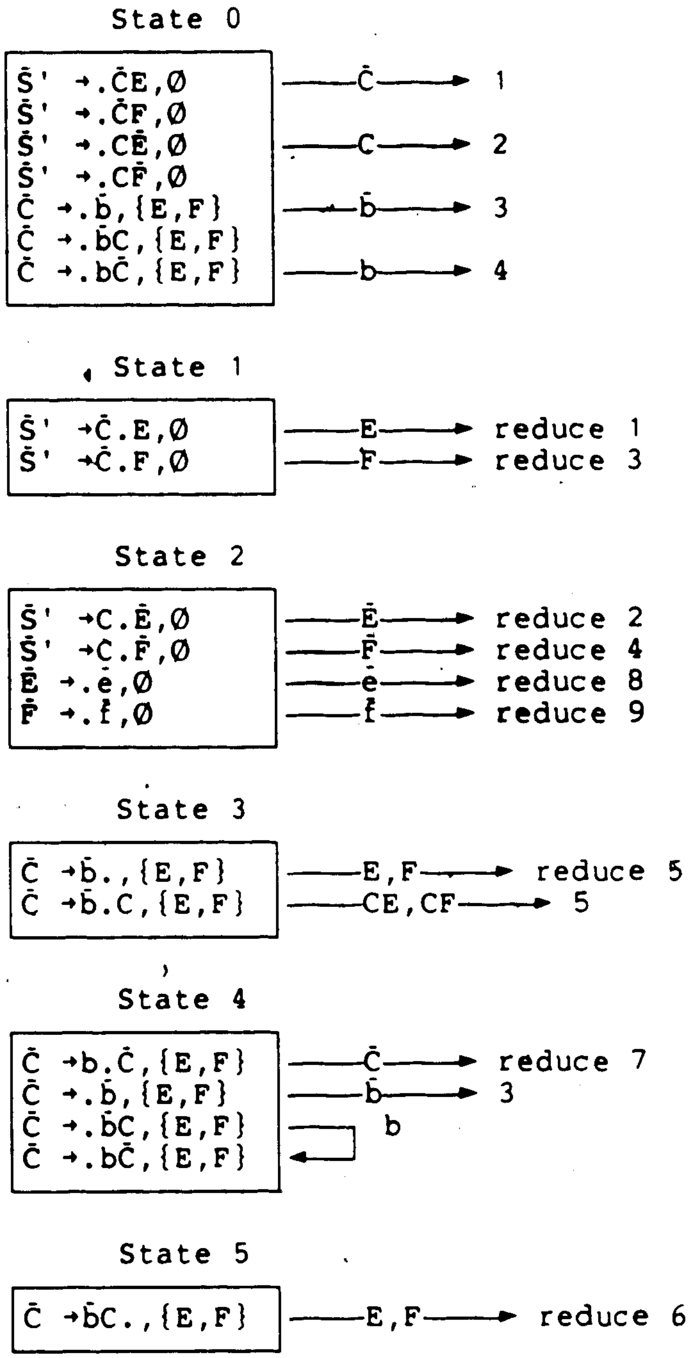
```
                  State 5
┌─────────────────────────┐
│ Concealed items:        │
│ C →b.,{E}               │
│ C →b.C,{E}              │───C,ON ───► reduce C →bC,OFF
├─────────────────────────┤
│ Nonconcealed items:     │
: SUBGOAL-RED(5) →.E,{ε}  :----E,OFF---►transfer1,reduce C →b
: SUBGOAL-SHIFT →.CE,{ε}  :----C,OFF---► 10
: C →.b,{E}               :---┐ b,OFF
: C →.bC,{E}              :◄--┘
└.........................┘
                  State 6
┌─────────────────────────┐
│ S →BC.F,{ε}             │───F,OFF───► reduce S →BCF
│ F →.f;{ε}               │───f,OFF───► reduce F →f
└─────────────────────────┘

                  State 7
┌─────────────────────────┐
│ Concealed items:        │
│ C →b.,{F}               │
│ C →b.C,{F}              │───C,ON ───► reduce C →bC,OFF
├─────────────────────────┤
│ Nonconcealed items:     │
: SUBGOAL-RED(5) →.F,{ε}  :---F,OFF---►transfer1,reduce C →b
: SUBGOAL-SHIFT →.CF,{ε}  :---C,OFF---► 11
: C →.b,{F}               :---┐ b,OFF
: C →.bC,{F}              :◄--┘
└.........................┘

                  State 8
┌.........................┐
: SUBGOAL-RED(3) →C.E,{ε}:---E,OFF---►transfer2,reduce A →a
: SUBGOAL-RED(4) →C.F,{ε}:---F,OFF---►transfer2,reduce B →a
: E →.e,{ε}              :---e,OFF---► reduce E →e
: F →.f,{ε}              :---f,OFF---► reduce F →f
└.........................┘
```

```
          State 9
┌.......................┐
: Concealed items       :
: C →b.,{E,F}           :
: C →b.C,{E,F}          :----C,ON ---► reduce C →bC,OFF
:-----------------------:
: Non-concealed items   :
: SUBGOAL-RED(5) →.E,{ε} :----E,OFF---►transfer1,reduce C →b
: SUBGOAL-RED(5) →.F,{ε} :----F,OFF---►transfer1,reduce C →b
: SUBGOAL-SHIFT →.CE,{ε} :----C,OFF---► 12
: SUBGOAL-SHIFT →.CF,{ε} :
: E →.e,{ε}             :----e,OFF---► reduce E →e
: F →.f,{ε}             :----f,OFF---► reduce F →f
: C →.b,{E,F}           :---┐ b,OFF
: C →.bC,{E,F}          :◄--┘
└.......................┘

          State 10
┌.......................┐
: SUBGOAL-SHIFT →C.E,{ε} :----E,OFF---► transfer 2, ON
: E →.e,{ε}             :----e,OFF---► reduce E →e
└.......................┘
          State 11
┌.......................┐
: SUBGOAL-SHIFT →C.F,{ε} :----F,OFF---► transfer 2, ON
: F →.f,{ε}             :----f,OFF---► reduce F →f
└.......................┘

          State 12
┌.......................┐
: SUBGOAL-SHIFT →C.E,{ε} :----E,OFF---► transfer 2, ON
: SUBGOAL-SHIFT →C.F,{ε} :----F,OFF---► transfer 2, ON
: E →.e,{ε}             :----e,OFF---► reduce E →e
: F →.f,{ε}             :----f,OFF---► reduce F →f
└.......................┘
```

$\bar{G}_1$:

$\bar{S}' \to \bar{C}E$ $\quad \bar{S}' \to C\bar{E}$ $\quad \bar{S}' \to \bar{C}F$
$\bar{S}' \to C\bar{F}$ $\quad \bar{C} \to \bar{b}$ $\quad \bar{C} \to \bar{b}C$
$\bar{C} \to b\bar{C}$ $\quad \bar{E} \to \bar{e}$ $\quad \bar{F} \to \bar{f}$

CFSM for $\bar{G}_1$:

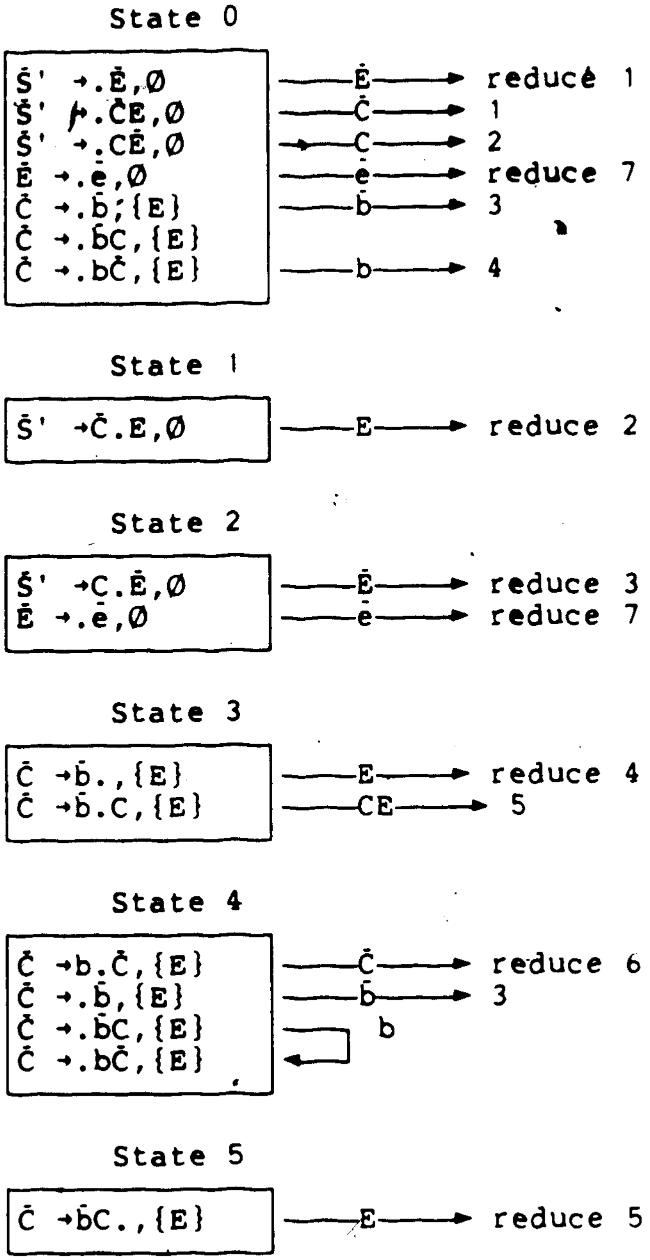


Diag. 4.3
$\bar{G}_1$ and its LR(2)-CFSM

$\dot{G}_2$:

Š' →Ē   Š' →ČE   Š' →CĒ
Č →b̄   Č →b̄C   Č →bČ
Ē →ė

CFSM for $\dot{G}_2$:

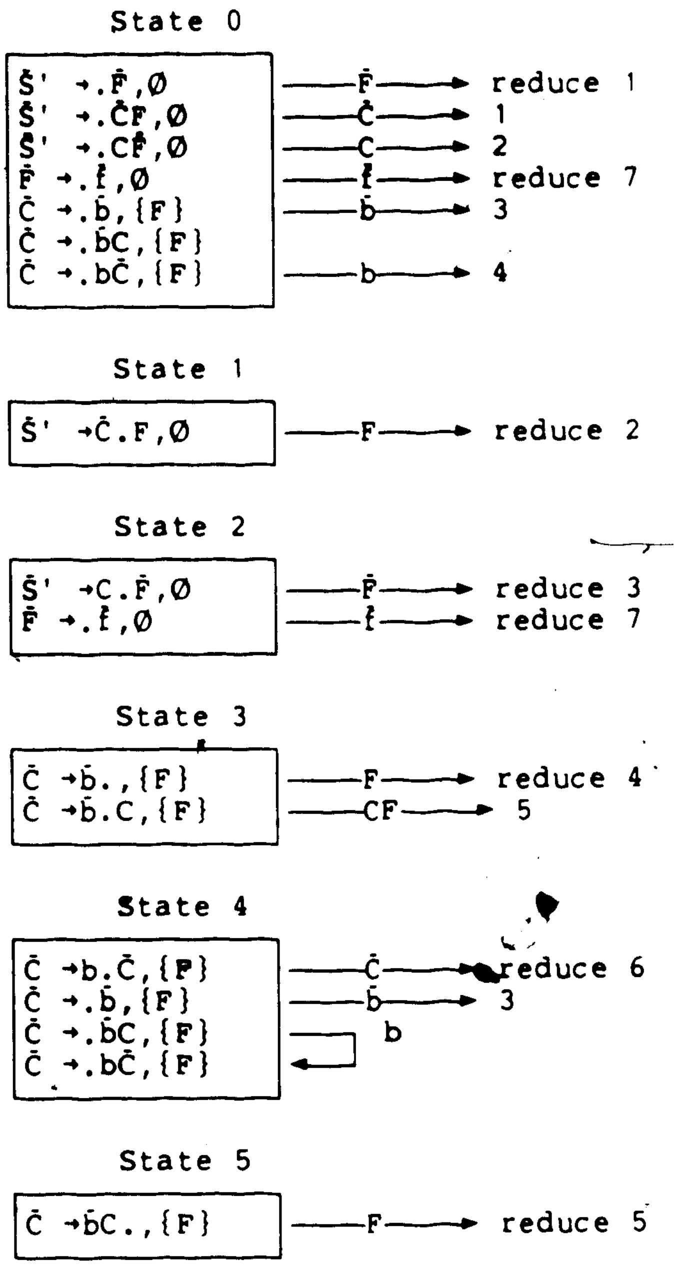


Diag. 4.4
$\dot{G}_2$ and its LR(2)-CFSM

Ḡ,:

S̄' →F̄ S̄' →C̄F S̄' →CF̄
C̄ →b̄ C̄ →b̄C C̄ →bC̄
F̄ →f̄

CFSM for Ḡ,:



Diag. 4.5
Ḡ, and its LR(2)-CFSM

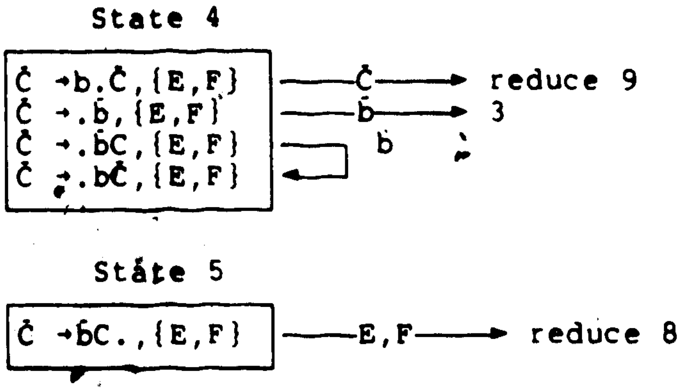
Ǧ$_4$:

```
Ŝ' →Ė          Ŝ' →Ḟ      Ŝ' →ČE
Ŝ' →CĖ         Ŝ' →ČF     Ŝ' →CḞ
Č →b̄           Č →b̄C      Č →b̄
Ė →ē           Ḟ →ḟ
```

CFSM for Ǧ$_4$:

```
      State 0

 ┌──────────────────┐
 │Ŝ' →.Ė,Ø          │ ────Ė────→ reduce 1
 │Ŝ' →.Ḟ,Ø          │ ────Ḟ────→ reduce 2
 │Ŝ' →.ČE,Ø         │ ────C────→ 1
 │Ŝ' →.CĖ,Ø         │ ────C────→ 2
 │Ŝ' →.ČF,Ø         │
 │Ŝ' →.CḞ,Ø         │
 │Ė →.ē,Ø           │ ────ē────→ reduce 10
 │Ḟ →.ḟ,Ø           │ ────ḟ────→ reduce 11
 │Č →.b̄,{E,F}       │ ────b̄────→ 3
 │Č →.b̄C,{E,F}      │
 │Č →.bČ,{E,F}      │ ────b────→ 4
 └──────────────────┘

      State 1

 ┌──────────────────┐
 │Ŝ' →Č.E,Ø         │ ────E────→ reduce 3
 │Ŝ' →Č.F,Ø         │ ────F────→ reduce 5
 └──────────────────┘

      State 2

 ┌──────────────────┐
 │Ŝ' →C.Ė,Ø         │ ────Ė────→ reduce 4
 │Ŝ' →C.Ḟ,Ø         │ ────Ḟ────→ reduce 6
 │Ė →.ē,Ø           │ ────ē────→ reduce 10
 │Ḟ →.ḟ,Ø           │ ────ḟ────→ reduce 11
 └──────────────────┘

      State 3

 ┌──────────────────┐
 │Č →b̄.,{E,Ḟ}       │ ────E,F────→ reduce 7
 │Č →b̄.C,{E,F}      │ ────CE,CF────→ 5
 └──────────────────┘
```

Diag. 4.6
Ǧ$_4$ and its LR(2)-CFSM


State 4
Č →b.Č,{E,F}
Č →.ḃ,{E,F}
Č →.ḃC,{E,F}
Č →.bČ,{E,F}
Č
reduce 9
ḃ
3
ḃ
State 5
Č →ḃC.,{E,F}
E,F
reduce 8

In the preceding discussion, the LRRL(k) grammars were not augmented with a special end-marker symbol "$". Nor did the LR(k) grammars have a "$^k" end-marker, which is customary to append to the right end of input sentences. So one might say that this section considered grammars with no end-markers. Too many authors are sloppy about the $^k augmentation of LR(k) grammars. Some authors leave out the $^k end-marker in the definition of LR(k) grammars. Others justify its presence by arguing that it ensures that there is always a K-symbol lookahead available to the parser. It can be shown that the presence of special end-markers has more ramifications than supplying the necessary amount of lookahead. Without end-marker symbols, a slightly different definition of LR(k) condition can be given in the following way.

Definition: LR(k) grammar with no end-marker

A context-free grammar G=(N,T,P,S) is LR(k) with no end-marker iff the three conditions:

(1) $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha Ax \underset{rm}{\Longrightarrow} \alpha\beta x$

(2) $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha' By \underset{rm}{\Longrightarrow} \alpha\beta z$

(3) $\{PF_k(x)\} \cap \{PF_k(z)\} \neq \emptyset$

imply that $\alpha Az = \alpha' By$.

It is obvious that when a grammar is augmented with $^k, the condition (3) degenerates to

$PF_k(x)=PF_k(z)$ same as the one in the traditional definition. Thus a grammar is LR(k) if its \$ -augmented form is LR(k) with no end-marker. One can easily observe that the LR(k) grammars with no end-markers defined here, have the property of generating languages that are prefix free and closed under concatenation, which is not true of the LR(k) grammars. Note that in the previous example only G_0 and $\bar{G}_0$ can be augmented with end-markers. One cannot augment G_i, i=1,...,4 or their respective reduced contexts grammars $\bar{G}_i$, because they generate only some segments of sentences in $L(G_0)$ or assuming dashes are omitted, some segments of sentential forms in $SF(G_0)$. The remainer of this section is concerned with only LR(k) grammars with no end-markers. Henceforth, the term will be used without the post modifier.

Now the formalization of the observations on the example grammar can be be provided by the following definition and theorem. The formalism is only for the type I basic LRRL(k) grammars. The type II grammars can be treated similarly.

Definition: LRRL(k) decision set of a grammar

Let $G_0=(N,T,P,S)$ be a context-free grammar and k a positive integer. The LRRL(k) decision set of G_0; $D(G_0,k)$ is a set of grammars defined recursively as follows:

(1) $\bar{G}_0$ the reduced contexts grammar of G_0 is an element of $D(G_0,k)$.

(2) For a grammar $\bar{G} \in D(G_0,k)$, if s is an LR(0) inadequate

state in its LR(k)-CFSM (i.e., a state with multiple reduce items or reduce and shift items), and G' is the grammar such that $G'=(\{S'\} \cup N,T,P',S')$ where $P'=P \cup \{S' \to \alpha \mid \alpha$ is a lookahead string in s$\}$, then the reduced contexts grammar of G', i.e., $\bar{G}'$ is an element of $D(G_{\bullet},k)$.

It can be shown that $D(G_{\bullet},k)$ is a finite set and its cardinality is bounded by $2^{|V| + \dots + |V| + |V|}$:

Theorem 4.7:

A context-free grammar G is LRRL(k) with ~~no~~ end-marker iff every grammar D in $D(G,k)$ the decision set of G is LR(k).

Proof: The proof is essentially the same as the theorem 4.6 with the difference that instead of prefixes of right sentential forms of G, prefixes of non-canonical sentential forms are to be used. Let $\{\bar{G}_1,\bar{G}_2,\dots,\bar{G}_n\}$ be a subset of $D(G,k)$ such that the sequence of grammars $G_1,G_2,\dots,$ and G_n are obtained by considering an LR(0) inadequate state in the LR(k)-CFSM's of $\bar{G},\bar{G}_1,\dots,$ and $\bar{G}_{n-1}$. Suppose G is LRRL(k) with no end-marker and without loss of generality assume $\bar{G},\bar{G}_1,\dots,$ and $\bar{G}_{n-1}$ are LR(k) grammars but $\bar{G}_n$ is not. Then under some viable prefix of $\bar{G}_n$ say $\gamma_n \alpha_n \bar{X}_n = \delta_n \alpha'_n \bar{X}_n$ there is a transition from the initial state of the LR(k)-CFSM of $\bar{G}_n$ to a state say s with conflicting items $[A \to \alpha_n \bar{X}_n .,\gamma]$ and $[B \to \alpha'_n \bar{X}_n .\beta_n ,\gamma']$ for some A,B,γ and γ' such that $PF_k(\gamma)=PF_k(\beta_n \gamma')$. Therefore there must be a state s' in the

LRRL(k)-CFSM of G with conflicting items $(A \to \alpha X_n., \gamma)$ and $(B \to \alpha' X_n.\beta_n, \gamma')$, which is a successor of the initial state under some string $\gamma_0 \alpha_0 X_0 \gamma_1 \alpha_1 X_1 \ldots \gamma_{n-1} \alpha_{n-1} X_{n-1} \gamma_n \alpha_n X_n$ such that $\gamma_0 \alpha_0 \bar{X}_0$, $\gamma_1 \alpha_1 \bar{X}_1$, ..., and $\gamma_{n-1} \alpha_{n-1} \bar{X}_{n-1}$ are viable prefixes of $\bar{G}$, $\bar{G}_1$, ..., and $\bar{G}_{n-1}$ respectively. Thus G is not LRRL(k), which contradicts the assumption.

Conversely, suppose every grammar in $D(G,k)$ is LR(k) but G is not LRRL(k) with no end-marker. If the state s in the LRRL(k)-CFSM of G is inadequate, then by a similar method one can show that some state s in the LR(k)-CFSM of one the grammars in $D(G,k)$ is inadequate. □

The above theorem provides a second definition for the LRRL(k) grammars. Let G be an LRRL(k) grammar with no end-marker and $D(G,k)=\{\bar{G},\bar{G}_1,\ldots,\bar{G}_m\}$ its decision set, then it is obvious that the grammars $G_1,\ldots,G_m$, i.e., those grammars that $\bar{G}_1,\ldots,\bar{G}_m$ are their reduced contexts grammars, are LRRL(k) grammars too with no end-marker. In fact if $G_1,\ldots,G_n$ $n \le m$ are those grammars that are obtained by considering the LR(0) inadequate states of $\bar{G}$, then one can say that G is LRRL(k) with no end-marker if and if G_i, $1 \le i \le n$, are LRRL(k) grammars with no marker.

4.3.3 Recursive definition of LRRL(k) grammars

The results of Section 4.3.2 can be put into a more rigorous form by supplying a recursive definition of LRRL(k) grammars in terms of derivations.

Definition A'-1:

A context-free grammar $G=(N,T,P,S)$ is type I basic LRRL(k) with no end-marker if and only if

(a) G is LR(0)

or

(b) the following three conditions together imply condition 4

(1) $S \xRightarrow[rm]{*} \alpha A x \xRightarrow[rm]{} \alpha\beta x \xRightarrow[rm]{*} yx$

is a rightmost derivation with corresponding leftmost derivation

$S \xRightarrow[lm]{*} zA\gamma \xRightarrow[lm]{} z\beta\gamma \xRightarrow[lm]{*} yx.$

(2) There exist rightmost derivations of the form

$S \xRightarrow[rm]{*} \alpha' A' x' \xRightarrow[rm]{} \alpha'\beta_1\beta_2 x' = \alpha\beta\beta_2 x' \xRightarrow[rm]{*} yx''$

with corresponding leftmost derivations

$S \xRightarrow[lm]{*} z'A'\gamma' \xRightarrow[lm]{} z'\beta_1\beta_2\gamma' = z\beta\beta_2\gamma' \xRightarrow[lm]{*} yx''$

and the set L is equal to

$\{PF_k(\beta_2\gamma') \mid \beta_2\gamma'$ appears in such leftmost derivations$\}$.

(3) $\{PF_k(\gamma)\} \underline{\cap} L \neq \emptyset$ or the grammar

$G'=(N\cup\{S'\},T,P',S')$ where

$P'=P\cup\{S' \rightarrow \sigma \mid \sigma \in Min(L\cup\{PF_k(\gamma)\})$

is not type I basic LRRL(k) with no end-marker.

(4) $\alpha=\alpha'$, $\beta_1\beta_2=\beta$ $(\beta_2=\epsilon)$ and $A'=A$

for all $\alpha', \beta_1 \beta_2$ and A' in (2).

Definition A'-2:

A context-free grammar $G=(N,T,P,S)$ is type I basic LRRL(k) if and only if $\$$-augmented grammar $G'=(N\cup\{GOAL\},\{\$\}\cup T,P\cup\{GOAL\rightarrow S\$\},GOAL)$ is a type I basic LRRL(k) grammar with no end-marker.

A similar definition for type II grammars can be provided with a slight change in the grammar G' of the condition (3).

The above recursive definition A'-1 relies on a number of grammars to be LRRL(k) (with no end-marker). On the surface there is no bound on the depth of recursion. However, it was shown earlier that only a finite number of grammars are involved in this recursion, i.e., grammars G and $G_1,\ldots,G_m$ such that their respective reduced context grammars $\bar{G}$ and $\bar{G}_i$, $1\leq i\leq m$, are the members of the decision set $D(G,k)$. Then one may perceive a circular decision problem here, that is to say the LRRL(k) condition on any grammar in the set $\{G,G_1,\ldots,G_m\}$ may circularly depend on the same condition. But, previously it was indicated that such a decision is reducible to testing whether each of the grammars in the decision set $D(G,k)=\{\bar{G},\bar{G}_1,\ldots,\bar{G}_m\}$ is LR(k).

It is in this sense, that one believes the work reported here introduces a new methodology in which grammars are recursively defined. Such definitions are valid when the decision problem can be reduced to testing of a finite number of predicates such that their truth does not circularly depend on each other. The method could be carried over to other grammar classes such as LR(k) and LC(k). The recursive definition of LR(k) grammars will depend on a decision set that consists of a finite number of LC grammars. Similarly the decision set for an LC(k) grammar would be a set of LL grammars.

As a concluding remark, one may notice that the issue discussed in this subsection is very similar to the definition of recursive data types in programming languages which itself seems to be inadequately addressed in the literature.

4.4 Complexity of LRRL(k) testing

Analyzing the complexity of the algorithms is an integral aspect of this research. Nevertheless, from the theoretical point of view, there is an important special issue that should be looked at very carefully. The issue is the complexity of LRRL(k) testing. In order to crystallize the problem some historic background as to the LR(k) testing is useful.

It is well known that the LR(k) parsers can have a number of states which is exponential in the size of the grammars that they parse. For any fixed value of k there was an $O(|G|^{4k+4})$ polynomial algorithm implicit in the original paper of Knuth on the LR(k) grammars [Knut 65], but for many years people somehow were carried away by practitioners who developed generators for LR(k) parsers. They assumed that the way to test whether a particular grammar is LR(k) is to try to construct an LR(k) parser, or in other words an LR(k)-CFSM for that grammar, and thus they believed that LR(k) testing is exponential in the size of grammar in the worst case. Even the most respected book on the theory of parsing by Aho and Ullman gave the exponential algorithm for the testing of LR(k) condition [AhUl 72a,p 391]. This practice is still widespread among the programming languages people. They usually run their grammar through a YACC-like parser generator to test whether it is LR(1), or, in the case of YACC, LALR(1).

Eventually Hunt, Szymanski and Ullman in a brilliant joint work obtained $O(|G|^{2k+2})$ and $O(|G|^{k+2})$ algorithms for LR(k) testing with a fixed value of k [HuSU 74,75]. They showed that their result carries over to LL(k), SLR(k) and LC(k) grammars. However they left the determining of the complexity of LALR(k) testing as an open problem. One might have expected to learn something from them about the complexity of LR(k,t) testing, since these grammars were developed by one the authors himself, but strangely enough they omitted the LR(k,t) grammars from the discussion.

Sippu, Soisalon-Soininen and Ukkonen [SiSU 83] discussed the complexity of LALR(k) testing. They contend that testing for the LALR(k) condition is based on the LALR(k) parser construction because the LALR(k) property is essentially given in terms of the LALR(k)-CFSM construction. So for fixed k, they conclude that the problem is **PSPACE-Complete**.

On the surface one might think that the LRRL(k) testing is exponential in the size of grammar since similar to LALR(k) grammars the LRRL(k) property originally is given in the form of CFSM construction. On the other hand if one considers the alternative definition, then to test a grammar G for the LRRL(k) condition, one needs to test each grammar in its decision set $D(G,k)=\{\hat{G},\hat{G}_1,\ldots,\hat{G}_m\}$ is LR(k). However each $\hat{G}_i$ is successively obtained from other members of $D(G,k)$ by actually constructing CFSM's for these members which implies an exponential algorithm. Furthermore the cardinality of $D(G,k)$ at the worst case is $O(2^{|V|^k})$.

Such observations in addition to the fact that no polynomial time algorithm is reported for LR(k,t) testing in the literature, and the fact that some prominent formal grammarians such as Soisalon-Soininen are content to accept the exponential CFSM construction for LALR(k) testing, which on the surface may seem to be a simpler test than LRRL(k), may soon diminish one's hope for obtaining a polynomial algorithm for the LRRL(k) testing. However, a considerable effort has been vested in finding such algorithms in this research, and one is presented in the following. The algorithm given here is for the type II basic LRRL(k) grammars. It is very simple and can be easily ported to type I and all the generalized grammars (except the GLRRL(k) grammars with bounded buffer) that are to be defined in Chapter 5 of this thesis. Nevertheless, because of a certain counting problem it cannot be applied to Szymanski's LR(k,t) grammars. It is important to note that such polynomial algorithms only test the properties for the corresponding classes and do not generate a parser.

The algorithm is based on constructing the non-deterministic version of the CFSM for a grammar. Let $G=(N,T,P,S)$ be a context-free grammar. Add production GOAL $\rightarrow S$ to P and let $V=N\cup T\cup\{\$\}$. Consider a non-deterministic finite state machine with 'potential' ϵ-arcs $M=(Q,V,\delta,q_0,\emptyset)$ where

V= The set of input symbols.

Q= The set of states of the form $[A \rightarrow\alpha.\beta,\sigma]$ where either $A \rightarrow\alpha\beta$ is a production in P and $\sigma\in V^*$ with $|\sigma|\leq k$, or A=SUBGOAL-RED(p), p is a production in P, $\sigma=\epsilon$ and $\alpha\beta$ is a string in V^{+} with length less than or equal to k.

q_0= The initial state [GOAL $\rightarrow.S,\$$].

δ= The transition mapping from $Q X(V\cup\{\epsilon\})$ into 2^Q is defined in the following way.

(1) $\delta([A \rightarrow\alpha.X\beta,\sigma],X)=\{[A \rightarrow\alpha X.\beta,\sigma]\}$.

(2) $\delta([A \rightarrow\alpha.X\beta,\sigma],\epsilon)=\{[X \rightarrow.\gamma,\beta+_k \sigma]|$ $X \rightarrow\gamma$ is a production in $P\}$.

(3) Also there are 'potential' transitions on empty string

of the form $\delta([A \rightarrow \alpha.,\sigma],\epsilon)=\{[\text{SUBGOAL-RED}(p) \rightarrow .\sigma,\epsilon]\}$, where p is the production $A \rightarrow \alpha$ in P. If after reading a string $\gamma=\gamma'\alpha$, one reaches to a state $q_1=[A \rightarrow \alpha.,\sigma]$, then a transition from state q_1 to $q_2=[\text{SUBGOAL-RED}(A \rightarrow \alpha) \rightarrow .\sigma,\epsilon]$ over the potential ϵ-arc is possible iff q_1 exhibits a conflict with some state q_1' with respect to the string γ as defined below. It should be clear that if the chain from q_0 to q_1 (which is traversed by reading γ) includes any potential ϵ-arc then the tail state of that arc must exhibit a conflict with some other state with respect to a prefix of γ that is read thus far.

Definition: Exhibition of a conflict

Let $q_1=[A \rightarrow \alpha.,\sigma]$ and $q_2=[B \rightarrow \alpha'.\beta,\sigma']$ be two states such that $(A \rightarrow \alpha.)\neq(B \rightarrow \alpha'.\beta)$. The states q_1 and q_2 exhibit a conflict with respect to a string $\gamma \in V^*$ iff the two states can be reached simultaneously on reading the string γ.

Explained in terms of deterministic CFSM construction, transitions of the type (1) represent shift moves. Type (2) transitions correspond to the closing of the states in CFSM, whereas type (3) transitions represent possible addition of auxiliary items when there is a shift-reduce/reduce-reduce ambiguity in a state.

The set of final states for the purpose of this discussion is irrelevant, and it is assumed to be the empty set.

One may observe that the non-deterministic machine corresponds to a 'quasi' linear right regular grammar with some potential productions. Similar observations were also made by some authors specially Heilbrunner [Heil 81] regarding the non-deterministic parsing automata for LR(k) grammars, in which they have called the corresponding grammar, an item grammar. However, one should note that an item grammar for an LR(k) grammar does not have potential productions and is a regular grammar in the true sense.

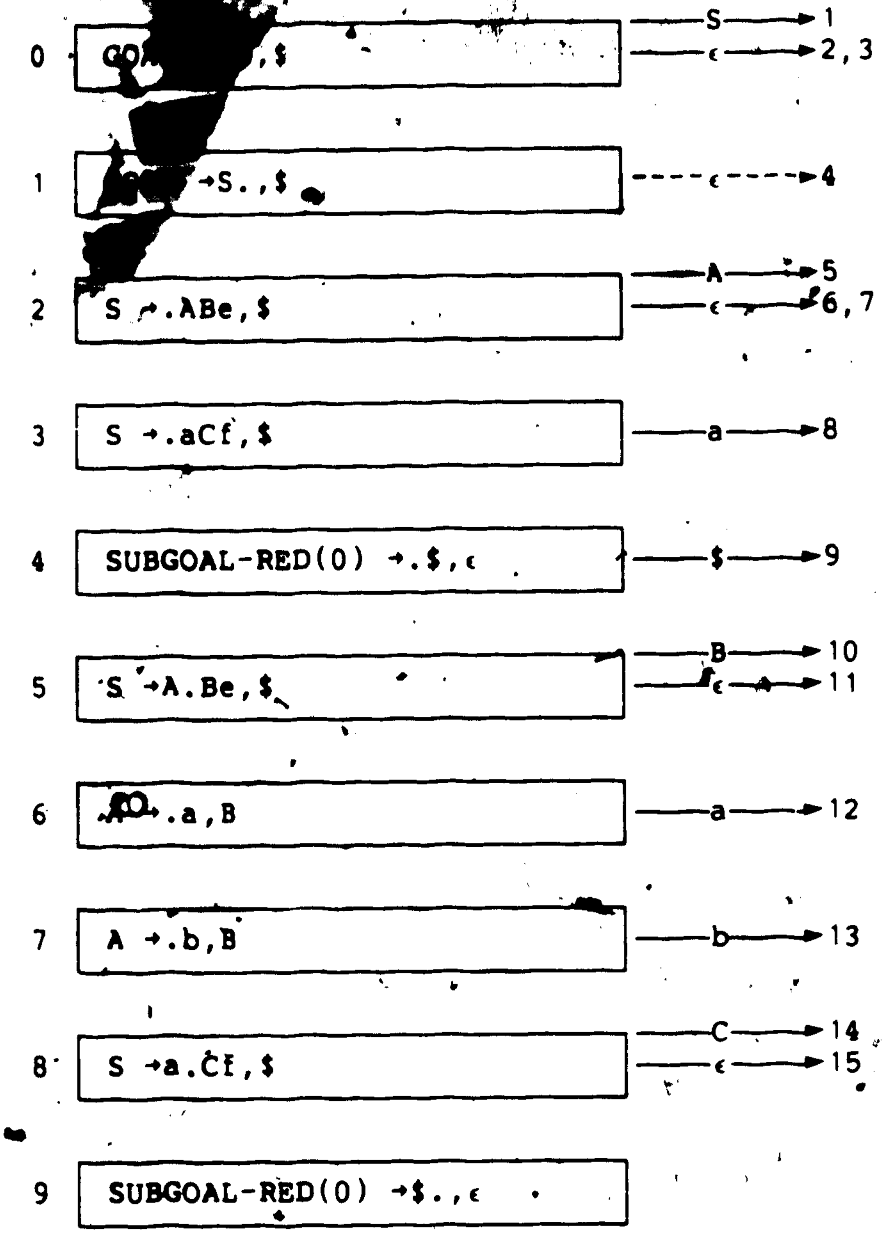
Example:

Diagram 4.7 shows the non-deterministic finite state machine for the grammar G and k=1, where G is:

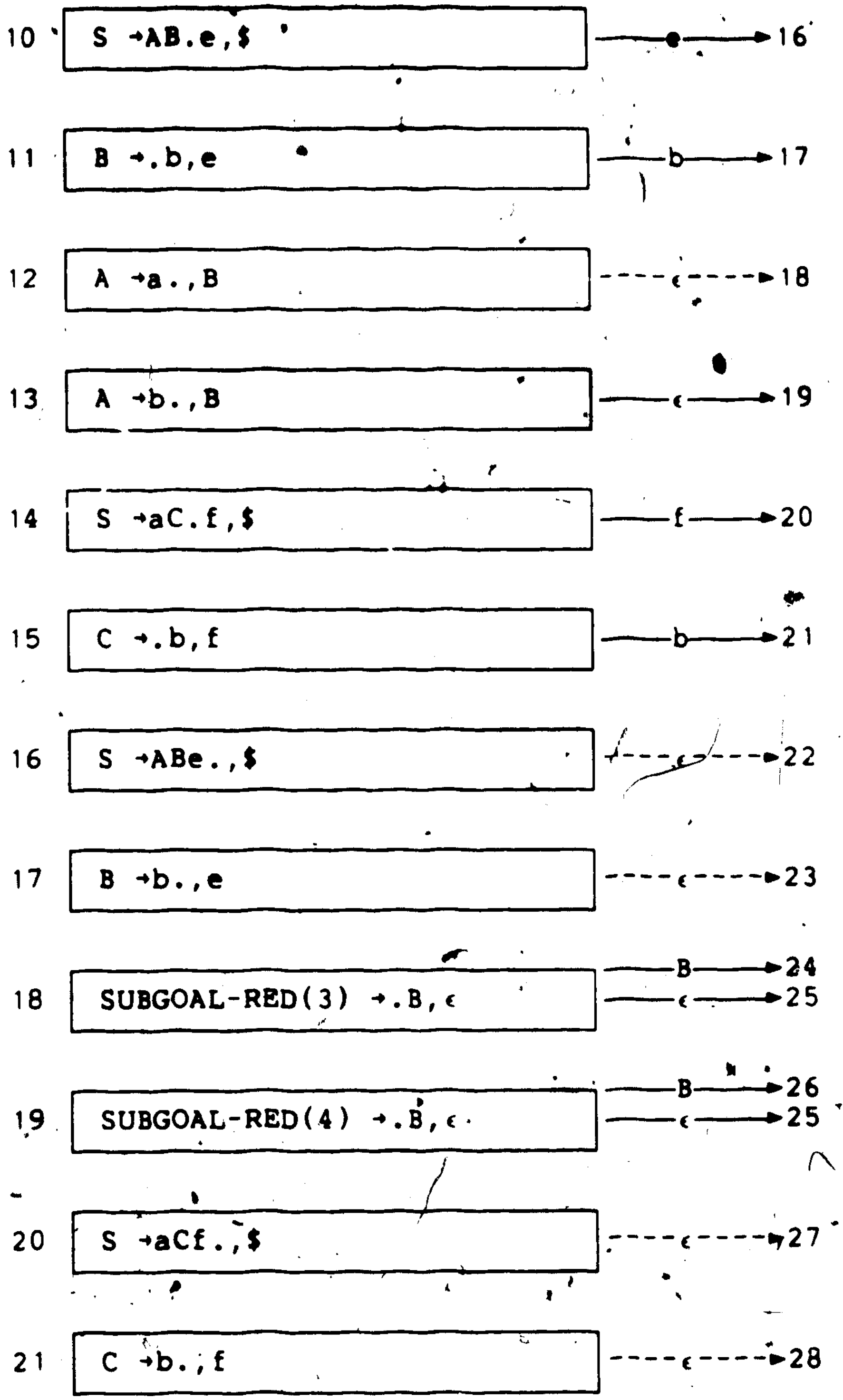
(1) $S \rightarrow A\ B\ e$
(2) $S \rightarrow a\ C\ f$
(3) $A \rightarrow a$
(4) $A \rightarrow b$
(5) $B \rightarrow b$
(6) $C \rightarrow b$

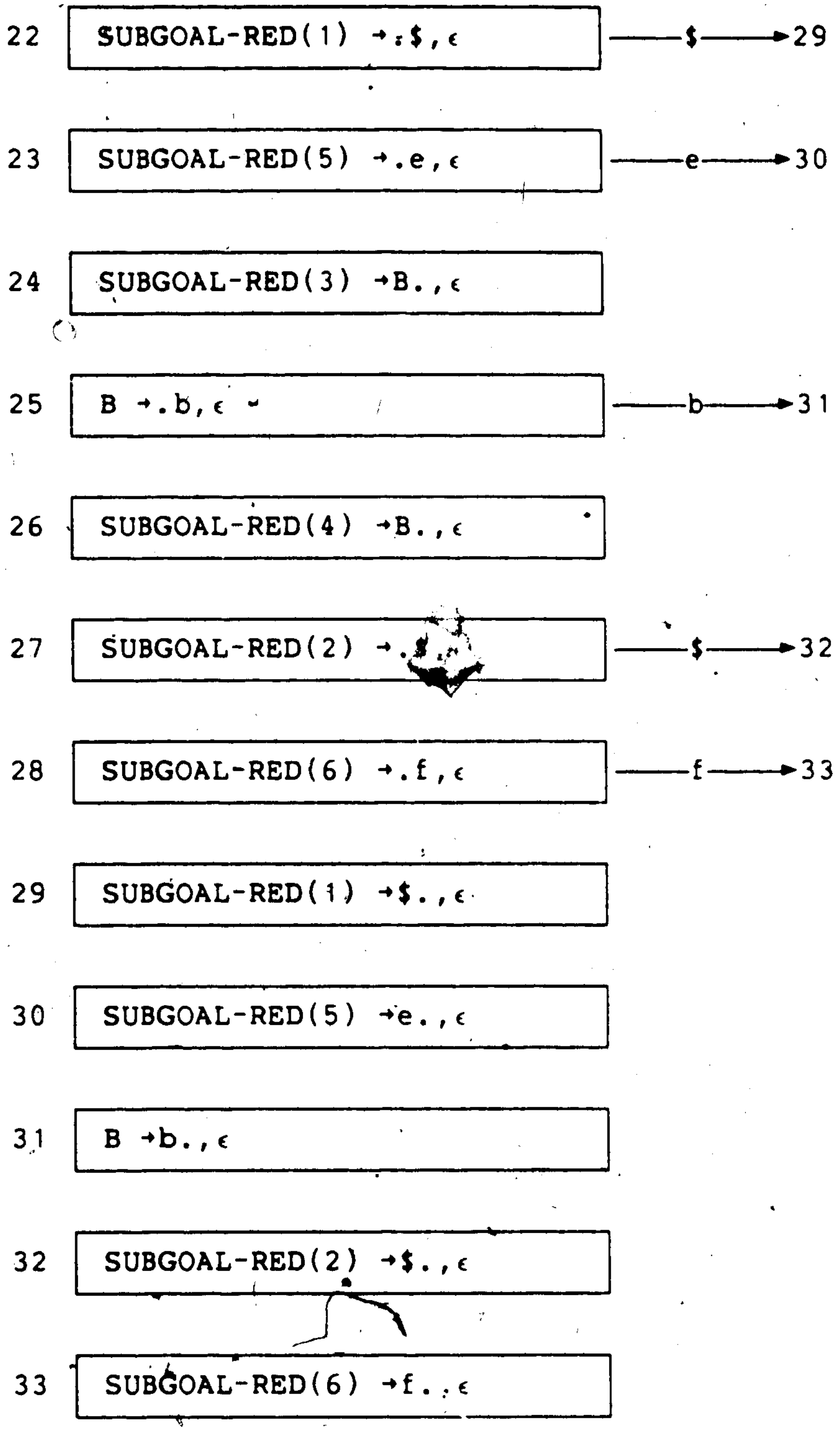
Broken arrows represent potential transitions on null string.

STATES:



Diag. 4.7
Non-deterministic Finite State
Machine with Potential transitions


10 S →AB.e,$ → 16
11 B →.b,e b → 17
12 A →a.,B ϵ → 18
13 A →b.,B ϵ → 19
14 S →aC.f,$ f → 20
15 C →.b,f b → 21
16 S →ABe.,$ ϵ → 22
17 B →b.,e ϵ → 23
18 SUBGOAL-RED(3) →.B,ϵ B → 24 ϵ → 25
19 SUBGOAL-RED(4) →.B,ϵ B → 26 ϵ → 25
20 S →aCf.,$ ϵ → 27
21 C →b.,f ϵ → 28


22 SUBGOAL-RED(1) →.$,ε —$→ 29
23 SUBGOAL-RED(5) →.e,ε —e→ 30
24 SUBGOAL-RED(3) →B.,ε
25 B →.b,ε —b→ 31
26 SUBGOAL-RED(4) →B.,ε
27 SUBGOAL-RED(2) →.$,ε —$→ 32
28 SUBGOAL-RED(6) →.f,ε —f→ 33
29 SUBGOAL-RED(1) →$.,ε
30 SUBGOAL-RED(5) →e.,ε
31 B →b.,ε
32 SUBGOAL-RED(2) →$.,ε
33 SUBGOAL-RED(6) →f.,ε

The condition for a grammar to be non-LRRL(k) is that two states exhibit a conflict which cannot be resolved with the use of lookaheads.

Definition: Exhibition of unresolvable conflict

If two states $q_1=[A \rightarrow \alpha.,\sigma]$ and $q_2=[B \rightarrow \alpha'.\beta,\gamma]$ in which $(\{\sigma\}\underline{\cap}\{\beta +_k \gamma\})\neq\emptyset$ and $\neg$(A=B=SUBGOAL-RED(p) and $\alpha=\alpha'$) exhibit a conflict (with respect to some string), then they exhibit an unresolvable conflict.

For example, in Diag. 4.7 the states 21 and 31 (i.e., $[C \rightarrow b.,f]$ and $[B \rightarrow b.,\epsilon]$) can be reached simultaneously by reading the string ab. Since $\{f\}\underline{\cap}\{\epsilon\}=\{\epsilon\}\neq\emptyset$ the two states exhibit an unresolvable conflict, and therefore G is not LRRL(1).

Generally simpler grammar problems (e.g., LR(k) testing) can be solved by computing a binary relation $R2 \subseteq Q \times Q$ on the states of such a non-deterministic machine, that represents mutual or simultaneous reachability of pairs of states after reading some string in V^*. One can check whether there is a pair $(q,q') \in R2$ that exhibit a conflict which cannot be resolved by the means of lookaheads. However, for harder problems such as LR(k,t) and LRRL(k) testing, such a binary relation cannot be computed unless with the aid of a ternary relation $R3 \subseteq Q \times Q \times Q$ which represents simultaneous reachability of triples of states after reading some string. To see the reason, consider the three states

$q_1=[A \rightarrow \alpha.,\sigma]$, $q_2=[A \rightarrow \alpha.,\gamma]$ and $q_3=[B \rightarrow \alpha'.,\tau]$, and suppose they are simultaneously reachable after reading some string. Clearly, if $q'_1=[\text{SUBGOAL-RED}(A \rightarrow \alpha) \rightarrow .\sigma,\epsilon]$ and $q'_2=[\text{SUBGOAL-RED}(A \rightarrow \alpha) \rightarrow .\gamma,\epsilon]$ are the successors of q_1 and q_2 under the potential ϵ-arcs then (q'_1,q'_2) must be in *R2*. However, $\{(q_i,q_j)|$ for all $i,j\in\{1,2,3\}\}\subseteq \mathit{R2}$ does not imply $(q'_1,q'_2)\in \mathit{R2}$ (see the above condition under which ϵ-transitions over the potential arcs are possible), but $(q_1,q_2,q_3)\in \mathit{R3}$ implies $(q'_1,q'_2,\cancel{q_3})\in \mathit{R3}$ and consequently $(q'_1,q'_2)\in \mathit{R2}$. By now one may have realized that the computation of *R3* is not possible without the use of *R4*, and of *R4* without the use of *R5* and so on, and ultimately one needs to compute *Rn*, where $n=\sum_{i=0}^{k} |V|^{i}$, i.e., a computation much more costly than deterministic CFSM construction. However, for the purpose of LRRL(k) testing, only computation of a binary relation $\mathbf{R2}\subseteq \mathit{R2}$ with the aid of a ternary relation $\mathbf{R3}\subseteq \mathit{R3}$ which itself can be computed directly, is sufficient. **R3** does not compute triple simultaneous reachability of such states as $q_1=[\text{SUBGOAL-RED}(p) \rightarrow .\sigma_1,\epsilon]$, $q_2=[\text{SUBGOAL-RED}(p) \rightarrow .\sigma_2,\epsilon]$ and $q_3=[\text{SUBGOAL-RED}(p) \rightarrow .\sigma_3,\epsilon]$, i.e., the triple $(q_1,q_2,q_3)\notin \mathbf{R3}$, and consequently triple simultaneous reachability of some states that follow these states will not be known. Therefore, some pairs of states such as $t_1=[\text{SUBGOAL-RED}(p') \rightarrow .\gamma_1,\epsilon]$ and $t_2=[\text{SUBGOAL-RED}(p') \rightarrow .\gamma_2,\epsilon]$ which are simultaneously reachable from the former states will not be in **R2** (i.e., $(t_1,t_2)\in \mathit{R2}$ but not $(t_1,t_2)\in \mathbf{R2}$).

Let $\{Q_0, Q_1, \ldots, Q_{2|P|}\}$ be a partition of Q such that $Q_i = \{[A_i \rightarrow \alpha_i ., \gamma] \mid A_i \rightarrow \alpha_i$ is the ith production in $P\}$ and $Q_{i+|P|} = \{[\text{SUBGOAL-RED('i')} \rightarrow .\gamma, \epsilon] \mid \gamma \in V^*\}$, for $i=0,\ldots,|P|-1$, and $Q_{2|P|}$ be the set of all those states indicating a shift move. The relation R3 is given by the algorithm 1 which uses a stack of triples of states. The relation R2 is simply $\{(q_1,q_2) \mid (q_1,q_1,q_2) \in R3\}$.

Algorithm 1: Computation of R3.

```
    Let STACK be a stack of triples of states.
    Routine ADD($q_1$,$q_2$,$q_3$);
      If ($q_1$,$q_2$,$q_3$)$\notin$R3 then
       R3:=R3$\cup${($q_1$,$q_2$,$q_3$)};
       PUSH ($q_1$,$q_2$,$q_3$) on the STACK
      end.

 1. STACK:=$\emptyset$;
    R3:=($q_0$,$q_0$,$q_0$), where $q_0$ is the initial state.
 2. While STACK is not empty
    begin
      POP ($q_1$,$q_2$,$q_3$) off the STACK;
 2.1  Process the type (1) transitions:
      If  $q_1$=[A $\rightarrow\alpha_1$.X$\beta_1$,$\sigma_1$] and
          $q_2$=[B $\rightarrow\alpha_2$.X$\beta_2$,$\sigma_2$] and
          $q_3$=[C $\rightarrow\alpha_3$.X$\beta_3$,$\sigma_3$]
      then ADD($q_1'$,$q_2'$,$q_3'$), where
            {$q_1'$}=$\delta$($q_1$,X), {$q_2'$}=$\delta$($q_2$,X) and {$q_3'$}=$\delta$($q_3$,X).
    end if.
2.2   Process normal (i.e., closure) $\epsilon$-transitions:
2.2.1 For all $t \in \delta(q_1,\epsilon)$ where
          the transition is a closure transition
          ADD($t,q_2,q_3$).
2.2.2 For all $t \in \delta(q_2,\epsilon)$ where
          the transition is a closure transition
          ADD($q_1,t,q_3$).
2.2.3 For all $t \in \delta(q_3,\epsilon)$ where
          the transition is a closure transition
          ADD($q_1,q_2,t$).
2.3   Process potential $\epsilon$-transitions:
      If not all of the $q_1,q_2$ and $q_3$ belong
      to the same $Q_i$ then
      begin
        For i=1,3
        Let $t_i$ := potential successor of $q_i$ if
            there exists one, otherwise $q_i$
        For i=1,3
        For j=1,3
        ADD ($t_1,t_i,t_j$).
      end.
```

The following theorem shows that the computation of R2 is sufficient for detecting the non-LRRL(k) property of a grammar.

Theorem 4.8:

There is a surjective mapping π from *R2* onto R2 such that if $(q_1,q_2) \in$ *R2* exhibits an unresolvable conflict then $\pi((q_1,q_2))=(q'_1,q'_2) \in R2$ also exhibits an unresolvable conflict.

Proof: First it is clear that the algorithm 1 computes the simultaneous triple reachability of those states that the paths from q_0 to these states do not simultaneously traverse potential ϵ-arcs with tail states that have common cores. Secondly suppose the end-marker '\$' is replaced with k symbols, say $\$^k$. With a careful consideration, one can see that the dual or triple reachability of those states with exactly k symbol lookaheads does require computation of *R3* or *R2*. Consider a situation in which the states $q_1=[A \rightarrow \alpha.,\sigma_1]$, $q_2=[A \rightarrow \alpha.,\sigma_2]$, $q_3=[A \rightarrow \alpha.,\sigma_3]$ and $q_4=[A' \rightarrow \tau_4\alpha.\tau'_4,\sigma_4]$ are simultaneously reachable from the states $s_1=[Z_1 \rightarrow \tau_1.A\tau'_1,\tau''_1]$, $s_2=[Z_2 \rightarrow \tau_2.A\tau'_2,\tau''_2]$, $s_3=[Z_3 \rightarrow \tau_3.A\tau'_3,\tau''_3]$ and $s_4=[A' \rightarrow \tau_4.\alpha\tau'_4,\sigma_4]$ (where $\sigma_i=PF_k(\tau'_i\tau''_i)$ for i=1,2,3) by reading the string α. Algorithm 1 does not compute the triple reachability of the states $q'_1=[\text{SUBGOAL-RED}(p) \rightarrow .\sigma_1,\epsilon]$, $q'_2=[\text{SUBGOAL-RED}(p) \rightarrow .\sigma_2,\epsilon]$ and $q'_3=[\text{SUBGOAL-RED}(p) \rightarrow .\sigma_3,\epsilon]$ where p is the production $A \rightarrow \alpha$. Now suppose the three states $[B_1 \rightarrow \beta_1.\beta'_1,\gamma_1]$, $[B_2 \rightarrow \beta_2.\beta'_2,\gamma_2]$ and $[B_3 \rightarrow \beta_3.\beta'_3,\gamma_3]$ are simultaneously reachable from q'_1, q'_2 and q'_3, i.e., $B_i\gamma_i$ is dominated by σ_i for i=1,2 and 3. If $|\gamma_i|=k$ for i=1,2 and 3 then one can see that these states are also simultaneously reachable via the states $s'_1=[Z_1 \rightarrow \tau_1 A.\tau'_1,\tau''_1]$, $s'_2=[Z_2 \rightarrow \tau_2 A.\tau'_2,\tau''_2]$ and $s'_3=[Z_3 \rightarrow \tau_3 A.\tau'_3,\tau''_3]$. Therefore the algorithm 1 computes the triple reachability of the former states anyway. The dual reachability of the two states with k-symbol lookaheads is similar. Thus, for such states $\pi((r,s))=(r,s)$. Note that the algorithm computes the dual reachability of any pair chosen from q'_1, q'_2 and q'_3, so $\pi((q'_i,q'_j))=(q'_i,q'_j)$ for i,j=1,2 and 3.

Now suppose the states $r_1=[Y_1 \rightarrow \mu_1 .B\mu'_1,\mu''_1]$, $r_2=[Y_2 \rightarrow \mu_2 .B\mu'_2,\mu''_2]$ and $r_3=[B' \rightarrow \mu_3 .\beta\mu'_3,\mu''_3]$, where at least one of the $|\mu'_i\mu''_i|$ is less than k, are simultaneously reachable from q'_1, q'_2 and q'_3. In this case it cannot be argued that the triple reachability of these states can be computed through the states s'_1, s'_2 and s'_3. Therefore the triple reachability of these states and in turn the triple reachability of $r'_1=[B \rightarrow \beta.,\mu'_1\mu''_1]$, $r'_2=[B \rightarrow \beta.,\mu'_2\mu''_2]$ and $r'_3=[B' \rightarrow \mu_3\beta.\mu'_3,\mu''_3]$, and consequently the dual reachability of $r''_1=[\text{SUBGOAL-RED}(p') \rightarrow .\mu'_1\mu''_1,\epsilon]$ and $r''_2=[\text{SUBGOAL-RED}(p') \rightarrow .\mu'_2\mu''_2,\epsilon]$, where p' is the production $B \rightarrow \beta$, may not be detected by the algorithm 1. Note that the dual reachability of any pair from $\{r_1,r_2,r_3\}$ and also any pair from $\{r'_1,r'_2,r'_3\}$ is computed by the algorithm, and such pairs are mapped on themselves. Now suppose that $t_1=[C_1 \rightarrow \alpha_1 .\alpha'_1,\alpha''_1]$ and $t_2=[C_2 \rightarrow \alpha_2 .\alpha'_2,\alpha''_2]$ are reachable from r''_1 and r''_2. If both C_1 and C_2 are elements of N, i.e., $C_1\alpha''_1$ and $C_2\alpha''_2$ are dominated by $\mu'_1\mu''_1$ and $\mu'_2\mu''_2$, respectively, then the simultaneous reachability of t_1 and t_2 could be concluded from the simultaneous reachability of $[Y_1 \rightarrow \mu_1 B.\mu'_1, \mu''_1]$ and $[Y_2 \rightarrow \mu_2 B.\mu'_2, \mu''_2]$. Thus (t_1, t_2) is mapped on itself. Otherwise, if one or both of C_1 and C_2 is the auxiliary symbol SUBGOAL-RED(p'), i.e., $\alpha_1\alpha'_1=\mu'_1\mu''_1$ and $\alpha''_1=\epsilon$ / $\alpha_2\alpha'_2=\mu'_2\mu''_2$ [illegible], then there are corresponding states to t_1/t_2 that exhibit the same characteristics. For instance, if C_1=SUBGOAL-RED(p') then (t_1, t_2) can be mapped on (t'_1, t_2), where depending on whether $|\alpha_1|<|\mu'_1|$, $t'_1=[Y_1 \rightarrow \mu_1 B\alpha_1 \bar{\mu}_1, \mu''_1]$ and $\bar{\mu}_1\mu''_1=\alpha'_1$, or $t'_1=[\text{SUBGOAL-RED}(Y_1 \rightarrow \mu_1 B\mu'_1) \rightarrow \bar{\mu}_1.\alpha'_1, \epsilon]$ and $\mu'_1\bar{\mu}_1=\alpha_1$. Cleatly, if (t_1, t_2) exhibits an unresolvable conflict then (t'_1, t_2) does so too.

Furthermore, the same kind of mapping applies to the successors of $[\text{SUBGOAL-RED}(C_1 \rightarrow \alpha_1\alpha'_1) \rightarrow .\alpha''_1, \epsilon]$ and $[\text{SUBGOAL-RED}(C_1 \rightarrow \alpha_1\alpha'_1) \rightarrow .\alpha''_2, \epsilon]$ if $C_1=C_2$ and $\alpha_1\alpha'_1=\alpha_2\alpha'_2$. □

The above theorem shows that computing triple reachability of states up to the point of first transitions over similar potential ϵ-arcs guarantees the simultaneous transitions into the states with lookaheads shorter than k symbols. Afterwards, for any pair of paths that traverse alike potential ϵ-arcs there is a pair of paths that do not go through similar potential ϵ-arcs, and if the first pair detect an unresolvable conflict then so do the second pair. As an example consider the fragment of a deterministic LRRL(2)-CFSM that is depicted in Diagram 4.8. In the non-deterministic version of the CFSM, the conflicting states q=[M →n.,ε] and q'=[M →n.,ε] are simultaneously reachable through two pairs of paths:

(1):

I. [Z $\to\alpha$.,AM] [SUBGOAL-RED(Z $\to\alpha$) $\to$.AM,ϵ] [A $\to$.a,M] [A $\to$a.,M] [SUBGOAL-RED(A $\to$a).M,ϵ] [M $\to$.n,ϵ] [M $\to$n.,ϵ]

II. [Z $\to\alpha$.,AN] [SUBGOAL-RED(Z $\to\alpha$) $\to$.AN,ϵ] [A $\to$.a,N] [A $\to$a.,N] [SUBGOAL-RED(A $\to$a).M,ϵ] [N $\to$.n,ϵ] [N $\to$n.,ϵ]

(2):

I. [Z $\to\alpha$.,AM] [SUBGOAL-RED(Z $\to\alpha$).AM,ϵ] [SUBGOAL-RED(Z $\to\alpha$)A.M,ϵ] [M $\to$.n,ϵ] [M $\to$n.,ϵ]

II. [Z $\to\alpha$.,AN] [SUBGOAL-RED(Z $\to\alpha$).AN,ϵ] [SUBGOAL-RED(Z $\to\alpha$)A.N,ϵ] [N $\to$.n,ϵ] [N $\to$n.,ϵ].

However, the algorithm 1 can only detect the dual reachability of these states only through the second pair of paths. Nevertheless this is sufficient for detecting the unresolvable conflict.

Similarly, the algorithm cannot detect the simultaneous reachability of the states [M $\to$m.,ϵ] and [SUBGOAL-RED(A $\to$a) $\to$m.,ϵ] through the paths:

I. [Z $\to\alpha$.,AM] [SUBGOAL-RED(Z $\to\alpha$) $\to$.AM,ϵ] [A $\to$.a,M] [A $\to$a.,M] [SUBGOAL-RED(A $\to$a) $\to$.M,ϵ] [M $\to$.m,ϵ] [M $\to$m.,ϵ]

II. [Z $\to\alpha$.,Am] [SUBGOAL-RED(Z $\to\alpha$) $\to$.Am,ϵ] [A $\to$.a,m] [A $\to$a.,m] [SUBGOAL-RED(A $\to$a) $\to$.m,ϵ] [SUBGOAL-RED(A $\to$m.,ϵ].

However, corresponding to these states, there are two states [M $\to$m.,ϵ] and [SUBGOAL-RED(Z $\to\alpha$) $\to$Am.,ϵ] such that their simultaneous reachability is detected by the algorithm through the paths:

I. [Z $\to\alpha$.,AM] [SUBGOAL-RED(Z $\to\alpha$) $\to$.AM,ϵ]

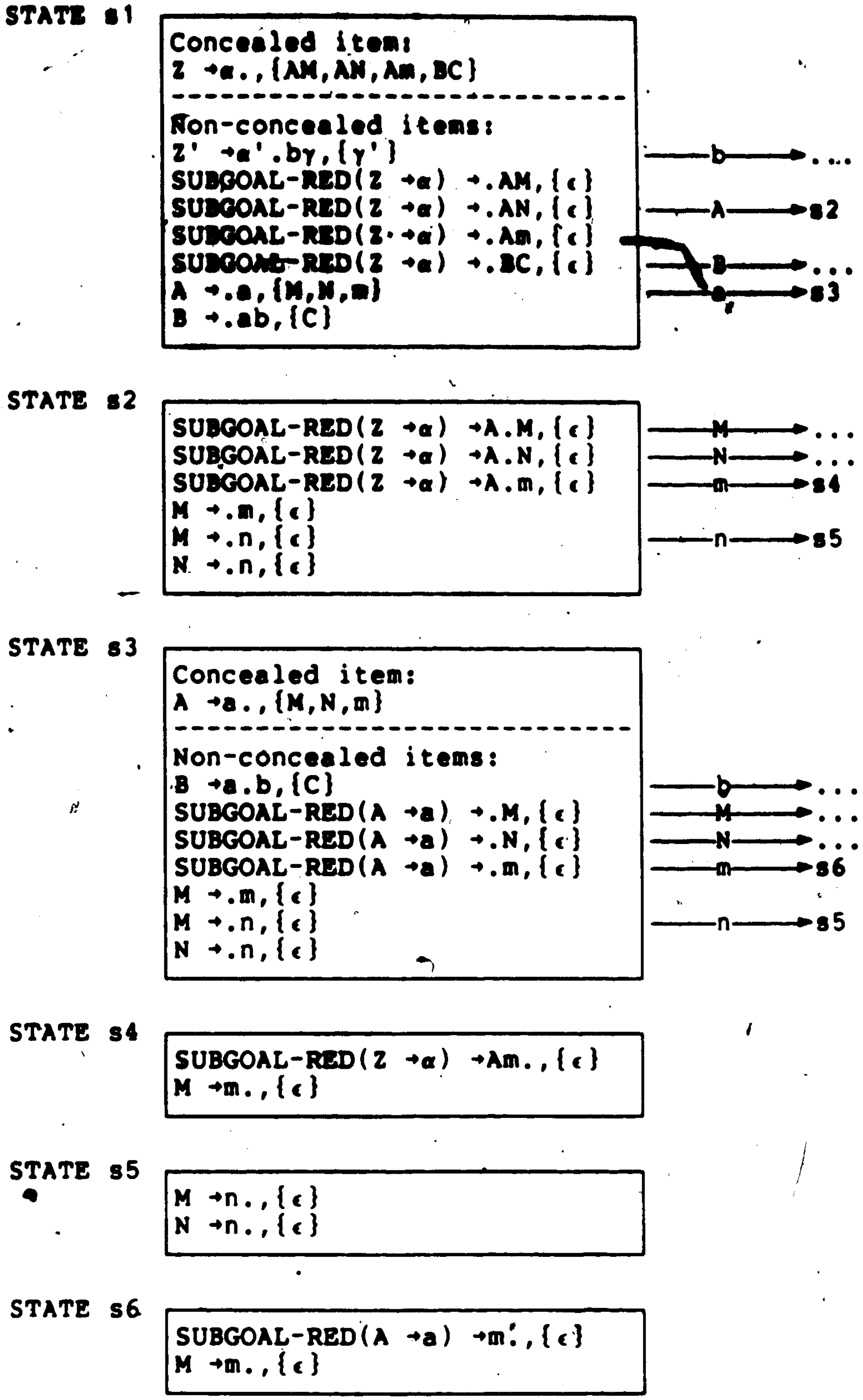
[SUBGOAL-RED(Z $\rightarrow\alpha$) $\rightarrow$A.M,ϵ] [M $\rightarrow$.m,ϵ] [M $\rightarrow$m.,ϵ]

II. [Z $\rightarrow\alpha$.,Am] [SUBGOAL-RED(Z $\rightarrow\alpha$) $\rightarrow$.Am,ϵ]

[SUBGOAL-RED(Z $\rightarrow\alpha$) $\rightarrow$A.m,ϵ] [SUBGOAL-RED(Z $\rightarrow\alpha$) $\rightarrow$Am.,ϵ],

which again is adequate to show the non-LRRL(2) property of the grammar.

Note that the method of testing given here would not work for the LR(k,t) grammars. For example, for the first pair of states, q and q', there would be two numbers indicating the number of bypassed phrases. If these numbers are h and h' for the first pair of paths, then they are h-1 and h'-1 for the second pair of paths.



STATE s1

```
Concealed item:
Z →α.,{AM,AN,Am,BC}
--------------------------------
Non-concealed items:
Z' →α'.bγ,{γ'}
SUBGOAL-RED(Z →α) →.AM,{ε}
SUBGOAL-RED(Z →α) →.AN,{ε}
SUBGOAL-RED(Z →α) →.Am,{ε}
SUBGOAL-RED(Z →α) →.BC,{ε}
A →.a,{M,N,m}
B →.ab,{C}
```

STATE s2

```
SUBGOAL-RED(Z →α) →A.M,{ε}
SUBGOAL-RED(Z →α) →A.N,{ε}
SUBGOAL-RED(Z →α) →A.m,{ε}
M →.m,{ε}
M →.n,{ε}
N →.n,{ε}
```

STATE s3

```
Concealed item:
A →a.,{M,N,m}
--------------------------------
Non-concealed items:
B →a.b,{C}
SUBGOAL-RED(A →a) →.M,{ε}
SUBGOAL-RED(A →a) →.N,{ε}
SUBGOAL-RED(A →a) →.m,{ε}
M →.m,{ε}
M →.n,{ε}
N →.n,{ε}
```

STATE s4

```
SUBGOAL-RED(Z →α) →Am.,{ε}
M →m.,{ε}
```

STATE s5

```
M →n.,{ε}
N →n.,{ε}
```

STATE s6

```
SUBGOAL-RED(A →a) →m.,{ε}
M →m.,{ε}
```

Diag. 4.8
A fragment of deterministic LRRL(2)-CFSM

Now the algorithm for (type II basic) LRRL(k) testing for a fixed k can stated as follows.

Algorithm 2: LRRL(k) testing

1. Construct the non-deterministic version of CFSM.
2. Compute R3 and obtain R2 from it.
3. Check that none of the pairs (q,q') in R2 exhibit an unresolvable conflict.

The computation of R3 for the non-deterministic machine of the example grammar, (shown in Diag. 4.7), shows that the pairs (0,0),(2,3),(6,3),(12,8),(18,15),(25,15) and (31,21) (among many others) belong to R2, with the last pair exhibiting an unresolvable conflict. Therefore, the algorithm 2 concludes that the example grammar is not LRRL(1).

The next theorem considers the complexity of LRRL(k) testing.

Theorem 4.9:

The non-LRRL(k) condition of a grammar G, with a fixed k, can be tested in P-space, and non-deterministically in time $O(|G|^{k+2})$ and deterministically in time $O(|G|^{3k+4})$.

Proof: In the non-deterministic machine M, the number of states of the form $[A \rightarrow \alpha.\beta,\sigma]$, where A is a non-terminal symbol, is less than or equal to $|G|(1+|V|+\ldots+|V|^{k} <(k+1)|G|^{k+1}$. The number of states of the form $[\mathrm{SUBGOAL}(p)\rightarrow\alpha.\beta,\epsilon]$ is at most $|P|(2|V|+\ldots+(k+1)|V|^{k})<k(k+1)|G|^{k+1}$. Therefore, the total number of states of M is less than $(k+1)^{2}|G|^{k+1}$. The number of transitions of the types (1) and (3) at most is equal to the number of the states, and thus is less than $(k+1)^{2}|G|^{k+1}$. For each state there are at most $|P|$ transitions of the type (2). Therefore, the size of δ is bounded by $(k+1)^{2}|G|^{k+2}$, and the size of M, i.e., $|M|=|Q|+|\delta|<(k+1)^{2}(|G|^{k+2}+|G|^{k+1})$. Thus, the non-deterministic machine M can be constructed in time $O(|M|)$, i.e., time $O(|G|^{k+2})$. So trivially, the non-LRRL(k) condition can be tested non-deterministically in time $O(|G|^{k+2})$.

For a deterministic test, one needs to construct the relation R3 according to the algorithm 4, which is the dominating factor. R3 can be stored as a $|Q|X|Q|X|Q|$ bit map, and the closure successors for each state can be listed. Each triple (q_1,q_2,q_3) will be pushed on the STACK at most once. Therefore, the total time spent for insertion is $O(|Q|^{3})$ or $O(|G|^{3k+3})$. The total time spent in statements 2.1 and 2.3, i.e., time for processing shift and potential ϵ-transitions is proportional to the size of R3, which is $O(|G|^{3k+3})$. The total time spent in statement 2.2.1 processing closure transitions is proportional to $|Q|^{3}|P|$, i.e., is $O(|G|^{3k+4})$. Similarly, times spent in steps 2.2.2 and 2.2.3 are $O(|G|^{3k+4})$. Therefore, a deterministic test at the worst case can be carried out in time $O(|G|^{3k+4})$.

From the size of M and the size of R3, it is trivial to see that both non-deterministic and deterministic tests can be carried out in polynomial space in the size of G. □

In practice, one can consider a stack of subsets of Q of the forms $\{q_1\}$, $\{q_1,q_2\}$ and $\{q_1,q_2,q_3\}$ instead of the stack of ordered tuples. Although, such optimization considerably contributes to the efficiency of the testing algorithm, it does not however change the order of the complexity.

From the diagnostics point of view, the algorithm 1 does not provide as much information as one wishes. Unlike deterministic CFSM construction, the algorithm 1 does not give the conflicting situations that involve more than two items, and does not provide the paths from the initial state to the conflicting items. Moreover, generally it is not expected that the deterministic CFSM for a real grammar to have an exponential number of states in terms of the size of the grammar. An approximate evidence for this is the size of LR(k) parsers for programming languages and the size of Marcus' parser for a deterministic subset of natural language. (It is unlikely that with no automatic generator, Marcus' parser would have been built by trial and error process if its size was exponential.)

Therefore, one may consider the purpose of this section is as one going through an exercise to prove that LRRL(k) testing is not an intrinsically exponential problem, which otherwise should have been left as open problem in the border region between the problems in P and the class of **NP-Complete** problems. Still, it remains an open problem (and from author's point of view it is doubtful) that the upper bound for the complexity of LRRL(k) testing can be sharpened by any method similar to the one given in [HuSU 75] for LR(k) testing.

4.5 Informal comparisons with alternative parsing methods

So far, it has been shown that the LRRL(k) grammars are not included in the class of LR(k) or the class of LR(k,t) grammars for any finite t. Also, it has been established that the class of Bounded Context Parsable BCP(m,n) grammars does not contain all the LRRL(k) grammars [Nozo 86]. The LRRL(2) grammar G_2:

$$\begin{aligned} S &\to d \\ S &\to A\ S\ a \\ S &\to B\ S\ b \\ A &\to a \\ B &\to a \end{aligned}$$

clearly shows that the LRRL(k) grammars are not LR-regular. Since in G_2, the parsing patterns for the reduction of the nth 'a', from the beginning of a sentence, to an 'A' or to a 'B' are given by $(A,B)^{n-1}\ a\#a^*\ d(a,b)^*\ a(a,b)^{n-1}$ and $(A,B)^{n-1}\ a\#a^*\ d(a,b)^*\ b(a,b)^{n-1}$. Therefore, the right contexts for the reduction of the productions A →a and B →a cannot be described in terms of a finite set of disjoint regular expressions.

The LRRL(k) grammars are included in the class of FSPA(k) grammars. The efficiency and the storage requirements of the LRRL(k) and FSPA(k) parsers for particular grammar are the same. However, in Chapter 2, it was discussed that the membership problem for the FSPA(k) grammars is undecidable, and thus without a reservation, one cannot set out to build a phrase finding automaton for an arbitrary grammar according to the FSPA scheme, since the construction is not guaranteed to terminate.

Marcus' parser, being partially top-down, cannot handle a grammar like G_2. However, in the case of grammars like G_1 of Chapter 2 which can be handled both by a Marcus-style parser (written in PIDGIN), and by an LRRL(k) parser, it would be interesting to compare the two parsers.

The parsing control mechanism for the Marcus-style parser is described by the PIDGIN grammar of Chapter 2. For the LRRL(k) parser, it can be given by the optimized Table 3.4 of Chapter 3 which requires much less storage. The size of the stack plus the buffer is the same for both parsers. Apart from those, in the case of Marcus-style parser the interpreter PARSIFAL must be resident in the main memory, while in the case of LRRL(k) parser, only the code for the algorithm C which is a very small program must be in the main memory. From the examples showing parsing of the sentence $a^2dbad^2b^2$, one can see that the number of steps for the two parsers are roughly equal, and is linear in the size of input sentence. For each step however, the LRRL(k) parser needs to access a single entry in the parse table (a memory location with an indirect address, if the table stored as an array), while in the case of Marcus-style parser, PARSIFAL has to interpret all the rules in the active packets. Thus the constant factor for the Marcus-style parser is much higher, and the parser considerably is slower. Therefore, for better efficiency Marcus' parser needs a PIDGIN compiler to compile PIDGIN grammars into executable codes, rather than interpreting them by PARSIFAL. But still it would not be as efficient as an LRRL(k) parser.

Lastly, the major factor that differentiates the two parsers is that the LRRL(k) parser is obtained automatically with no effort by simply running the context-free grammar through the parser generator. While, a considerable amount of trial and error is required to produce a PIDGIN grammar.

By the foregoing discussion, it becomes apparent that in parsing grammars like G_1 or G_2, the real alternatives that should be compared against the LRRL parsers, in terms of their efficiencies, are LR(k,∞), Earley's, backtracking and parallel parsers.

4.5.1 Comparison with LR(k,∞) parsers

An LR(k,∞) parser runs in a number of steps linearly proportional to the length of an input sentence. However, each step required computation of a set of LR(k,∞) items which is roughly equal to building a CFSM state in the construction of an LRRL(k) parser. Therefore, the constant factor in the time complexity is very large. In terms of storage requirements, it employs a stack of sets of items which is very unattractive in comparison with the same size stack of pairs of a state number and a symbol that LRRL parsers use. Therefore, one can conclude that an LRRL(k) parser for a particular grammar is much more efficient than an LR(k,∞) parser for the same grammar.

However, the coverage of the LR(k,∞) parsers are much broader and in fact they can parse some non-deterministic unambiguous grammars. On other hand, because of the undecidable membership one would not know that an arbitrary grammar can be parsed in that fashion.

4.5.2 Comparison with Earley's parser

Earley's parser [Earl 70] and variations of it by Valiant [Vali 75], and Graham, Harrison and Ruzzo [GrHR 80] are the most efficient general context-free parsing algorithms. Earley's algorithm requires $O(n^3)$ time and $O(n^2)$ space at the worst case for parsing ambiguous grammars (where n is the length of input sentence). Unambiguous grammars require $O(n^2)$ time. The two variations run in subcubic time. The CKY (Cocke-Kasami-Younger) algorithm [Youn 67] can be considered as the top-down version of

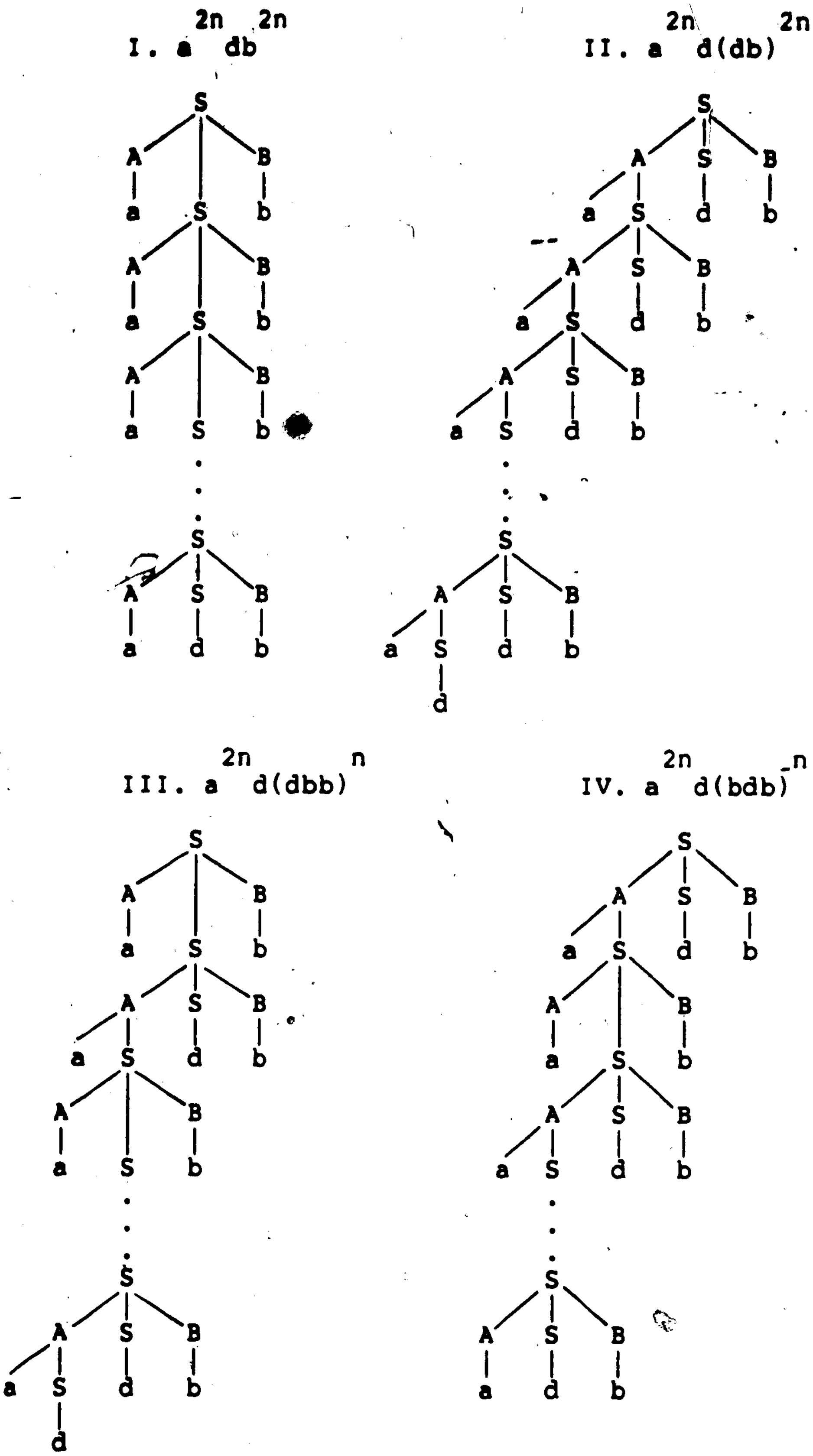
Earley's algorithm that only handles the grammars given in Chomsky normal form.

Earley's parser builds a list of n states for an input of size n. Each state is a collection of items and unlike LRRL(k) and LR(k,∞) they cannot be popped from the stack as the parse proceeds. Therefore, even in terms of average number of stack nodes the algorithm is quite expensive. Although, for finite state grammars such as LRRL(k), it can be shown that the list of states can be constructed in $O(n)$ time, still there is a substantial overhead in that the algorithm manipulates the list as it proceeds.

Therefore, one can conclude that it would be very inefficient to parse an LRRL(k) grammar with Earley' algorithm.

4.5.3 Comparison with backtracking parsers

The recursive-descent and ATN (Augmented Transition Networks) parsers may be the most popular ones among the parsers with unlimited backtracking, with the latter having the capability to handle the non-context-free features of a language. In order to compare LRRL parsers with these methods, it would be constructive to consider the four sentences in $L(G_1)$ and their syntax trees depicted in Figure 4.4.



I. $a^{2n}db^{2n}$

II. $a^{2n}d(db)^{2n}$

III. $a^{2n}d(dbb)^{n}$

IV. $a^{2n}d(bdb)^{n}$

Fig. 4.4

From the example sentences, it is obvious that a recursive-descent parser will backtrack 2^{2n} times to the 2n-th 'a' in a sentence, 2^{2n-1} times to the (2n-1)th 'a',...,and twice to the first 'a'. An ATN parser can do better if HOLD registers are used intelligently. If the parser recognizes that a segment of input sentence is parsed successfully in one way, then it can hold the parsed construct in a special register and thus will not need to reparse the segment in the same way. For example, in the grammar G_1, a 'd' can only be reduced to an 'S', and consequently when a segment of the input parsed as 'ASB' the dominating 'S' can be held in a register.

Writing of recursive-descent parsers is very easy, (the feature that makes them popular), specially in programming languages such as PROLOG and SNOBOL. In these languages, roughly listing of productions is the only thing that one needs and the bulk of work is passed to the programming language system. Such parsers extensively backtrack, and use of them for LRRL(k) grammars must be considered disastrous.

The bottom-up backtracking parsers, e.g., the one that is based on an LR(k) table with multiple entries, generally do better than top-down recursive-descent parsers. However, their performance in the case of LRRL(k) grammars like G_1 is not any better than recursive-descent parsers.

4.5.4 Comparison with parallel parsers

Parallel parsing, strangely enough, has become to denote two different concepts in parsing (approximating MISD and MIMD concepts in multiprocessor environments). The first is that when a parser encounters an ambiguous decision point with n choices, it branches into n processes each getting a copy of the stack, and following a different parsing path. Therefore, in the example sentences, by the time that mth 'a' is reached there would be 2^m processes manipulating the same number of stacks.

In the second meaning, the input segment is partitioned into segments, and n processes, usually running on n processors, each get a segment of input. Essentially there are two categories of these parsers. In one which is developed by Fischer [Fisc 75], each parser on encountering an ambivalent point follows multiple paths in pseudo-parallel, i.e, manipulates multiple stacks. In the second kind introduced by Mickunas and Schell [MiSc 78], each parser on encountering an ambiguous decision point transfers the content of its stack to the left neighbouring process.

Parallel parsers in the second sense can be constructed for variety of grammar classes, and as one can see from Fischer's model, each parser in this method can behave like a parallel parser in the first sense. However, the efficiency of these parsers are in doubt. In the worst case, each of the n parallel parsers in the first category may follow more than n paths on 1/n length of the input. Thus, if the grammar is linearly parsable on a single processor, the multiprocessor parser will do worse than a single processor. In the second category, all the input may be passed to the first (leftmost) processor while the other n-1 processors sit idle. Cohen, Hickey and Katcoff [CoHK 82] in an analytical study show that the speedup in such parsers is not related to the class of grammars, but depends on the characteristics of individual syntax trees such as their heights. Nevertheless, it should be mentioned that the work of Mickunas and Schell in producing parallel scanners is quite remarkable and greatly contributes to the efficiency of the scanning phase, which is the slowest phase in compilers, in a multiprocessor environment.

The MLR parser recently developed by Tomita [Tomi 84,85] is an interesting parallel parser in the first sense. The MLR parser is based on an LR(0) parse table with multiple entries. Such a table for the grammar G_1 is given by Table 4.1. The MLR parser initially starts as single process. As soon as it reaches to a state with n different actions, the process is split into n processes that are executed individually in parallel. However, they are synchronized in a way that they always look at the same input symbol. If the system finds that two or more processes are in the same state, these processes will be combined. Therefore, the number of processes at any given time is bounded by the number of states in the parse table. Instead of multiple stacks there is a multilinked structure as shown in Figure 4.5 for the example grammar. This structure in fact combines the common segments of the traditional multiple stacks. This combining mechanism guarantees that no part of an input_sentence is parsed more than once in the same manner.

	Symbols						
States	a	b	d	S	A	B	$
0	g3,r3		r1	g1	g2		
1							r0
2	g3,r3		r1	g4	g2		
3	g3,r3		r1	r4	g2		
4		r5				r2	

Grammar G_1:
(0) GOAL →S
(1) S →d
(2) S →A S B
(3) A →a
(4) A →a S
(5) B →b

Table 4.1
LR(0) parse table with multiple entries

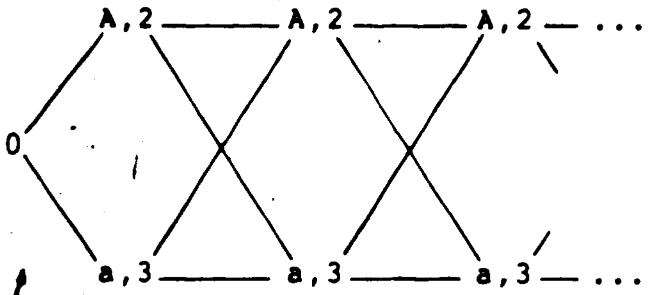


Fig. 4.5

The multilink structure for MLR parser

Though Tomita's approach is an attractive one, from software engineering point of view it does seem to have problems. Combining independent processes into a single one is not an easy task. Furthermore, complexity of such combining, i.e., synchronizing and management of parent-child relationship of processes, and management of the shared data is not dealt with and simply relegated to some entity which is only referred to as 'the system'. One may easily see that if the algorithm is to be run on a multiprocessor system the cost of overhead in terms of communication, i.e., concrete complexity, is prohibitive. On the other hand, if it is implemented on a single processor, then it is only Earley's algorithm in disguise, with the exception that the items, without their initial state numbers, are given in precompiled form. Note that Earley's algorithm also does not parse a segment of input more than once in the same manner.

Again the LRRL(k) parsers in comparison with the MLR parser will do much better. Ignoring all the difficulties with processes management, by the time one reaches the mth 'a' in the example sentences, 2m processes will be created in the MLR parser. The two most recent processes will be active and the others will be in wait mode. A multilink structure of the size 2m nodes will created. While, the LRRL(k) parser will have only a stack of m nodes. However, it should be reminded that the MLR parser is a general purpose parser.

4.5.5 Comparison with an oracle model

Perhaps the most interesting comparison will be a one against an oracle model based on LR parsers. In this model it is assumed that when the parser reaches an ambiguous point, i.e., a state with multiple actions, the oracle will choose the correct action. Figure 4.6 shows that the oracle model parses the sentence $a^2dbad^2b^2 \in L(G_1)$ in 22 steps (i.e., one more than the number of nodes in the parse tree). It took 28 steps for the type I basic LRRL(2) parser to parse the same sentence (Fig. 3.3 of Chapter 3). In fact if a sentence contains m number of a's, then the LRRL(2) parser will take 2m more steps than the oracle model to parse sentence. This is true because the a's are the ambiguous elements in the grammar G_1. To resolve the role of each 'a', the LRRL(2) parser transfers its two fully reduced right context to the buffer, which later are shifted back onto the stack contributing two more steps in parsing time.

In terms of storage requirements, one observes that the LRRL(2) parse table (Table 3.4) is only marginally (i.e., one extra row) larger than the parse table in the oracle model (Table 4.1).

State stack	Parse stack		Current symbol	Remainder of input
-	-		-	aadbaddbb$
0	-		a	adbaddbb$
0,3	a		a	dbaddbb$
0,3	aa	reduce 3	A	dbaddbb$
0,3,2	aA		d	baddbb$
0,3,2	aAd	reduce 1	S	baddbb$
0,3,2,4	aAS		b	addbb$
0,3,2,4	aASb	reduce 5	B	addbb$
0,3,2,4	aASB	reduce 2	S	addbb$
0,3	aS	reduce 4	A	addbb$
0,2	A		a	ddbb$
0,2,3	Aa		d	dbb$
0,2,3	Aad	reduce 1	S	dbb$
0,2,3	AaS	reduce 4	A	dbb$
0,2,2	AA		d	bb$
0,2,2	AAd	reduce 1	S	bb$
0,2,2,4	AAS		b	b$
0,2,2,4	AASb	reduce 5	B	b$
0,2,2,4	AASB	reduce 2	S	b$
0,2,4	AS		b	$
0,2,4	ASb	reduce 5	B	$
0,2,4	ASB	reduce 2	S	$
0,1	S		$	-
0	reduce 0: accept		$	-

Fig. 4.6
Parsing $a^2dbad^2b^2$ in oracle model

CHAPTER 5

GENERALIZATIONS OF LRRL(k) GRAMMARS

Chapters 3 and 4 dealt with development and properties of two types of basic LRRL grammars. The present chapter is concerned with the generalizations of these grammars. In addition, the subclasses of LRRL grammars and the grammars with ϵ-rules are also discussed in this chapter. The chapter concludes with a detailed examination of Hunt's statement regarding the generalized forms of LR grammars.

The difference between the two types of basic grammars is that the parsers for the type I grammars backtrack to a delayed decision point regardless of whether it is a reduction or a shift operation. The parsers for the type II grammars only backtrack to a deferred reduction point. The type I grammars are included in the type II grammars. The generalizations can be considered equally for the type I or the type II grammars, preserving the fundamental difference between the two types in terms of backtracking and inclusion of the type I in the type II grammars. However, in this chapter only the generalizations of the type II grammars will be considered.

Although the class of basic LRRL(k) grammars is both intuitively and formally interesting, there are two points that may be considered as pitfalls for these grammars:

(i) The class of basic LRRL(k) grammars with a fixed k does not include the class of LR(k) grammars with the same k.

(ii) The class of basic LRRL(k) grammars with a fixed k is not included in the class of basic LRRL(k+1), but in the class $\bigcup_{j=k+1}^{2k+1}$ basic LRRL(j).

However, the two problems can be easily remedied in the form of modified LRRL(k) grammars. Sections 5.1 and 5.2 deal with the ELRRL(k) and MLRRL(k) grammars that overcome these problems. Section 5.3 describes the GLRRL(k) grammars that include the LR(k,t) grammars as a proper subset.

The subclasses of the LRRL grammars are considered in Section 5.4. Finally, Section 5.5 deals with the LRRL grammars with ϵ-productions.

5.1 The ELRRL(k) grammars

Most of the LR(k) grammars fall within the class of the basic LRRL(k) grammars for the same k. However, despite overwhelming generality of the latter grammars, there are some subtle LR(k) grammars that are not basic LRRL(k) for the same k or are not basic LRRL(k) for any value of k. An example is the grammar G_1:

$S \rightarrow A\ B\ m \mid A\ B'\ n \mid A'\ C\ p$

$A \rightarrow a \qquad A' \rightarrow a$

$B \rightarrow b \qquad B' \rightarrow b \qquad C \rightarrow c$

G_1 is LR(1) but not LRRL(1). However it is LRRL(2). On the other hand the LR(1) grammar G_2:

$S \rightarrow A\ B\ m \mid B\ m \qquad A \rightarrow C \mid A\ C \qquad B \rightarrow a \qquad C \rightarrow a$

is not LRRL(k) for any k. When the initial 'a' in the input sentence is encountered, the decision to reduce it to 'B' or 'C' relies on the parsability of the right context. The right context for 'B' is 'm'. However for any k, 'C^{k-1} B' and 'C^{k}' are right contexts for the reduction to 'C', and these two strings cannot be recognized from each other since they derive the same terminal string. The problem of course lies in the insistence that the right context be parsable.

It is desirable to combine properties of the basic LRRL(k) parsing with those of LR(k), to obtain a new class of **ELRRL(k):** Extended LRRL(k) grammars. The class of ELRRL(k) grammars will include all the LR(k) grammars with the same k.

To achieve this goal, the step (3-ii) of the CFSM construction algorithm **A-II** (Section 3.2.11) is modified so that it first checks to see if a resolution can be made only by looking ahead at the terminal symbols in the input sentence as it is illustrated below.

```
For i:=m+1,...,n let
   $l_i$ ={ $\mathrm{FIRST}_k$ (σ)| σ∈$L_i$ }.
$l$={ $\mathrm{FIRST}_k$ (σ)| σ∈SHL(s)}.
IF
   $l_i \cap l = \emptyset$ for i=m+1,...,n
   and
   $l_i \cap l_j = \emptyset$ for i,j=m+1,...,n, i≠j
Then
   conflicts in the state can be resolved just by looking
   ahead at the few terminal symbols in the remainder
   of input sentence. Or equally,
   Add items SUBGOAL-RED($p_i$) →.x,{ε}
   for all x∈$l_i$, for i=m+1,...,n;
   conceal the conflicting reduction items $I_i$
   for i=m+1,...,n, and proceed to step (3-iii).
Else
   Perform step (3-ii) as before.
```

To obtain a rather formal definition for type I ELRRL grammars, one only needs to change the condition (a) of the definition A'-1 in Section 4.3.3 to read "(a) G is LR(k)". The derivations oriented definition of the (type II) ELRRL grammars (i.e., the ones that are discussed here) can be obtained by changing the same condition in an equivalent definition for type II basic grammars.

5.2 The MLRRL(k) grammars

Suppose a grammar G is basic LRRL(k) but not basic LRRL(k') for some k'>k. The implication is that when an LRRL(k')-CFSM for G is constructed, conflicts in some states will not be resolvable. But, if one considers an auxiliary item such as SUBGOAL-RED(p_i) $\to .\alpha, \{\epsilon\}$ in the LRRL(k)-CFSM, there would be corresponding auxiliary items $\{$ SUBGOAL-RED(p_i) $\to .\alpha\beta_j, \{\epsilon\} \mid j=1,\ldots,m \}$ in the LRRL(k')-CFSM. Since G is LRRL(k) there exists a state in LRRL(k')-CFSM such that its basis consists of items SUBGOAL-RED(p_i) $\to \alpha.\beta_j, \{\epsilon\}$, $j=1,\ldots,m$. However, further expansions of β_j's may not end up in resolvable states. To remedy this situation, one can modify the CFSM construction algorithm **A-II** in such a way that states with basis consisting of only auxiliary items SUBGOAL-RED(p) $\to \alpha.\beta_j, \{\epsilon\}$, $j=1,\ldots,m$ to be considered as final states and only $|\alpha|$ symbols (instead of usual $|\alpha\beta_j|$ symbols) be transferred onto the buffer. The modified algorithm defines the **MLRRL(k)**: Modified LRRL(k) grammars.

Clearly, the class of MLRRL(k+1) grammars includes MLRRL(k) grammars, and the class of basic LRRL(k) grammars is contained in the MLRRL(k) class. Obviously, one may combine properties of the LR(k) grammars with those of the MLRRL(k) grammars obtaining a new class of **EMLRRL(k)**: Extended Modified LRRL(k) grammars.

5.3 The GLRRL(k) grammars

The lookahead policy used in basic LRRL(k) grammars may be viewed as the opposite extreme to the one employed in LR(k) grammars. In LR(k) parsing the lowest level of nodes, i.e., terminals are used as lookaheads, while in basic LRRL(k) parsers the highest level of nodes that follow the current construct, (i.e., the fully reduced right context), act as lookaheads. The ELRRL(k) or EMLRRL(k) techniques combine the two policies. An immediate question arises here is that whether reductions can be based on some intermediate-level nodes of the right contexts, i.e., the decision for the reduction of a questionable phrase can be reached before the complete parse of the k-symbol fully reduced right context is obtained. The answer is positive and indeed several variety of grammar classes with decidable memberships can be defined in this fashion. However, it seems always there are two decisions to be made in such definitions. One is the size of buffer, i.e., the number of nodes that are parsed before the deferred reduction of the first questionable phrase is made. Obviously, all these nodes are descendents of the k fully reduced right context of the questionable phrase. The size of the buffer could be taken to be k as before, or some other number greater than k, (e.g., a multiple of k), or even an unbounded buffer can be used. The second is whether one should choose to leave some of the phrases in the lookahead string unreduced before returning to the questionable phrase. In fact, this has to do with the sequencing of reductions, that is, if at a point one realizes that more than one phrase in the sentential form can be reduced then which phrase should be reduced first. It seems that Szymanski has also encountered the problem of such an ordering which can be used profitably by a parser, and he has suggested it as a further research in [Szym 73,p 169]. The LR(k,t) parsers in the way they are constructed reduce the leftmost phrase first.

Here, two classes of parsers will be considered. The parsers in the first class employ unbounded buffers, and parsers in the second class use finite size buffers. The lookahead policy adopted is the use of at most k terminal symbols or at most k symbols reduced right context at arbitrary level, with the exception that the last phrase (if any) in the lookahead string may be left unreduced if it cannot be reduced without the use of some lookaheads. The scheme may be illustrated better by using Figure 5.1. Suppose at some stage in parsing, the reduction of the phrase $X_1...X_t$ to the non-terminal X becomes questionable, and one proceeds to parse the k-symbol fully reduced right context for this reduction, i.e., $A_1...A_k$. It may be that by the time the parse of B_m is obtained, it becomes apparent that the reduction of $X_1...X_t$ to X was appropriate. In generalized LRRL parsing, one is allowed to back up to the deferred reduction point if m≤k. Otherwise, one continues with the parsing of right context, obtaining higher level nodes until one reaches a state in which the number of nodes that lie to the right of the questionable phrase is less than or equal to k, or ends up with the complete parse of the string $A_1 \ldots A_k$. In addition, if $B_{n+i} \ldots B_m$ is a phrase that can be reduced to some non-terminal B without using any lookaheads, that reduction is made before backtracking to reduce the X-phrase. This reduction contributes to the efficiency of the parser since fewer nodes are put in the buffer. In LR(k,t) parsing such phrases are left unreduced before backing up.

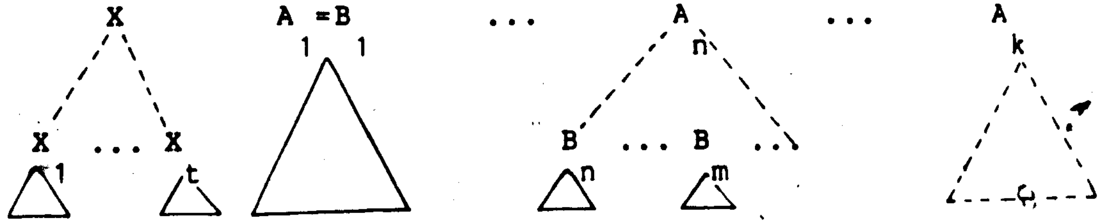


Fig 5.1
Lookaheads in GLRRL parsing

The difference between the two classes is that with an unbounded buffer the lookahead policy is applied recursively with no reservations at any ambivalent point in parsing the lookahead context itself. In the model with bounded buffer, the policy is applied recursively as long as the number of nodes which should be deleted from the buffer while backtracking to the first questionable phrase does not exceed a certain limit n. Otherwise, the policy used in basic LRRL parsing, (i.e., parsing of fully reduced context), is employed. Note that at this stage, i.e., while parsing the last fully reduced context, the parser forgets the surpassing of the above limit, and the whole policy recursively starts all over again. However, the parser will not backtrack to this point until it obtains the complete parse of the fully reduced context, which at most is k symbols. Thus, when the parser backtracks to the first questionable phrase there would be at most n+k nodes in the buffer. The lookahead policy in the bounded buffer case may be better understood by considering that the accumulated backtrackings over the nodes which are not fully reduced contexts never exceeds n symbols.

The derived classes of grammars in these methods will be denoted by **GLRRL(k)**: Generalized LRRL(k) grammars with unbounded and bounded buffers respectively. It will be shown that these grammars with unbounded buffers, and with a suitable size bounded buffers properly include Szymanski's LR(k',t) grammars for k=k't, and thus they provide the most powerful class of grammars that are based on LR parsing concepts while they are known to retain the decidability of the membership problem in the class.

The development of a CFSM for these classes is slightly more involved than that of the basic LRRL(k) grammars and is described in the following subsections.

5.3.1 GLRRL(k) parser with unbounded buffer

The items used in the construction of a GLRRL(k)-CFSM, in addition to the usual core and lookahead segments, have two tags that carry extra information. Thus each item is of the form $(A \rightarrow \alpha.\beta,L,[p,m])$. The tag p is an element of $\{-,0,1,\ldots,|P|\}$, and $m\epsilon\{-,0,1,\ldots,k\}$. Furthermore, for an item either both tags are numbers, or both are '-' (which in this case, both tags can be removed. However, for sake of clarity they are retained in the algorithm and the examples that follow). When p=i (a number), it indicates that the last phrase whose reduction was postponed, is the ith production in the grammar. In other words, an item not only indicates a partial parse of the current construct, it may also show in the context of what suspected phrase whose reduction is deferred, the current construct occurs. When m is a number, it indicates the number of nodes that lie between the last questionable phrase and the current point, i.e., the number of symbols that are on the top of the stack above the last suspected phrase. Clearly, during parsing of a sentence, if one reaches to a state that all the items in that state have the same tags [p,m], then the reduction of the last questionable phrase at m states back via the production p becomes evident. In three situations both tags are '-'. (1) When no phrase is bypassed. (2) When the current construct occurs in more than one context, i.e., when in more than one way one can say what phrase has been bypassed last. (In terms of a multiple path parser this simply means that the current construct is on more than one path). (3) When the current construct lies more than k symbols far from the last bypassed phrase, i.e., when the number of nodes on the top of the stack above the last questionable phrase is more than k. One may note that by this time, it may become apparent that the current construct is in unique context and thus one can decide about the reduction of the last bypassed phrase. But, since the number of nodes that one needs to backtrack over them is more than k, these nodes do not really constitute a k-symbol lookahead for the questionable phrase. However, one may see that a more general class of grammars (with table driven parsers and decidable membership) can defined that way (i.e., by allowing backtrackings over more than k symbols). The parsing algorithm for such grammars will require $O(n^2)$ time. Therefore, those grammars will not preserve an important property of the LR(k) grammars, i.e., the linear parsability, and are not pursued further in this thesis. Further, it is crucial to observe that in such a case the tag m cannot be used, and special devices like use of special symbols that are pushed on the stack after the questionable phrases, or pointers should be considered. These will enable the parsing algorithm to backtrack over a correct number of nodes.

Note that in GLRRL(k) items, there is only one auxiliary symbol SUBGOAL instead of $|P|+1$ SUBGOAL-RED(p) symbols. Because in the auxiliary item (SUBGOAL $\rightarrow \alpha . \beta, \{\epsilon\}, [p, |\alpha|]$), the tag p serves the same purpose as '(p)' does in an auxiliary basic LRRL(k) item. Furthermore, since an auxiliary item does not have more than k symbols on the right hand side (i.e., $|\alpha\beta| \leq k$), it never loses its tags by shifting the dot to the right end.

The following algorithms describe the closure of a set of GLRRL(k) items and construction of a GLRRL(k)-CFSM, for the case where an unbounded buffer to be used.

Close (s: Set of GLRRL(k) items)

```
Repeat
   For an item $(A \to \alpha.B\beta, L, [p,m])$ in s and $B \in N$
   and for all $B \to \gamma$ in P
      let $I_1 := (B \to .\gamma, L', [p,m])$ where $L' = \{BETA\} +_k L$;
      if there is an item $I_2 = (B \to .\gamma, L'', [p',m']$ in s
      then if $p' = p$ and $m = m'$ then
           replace $I_2$ by $(B \to .\gamma, Min(L' \cup L''), [p,m])$
           else replace $I_2$ by $(B \to .\gamma, Min(L' \cup L''), [-,-])$
      else add $I_1$ to s;
Until no change can be made in the set of items s.
```

Algorithm 5.1: Construction of GLRRL(k) CFSM - unbounded buffer

```
(1) Add GOAL symbol to N and production
    0: GOAL $\to$S to P.

(2) Build the initial state $s_0$:
       Let the basis of $s_0$ be { $(GOAL \to .S, \{\$\}, [-,-])$ }.
       Close the basis of $s_0$.
```

Let the set of states of CFSM, $\Omega=\{s_0\}$.

(3) <u>Repeat</u>

For a non-final state t whose successors are not yet determined build its successor states under all applicable symbols X.

Construction of a successor state s for the given state t under symbol X:

```
(i) If there is no item of the form
    $(A \rightarrow \alpha.X\beta,L,[p,m])$ in t, then t has no successor
    under symbol X.
    Else
        let basis of $s=\emptyset$.
        For each item $(A \rightarrow \alpha.X\beta,L,[p,m])$ in t
        If $p\neq$'-' and $m<k$ then add the item
        $(A \rightarrow \alpha X.\beta,L,[p,m+1])$ to the basis of s
        else add the item
        $(A \rightarrow \alpha X.\beta,L,[-,-])$ to the basis of s.
```

(ii) Check if s is a final state:

If s contains a single item of the form $(A \rightarrow \alpha X.,L,[p,m'])$ then it is a final state; (If $A\in N$, then s indicates the reduction of $A \rightarrow \alpha X$, else, i.e., when A=SUBGOAL, s indicates the reduction of the production p at m' states back.)

label s as a final state and go to step (v).

If all items in s are of the form

$(A_j \to \alpha X.\beta_j, L_j, [p,m'])$

$j=1,\ldots,n$ and $p \neq$ '-' then s is a final state;

(s indicates the reduction of the production p at m' states back.)

label s as a final state and proceed to step (v).

(iii) Check for inadequacy in the basis of s:

If there are items of the form

$I_j : (A_{j'} \to \alpha_j X.\beta_j, L_j, [p_j, m_j]),$

$j=1,\ldots,n_1$ and $\beta_j \neq \epsilon$

and

$I_j : (A_j \to \alpha_j X., L_j, [p_j, m_j],$

$j=n_1+1,\ldots,n_2$

where $n_1 \geq 1$ and $n_2 > n_1$

(i.e., shift-reduce/reduce-reduce conflicts)

or

$n_1 = 0$ and $n_2 > 1$

(i.e., only reduce-reduce conflicts)

Then

let the shift lookaheads of s be

```
SHL(s)=Min $\bigcup_{j=1}^{n_1}(\{\beta_j\}+L_{kj})$.
If $L_j \cap SHL(s) = \emptyset$ for $j=n_1+1,\ldots,n_2$
   (i.e., possible shift-reduction
    resolution)
   and
   $L_i \cap L_j = \emptyset$ for $i,j=n_1+1,\ldots,n_2$ , $i \neq j$
   (i.e., possible resolution of
    reduce-reduce conflicts)
Then
   Check if resolution can be made by
   by looking at the terminal symbols:
   Let $l=\{FIRST_k(\gamma) \mid \gamma \in SHL(s)\}$.
   For $j=n_1+1,\ldots,n_2$ let
   $l_j=\{FIRST_k(\gamma) \mid \gamma \in L_j\}$.
   If $l_j \cap l = \emptyset$ for $j=n_1+1,\ldots,n_2$ and
      $l_i \cap l_j = \emptyset$ for $i,j=n_1+1,\ldots,n_2$, $i \neq j$
   then
     for $j=n_1+1,\ldots,n_2$
       for all $x \in l_j$
          add (SUBGOAL $\to .x,\{\epsilon\},[p'_j,0]$)
          to s, where $p'_j$ is the production
          $A_j \to \alpha_j X$.
   else
     for $j=n_1+1,\ldots,n_2$
       for all $\gamma \in L_j$
          add (SUBGOAL $\to .\gamma,\{\epsilon\},[p'_j,0]$)
          to s, where $p'_j$ is the production
```

```
            $A_j \to \alpha_j X.$
            Conceal (delete) the conflicting basis
            items $I_{n_1+1},...,I_{n_2}$ :
            for $j=n_1+1,...,n_2$
            delete $I_j$ : $(A_j \to \alpha_j X., L_j, [p_j, m_j])$.
        Else
            conclude the grammar is not GLRRL(k)
            parsable with an unbounded buffer,
            and exit.

    (iv) Close the basis of s.

    (v)  Add state s to Ω if there is no state t'
         in Ω with the same items as s, or with
         items which match those of s apart from the
         lookahead sets, and for lookahead set L'
         in t' it is true that $L'=Min(L \cup L')$ and
         $L'=L \cap L'$, where L is the corresponding
         lookahead set in s.
         Otherwise let t' be the successor of t.

Until no more state can be added to Ω.

(4) Conclude that the grammar is GLRRL(k) with an
          unbounded buffer.
```

Theorem 5.1

The construction algorithm 5.1 terminates.

Proof: The proof is similar to **Theorem 4.4**. The domains of p and m tags are finite sets. Thus, the number of GLRRL(k) items is finite, which in turn implies that the number of states that can be generated is finite. □

Definition:

A (reduced ϵ-free) context-free grammar G=(N,T,P,S) is GLRRL(k) with unbounded buffer iff a CFSM for G can be constructed according to the above algorithm.

Corollary 5.1:

It is decidable whether an arbitrary context-free grammar is GLRRL(k) for a fixed number k and an unlimited size buffer.

The following example considers a GLRRL(k) grammar with an unbounded buffer.

Example 1:

Let H_1 be a context-free grammar with the following productions.

(1) $S \rightarrow A\ B\ M$

(2) $S \rightarrow B\ M$

(7) $C \rightarrow E\ C$

(8) $C \rightarrow n$

(3) $A \rightarrow A\ C$

(4) $A \rightarrow C$

(5) $B \rightarrow E\ B$

(6) $B \rightarrow n$

(9) $E \rightarrow a$

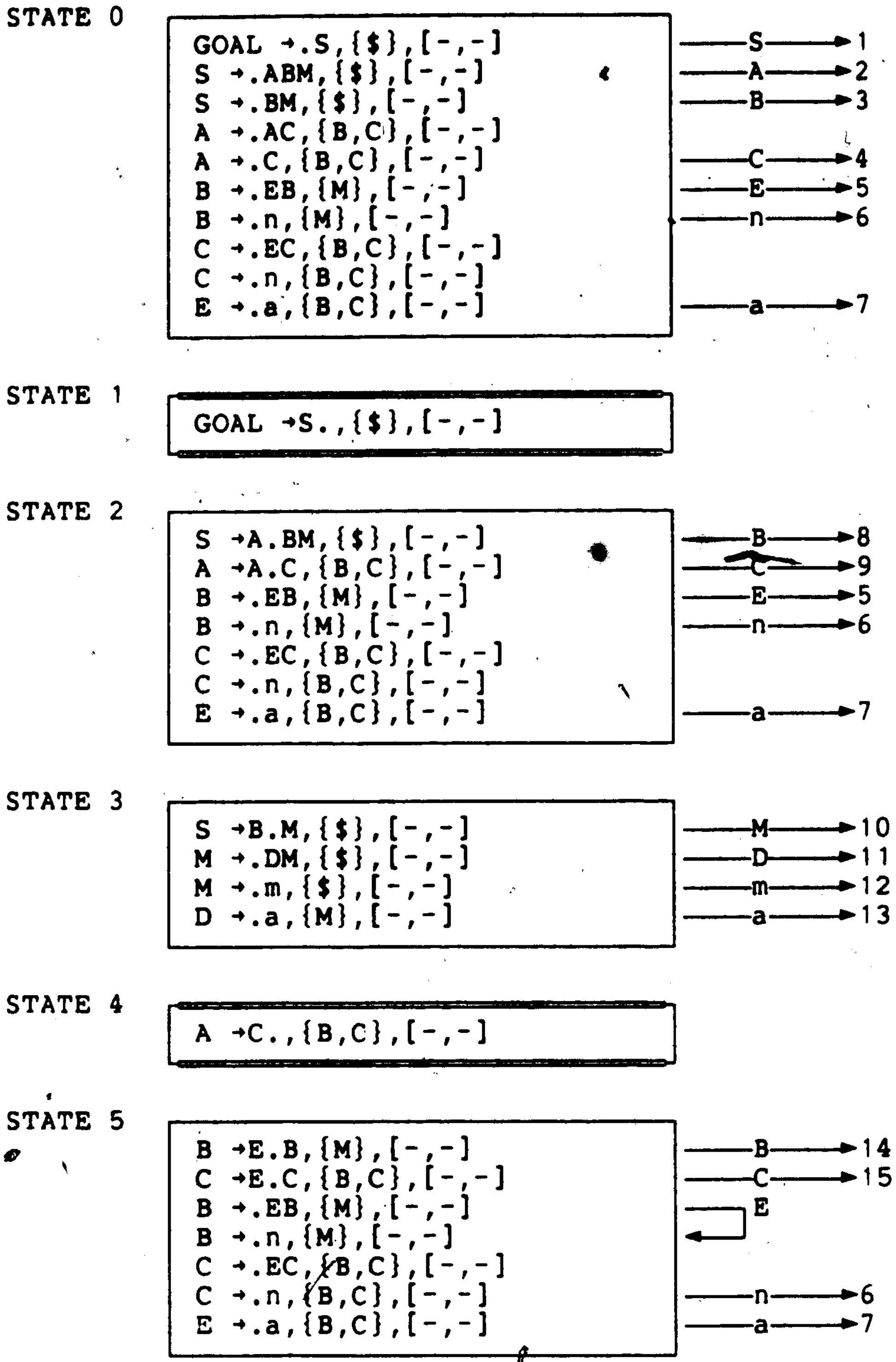
(10) $M \rightarrow D\ M$

(11) $M \rightarrow m$

(12) $D \rightarrow a$

The grammar H_1 is not basic LRRL(k) for any k. The first 'n' in a sentence can be reduced either to a 'C' or to a 'B'. The fully reduced context for the reduction $B \rightarrow n$ is M$. However, for any value of k, $C^{k-1}B$ and C^{k} are fully reduced contexts for the reduction $C \rightarrow n$. These two string cannot be recognized from each other, because both derive the terminal string $(a^{*}n)^{k}$. It is also easy to see that H_1 is not LR(k,t) for any finite k and t. Diagram 5.1 shows a GLRRL(1)-CFSM for H_1 constructed according to the algorithm 5.1, and Table 5.1 is the corresponding parse table. Figure 5.2 illustrates the parsing of a sentence in $L(H_1)$. The parse tree of the sentence is shown in Figure 5.3.

Explained intuitively, the role of each 'n' in a sentence becomes apparent when it is immediately followed by another 'n' or by an 'm', or when the 'a' that follows it, is reduced to an 'E' or a 'D'. In turn, the role of each 'a' appearing after the first 'n' in a sentence becomes obvious when it is immediately followed by an 'n' or 'm', or when the 'a' to its right is reduced to an 'E' or a 'D'.



STATE 0

GOAL →.S,{$},[-,-]
S →.ABM,{$},[-,-]
S →.BM,{$},[-,-]
A →.AC,{B,C},[-,-]
A →.C,{B,C},[-,-]
B →.EB,{M},[-,-]
B →.n,{M},[-,-]
C →.EC,{B,C},[-,-]
C →.n,{B,C},[-,-]
E →.a,{B,C},[-,-]

STATE 1

GOAL →S.,{$},[-,-]

STATE 2

S →A.BM,{$},[-,-]
A →A.C,{B,C},[-,-]
B →.EB,{M},[-,-]
B →.n,{M},[-,-]
C →.EC,{B,C},[-,-]
C →.n,{B,C},[-,-]
E →.a,{B,C},[-,-]

STATE 3

S →B.M,{$},[-,-]
M →.DM,{$},[-,-]
M →.m,{$},[-,-]
D →.a,{M},[-,-]

STATE 4

A →C.,{B,C},[-,-]

STATE 5

B →E.B,{M},[-,-]
C →E.C,{B,C},[-,-]
B →.EB,{M},[-,-]
B →.n,{M},[-,-]
C →.EC,{B,C},[-,-]
C →.n,{B,C},[-,-]
E →.a,{B,C},[-,-]

Diag. 5.1
GLRRL(1)-CFSM for grammar H_1

```
STATE 6    Deleted items:
           B →n.,{M},[-,-]
           C →n.,{B,C},[-,-]
           -------------------------
           SUBGOAL →.M,{ε},[6,0]         ——M——→16
           SUBGOAL →.B,{ε},[8,0]         ——B——→17
           SUBGOAL →.C,{ε},[8,0]         ——C——→18
           M →.DM,{ε},[6,0]              ——D——→19
           M →.m,{ε},[6,0]               ——m——→20
           B →.EB,{ε},[8,0]              ——E——→21
           B →.n,{ε},[8,0]               ——n——→22
           C →.EC,{ε},[8,0]
           C →.n,{ε},[8,0]
           D →.a,{M},[6,0]               ——a——→23
           E →.a,{B,C},[8,0]

STATE 7    E →a.,{B,C},[-,-]

STATE 8    S →AB.M,{$},[-,-]             ——M——→24
           M →.DM,{$},[-,-]              ——D——→11
           M →.m,{$},[-,-]               ——m——→12
           D →.a,{M},[-,-]               ——a——→13

STATE 9    A →AC.,{B,C},[-,-]

STATE 10   S →BM.,{$},[-,-]

STATE 11   M →D.M,{$},[-,-]              ——M——→25
           M →.DM,{$},[-,-]              ——┐D
           M →.m,{$},[-,-]               ←—┘
           D →.a,{M},[-,-]               ——m——→12
                                         ——a——→13

STATE 12   M →m.,{$},[-,-]
```

STATE 13

```
D →a.,{M},[-,-]
```

STATE 14

```
B →EB.,{M},[-,-]
```

STATE 15

```
C →EC.,{B,C},[-,-]
```

STATE 16

```
SUBGOAL →M.,{ε},[6,1]
```

STATE 17

```
SUBGOAL →B.,{ε},[8,1]
```

STATE 18

```
SUBGOAL →C.,{ε},[8,1]
```

STATE 19

```
M →D.M,{ε},[6,1]
```

STATE 20

```
M →m.,{ε},[6,1]
```

Note that the reduction of M →m is given preference over the reduction of B →n at one state back by the parsing algorithm. State 30 has the similar property.

STATE 21

```
B →E.B,{ε},[8,1]
C →E.C,{ε},[8,1]
```

STATE 22

```
B →n.,{ε},[8,1]
C →n.,{ε},[8,1]
```

```
STATE 23   Deleted items:
           First set:
           D →a.,{M},[6,1]
           E →a.,{B,C},[8,1]
           --------------------------
           Second set:
           D →a.,{M},[12,1]
           E →a.,{B,C},[8,1]
           --------------------------
           SUBGOAL →.M,{ε},[12,0]      ——M——→26
           SUBGOAL →.B,{ε},[9,0]       ——B——→27
           SUBGOAL →.C,{ε},[9,0]       ——C——→28
           M →.DM,{ε},[12,0]           ——D——→29
           M →.m,{ε},[12,0]            ——m——→30
           B →.EB,{ε},[9,0]            ——E——→31
           B →.n,{ε},[9,0]             ——n——→32
           C →.EC,{ε},[9,0]
           C →.n,{ε},[9,0]
           D →.a,{M},[12,0]            ——┐ a
           E →.a,{B,C},[9,0]           ←—┘

STATE 24   S →ABM.,{$},[-,-]

STATE 25   M →DM.,{$},[-,-]

STATE 26   SUBGOAL →M.,{ε},[12,1]

STATE 27   SUBGOAL →B.,{ε},[9,1]

STATE 28   SUBGOAL →C.,{ε},[9,1]

STATE 29   M →D.M,{ε},[12,1]

STATE 30   M →m.,{ε},[12,1]
```

STATE 31

$B \to E.B, \{\epsilon\}, [9,1]$
$C \to E.C, \{\epsilon\}, [9,1]$

STATE 32

$B \to n., \{\epsilon\}, [9,1]$
$C \to n., \{\epsilon\}, [9,1]$

States

Symbols	0	1	2	3	5	6	8	11	23
a	r9		r9	r12	r9	g23	r12	r12	g23
m				r11		r11	r11	r11	r11
n	g6		g6		g6	t1,r8			t1,r9
A	g2								
B	g3		g8		r5	t1,r8			t1,r9
C	r4		r3		r7	t1,r8			t1,r9
D				g11		t1,r6	g11	g11	t1,r12
E	g5		g5		g5	t1,r8			t1,r9
M				r2		t1,r6	r1		
S	g1								
$		r0							

Table 5.1
GL~~R~~RL(1) parsing table for H,

State stack	Symbol stack		Current symbol	Buffer	Remainder
-	-		-	-	anaanaaanaam$
0	-		a	-	naanaaanaam$
0	a	r9	E	-	naanaaanaam$
0,5	E		n	-	aanaaanaam$
0,5,6	En		a	-	anaaanaam$
0,5,6,23	Ena		a	-	naaanaam$
0,5,6,23,23	Enaa		n	-	aaanaam$
0,5,6,23,23	Enaan	t1,r9	E	n	aaanaam$
0,5,6,23	EnaE	t1,r9	E	En	aaanaam$
0,5,6	EnE	t1,r8	C	EEn	aaanaam$
0,5	EC	r7	C	EEn	aaanaam$
0	C	r4	A	EEn	aaanaam$
0,2	A		E	En	aaanaam$
0,2,5	AE		E	n	aaanaam$
0,2,5,5	AEE		n	-	aaanaam$
0,2,5,5,6	AEEn		a	-	aanaam$
0,2,5,5,6,23	AEEna		a	-	anaam$
0,2,5,5,6,23,23	AEEnaa		a	-	naam$
0,2,5,5,6,23,23,23	AEEnaaa		n	-	aam$
0,2,5,5,6,23,23,23	AEEnaaan	t1,r9	E	n	aam$
0,2,5,5,6,23,23	AEEnaaE	t1,r9	E	En	aam$
0,2,5,5,6,23	AEEnaE	t1,r9	E	EEn	aam$
0,2,5,5,6	AEEnE	t1,r8	C	EEEn	aam$
0,2,5,5	AEEC	r7	C	EEEn	aam$
0,2,5	AEC	r7	C	EEEn	aam$
0,2	AC	r3	A	EEEn	aam$
0,2	A		E	EEn	aam$
0,2,5	AE		E	En	aam$
0,2,5,5	AEE		E	n	aam$
0,2,5,5,5	AEEE		n	-	aam$
0,2,5,5,5,6	AEEEn		a	-	am$
0,2,5,5,5,6,23	AEEEna		a	-	m$
0,2,5,5,5,6,23,23	AEEEnaa		m	-	$
0,2,5,5,5,6,23,23	AEEEnaam	r11	M	-	$
0,2,5,5,5,6,23,23	AEEEnaaM	t1,r12	D	M	$
0,2,5,5,5,6,23	AEEEnaD	t1,r12	D	DM	$
0,2,5,5,5,6	AEEEnD	t1,r6	B	DDM	$
0,2,5,5,5	AEEEB	r5	B	DDM	$
0,2,5,5	AEEB	r5	B	DDM	$
0,2,5	AEB	r5	B	DDM	$
0,2,8	AB		D	DM	$
0,2,8,11	ABD		D	M	$
0,2,8,11,11	ABDD		M	-	$
0,2,8,11,11	ABDDM	r10	M	-	$
0,2,8,11	ABDM	r10	M	-	$
0,2,8	ABM	r1	S	-	$
0,1	S		$	-	-
0	r0: accept				

Fig. 5.2
Parsing of the sentence ana'na'na'm

Example 1 demonstrated a kind of grammar that needs unlimited length buffer. The maximum number of nodes that are deposited in the buffer is one more than the maximum number of intervening a's that lie between two n's in a sentence. Obviously, this number could be arbitrarily large. On the other hand, Example 2 shows a grammar that does not need an unbounded buffer.

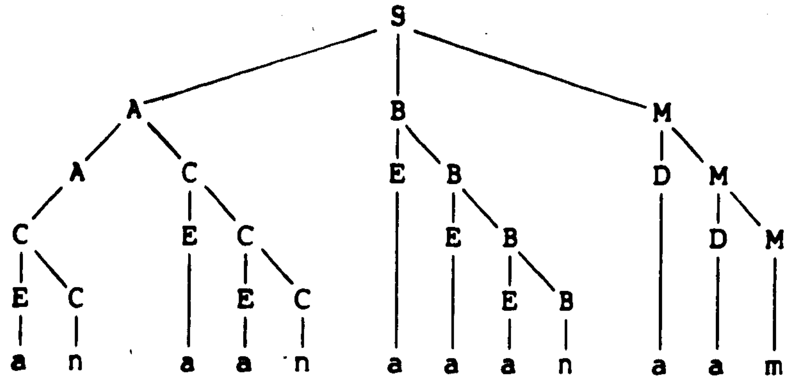


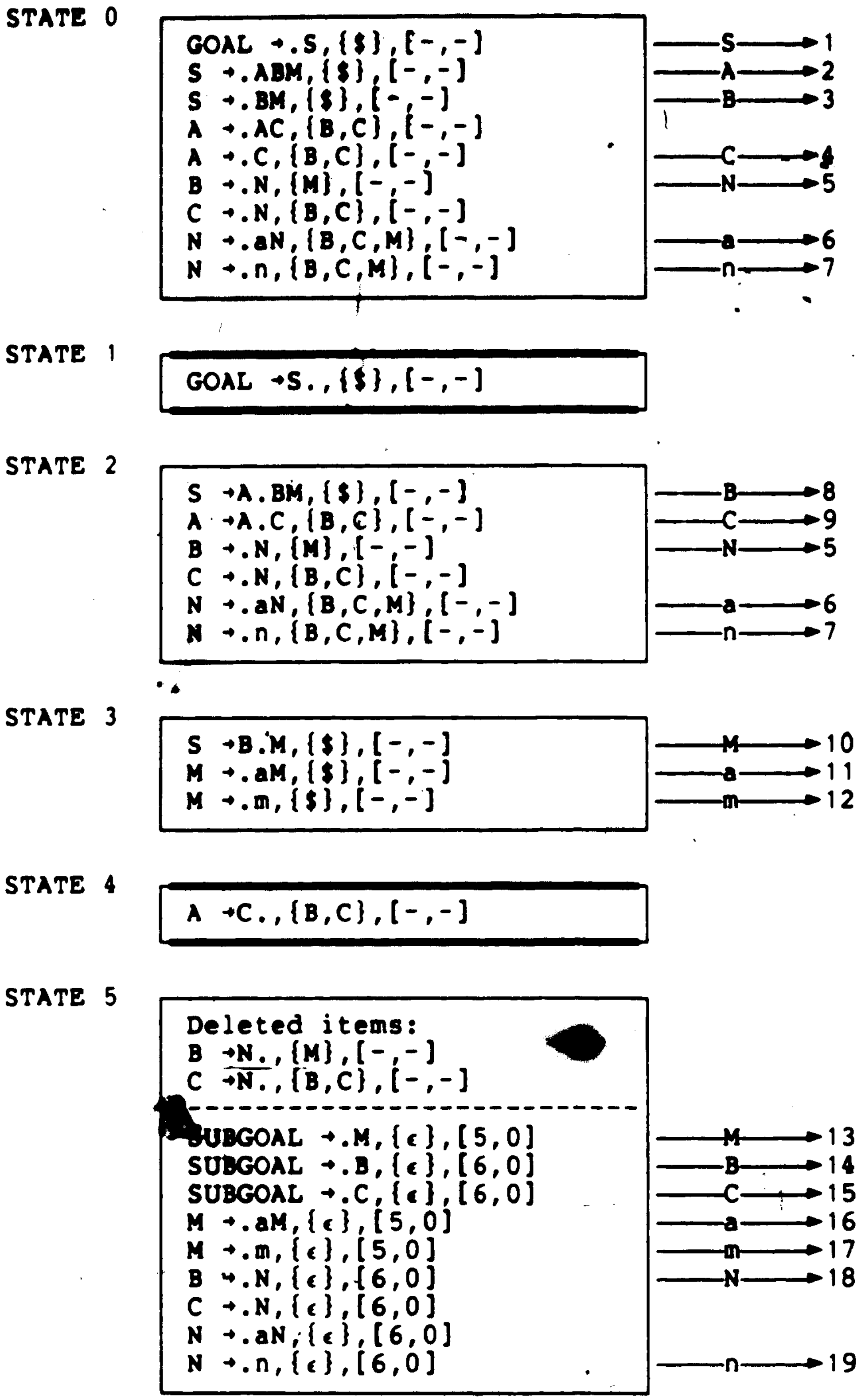
Fig. 5.3
Parse tree of $ana^2na^3na^2m$

Example 2:

Consider the grammar H_2 that generates the same language as H_1, but with different set of productions:

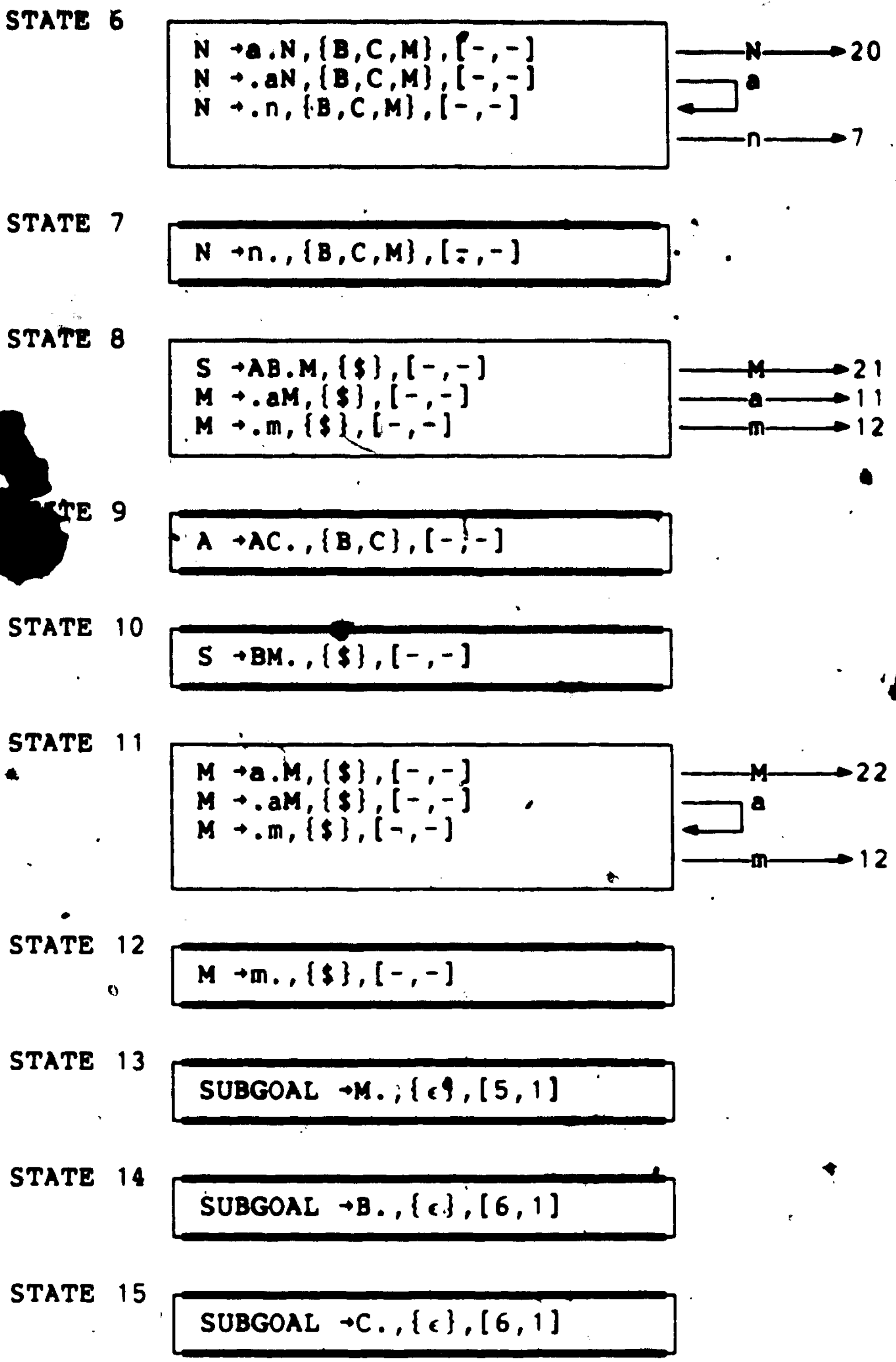
(1) S → A B M	(6) C → N
(2) S → B M	(7) N → a N
(3) A → A C	(8) N → n
(4) A → C	(9) M → a M
(5) B → N	(10) M → m

Diagram 5.2 shows the GLRRL(1)-CFSM for H_1 generated by the algorithm 5.1. The corresponding parse table appears in Table 5.2.



```
STATE 0
GOAL →.S,{$},[-,-]              ——S——→1
S →.ABM,{$},[-,-]               ——A——→2
S →.BM,{$},[-,-]                ——B——→3
A →.AC,{B,C},[-,-]
A →.C,{B,C},[-,-]               ——C——→4
B →.N,{M},[-,-]                 ——N——→5
C →.N,{B,C},[-,-]
N →.aN,{B,C,M},[-,-]            ——a——→6
N →.n,{B,C,M},[-,-]             ——n——→7

STATE 1
GOAL →S.,{$},[-,-]

STATE 2
S →A.BM,{$},[-,-]               ——B——→8
A →A.C,{B,C},[-,-]              ——C——→9
B →.N,{M},[-,-]                 ——N——→5
C →.N,{B,C},[-,-]
N →.aN,{B,C,M},[-,-]            ——a——→6
N →.n,{B,C,M},[-,-]             ——n——→7

STATE 3
S →B.M,{$},[-,-]                ——M——→10
M →.aM,{$},[-,-]                ——a——→11
M →.m,{$},[-,-]                 ——m——→12

STATE 4
A →C.,{B,C},[-,-]

STATE 5
Deleted items:
B →N.,{M},[-,-]
C →N.,{B,C},[-,-]
-----------------------------
SUBGOAL →.M,{ε},[5,0]           ——M——→13
SUBGOAL →.B,{ε},[6,0]           ——B——→14
SUBGOAL →.C,{ε},[6,0]           ——C——→15
M →.aM,{ε},[5,0]                ——a——→16
M →.m,{ε},[5,0]                 ——m——→17
B →.N,{ε},[6,0]                 ——N——→18
C →.N,{ε},[6,0]
N →.aN,{ε},[6,0]
N →.n,{ε},[6,0]                 ——n——→19
```

Diag. 5.2
GLRRL(1)-CFSM for grammar H_3



STATE 6

N →a.N,{B,C,M},[-,-]
N →.aN,{B,C,M},[-,-]
N →.n,{B,C,M},[-,-]

STATE 7

N →n.,{B,C,M},[-,-]

STATE 8

S →AB.M,{$},[-,-]
M →.aM,{$},[-,-]
M →.m,{$},[-,-]

STATE 9

A →AC.,{B,C},[-,-]

STATE 10

S →BM.,{$},[-,-]

STATE 11

M →a.M,{$},[-,-]
M →.aM,{$},[-,-]
M →.m,{$},[-,-]

STATE 12

M →m.,{$},[-,-]

STATE 13

SUBGOAL →M.,{ε},[5,1]

STATE 14

SUBGOAL →B.,{ε},[6,1]

STATE 15

SUBGOAL →C.,{ε},[6,1]

STATE 16

```
M →a.M,{ε},[5,1]        ——M——→23
N →a.N,{ε},[6,1]        ——N——→24
M →.aM,{ε},[5,1]        ——a——→25
M →.m,{ε},[5,1]         ——m——→26
N →.aN,{ε},[6,1]
N →.n,{ε},[6,1]         ——n——→27
```

STATE 17

```
M →m.,{ε},[5,1}
```

STATE 18

```
B →N.,{ε},[6,1]
C →N.,{ε},[6,1]
```

STATE 19

```
N →n.,{ε},[6,1]
```

STATE 20

```
N →aN.,{B,C,M},[-,-]
```

STATE 21

```
S →ABM.,{$},[-,-]
```

STATE 22

```
M →aM.,{$},[-,-]
```

STATE 23

```
M →aM.,{ε},[-,-]
```

STATE 24

```
N →aN.,{ε},[-,-]
```

STATE 25

```
M →a.M,{ε},[-,-]        ——M——→23
N →a.N,{ε},[-,-]        ——N——→24
M →.aM,{ε},[-,-]        ——a——↺
M →.m,{ε},[-,-]
N →.aN,{ε},[-,-]        ——m——→26
N →.n,{ε},[-,-]         ——n——→27
```

STATE 26

M →m.,{ε},[-,-]

STATE 27

N →n.,{ε},[-,-]

	States									
Symbols	0	1	2	3	5	6	8	11	16	25
a	g6		g6	g11	g16	g6	g11	g11	g25	g25
m				r10	r10		r10	r10	r10	r10
n	r8		r8		r8	r8			r8	r8
A	g2									
B	g3		g8		t1,r6					
C	r4		r3		t1,r6					
M				r2	t1,r5		r1	r9	r9	r9
N	g5		g5		t1,r6	r7			r7	r7
S	g1									
$		r0								

Table 5.2
GLRRL(1) parse table for H_2

The Parse tree of the example sentence $ana^2na^3na^2m$ is shown in Figure 5.4, and the trace of the parser is illustrated in Figure 5.5. As one can observe each sequence of a's followed by an n independent of its right context is reduced to an **N**. However, for the reduction of an **N** to a **B** or a **C** one needs to obtain the parse of the one-symbol reduced right context which is either an **M** or an **N**. The grammar H_2 is not basic LRRL(k) for the same reason as given for the grammar H_1. Nevertheless the parser for H_2 uses only a bounded buffer. This is true because H_2 is an LR(1,2) grammar.

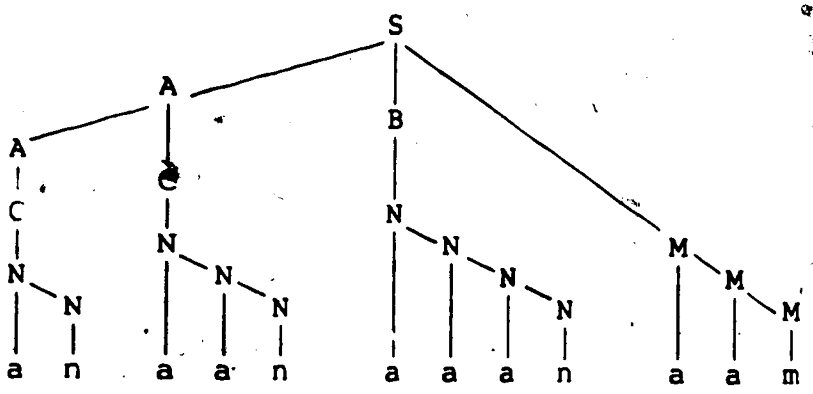


Fig. 5.4
Parse tree of $ana^2na^3na^2m \in L(H_2)$

State stack	Symbol stack		Current symbol	Buff-er	Remainder
-	-		-	-	anaanaaanaam$
0	-		a	-	naanaaanaam$
0,6	a		n	-	aanaaanaam$
0,6	an	red 8	N	-	aanaaanaam$
0,6	aN	red 7	N	-	aanaaanaam$
0,5	N		a	-	anaaanaam$
0,5,16	Na		a	-	naaanaam$
0,5,16,25	Naa		n	-	aaanaam$
0,5,16,25	Naan	red 8	N	-	aaanaam$
0,5,16,25	NaaN	red 7	N	-	aaanaam$
0,5,16	NaN	red 7	N	-	aaanaam$
0,5	NN	trans 1 red 6	C	N	aaanaam$
0	C	red 4	A	N	aaanaam$
0,2	A		N	-	aaanaam$
0,2,5	AN		a	-	aanaam$
0,2,5,16	ANa		a	-	anaam$
0,2,5,16,25	ANaa		a	-	naam$
0,2,5,16,25,25	ANaaa		[illegible]	-	aam$
0,2,5,16,25,25	ANaaan	red 8	[illegible]	-	aam$
0,2,5,16,25,25	ANaaaN	red 7	N	-	aam$
0,2,5,16,25	ANaaN	red 7	N	-	aam$
0,2,5,16	ANaN	red 7	N	-	aam$
0,2,5	ANN	trans 1 red 6	C	N	aam$
0,2	AC	red 3	A	N	aam$
0,2	A		N	-	aam$
0,2,5	AN		a	-	am$
0,2,5,16	ANa		a	-	m$
0,2,5,16,25	ANaa		m	-	$
0,2,5,16,25	ANaam	red 10	M	-	$
0,2,5,16,25	ANaaM	red 9	M	-	$
0,2,5,16	ANaM	red 9	M	-	$
0,2,5	ANM	trans 1 red 5	B	M	$
0,2,8	AB		M	-	$
0,2,8	ABM	red 1	S	-	$
0,1	S		$	-	-
0	reduce 0: accept				

Fig 5.5
Parsing the sentence anaʼnaʼnaʼm$\in L(H_2)$

5.3.2 GLRRL(k) grammars with bounded buffer

The grammar H_2, showed that not all the parsers produced by the algorithm 5.1 use unlimited number of buffer cells. In general, one can test whether a GLRRL(k) parser will use more than a given number l cells of the buffer. The test can be carried out by adding extra information to the items of a CFSM. However, a more interesting option is to try to alter the algorithm in a way that ensures the resulting parser does not use more than a certain number of buffer cells, even where the original algorithm produces a parser that uses a buffer unlimited in size. Such a goal can be achieved in few different ways. The method described here is essentially the one that was outlined in the beginning of Section 5.3. In this method, one employs the GLRRL(k) lookahead policy (i.e., reduced contexts at arbitrary levels) as long as the number of nodes, over which a backtracking takes place in order to reduce the first bypassed phrase, does not exceed a limit n. At this point, if there is a local ambiguity, that would be resolved by the ELRRL(k) lookahead policy. The latter policy ensures that no more than k terminal symbols or fully reduced nodes will reside in the buffer when one backtracks to this point. Therefore, at the time the first bypassed phrase is reduced, the total number of nodes in the buffer will not exceed n+k. However, the interesting feature of this scheme is that the method itself can be used recursively to parse the last k-symbol fully reduced context.

A simple way to implement this limitation is to translate it into the number of bypassed questionable phrases in a sentential form. In GLRRL(k) parsing, if m questionable phrases in a sentential form are bypassed, then at most km nodes will be accumulated in the buffer during the backing up to the first questionable phrase. Therefore, in the new algorithm, if one chooses n to be kt-k, then the lookaheads that resolve the decision about the reduction of the t-th questionable phrase must be k-symbol fully reduced right context (or k-symbol terminal lookahead) rather than k-symbol reduced right context at an arbitrary level. However, for any questionable phrase after the t-th one, except the jt-th questionable phrases (j>1), the GLRRL(k) lookahead policy may be used. As one can see, such a strategy periodically enforces the lookaheads accumulated in the buffer to be reduced to at most k-node fully reduced contexts.

The corresponding class of grammars that can be parsed in this fashion are called GLRRL(k) grammars with bounded buffer of size kt. Obviously, these grammars are a proper subclass of the GLRRL(k) grammars with unbounded buffer.

The items used in the construction of a CFSM for a GLRRL(k) parser with a bounded buffer, are essentially the same as the ones that are employed in the unbounded buffer case. The differences are that here an auxiliary item can have a numeric p-tag while its m-tag is '-'. However, any other item that is obtained through the closure of such an auxiliary item has its p-tag set to '-'. Furthermore, a third tag $q \in \{0,1,\ldots,t\}$ representing the number of bypassed phrases is added to the items. Consequently, the closure algorithm in this case is slightly different and is given below.

Close (s: Set of GLRRL(k) items) - Bounded buffer case

```
Repeat
    For an item $(A \rightarrow \alpha.B\beta, L, [p,m,q])$ in s and $B \in N$
    and for all $B \rightarrow \gamma$ in P
       let $L' = \{\beta\} +_k L$;
       If  m='-' then let p'='-' else let p'=p;
       If  q=t then let q'=0 else let q'=q;
       let $I_1 := (B \rightarrow .\gamma, L', [p', m, q'])$;
       if there is an item $I_2 = (B \rightarrow .\gamma, L'', [p'', m', q'']$ in s
       then let r=max(q',q");
            if p"=p' and m'=m then
            replace $I_2$ by $(B \rightarrow .\gamma, Min(L' \cup L''), [p', m, r])$
            else replace $I_2$ by $(B \rightarrow .\gamma, Min(L' \cup L''), [-, -, r])$
       else add $I_1$ to s;
 Until no change can be made in the set of items s.
```

The following describes the algorithm for generating a CFSM in the bounded buffer case. Starred segments show the primary differences with the algorithm 5.1.

Algorithm 5.2: Construction of GLRRL(k) CFSM - bounded buffer

```
(1) Add GOAL symbol to N and production
    0: GOAL →S to P.

(2) Build the initial state $s_0$:
       Let the basis of $s_0$ be { (GOAL →.S,{$},[-,-,0]) }.
       Close the basis of $s_0$.
       Let the set of states of CFSM, $\Omega$={$s_0$}.

(3) Repeat
       For a non-final state t whose successors are not yet determined build its successor states under all applicable symbols X.

       Construction of a successor state s for the given state t under symbol X:

       (i) If there is no item of the form (A →$\alpha$.X$\beta$,L,[p,m,q]) in t, then t has no successor under symbol X.
           Else
               let basis of s=∅.
               For each item (A →$\alpha$.X$\beta$,L,[p,m,q]) in t
(*)            If m is a number < k then add the item (A →$\alpha$X.$\beta$,L,[p,m+1,q]) to the basis of s
```

```
        else
(*)         if A=SUBGOAL then add
                $(A \to \alpha X.\beta, L, [p,-,q])$
            else add $(A \to \alpha X.\beta, L, [-,-,q])$
                to the basis of s.
```

(ii) Check if s is a final state:

If s contains a single item of the form $(A \to \alpha X., L, [p,m',q])$ then it is a final state; (If $A \in N$, then s indicates the reduction of $A \to \alpha X$, else, i.e., when A=SUBGOAL, s indicates the reduction of the production p

(*) at $|\alpha X|$ states back.)

label s as a final state and go to step (v).

If all items in s are of the form $(A_j \to \alpha X.\beta_j, L_j, [p,m',q])$ $j=1,\ldots,n$ and $m' \neq$ '-' then s is a final state; (s indicates the reduction of the production p at m' states back.)

label s as a final state and proceed to step (v).

(iii) Check for inadequacy in the basis of s:

If there are items of the form

$I_j : (A_j \to \alpha_j X.\beta_j, L_j, [p_j, m_j, q_j]),$

```
    $j=1,...,n_1$ and $\beta_j \neq \epsilon$
    and
    $I_j : (A_j \to \alpha_j X_j.,L_j,[p_j,m_j,q_j])$,
    $j=n_1+1,...,n_2$ ,
    where $n_1 \geq 1$ and $n_2 > n_1$
    (i.e., shift-reduce/reduce-reduce conflicts)
    or
    $n_1=0$ and $n_2>1$
    (i.e., only reduce-reduce conflicts)
Then
     let the shift lookaheads of s be
     $SHL(s)=Min \cup_{j=1}^{n_{1o}} (\{\beta_j\}+_k L_j)$.
     If $L_j \cap SHL(s) = \emptyset$ for $j=n_1+1,...,n_2$
        (i.e., possible shift-reduction
         resolution)
        and
        $L_i \cap L_j = \emptyset$ for $i,j=n_1+1,...,n_2$ , $i \neq j$
        (i.e., possible resolution of
         reduce-reduce conflicts)
     Then
        Check if resolution can be made by
        looking at the terminal symbols:
        Let $l=\{FIRST_k(\gamma) | \gamma \in SHL(s)\}$.
        For $j=n_1+1,...,n_2$ let
        $l_j=\{FIRST_k(\gamma) | \gamma \in L_j\}$.
        If $l_j \cap l = \emptyset$ for $j=n_1+1,...,n_2$ and
```

```
          $l_i \cap l_j = \emptyset$ for $i,j=n_1+1,\ldots,n_2$, $i \neq j$
      then
        for $j=n_1+1,\ldots,n_2$
          for all $x \in l_j$
            Let $q'=q_j+1$;
(*)         If $q'=t$ then let $m'='-'$
               else let $m'=0$;
            add (SUBGOAL $\rightarrow .x,\{\epsilon\},[p'_j,m',q']$)
            to s, where $p'_j$ is the production
            $A_j \rightarrow \alpha_j X.$
      else
        for $j=n_1+1,\ldots,n_2$
          for all $\gamma \in L_j$
(*)         Let $q'=q_j+1$;
            If $q'=t$ then let $m'='-'$
               else let $m'=0$;
            add (SUBGOAL $\rightarrow .\gamma,\{\epsilon\},[p'_j,m',q']$)
            to s, where $p'_j$ is the
            production $A_j \rightarrow \alpha_j X.$
      Conceal (delete) the conflicting basis
      items $I_{n_1+1},\ldots,I_{n_2}$ :
      for $j=n_1+1,\ldots,n_2$
      delete $I_j$ : $(A_j \rightarrow \alpha_j X.,L_j,[p_j,m_j,q_j])$.

   Else
      conclude the grammar is not GLRRL(k)
      parsable with a bounded buffer,
            and exit.

      (iv) Close the basis of s.

      (v)  Add state s to Ω if there is no state t'
           in Ω with the same items as s, or with
           items which match those of s apart from the
           lookahead sets, and for lookahead set L'
           in t' it is true that $L'=\mathrm{Min}(L\cup L')$
           and $L'=L\underline{\cap}L'$, where L is the corresponding
           lookahead set in s.
           Otherwise let t' be the successor of t.

   Until no more state can be added to Ω.

(4) Conclude that the grammar is GLRRL(k) with a
          bounded buffer.
```

Theorem 5.2:

The construction algorithm 5.2 terminates.

Proof: Is similar to Theorem 5.1, with the addition that the domain of the q-tag is also a finite set. □

Definition:

A (reduced ϵ-free) context-free grammar G=(N,T,P,S) is GLRRL(k) with a buffer of size kt iff a CFSM for it can be constructed according to the above algorithm.

Corollary 5.2

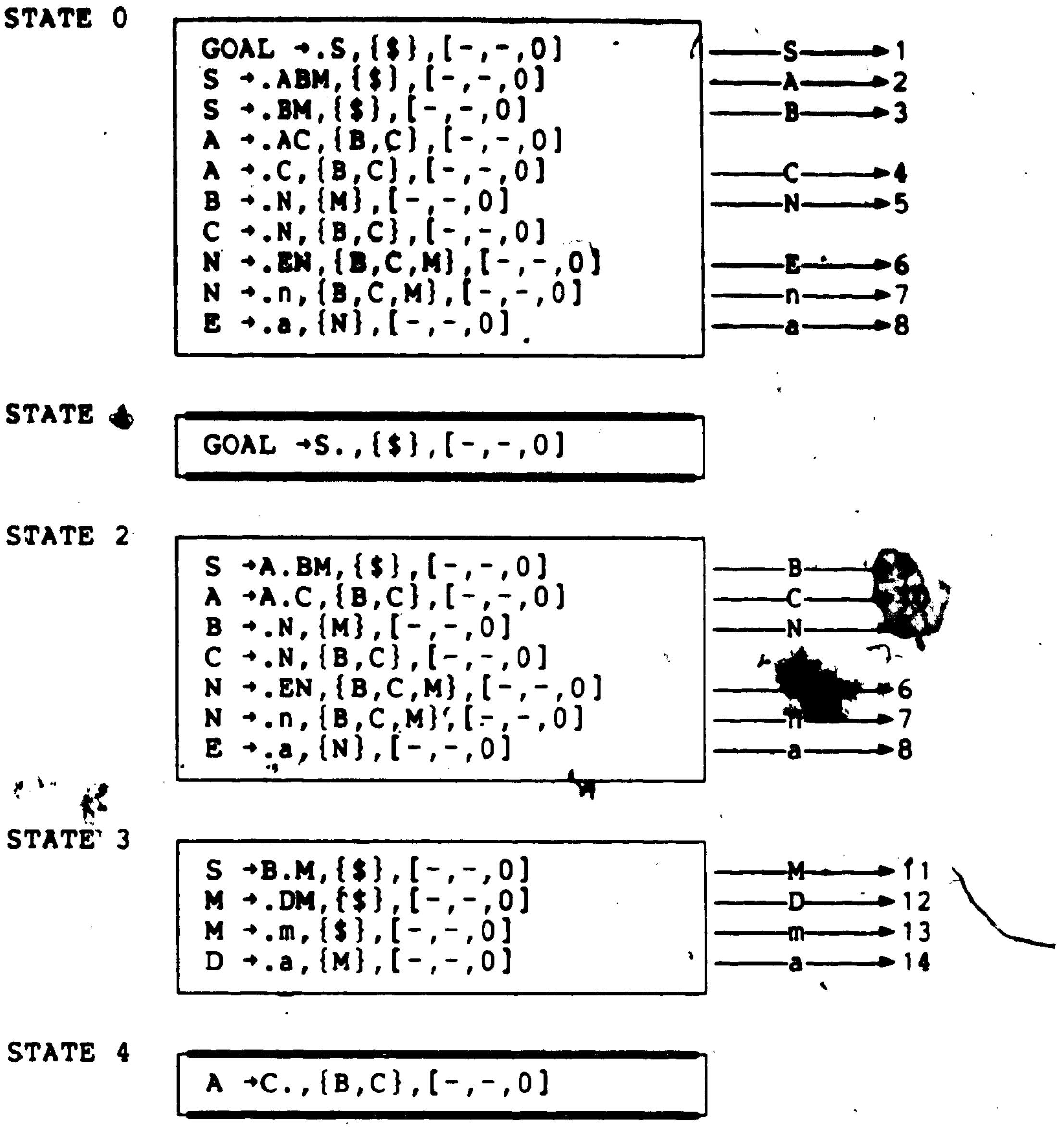
It is decidable whether an arbitrary context-free grammar is GLRRL(k) for a fixed number k and a buffer of size kt.

Example 3:

Consider the grammar H_3 that again generates the same language as H_1 and H_2 but with the productions:

(1) S → A B M	(7) N → E N
(2) S → B M	(8) N → n
(3) A → A C	(9) E → a
(4) A → C	(10) M → D M
(5) B → N	(11) M → m
(6) C → N	(12) D → a

Diagram 5.3 illustrates the GLRRL(1)-CFSM, with a buffer of size kt=2, obtained by applying the algorithm 5.2 to the grammar H_3. The corresponding parse table is shown in Table 5.3.



STATE 0

GOAL →.S,{$},[-,-,0]
S →.ABM,{$},[-,-,0]
S →.BM,{$},[-,-,0]
A →.AC,{B,C},[-,-,0]
A →.C,{B,C},[-,-,0]
B →.N,{M},[-,-,0]
C →.N,{B,C},[-,-,0]
N →.EN,{B,C,M},[-,-,0]
N →.n,{B,C,M},[-,-,0]
E →.a,{N},[-,-,0]

STATE 1

GOAL →S.,{$},[-,-,0]

STATE 2

S →A.BM,{$},[-,-,0]
A →A.C,{B,C},[-,-,0]
B →.N,{M},[-,-,0]
C →.N,{B,C},[-,-,0]
N →.EN,{B,C,M},[-,-,0]
N →.n,{B,C,M},[-,-,0]
E →.a,{N},[-,-,0]

STATE 3

S →B.M,{$},[-,-,0]
M →.DM,{$},[-,-,0]
M →.m,{$},[-,-,0]
D →.a,{M},[-,-,0]

STATE 4

A →C.,{B,C},[-,-,0]

Diag. 5.3
GLRRL(1)-CFSM for grammar H,

STATE 5

```
Deleted items:
B →N.,{M},[-,-,0]
C →N.,{B,C},[-,-,0]
------------------------------------
SUBGOAL →.M,{ε},[5,0,1]        ────M────►15
SUBGOAL →.B,{ε},[6,0,1]        ────B────►16
SUBGOAL →.C,{ε},[6,0,1]        ────C────►17
M →.DM,{ε},[5,0,1]             ────D────►18
M →.m,{ε},[5,0,1]              ────m────►19
D →.a,{M},[5,0,1]              ────a────►20
B →.N,{ε},[6,0,1]              ────N────►21
C →.N,{ε},[6,0,1]
N →.EN,{ε},[6,0,1]             ────E────►22
N →.n,{ε},[6,0,1]              ────n────►23
E →.a,{N},[6,0,1]
```

STATE 6

```
N →E.N,{B,C,M},[-,-,0]         ────N────►24
N →.EN,{B,C,M},[-,-,0]         ───┐ E
N →.n,{B,C,M},[-,-,0]          ◄──┘
E →.a,{N},[-,-,0]              ────n────►8
                               ────a────►7
```

STATE 7

```
N →n.,{B,C,M},[-,-,0]
```

STATE 8

```
E →a.,{N},[-,-,0]
```

STATE 9

```
S →AB.M,{$},[-,-,0]            ────M────►25
M →.DM,{$},[-,-,0]             ────D────►12
M →.m,{$},[-,-,0]              ────m────►13
D →.a,{M},[-,-,0]              ────M────►14
```

STATE 10

```
A →AC.,{B,C},[-,-,0]
```

STATE 11

```
S →BM.,{$},[-,-,0]
```

```
STATE 12
  M →[illegible]$},[-,-,0]          ——M——►26
  M →.DM,{$},[-,-,0]                ——D (to State 12)
  M →.m,{$},[-,-,0]
  D →.a,{M},[-,-,0]                 ——m——►13
                                    ——a——►14

STATE 13
  M →m.,{$},[-,-,0]

STATE 14
  D →a.,{M},[-,-,0]

STATE 15
  SUBGOAL →M.,{ε},[5,1,1]

STATE 16
  SUBGOAL →B.,{ε},[6,1,1]

STATE 17
  SUBGOAL →C.,{ε},[6,1,1]

STATE 18
  M →D.M,{ε},[5,1,1]

STATE 19
  M →m.,{ε},[5,1,1]

STATE 20
  Deleted items:
  First set:
  D →a.,{M},[5,1,1]
  E →a.,{N},[6,1,1]
  Second set:
  D →a.,{M},[12,1,1]
  E →a.,{N},[6,1,1]
  ----------------------------
  SUBGOAL →.M,{ε},[12,-,2]          ——M——►27
  SUBGOAL →.N,{ε},[9,-,2]           ——N——►28
  M →.DM,{ε},[-,-,0]                ——D——►29
  M →.m,{ε},[-,-,0]                 ——m——►30
  D →.a,{M},[-,-,0]                 ——a——►31
  N →.EN,{ε},[-,-,0]                ——E——►32
  N →.n,{ε},[-,-,0]                 ——n——►33
  E →.a,{N},[-,-,0]
```

STATE 21

```
B →N.,{ε},[6,1,1]
C →N.,{ε},[6,1,1]
```

STATE 22

```
N →E.N,{ε},[6,1,1]
```

STATE 23

```
N →n.,{ε},[6,1,1]
```

STATE 24

```
N →EN.,{B,C,M},[-,-,0],
```

STATE 25

```
S →ABM.,{$},[-,-,0]
```

STATE 26

```
M →DM.,{$},[-,-,0]
```

STATE 27

```
SUBGOAL →M.,{ε},[12,1,2]
```

STATE 28

```
SUBGOAL →N.,{ε},[9,1,2]
```

STATE 29

```
M →D.M,{ε},[-,-,0]      ——M——→34
M →.DM,{ε},[-,-,0]      ——D (to itself)
M →.m,{ε},[-,-,0]
D →.a,{M},[-,-,0]       ——m——→30
                        ——a——→14
```

STATE 30

```
M →m.,{ε},[-,-,0]
```

STATE 31

```
Deleted items:
D →a.,{M},[-,-,0]
E →a.,{N},[-,-,0]
------------------------------
SUBGOAL →.M,{ε},[12,0,1]        ——M——→35
SUBGOAL →.N,{ε},[9,0,1]         ——N——→36
M →.DM,{ε},[12,0,1]             ——D——→37
M →.m,{ε},[12,0,1]              ——m——→38
D →.a,{M},[12,0,1]              ——a——→20
N →.EN,{ε},[9,0,1]              ——E——→39
N →.n,{ε},[9,0,1]               ——n——→40
E →.a,{N},[9,0,1]
```

STATE 32

```
N →E.N,{ε},[-,-,0]              ——N——→41
N →.EN,{ε},[-,-,0]              ——E
N →.n,{ε},[-,-,0]
E →.a,{N},[-,-,0]               ——n——→33
                                ——a——→8
```

STATE 33

```
N →n.,{ε},[-,-,0]
```

STATE 34

```
M →DM.,{ε},[-,-,0]
```

STATE 35

```
SUBGOAL →M.,{ε},[12,1,1]
```

STATE 36

```
SUBGOAL →N.,{ε},[9,1,1]
```

STATE 37

```
M →D.M,{ε},[12,1,1]
```

STATE 38

```
M →m.,{ε},[12,1,1]
```

STATE 39

```
N →E.N,{ε},[9,1,1]
```

STATE 40

```
N →n.,{ε},[9,1,1]
```

STATE 41

```
N →EN.,{ε},[-,-,0]
```

	States											
Symbols	0	1	2	3	5	6	9	12	20	29	31	32
a	r9		r9	r12	g20	r9	r12	r12	g31	r12	g20	r9
m				r11	r11		r11	r11	r11	r11	r11	
n	r8		r8		r8	r8			r8		r8	r8
A	g2											
B	g3		g9		t1,r6							
C	r4		r3		t1,r6							
D				g12	t1,r5		g12	g12	g29	g29	t1,r12	
E	g6		g6		t1,r6	g6			g32		t1,r9	g32
M				r2	t1,r5		r1	r10	t1,r12	r10	t1,r12	
N	g5		g5		t1,r6	r7			t1,r9		t1,r9	r7
S	g1											
$		r0										

Table 5.3
GLRRL(1) parse table for H_3

The trace of the parser when analyzing the example sentence anaʼnaʼnaʼm is shown in Figure 5.6. The parse tree of the sentence is depicted in Figure 5.7.

State stack	Symbol stack		Current symbol	Buff-er	Remainder
-	-		-	-	anaanaaanaam$
0	-		a	-	naanaaanaam$
0	a	red 9	E	-	naanaaanaam$
0,6	E		n	-	aanaaanaam$
0,6	En	red 8	N	-	aanaaanaam$
0,6	EN	red 7	N	-	aanaaanaam$
0,5	N		a	-	anaaanaam$
0,5,20	Na		a	-	naaanaam$
0,5,20,31	Naa		n	-	aaanaam$
0,5,20,31	Naan	red 8	N	-	aaanaam$
0,5,20,31	NaaN	trans 1, red 9	E	N	aaanaam$
0,5,20,32	NaE		N	-	aaanaam$
0,5,20,32	NaEN	red 7	N	-	aaanaam$
0,5,20	NaN	trans 1, red 9	E	N	aaanaam$
0,5	NE	trans 1, red 6	C	EN	aaanaam$
0	C	red 4	A	EN	aaanaam$
0,2	A		E	N	aaanaam$
0,2,6	AE		N	-	aaanaam$
0,2,6	AEN	red 7	N	-	aaanaam$
0,2,5	AN		a	-	aanaam$
0,2,5,20	ANa		a	-	anaam$
0,2,5,20,31	ANaa		a	-	naam$
0,2,5,20,31,20	ANaaa		n	-	aam$
0,2,5,20,31,20	ANaaan	red 8	N	-	aam$
0,2,5,20,31,20	ANaaaN	trans 1, red 9	E	N	aam$
0,2,5,20,31	ANaaE	trans 1, red 9	E	EN	aam$
0,2,5,20,32	ANaE		E	N	aam$
0,2,5,20,32,32	ANaEE		N	-	aam$
0,2,5,20,32,32	ANaEEN	red 7	N	-	aam$
0,2,5,20,32	ANaEN	red 7	N	-	aam$
0,2,5,20	ANaN	trans 1, red 9	E	N	aam$
0,2,5	ANE	trans 1, red 6	C	EN	aam$
0,2	AC	red 3	A	EN	aam$
0,2	A		E	N	aam$
0,2,6	AE		N	-	aam$
0,2,6	AEN	red 7	N	-	aam$
0,2,5	AN		a	-	am$
0,2,5,20	ANa		a	-	m$
0,2,5,20,31	ANaa		m	-	$
0,2,5,20,31	ANaam	red [illegible]	M	-	$
0,2,5,20,31	ANaaM	trans 1, red 12	D	M	$
0,2,5,20,29	ANaD		M	-	$
0,2,5,20,29	ANaDM	red 10	M	-	$
0,2,5,20	ANaM	trans 1, red 12	D	M	$
0,2,5	AND	trans 1, red 5	B	DM	$
0,2,9	[illegible]		D	M	$
0,2,9,12	[illegible]		M	-	$
0,2,9,12	[illegible]	red 10	M	-	$
0,2,9	[illegible]	red 1	S	-	$
0,1	S		$	-	-
0	reduce 0: accept				

Fig. 5.6

One may observe that as soon as two symbols accumulate in the buffer, the parser starts to reduce them. In parsing the sentence, the decision for the reduction of an 'N' to a 'C' or a 'B' is reached by examining the next 'E' or 'D' symbol that follows it in the sentential form. These symbols are only reduced contexts, (not fully reduced ones), for the decision. In turn, the decision for the reduction of every second 'a', in a sequence of 'a's appearing after the first 'n' in the sentence, to an 'E' or 'D' is based on its reduced right context, i.e., the next 'E' or 'D'. The decision for the reduction of others is reached by having the fully reduced right context, i.e., the next 'N' or 'M'.

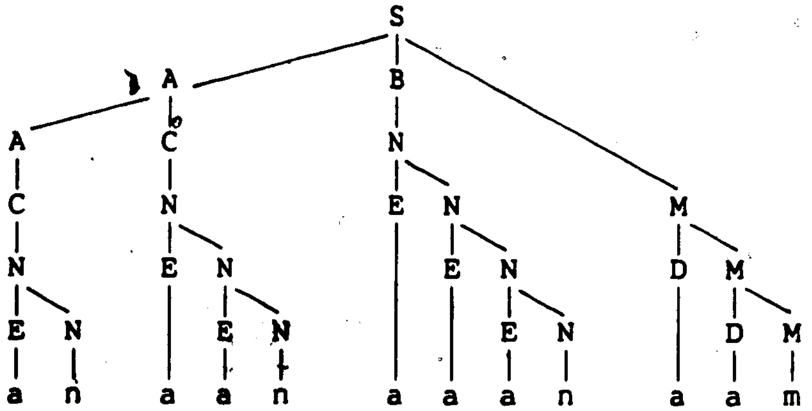


Fig. 5.7
Parse tree of $ana^2na^3na^2m \in H_3$

It is important to note t[illegible] [illegible]nded-buffer parser for the grammar H_3, much like the unbounded-buffer parser for H_1, postpones an unlimited number of reductions of the 'a' symbols appearing in a sentence after the first 'n', Thus, neither H_1 nor H_3 is an LR(k,t) grammar for any finite k and t. Furthermore, the grammar H_3, similar to H_1, is not a basic LRRL(k) grammar for any finite k. Because, for any fixed k, "C^k" and "$C^{k-1}B$" are fully reduced right contexts for the reduction of an 'N' to a 'C'. These two different contexts cannot be parsed completely for the reason that both derive the same non-terminal string "N^k".

Lastly, an interesting phenomenon is that a parser generated by the algorithm 5.1 for the grammar H_3 will use an unlimited number of buffer cells. Because the grammar is not LR(k,t), the parser will deposit arbitrary numbers of E's and D's in the buffer, and the parsing algorithm does not have any mechanism to enforce it to clean up the buffer by reducing the nodes in the buffer. On the other hand, one should note the algorithm 5.1 is a more powerful scheme than the algorithm 5.2. Applying the algorithm 5.2 to the grammar H_1 will result in a failure.

5.3.3 Properties of GLRRL(k) grammars

The GLRRL(k) grammars retain most of the properties of the basic LRRL(k) grammars.

Decidable membership: In the two previous sections it was shown that the membership problem for the class of GLRRL(k) grammars, for fixed k, both in bounded and unbounded buffer cases is decidable. Furthermore, for the case of unbounded buffer, one can easily develop a polynomial time algorithm (similar to the one given in Section 4.4) for GLRRL(k) testing, which details of it are not given in this thesis. However, it seems that the method developed in Section 4.4 is not applicable in the bounded buffer case. The reason for this is the counting of bypassed questionable phrases, which also denies the application of the method to the LR(k,t) grammars. Therefore, the exponential time CFSM construction algorithm 5.2 remains the best algorithm yet for the GLRRL(k) testing with bounded buffer.

Unambiguity and linear parsability: Similar to the basic LRRL(k) grammars, one can show that GLRRL(k) grammars are unambiguous. The linear complexity of the parsing algorithm follows from the fact that at the worst case for the reduction of each phrase k lookaheads will be deposited in the buffer. These nodes are pushed back on the stack in the future accounting for k additional shift operations. The two properties are also immediate from the fact that a CFSM provides a set of regular parsing patterns for a GLRRL(k) grammar.

In addition to the above properties, the GLRRL grammars properly include the LR(k,t) grammars. By definition, the GLRRL(k) grammars with bounded buffer are trivially included in the class of GLRRL(k) grammars with unbounded buffer. Failure of the algorithm 5.2 on the grammar H_1 shows that this inclusion is a proper one. Lemma 5.1 together with Theorem 5.3 ascertain the inclusion of the LR(k,t) grammars in the class of GLRRL grammars with bounded buffer.

Lemma 5.1:

The number of nodes that could be accumulated in the buffer of an LR(k,t) parser never exceeds kt.

Proof: Suppose an LR(k,t) parser in parsing an input sentence $x \in T^{*}\$^{k}$ reaches to a configuration:

Stack	Buffer	Remainder
$\alpha X_1 \ldots X_m$	$a_1 \ldots a_k$	x'

and finds that $X_1 \ldots X_m$ may be reduced to the non-terminal symbol **X**, but the terminal lookahead string $a_1 \ldots a_k$ in the buffer does not provide sufficient right context for such a reduction. That is $X_1 \ldots X_m$ becomes the first questionable phrase that its reduction should be postponed. The parser moves forward and reduces some phrases, and then backs up to the X-phrase while depositing the reduced phrases in the buffer as shown by the following configuration:

Stack	Buffer	Remainder
$aX_1 \ldots X_m$	$Y_1 \ldots Y_k \ldots Y_{n-k+1} \ldots Y_n$	x''

where $Y_{n-k+1} \ldots Y_n$ is a terminal string and x'' is the unseen part of the sentence. Since the parser has moved forward beyond the segment of input that is derived from $Y_1 \ldots Y_k$, it means that at least one of these nodes dominated a questionable phrase whose reductions could not have been inferred by looking at some prefix of the terminal string FIRST $(Y_k Y_{k+1} \ldots Y_{2k})$. (For a maximum effect in the value of n, one can suppose that the questionable phrase was the Y_k-phrase or some rightmost descendant of it). Similarly, one can argue that every k node after Y_k, except $Y_{n-2k+1} \ldots Y_{n-k}$, must have dominated at least one questionable phrase. Since the number of bypassed reductions including the reduction of X-phrase could have not exceeded t-1, thus $n \le kt$. □

Theorem 5.3:

An LR(k,t) grammar can be parsed by an GLRRL(kt) parser with a buffer of size kt^2.

Proof: The lookahead policy that decides the reductions of the first t-1 bypassed phrases in GLRRL(k) parsing with bounded buffer is the same as the one employed in LR(k,t) parsing. Except, if $Y_1 \ldots Y_k$ is the lookahead that decides the reduction of $X_1 \ldots X_m$ to the nonterminal X, and $Y_i \ldots Y_k$ itself is a phrase that can be reduced to a nonterminal say Y without referring to any symbol to the right of Y_k, then reduction of the Y-phrase and probably its dominators takes place before backing up to the X-phrase. This partly is due the fact that LRRL parsers examine the lookaheads one at a time, and become aware of those reductions. Secondly, such reductions are given priority over the reduction of X-phrase in the CFSM construction. The lookahead policy for the t-th phrase could as well be k terminal symbols, which always is so in an LR(k,t) parser. Therefore, one may observe that if an LR(k,t) grammar G is such that in parsing L(G) by the LR(k,t) parser, the contents of the buffer are always some descendants of the k-symbol fully reduced right context of the first bypassed phrase, then the grammar G can be parsed by an GLRRL(k) parser with a buffer of size kt (e.g., the grammar H_3 in Section 5.3.1). However, this is not true in general. The contents of the buffer could be the first kt symbols of the fully reduced right context of the first bypassed phrase, or some descendants of them. Therefore, it can be concluded that in order to parse an LR(k,t) grammar by the GLRRL method, a GLRRL(kt) parser with a buffer of size kt^2 is sufficient. □

In conjunction with the above theorem, one can show that when the algorithm 5.2, with fixed parameters kt and t, is applied to an LR(k,t) grammar, the q-tag will be incremented at most to t and will never be reinitialized to zero. Furthermore, an LR(k,t) grammar will not utilize all the kt^2 cells of the GLRRL(kt) parser. Only, at most kt cells will be used.

Now by considering the above theorem, the following corollary may be stated.

Corollary 5.3:

The class of GLRRL(k) grammars with bounded buffer of size kt properly include the class of LR(k/t,t) grammars.

Proof: The inclusion is immediate from Theorem 5.3. It is proper because some GLRRL(k) grammars with bounded buffer like H, are not LR(k',t') for any finite values of k' and t'. □

Ignoring all the peculiarities of different classes, one may intuitively state that in the construction of an LR(k,t) phrase finding automaton, two mechanisms prevents generating of an unlimited number of states. One is the limitation on the length of the lookahead strings associated with the items, and the other is the limit on the number of bypassed phrases. In the construction of a GLRRL(k) CFSM with unbounded buffer, only one mechanism, i.e., the limit on the size of lookahead strings (associated with the items) achieves the same goal. While in bounded buffer case the number of bypassed phrases is counted, but the arithmetic in fact is modulo t. One may even resemble the last method to a system that implements Szymanski's LR(k,t) method in a periodic fashion. Lastly, one may observe that if one sets out to construct a phrase finding automaton according to the LR(k,∞) scheme, there is no mechanism to prevent one from generating infinite number of states.

Perhaps a remark is in place here. The class of GLRRL(k) grammars with bounded buffer developed in this chapter is not the only way (and perhaps is not even the best way) to define a subclass of GLRRL(k) grammars that use limited number of buffer cells. In fact it is not guaranteed that a GLRRL(k) parser with a buffer of size kt will fully utilize its buffer. However, the implementation reported here is the simplest one and it easily shows the inclusion of the LR(k,t) grammars in the corresponding bounded-buffer GLRRL grammars.

It is important to note that none of the modifications discussed here affects the basic characteristics of the LRRL grammars, specially the decidability of the membership problem. Furthermore, apart from use of unbounded buffers by some GLRRL parsers, all the parsers developed in this research retain the basic advantages of the LR parsers, i.e., automatic generation, table drivability and linear complexity. Indeed a casual user who obtains one of these parsers through automatic generation may not even notice that his/her grammar was not LR(k).

Figure 5.8 gives a lattice diagram of inclusion of different classes of grammars so far discussed in this thesis for given values of k and t. Those in bold characters are proposed in this research.

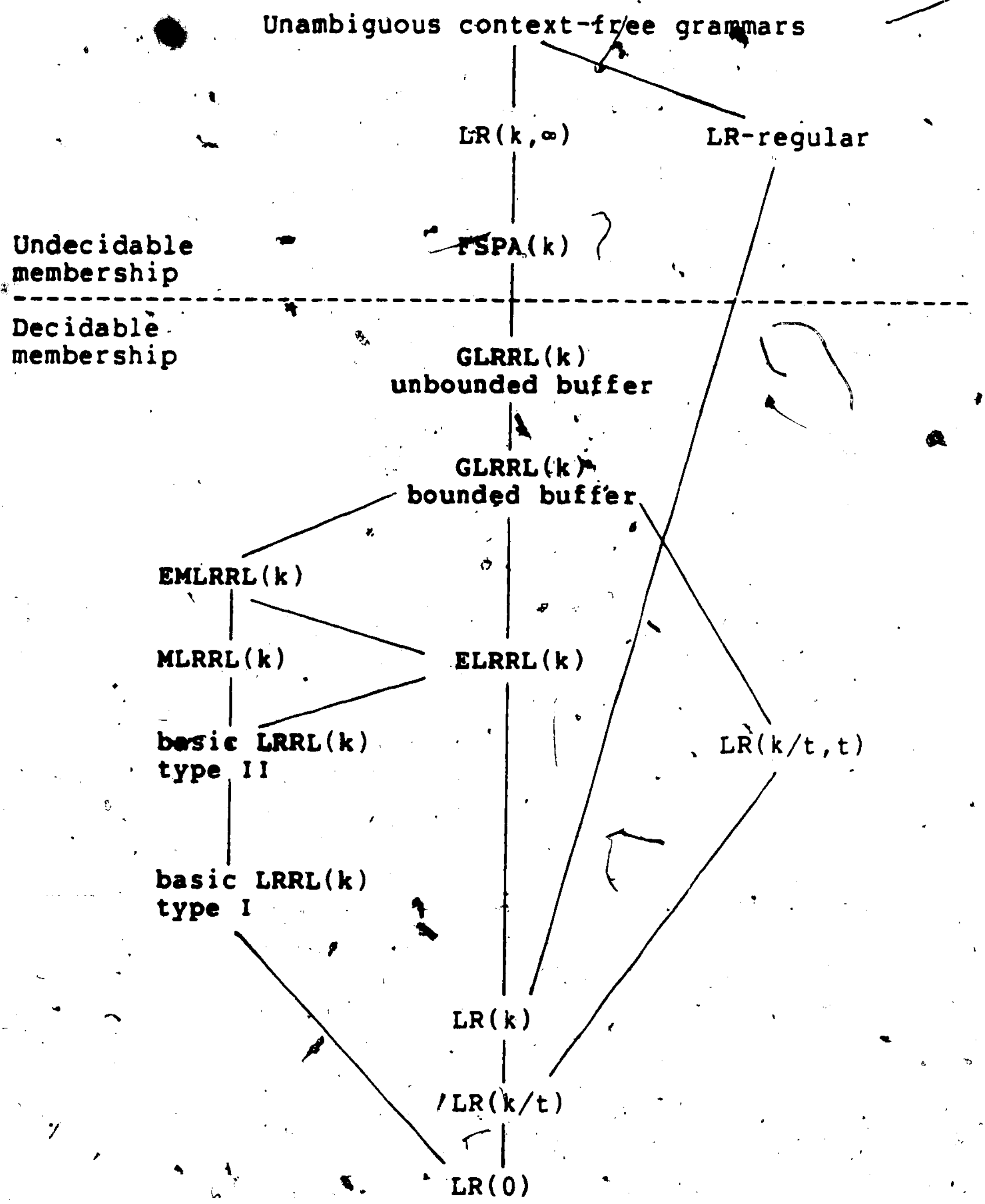


Fig. 5.8
Inclusion of grammar classes

Turning to the language space, one can show that all the parsers with bounded buffers, i.e., basic LRRL(k), ELRRL(k), MLRRL(k), EMLRRL(k) (which have buffers of size k) and GLRRL(k) parsers with buffer of size kt, accept only deterministic context-free languages. This can be proven by simulating the parser by a deterministic pushdown automaton with stack symbols consisting of pairs of a parser state and contents of buffer. A similar proof is given for the Marcus parser in Appendix I. On the other hand ELRRL(k), EMLRRL(k) and bounded-buffer GLRRL(k) grammars include LR(k) grammars. Thus the classes of ELRRL, EMLRRL and bounded-buffer GLRRL languages are the same and are equal to the class of deterministic context-free languages. However, the introduction of unbounded-buffer GLRRL(k) grammars has led to an interesting open problem in the language space. Obviously, unbounded GLRRL languages include the deterministic context-free languages. But, it is not known whether this inclusion is a proper one, or it is an equivalence. The unbounded buffer in fact yields a two-stack parser. Generally, deterministic two-stack parsers may hide the non-determinism of a context-free language. For example, LR(k,∞), FSPA(k), LR-regular and BCP(m,n) parsers are capable to handle some non-deterministic context-free languages such as $L=\{a^n b^{2n} c a^{2m} b^m \mid m,n\geq 0\} \cup \{a^n b^n d a^m b^m \mid m,n\geq 0\}$.

It seems unplausible that an unbounded-buffer GLRRL(k) parser can be simulated by a single stack pushdown automaton. On the other hand, there does not seem to be a counter example for the equivalence. Therefore, the following question is not answered in this research and remains open.

Open problem:

- Is the class of unbounded-buffer GLRRL languages equivalent to the class of deterministic context-free languages?

A similar problem regarding a Marcus-type parser with unbounded buffer is also discussed in Appendix I.

5.4 Subclasses of LRRL(k) grammars

In this section three subclasses of LRRL grammars are discussed.

5.4.1 Marcus parable MP(k) grammars

For each class of LRRL grammars, it is possible to define a subclass of them that can be handled by a Marcus-type parser in a partially top-down manner. Since a Marcus-style parser backtracks to a postponed shift (i.e. attachment) point, these grammars indeed need to be subclasses of type I LRRL grammars in each case. As an example case, the following considers a subclass of type I basic LRRL(k) grammars that a Marcus-type parser for them exists.

In order to characterize these grammars, one needs to decouple the parse tree construction mechanism of the Marcus parser from its finite state control mechanism that is used for the recognition of the productions in the grammar. In constructing a CFSM for these grammars two kinds of ambivalent situations must be considered. First is a situation in which more than one nonterminal symbol appear on the left hand sides of the basis items in a state. Marcus' parser, due to its top-down nature, requires that on entry to a state, the nonterminal symbol on the left hand sides of the basis items to be unique. In this way the next construct is predicted and the corresponding node could be created. Thus, if a state t contains items $(A \rightarrow .C\alpha, L)$ and $(B \rightarrow .C\beta, L')$ then its successor under the symbol C would be inadequate. However, if $(\{C\alpha\} +_k L) \cap (\{C\beta\} +_k L') = \emptyset$ then the conflict may be resolved by parsing ahead the k-symbol lookahead. In this situation the state t will have three different successors under the symbol C. Under the FLAG value OFF, the successor state s, will contain basis items $\{(\text{SUBGOAL-PRED}(A) \rightarrow C.\gamma, \{\epsilon\}) \mid C\gamma \in (\{C\alpha\} +_k L)\}$ and $\{(\text{SUBGOAL-PRED}(B) \rightarrow C.\delta, \{\epsilon\}) \mid C\delta \in (\{C\beta\} +_k L')\}$. This new form of auxiliary productions, i.e., SUBGOAL-PRED(X) $\rightarrow \tau$, indicates a deferred prediction of the nonterminal X. Upon completion one of these auxiliary productions the lookaheads, i.e., the right hand side of the auxiliary production, will be deposited in the buffer and the prediction of the nonterminal X at k states back will be signaled by setting the FLAG to a new value Pred-X. Therefore, in this scheme the FLAG variable besides the previous OFF and ON values could take any of the |N| new values Pred-X where X∈N. In the above situation, the state s_1 the successor of t under the FLAG value Pred-A and the symbol C will contain the basis item (A →C.α,L), and the state s_2 the successor of t under the FLAG value Pred-B and the symbol C will include the basis item (B →C.β,L'). Note that predicting of a new nonterminal can also be in conflict with a shift operation on a longer basis item (i.e., with a minimal attachment), as it is dealt with in the following algorithm.

The second kind of conflicts that may arise in the states of a CFSM, are the reduce-shift conflicts. This is the situation in which two items (A →α.,L) and (A →α.β,L') appear in the same state. To resolve this kind of conflict, the scheme used in the type I basic LRR(k) grammars will be employed. That is auxiliary items

{(SUBGOAL-RED(p) →.γ,{ε})|γ∈L, and p is the production A →α}

and {(SUBGOAL-SHIFT →.σ,{ε}|σ∈({β}$+_{k}$ L')} will be added to the state (provided that the lookaheads indicate a possible resolution of the reduce-shift conflict), and the original conflicting items will be concealed.

Algorithm 5.3 describes the construction of a CFSM in this fashion. The resulting grammars in this method will be denoted by basic MP(k), i.e., Marcus-parsable with k-symbol fully reduced lookaheads.

Algorithm 5.3:

The following algorithm decides if a given grammar is partially top-down parsable with k-symbol fully reduced lookaheads. The algorithm will produce a CFSM if the answer is positive.

(1) Add **GOAL** s[illegible] and production
GOAL →S to [illegible]

[illegible] Build the initial state s_0:
Let the basis of s_0 be { (GOAL →.S,{\$}) }
and its concealed set be ∅.
Close non-concealed basis of s_0.
Let [illegible] set of states of CFSM, $\Omega=\{s_0\}$.

(3) <u>Repeat</u>
For a state t whose successors are not yet determined build its successor states under all applicable symbol X and FLAG values (OFF,ON,Pred-Y;Y∈N) as described below.

(a)- Construction of a successor state s for the given state t under symbol X and FLAG=OFF:

(i) If there is no non-concealed item of the form $(A \rightarrow \alpha.X\beta,L)$ in t, then t has no successor under symbol X and FLAG=OFF.

```
Else
  Check for predict-predict and predict-shift
  conflicts:
  If all non-auxiliary and nonconcealed items
     in t with an X to the right of dot are of
     the form $(A \to \alpha.X\beta_j, L_j)$
     for $j=1,\ldots n$
  then there is no predict-predict or predict-
      shift conflict, and thus let basis of s be
      $\{(A \to \alpha X.\beta_j, L_j) \mid j=1,\ldots,n\}$;
      Check for reduce-shift conflicts in
      the basis of s:
      If there are items of the form
        $I_j : A \to \alpha X.\beta_j, L_j$;
        $j=2,\ldots,n$ and $\beta_j \neq \epsilon$
        and
        $I_1 : A \to \alpha X., L_1$;
        where $n \geq 2$
      then
        let the shift lookaheads of s be
        $\mathrm{SHL}(s)=\mathrm{Min} \bigcup_{j=2}^{n} (\{\beta_j\} +_k L_j)$.
        If $L_1 \cap \mathrm{SHL}(s)=\emptyset$
         (i.e. possible shift-reduction
          resolution)
        then
          conceal the original conflicting basis
          items $I_1,\ldots,I_n$, and add new
        non-concealed basis, i.e.
        add (SUBGOAL-RED($p_1$) $\rightarrow .\gamma,\{\epsilon\}$)
        to s for all $\gamma \in L_1$;
        where $p_1$ is the production $A \rightarrow \alpha X$.
        Also add (SUBGOAL-SHIFT $\rightarrow .\gamma,\{\epsilon\}$) to s
        for all $\gamma \in$ SHL(s).
      Else
        conclude the grammar is not
        basic MP(k) and exit.
  Else (i.e., when there is a predict-predict or
        predict-shift conflict in t.)
        Let SHL(t)=$\{(X\beta_j) +_k L_j\}$
        such that $(A \rightarrow \alpha.X_j, L_j)$
        is an item in t and $\alpha \neq \epsilon\}$.
        Let PL($B_i$)=$\{(X\beta_{ij}) +_k L_{ij}$
        such that $(B_i \rightarrow .X\beta_{ij}, L_{ij})$
        is an item in t}.
        If for all i,j and $i \neq j$
           SHL(t) $\rightarrow$PL($B_i$)=$\emptyset$
           (i.e., possible resolution of
            predict-shift conflicts)
           and
           PL($B_i$) $\rightarrow$PL($B_j$)=$\emptyset$
           (i.e., possible resolution of
            predict-predict conflicts)
        then
           Conceal all the items
```

γ

$(A \to \alpha . X\beta_{j})$

in state t, where $\alpha \neq \epsilon$)

Let the basis of s=

$\{(\text{SUBGOAL-SHIFT} \to X.\gamma, \{\epsilon\}) \mid X\gamma \in \text{SHL}(t)\}$

and for all i, and $X\delta \in$ [illegible](B_i)

add the item

$(\text{SUBGOAL-PRED}(B_i) \to X.\delta, \{$[illegible]$\})$

to s.

Else

conclude the grammar is not basic MP(k) and exit.

(ii) Close the non-concealed basis of s.

(iii) Add state s to Ω if there is no state t' in Ω with the same items as s, or with items which match those of s apart from the lookahead sets, and for lookahead set L' in t' it is true that L'=Min(L∪L') and L'≠L∩L', where L is the corresponding lookahead set in s.

(i.e., each $\sigma' \in L'$ is a prefix of some $\sigma \in L$ and each $\sigma \in L$ has a prefix σ' in L'.

This latter condition is introduced as a means of optimization that reduces lengths of necessary lookaheads.)

Otherwise let t' be the successor of t.

(b)- Construction of a successor state s for the given state t under symbol X and FLAG=ON :

```
(i) If there is no concealed item of the form
        $A \rightarrow \alpha.X\beta,L$ in t,
    then state t has no successor under symbol X
        and FLAG value ON.
    Else
        let non-concealed basis of s be $\emptyset$.
        For each concealed item $(A \rightarrow \alpha.X\beta,L)$
        in t add $(A \rightarrow \alpha X.\beta,L)$ to the basis of s.
        Check for shift-reduce conflicts in s
        as in part (a) and add any necessary
        SUBGOAL-RED and SUBGOAL-SHIFT auxiliary
        items.

(ii)-(iii) Repeat steps (ii) and (iii) of
           part (a) for state s.
```

(c)- Construction of a successor state s for the given state t under symbol X and FLAG=Pred-A:

```
(i) If state t had a predict-predict or
    predict-shift conflicts under the symbol X,
```

then t has successors under symbol X and some FLAG values other than On and OFF. In this case the basis of s is $\{(A \rightarrow X.\beta_j, L_j)\}$ such that each $(A \rightarrow .X\beta_j, L)$ is a conflicting item in t predicting A as the next construct. After obtaining the basis, check for shift-reduce conflicts and add any necessary SUBGOAL-RED or SUBGOAL-SHIFT auxiliary items a[illegible]rt (a).

(ii)-(iii) Repeat the steps (ii) and (iii) of part (a) for state s.

<u>Until</u> no more state can be added to Ω.

(4) Conclude that the grammar is basic MP(k).

The CFSM for a grammar, obtained by applying the algorithm 5.3, can act as a finite state control mechanism both for a bottom-up parser and for a partially top-down one. In the bottom-up parser, as before, the completed right hand sides of the auxiliary productions are deposited in the buffer, and therefore one does need more that a k-cell buffer. In the partially top-down parser, the original nonterminal symbols that were predicted but their daughters are not yet completely parsed reside on the stack. This scheme is applicable to the MP(k) grammars, because of the uniqueness of the original nonterminal symbol that may appear on the left hand side of the basis items in a state. In this method, shifting of a symbol is treated as the attachment to its parent.

On the other hand, auxiliary nonterminals such as SUBGOAL-PRED(X) or SUBGOAL-PRED(p) do not have a corresponding node on the stack. Instead, each symbol appearing on the right hand side of an auxiliary production, upon completion of its parse is deposited in the buffer. Therefore, in general the number of buffer cells could be unlimited. However, one can consider a more restricted subclass of these grammars that use limited size buffers. This can be achieved by associating a tag l with the items of the CFSM. The l-tag will indicate the number of occupied cells in the buffer upon reaching the state.

One should also note that on transferring a node into the buffer, a pointer (which in fact defines a window) will be moved one cell forward to [illegible] on co[illegible] of auxiliary production the pointer will b[illegible] backward to the left a number of cells equal to the size of the right hand side of the auxiliary production.

One may observe that in this method a node corresponding to the current construct is created when its first daughter or some fully re[illegible]ight context becomes available. This scheme is v[illegible]imilar to Marcus' parser except that Marcus' parser can[illegible] create some dominators of the current construct if t[illegible] can be uniquely determined. This feature can be added to the present parser by analyzing the closure graph of the items in each state.

In this section, only the lookahead policy of the basic LRRL grammars (i.e., the complete parsing of the fully reduced contexts), was applied in a partially top-down parser. One can construct similar parsers that employ the ELRRL, MLRRL or GLRRL lookahead policies (i.e., an incomplete parse of the fully reduced contexts). A parser based on GLRRL lookahead policy would parse the grammar G_1 of Chapter 2, exactly in the same manner that the Marcus-style parser described by the PIDGIN grammar G_2 does.

5.4.2 Simple LRRL(k) grammars

Similar to SLR(k) grammars [Dere 71], the simple LRRL(k) grammars may be defined by employing non-exact lookaheads. In this scheme, associated with each item, there is a number 1≤k that indicates the number of lookaheads allowable for that item. When an inadequacy arises in a state of LR(0)-CFSM, lookaheads are obtained by computing RED-NEXT sets, where $\text{RED-NEXT}_k(S)=\{\$\}$, and for a non-terminal A, $\text{RED-NEXT}_k(A)=\cup(\{\beta\} +_k \text{RED-NEXT}_k(B))$ such that $B \rightarrow \alpha A\beta \in P$.

RED-NEXT$_k$(A) is the set of k symbol strings that can appear next to the right of A in the set of leftmost sentential forms. When two items A $\rightarrow \alpha.$,n and B $\rightarrow \alpha.\beta$,m are in conflict, new items of the forms SUBGOAL-RED(p) $\rightarrow .\sigma,0$; $\sigma \in$RED-NEXT$_n$ (A) and SUBGOAL-SHIFT $\rightarrow .\sigma,0$; $\sigma \in \{\beta\} +_k$ RED-NEXT$_k$ (B) will be added to the inadequate state, provided that they indicate a possible resolution.

Similar grammars were also introduced in [Szym 73] and again discussed in [Tai 79] for k=1.

5.4.3 LALR analogues of LRRL(k) grammars

Discussed in the optimization of CFSM construction, was a class of LRRL grammars analogous to LALR grammars where states with similar items and non-conflicting lookahead sets may be merged.

The CFSM for these grammars either can be constructed from a corresponding LRRL(k)-CFSM by merging the states with common core segments. Or, it can be constructed from an LR(0)-CFSM by establishing propagate links as in LALR(k) grammars.

5.5 LRRL(k) grammars with ϵ-rules

So far, it was assumed that all the context-free grammars under consideration in this thesis are ϵ-free. ϵ-rules can be incorporated into LRRL(k) grammars in two different ways. One approach is similar to the LR(k,t) grammars, in which no distinction is made regarding the lookaheads that may derive null string. In this approach, a right context may shrink away to null string, giving absolutely no additional information for making parsing decisions. LR(k,t) parsers that employ this strategy, also suffer from a certain anomaly in that they parse some grammars with ϵ-rules which technically are not LR(k,t) [Szym 73, p167].

In the second approach, which is adopted here, one takes into consideration only those lookaheads that do not derive null strings. A nonterminal symbol A that derives both non-null strings and ϵ, will be represented by $\bar{A}$ in the lookahead strings. In the subsequent manipulations $\bar{A}$ would be prohibited to derive a null string. Therefore, given a context-free grammar G=(N,T,P,S), one needs to subcategorize the nonterminal symbols N into N_1, N_2 and N_3, where

N_1= The set of nonterminals that do not derive a null string.

N_2= The set of nonterminals that derive both non-null and null strings. The instances of these nonterminals that yield non-null strings will be denoted by $\bar{N}_2$.

N_3= The set of nonterminals that only derive null strings.

Such a categorization can be obtained by a straightforward iterative algorithm.

The following preliminary definitions will be needed in the presentation of algorithms for these grammars.

Definition: ϵ-free concatenation

Let $\alpha = X_1 \ldots X_n$ be a string such that none of X_i in it derives a null string. Then ϵ-free concatenation of a symbol X with α is

$$\epsilon\text{-free cat }(X,\alpha) = \begin{cases} X\alpha & \text{If } X \in N_1 \cup T \\ \bar{X}\alpha & \text{If } X \in N_2 \\ \alpha & \text{If } X \in N_3 \end{cases}$$

ϵ-free concatenation of a string $\beta = Y_1 \ldots Y_m$ with string α is recursively defined by:

$$\epsilon\text{-free cat}(\beta,\alpha) = \epsilon\text{-free cat}(Y_1, \epsilon\text{-free cat}(Y_2 \ldots Y_m, \alpha)).$$

Definition: k-bounded ϵ-free concatenation of two sets

Let L_1 be set of strings and L_2 a set of strings of non-null deriving symbols. The k-bounded ϵ-free concatenation of L_1 with L_2 defined as:

$$L_1 \underset{k}{\pm} L_2 = \{PF_k(\epsilon\text{-free cat}(\beta,\alpha)) \mid \beta \in L_1 \text{ and } \alpha \in L_2\}.$$

Definition: k-symbol ϵ-free fully reduced right context

The k-symbol ϵ-free fully reduced right context of a phrase in a parse tree consists of the k non-null deriving nodes that follow the phrase in the leftmost derivation of the tree. Thus these nodes dominate any sequence of k subtrees to the immediate right of the phrase that have non-null frontiers (i.e., a k-symbol ϵ-free reduced right context of the phrase).

The following subsections present the closure and CFSM construction algorithms for the type II basic LRRL(k) grammars in the presence of ϵ-productions. The parsers for these grammars employ k-symbol ϵ-free fully reduced right contexts in making parsing decisions. Similar algorithms can be developed for the other classes of LRRL grammars.

5.5.1 Closure operation in the presence of ϵ-rules

Recall that a nonterminal symbol A may be shown as $\bar{A}$ in an item. In closing a state, one should be aware that such symbols are not supposed to derive a null string. The following algorithm handles such a provision.

```
Add(I: item, s: set of items)
    Suppose $I=(B \to .\gamma,L)$. If there is an item
    $I'=(B \to .\gamma,L')$ in s, then replace I' by
    $I''=(B \to .\gamma,L'')$, where $L''=Min(L \cup L')$.
    Else let $s=s \cup \{I\}$.

Close(s: set of items)
   Repeat
     For an item $A \to \alpha.X\beta,L$ in s
       If X is a nonterminal symbol B in N then
          for all $B \to \gamma \in P$
            Add($(B \to .\gamma,\{\beta\} \pm_k L)$,s).
       Else
    If X is a nonterminal $\bar{B}$ in $\bar{N}_2$ then
        for all $B \to \gamma$ in P
        such that $\gamma \neq \epsilon$ and $\gamma \notin N_3^*$
        If $\gamma = X_1 \ldots X_n$ and at least
        one of $X_i$'s is in $N_1 \cup T$
        then Add(($\hat{B} \to .\gamma, \{\beta\} \pm_k L$),s).
        Else for each i=1,...,n
            Add(($\bar{B} \to .X_1 \ldots \bar{X}_i \ldots X_n, \{\beta\} \pm_k L$),s).

Until no change can be made in set of items s.
```

5.5.2 Construction of CFSM for grammars with ϵ-rules

Let G=(N,T,P,S) be a reduced context-free grammar. Let (N_1,N_2,N_3) be the partitioning of N as described above. The following algorithm decides whether G is type II basic LRRL(k) for fixed number k. The algorithm produces a CFSM if the answer is positive.

Algorithm 5.4:

```
(1) Add GOAL symbol to N and production
    0: GOAL $\to$S to P.

(2) Build the initial state $s_0$:
        Let $s_0$ = { (GOAL $\to$.S,{$}) }.
        Close ($s_0$).
        Let the set of states of CFSM, $\Omega$={$s_0$}.
```

(3) Repeat

For a state t whose successors are not yet determined build its successor states under all applicable symbol X.

Construction of a successor state s for the given state t under symbol X:

(i) Let s=∅.

For each item $(A \to \alpha.X\beta, L)$ in t add $(A \to \alpha X.\beta, L)$ to s.

Furthermore if X is a nonterminal $\bar{B}$ in $\bar{N}_2$ then for each item $(A \to \alpha.B\beta, L)$ in t, also add $(A \to \alpha B.\beta, L)$ to s.

(ii) Close(s).

(iii) Check for inadequacy in s:

If there are items of the form

$$I_j : A_j \to \alpha_j .B_j \beta_j ,L_j ;$$
$$j=1,\dots,m$$

and

$$I_j : A_j \to \alpha_j ., L_j ;$$
$$j=m+1,\dots,n$$

where m≥1 and n>m

(i.e. shift-reduce/reduce-reduce conflicts) or

m=0 and n>1

(i.e. only reduce-reduce conflicts)

Then

let the shift lookaheads of s be

$SHL(s)=\mathrm{Min} \bigcup_{j=1}^{m} B_j(\{\beta_j\}+_k L_j)$.

(Note the resemblance to ε-free $FIRST_k$ in LR(k) grammars.).

If $L_j \cap SHL(s)=\emptyset$ for j=m+1....,n

(i.e. possible shift-reduction resolution)

and

for i,j=m+1,...,n ,i≠j,

items I_i and I_j are the same when the dashes over the symbols are ignored or $L_j \cap L_j=\emptyset$

(i.e. possible resolution of reduce-reduce conflicts. Note that two items $(A \to \alpha.,L)$ and $(\bar{A} \to \bar{\alpha}.,L')$ are assumed not to pose a conflict.)

Then

conceal the conflicting items in s:

$I_{m+1},...,I_n$ and add new items corresponding to their lookaheads, i.e.,

```
            For j=m+1,...,n
               add SUBGOAL-RED($p_j$) $\rightarrow . \gamma , \{ \epsilon \}$
               to s for all $\gamma \in L_j$;
               where $p_j$ is the production $A_j \rightarrow \alpha_j$ .
            Goto step (ii).
         Else
            conclude the grammar is not type II
            basic LRRL(k) and exit.
      Else, i.e., when there is no conflicting
            items in s, proceed to step (iv).

   (iv) Add state s to $\Omega$ if there is no state t'
        in $\Omega$ with the same items as s, or with
        items which match those of s apart from the
        lookahead sets, and for lookahead set L'
        in t' it is true that $L' = Min(L \cup L')$ and
        $L' = L \cap L'$, where L is the corresponding
        lookahead set in s.
        Otherwise let t' be the successor of t.

Until no more state can be added to $\Omega$.

(4) Conclude that the grammar is type II basic LRRL(k).
```

The parsing algorithm for these grammars is exactly the same as the one used for the grammars with no ϵ-productions.

Thus the parsing algorithm is not repeated here and only an example will be presented.

Example:

Let G be a context-free grammar with productions:

(1) S → A C

(2) S → B D

(3) A → a

(4) B → a

(5) C → E F

(6) C → ϵ

(7) D → M N

(8) E → [illegible]

(9) E → ϵ

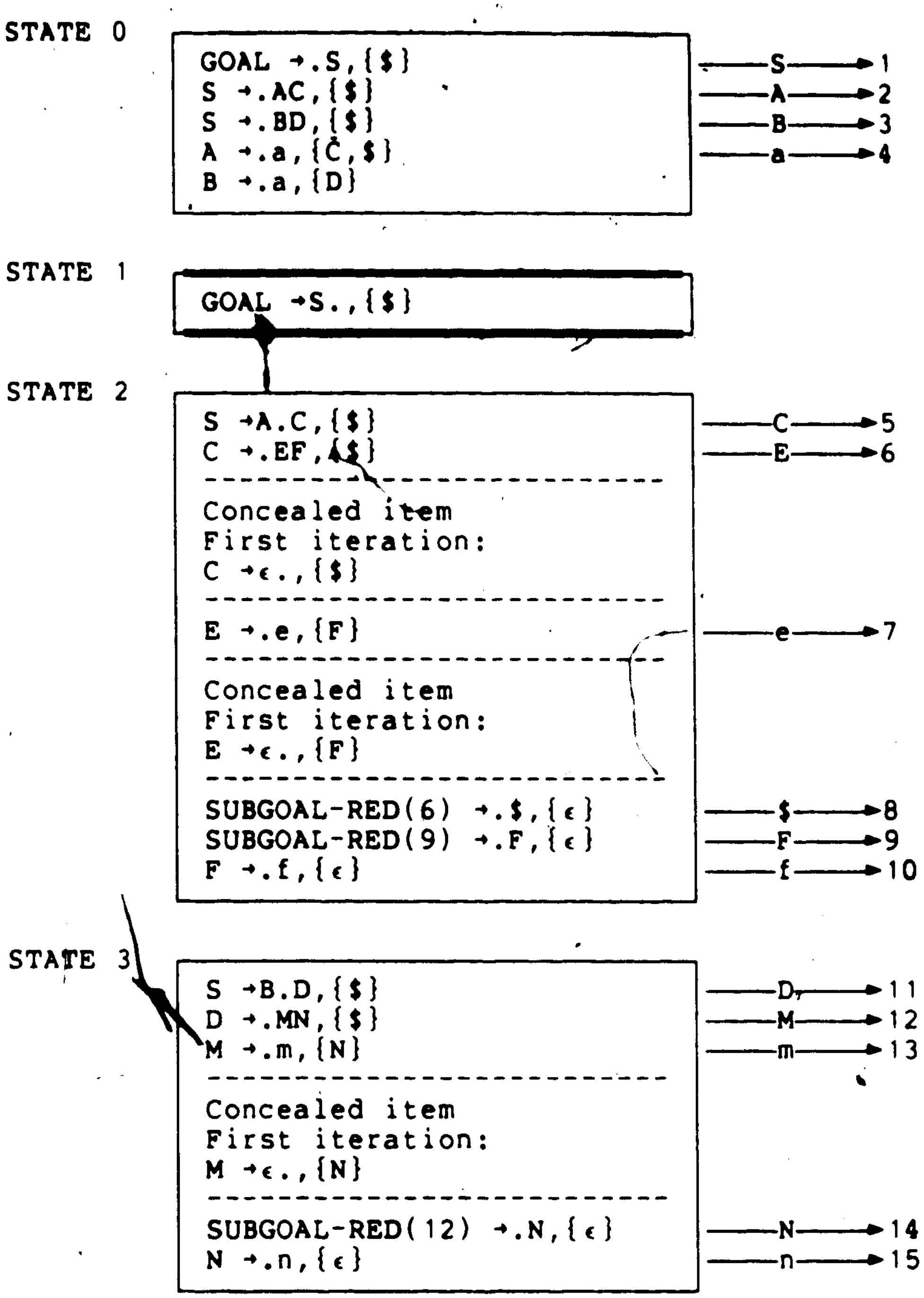
(10) F → f

(11) M → m

(12) M → ϵ

(13) N → n

In the above grammar nonterminals C,M and E derive both null and non-null strings. Diagram 5.4 shows a type II basic LRRL(1)-CFSM for G constructed by the algorithm 5.4. The corresponding parse table is shown by Table 5.4.



Diag. 5.4
LRRL(1)-CFSM for a grammar with ε-rules

STATE 4

```
Concealed items
First iteration:
A →a.,{Ĉ,$}
B →a.,{D}
---------------------------
SUBGOAL-RED(3) →.Ĉ,{ε}        ——Ĉ——→16
SUBGOAL-RED(3) →.$,{ε}        ——$——→17
SUBGOAL-RED(4) →.D,{ε}        ——D——→18
Ĉ →.EF,{ε}                    ——E——→19
E →.e,{F}                     ——e——→7
---------------------------
Concealed item
Second iteration:
E →ε.,{F}
---------------------------
D →.MN,{ε}                    ——M——→20
M →.m,{N}                     ——m——→13
---------------------------
Concealed item
Second iteration:
M →ε.,{N}
---------------------------
SUBGOAL-RED(9) →.F,{ε}        ——F——→9
SUBGOAL-RED(12) →.N,{ε}       ——N——→14
F →.f,{ε}                     ——f——→10
N →.n,{ε}                     ——n——→15
```

STATE 5

```
S →AC.,{$}
```

STATE 6

```
C →E.F,{$}                    ——F——→21
F →.f,{$}                     ——f——→10
```

STATE 7

```
E →e.,{F}
```

STATE 8

```
SUBGOAL-RED(6) →$.,{ε}
```

STATE 9

```
SUBGOAL-RED(9) →F.,{ε}
```

STATE 10

```
F →f.,{ε}
```

```
STATE 11   S →BD.,{$}

STATE 12   D →M.N,{$}              ——N——→22
           N →.n,{$}               ——n——→15

STATE 13   M →m.,{N}

STATE 14   SUBGOAL-RED(12) →N.,{ε}

STATE 15   N →n.,{ε}

STATE 16   SUBGOAL-RED(3) →Č.,{ε}

STATE 17   SUBGOAL-RED(3) →$.,{ε}

STATE 18   SUBGOAL-RED(4) →D.,{ε}

STATE 19   Č →E.F,{ε}              ——F——→23
           F →.f,{ε}               ——f——→10

STATE 20   D →M.N,{ε}              ——N——→24
           N →.n,{ε}               ——n——→15

STATE 21   C →EF.,{$}

STATE 22   D →MN.,{$}
```

STATE 23

Ĉ →EF.,{ε}

STATE 24

D →MN.,{ε}

States

Symbols	0	1	2	3	4	6	12	19	20
a	g4								
e			r8		r8				
f			r10		r10	r10		r10	
m				r11	r11				
n				r13	r'3		r13		r13
A	g2								
B	g3								
C			r1						
Ĉ			r1		t1,r3				
D				r2	t1,r4				
E			g8		g19				
F			t1,r9		t1,r9	r5		r5'*	
M				g12	g20				
N				t1,r12	t1,r12		r7		r7
S	g1								
$		r0	t1,r6		t1,r3				

* Production 5' is Ĉ →EF.

Table 5.4
Parse table for an LRRL(1) a grammar with ε-rules

Figure 5.9 illustrates parsing of the sentence af ∈ L(G). The corresponding parse tree is depicted in Figure 5.10.

State stack	Symbol stack		Current symbol	Buffer	Remainder
-	-		-	-	af$
0	-		a	-	f$
0,4	a		f	-	$
0,4	af	red 10	F	-	$
0,4	aF	trans 1, red 9	E	F	$
0,4,19	aE		F	-	$
0,4,19	aEF	red 5'	Č		$
0,4	aČ	trans 1, red 3	A	Č	$
0,2	A		Č	-	$
0,2	AC	red 1	S	-	$
0,1	S		$	-	-
0	reduce 0: accept				

Fig. 5.9
Parsing of the sentence "af"

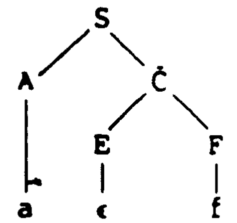


Fig. 5.10
Parse tree of the sentence "af"

The grammar G was chosen in Example 4 to show the handling of ε-productions in a general way with a finite language. However, a more interesting example of LRRL(1) grammars is the grammar H given in [Szym 73, p167] and shown here. This grammar technically is not LR(k,t). Nevertheless, the way LR(k,t) parsers are constructed enables them to handle this grammar.

H:

$$\begin{array}{ll} S \rightarrow A\ a & S \rightarrow B\ b \\ A \rightarrow d\ A\ E & A \rightarrow d\ E \\ B \rightarrow d\ B\ E & B \rightarrow d\ E \\ E \rightarrow \epsilon & \end{array}$$

5.6 Conclusion

This section concludes the discussion of context-free syntax of languages. Chapters 3 through 5 introduced new classes of unambiguous context-free grammars that in parsing them, one employs more complex lookaheads than one is allowed in LR parsing. It was shown that many of these classes properly include LR(k) grammars, and they provide a basis for constructing Marcus type parsers. Furthermore, introduction of LRRL(k) grammars and previous existence of LR(k,t) grammars put a question mark against what Hunt has concluded in 1982 [Hunt 82].

The concept of regular separability was stated in Section 2.3.1. Hunt, in dealing with decidability of grammar problems such as cover problems, generalizes this concept to *C*-separability of two languages, where *C* is an arbitrary class of languages [Hunt 82]. *C*-separability can be defined in the following way.

Definition: $\mathcal{C}$-separability

Let G and H be grammars in a class Γ, and $\mathcal{C}$ be a class of languages. L(G) and L(H) are said to be $\mathcal{C}$-separable if and only if there exists a language $S \in \mathcal{C}$ such that $L(G) \subset S$ and $L(H) \cap S = \emptyset$.

S is said to be a $\mathcal{C}$-envelope for L(G) [Nijh 80].

A special emphasis has been put on *definite event* separability. The class of definite event languages is defined as follows.

Definition: Definite event languages

A language L is said to be definite event if and only if there exists a finite alphabet Σ and finite sets $F_1 \subseteq \Sigma^*$ and $F_2 \subseteq \Sigma^*$ such that $L = F_1 \Sigma^* \cup F_2$.

One can easily observe that the definite event separability of two arbitrary languages is undecidable. In fact the LR(k) condition, for *some* k, for an arbitrary context-free grammar is equivalent to this problem. The following is a general theorem that considers undecidable relations between pairs of grammars.

Theorem 5.4 (Hunt 1982):

Let Γ be a class of grammars. Let ρ_1 and ρ_2 be binary relations on Γ defined by

(1) G ρ_1 H if and only if the languages L(G) and L(H) are

definite event separable.

(2) $G \rho_2 H$ if and only if the intersection of the languages $L(G)$ and $L(H)$ is a context-free language.

Let ρ be any binary relation on Γ between ρ_1 and ρ_2, i.e. for all $G,H \in \Gamma$, if $G \rho_1 H$ then $G \rho H$, and if $G \rho H$ then $G \rho_2 H$. Then it is undecidable, for grammars G and H in Γ, whether $G \rho H$.

From the above theorem, Hunt has concluded that generalizing the concepts of LR parsing techniques must produce a class of grammars with an undecidable membership problem. One may observe that the relations between the grammars in the theorem are given in terms of some relationships between the languages that these grammars generate. Intuitively, the relevance of the theorem to various parsing techniques is that at a point where reduction of a phrase is in question, the two sets of terminal lookaheads that provide sufficient right context for the correct decision are two context-free languages L_{yes} and L_{no}. In order to make a correct decision these languages must be separatable. In the case of LR(k) parsing, the lookahead languages are finite, however problems arise when these languages are not finite. For example, in LR-regular parsing the lookaheads for reduction are the set of all possible terminal strings than can follow the phrase in the sentence. The criteria used for definitions of LR-regular and FSPA(1) grammars require that L_{yes} and L_{no} to be regular separable. While, the criteria for LR(1,∞) grammars implies that L_{yes} and L_{no} to have empty intersection. A condition which can be verified if one was able to show that in general $L_{yes} \cap L_{no}$ is a context-free language.

One may see that shifting of attention from grammars to relations on languages, through finding envelopes for them, inevitably leads to undecidable problems. In LRRL(k) parsing, although the lookahead languages may be infinite, but the conditions are stated in terms of the grammar that generates them. By introducing the auxiliary items in a state of the CFSM, one may assume that a grammar G_{yes} with the start symbol SUBGOAL-RED(Phrase) is defined that generates L_{yes}. Similarly, a grammar G_{no} with a competing SUBGOAL symbol as its start symbol generates L_{no}. However, in LRRL(k) scheme only one grammar that generates $L(G_{yes}) \cup L(G_{no})$ is introduced. The LRRL(k) criteria requires that the latter grammar to be LRRL(k) with no end-marker.

Thus if one wishes to describe the criterion for any class of LRRL(k) grammars, with a fixed k, as a relation τ_k between G_{yes} and G_{no}, then the relation τ_k can be stated as:

- Let $H=(N_1,T_1,P_1,S_1)$ and $J=(N_2,T_2,P_2,S_2)$ be context-free grammars. $H \; \tau_k \; J$ if and only if the grammar $G=(N_1 \cup N_2 \cup \{S\},\ T_1 \cup T_2,\ P_1 \cup P_2 \cup \{S \rightarrow S_1,\ S \rightarrow S_2\},\ S)$ has an LRRL(k)-CFSM with no end-marker.

Clearly, $H \; \tau_k \; J$ implies that $L(H) \cap L(J) = \emptyset$. However, the relation τ_k neither implies nor is implied by the relation

ρ, of Theorem 5.4.

CHAPTER 6

TOWARDS ATTRIBUTED LRRL(k) PARSING

This chapter is mainly concerned with attributed grammars, or if one wishes the context-sensitivity of a language. Rather than giving a complete algorithm for parsing general attributed grammars, special topics are discussed separately. In fact, the content of this chapter may be considered as a groundwork for further research on a general deterministic attributed parsing technique. Among the issues discussed are evaluation of attributes in attributed LRRL(k) grammars, attribute directed parsing, and semantic disambiguation by means of the 'wait and see' policy. Also, applications of LRRL and attributed LRRL parsers in syntactic error correction, and binding of the traces in Marcus-type parsers are discussed in this chapter.

6.1 Background

Attribute grammars were devised by Knuth [Knut 68]. Affix grammars, that are formally related to attribute grammars, were introduced in an implementation of Algol 68 by Koster [Kost 71]. Attribute grammars are currently assumed to the the most promising tools available in design of compiler writing systems. They are clean extensions of context-free grammars that are suitable for describing the context-sensitive syntax of programming languages (sometimes called the static semantics), e.g., scope of identifiers, and semantics of those languages. It has been reported [Watt 77] that Koster has used affixes in generating natural language as far back as 1965.

The definition of attribute grammars varies with different authors [Knut 68], [Boch 76], [Kast 80], [Tien 80] and [JaPo 81], and in terms of the fine details of definition, no generally accepted standard definition seems to have emerged yet. The basic idea is to augment each symbol of a context-free grammar with a set of attributes, and associate a set of semantic rules or evaluation rules with each production of the grammar. These rules evaluate the value of each attribute of a symbol in the derivation tree of a sentence. The definition adopted in this thesis is similar to the one given in [Kast 80] which appears to be more complete than others.

6.1.1 Attributed grammars

An Attribute grammar (or attributed grammar) **AG** is defined by 5-tuple AG=(G,A,D,F,C). G=(N,T,P,S) is a reduced context-free grammar. P is the set of syntactic rules. Each syntactic rule p∈P has the form p: $X_0 \rightarrow X_1 \ldots X_n$, n ≥0. X_i denotes an occurrence of a symbol of N for i=0 and V=N∪T for i>0. In the following, X_i will denote an occurrence of symbol X in such a rule p. A is a set of attributes. Each attribute is associated to exactly one symbol X∈V. A_X is the set of attributes associted to X. The elements of A_X are denoted by X.a, X.b,... and $A=\cup_{X\in V} A_X$, and $A_X \cap A_Y \neq \emptyset$ implies X=Y. A is partitioned into two disjoint sets AI and AS, the inherited and the synthesized attributes. A=AI∪AS and AS∩AI=∅. Hence each A_X is partitioned into two subsets AI_X and AS_X. (Consider a symbol X and a phrase p derived from X. Each inherited attribute of X is supposed to convey information about the context of p, and each synthesized attribute of X is supposed to convey information about p itself.) For each occurrence X_i of a symbol X in a rule p there is an attribute occurrence X_i.a for all attribute a∈A_x. $A_p = \cup_{i=0}^{n} \cup_{a\in A} X_i.a$ is the set of all attribute occurrences in p. With attribute X.a, a fixed domain DOM(X.a) is associated. The values defined for attribute occurrences are taken from the domain of the attributes, i.e., DOM(X_i.a)=DOM(X.a) for X_i=X. D is the set {DOM(a)|a∈A}.

Furthermore, for each syntactic rule p the set of defining (or defined) attribute occurrences is taken to be $DEF_p = \{X_i.a \mid (i=0 \text{ and } a \in AI) \text{ or } (i>0 \text{ and } a \in AS)\}$ and applied occurrences is $APP_p = \{X_i.a \mid (i=0 \text{ and } a \in AS) \text{ or } (i>0 \text{ and } a \in AI)\}$. Applied attribute occurrences are evaluated from defined occurrences by set of F_p semantic functions associated to a rule $p \in P$, and $F = \cup_{p \in P} F_p$. Each semantic function is assumed to be in normal form, i.e., it defines the value of an attribute occurrence in p depending on $k \geq 0$ attribute occurrences in p: $f \in F_p$ and

$f \subset \times_{j=0}^{k} DOM(b_j \in DEF_p) \to DOM(a \in APP_p)$ for $k \geq 0$.

Note that in [Kast 80] applied and defining occurrences are used in opposite place of each other, but consensus among authors seems to be that these terms should be used as given in the preceding paragraph.

C is a set of semantic conditions, one associated to each rule

$C_p \in \times_{j=1}^{k} DOM(a_j \in A_p) \to \{true, false\}$.

A sentence $s \in L(G)$ is a sentence in $L(AG)$ if and only if for each application of a rule p in the derivation of s the values of the corresponding attribute occurrences meet the condition C_p.

6.1.2 Extended attribute grammars

In 1983, Watt and Madsen [WaMa 83] introduced the extended attribute grammars. EAG's are intended to preserve all the desirable properties of attribute grammars, but at the same time to be more concise and readable and like van Wijngaarden grammars to be generative. The basic idea behind EAG is to allow attribute expressions rather than just attribute occurrences. Moreover the restriction that each attribute occurrence must be in only one defining position is relaxed. These relaxations allow all relationship among the attributes in each syntactic rule to be expressed implicitly, so that explicit evaluation rules and constraints become unnecessary. The following gives the definition of an extended attribute grammar.

An extended attribute grammar is a 5-tuple EAG=(D,V,S,A,P) where $D=(D_1,D_2,\ldots,f_1,f_2,\ldots)$ is an algebraic structure with domains $D_1,D_2,\ldots$ and partial functions $f_1,f_2,\ldots$ operating on Cartesian products of these domains. Each object in each of these domains is called an attribute. V is the vocabulary of grammar, a finite set of symbols which is partitioned into the non-terminals N and terminal vocabulary T. Associated with each symbol in V is a fixed number of attribute positions. Each attribute position has a fixed domain chosen from D, and is classified as either inherited or synthesized. A symbol A with n inherited and m synthesized positions is written as $A\downarrow\underset{n}{\underline{\ldots}}\uparrow\underset{m}{\underline{\ldots}}$ · S, a member of N, is the distinguished non-terminal of G, i.e., start symbol. A is a finite collection of attribute variables. Each variable has a fixed domain chosen from D. An attribute expression is one of the following:

(a) a constant attribute or

(b) an attribute variable or

(c) a function application $f(e_1,\ldots,e_n)$ where $e_1,\ldots,e_n$ are attribute expressions and f is an appropriate (partial) function chosen from D.

Let X be any symbol in V and let X have k attribute positions, whose domains are $D_1,\ldots,D_k$ respectively. If $a_1,\ldots,a_k$ are attributes in the domains $D_1,\ldots,D_k$ respectively, then $<X+a_1+\ldots+a_k>$ is an attributed symbol. Each + stands for either ↓ or ↑, prefixing an inherited or synthesized position as the case may be.

If $e_1,\ldots,e_k$ are attribute expressions whose ranges are included in $D_1,\ldots,D_k$ then $<X+e_1+\ldots+e_k>$ is an attributed symbol form.

P is a finite set of production rule forms each of the form:

$$F_0 \rightarrow F_1 \ldots F_m$$

$F_0, F_1, \ldots, F_m$ are attributed symbol forms, F_0 being non-terminal form.

The language generated by EAG is defined as follows. Let $F_0 \rightarrow F_1 \ldots F_m$ be a rule. Take an attribute variable x which occurs in this rule, select any attribute a for x in the domain of x and systematically substitute a for x throughout the rule. Repeat such substitution until no variable remains. Then evaluate all the attribute expressions provided all of them have defined values. This yields a production rule:

$$A \rightarrow A_1 \ldots A_m$$

$A_1 \ldots A_m$ is the direct production of attributed non-terminal A. A production of A is either

(a) a direct production of A or

(b) the sequence of attributed symbols obtained by replacing, in some production of A, some attributed non-terminal A' by a direct production of A'.

A terminal production of A is a production of A which consists entirely of attributed terminals. A sentence of EAG is a terminal production of S. The language generated by EAG is the set of such sentences.

The following examples describe some EAG's along with their equivalent attribute grammars.

Example 1: Assignment statement in an Algol 68-like language

The left hand side of each assignment must be an identifier of the mode **ref(MODE)**, where **MODE** is the mode of the right hand side. Each identifier must be declared and its mode is determined by its declaration. **ENV** is used for the set of declarations in the following rules. The EAG is given by the rules:

```
(1) <Assignment ↓ENV> →
    <Identifier ↓ENV ↑ref(MODE)> := <Expression ↓ENV ↑MODE>
(2) <Identifier↓ENV ↑ENV[NAME]> → <Name ↑NAME>
```

Equivalently, an attribute grammar can be given by:

```
(1) <Assignment ↓ENV> →
    <Identifier ↓ENV1 ↑MODE1> := <Expression ↓ENV2 ↑MODE2>
         evaluate ENV1 ← ENV
         evaluate ENV2 ← ENV
         where MODE1 = ref(MODE2)
(2) <Identifier ↓ENV ↑MODE> → <Name ↑NAME>
         evaluate MODE ← ENV[NAME]
```

Example 2: Declaration and use of variables

Declaration and use of variables in a simple language can be modeled by the following EAG. The constraints are that every variable used must be declared and no variable declared twice.

EAG rules:

```
(1) <Program ↑C1 and C2> →
    <Declarations ↑SET ↑C1> <Statements ↓SET ↑C2>
(2) <Declarations ↑{NAME} ↑true> → dec <Var ↑NAME>
(3) <Declarations ↑SET∪{NAME} ↑C and NAME∉SET> →
    <Declarations ↑SET ↑C> dec <Var ↑NAME>
(4) <Statements ↓SET ↑NAME∈SET> → use <Var ↑NAME>
(5) <Statements ↓SET ↑C and NAME∈SET> →
```

```
    <Statements ↓SET ↑C> use <Var↑NAME>
```

The following is an equivalent AG:

```
(1) <Program ↑CORRECT> →
    <Declarations ↑SET1 ↑C1> <Statements ↓SET2 ↑C2>
          evaluate CORRECT ← C1 and C2
          evaluate SET2 ← SET1
(2) <Declarations ↑SET ↑C> → dec <Var ↑NAME>
          evaluate SET ← {NAME}
          evaluate C ← true
(3) <Declarations ↑SET1 ↑C1> →
    <Declarations ↑SET2 ↑C2> dec <Var ↑NAME>
          evaluate SET1 ← SET2∪{NAME}
          evaluate C1 ← C2 and NAME∉SET2
(4) <Statements ↓SET ↑C> → use <Var ↑NAME>
          evaluate C ← NAME∈SET
(5) <Statements ↓SET1 ↑C1> →
    <Statements ↓SET2 ↑C2> use <Var ↑NAME>
          evaluate SET2 ← SET1
          evaluate C1 ← C2 and NAME∈SET1
```

It is seen in the above examples that EAG's have equivalent AG's. In fact, this is true for all extended attribute grammars, and an equivalent AG can be generated automatically [WaMa 83].

The purpose of including EAG's in this thesis is that they provide a better means in terms of parsing discussions, and in the remainder of this chapter AG and EAG will be used interchangeably.

6.2 Attribute grammars and LRRL parsing

When a context-free grammar, in particular an LRRL(k) grammar is augmented with attributes, the following questions must be addressed.

1. How the attributes will be evaluated and in what order.
2. If the evaluation of attributes to be meshed with a particular parsing method (i.e., on the fly evaluation of attributes) then what kind of attribute propagation is harmonious with the parsing method. How the parser is to be augmented to compute as many attributes as possible during the parsing phase.
3. How one takes advantage of attributes in parsing, i.e., attribute-directed parsing.
4. How a parser-evaluator can cope with such phenomena as local semantic ambiguity, e.g., lexical ambiguity, multiple word meaning in natural languages or overloaded symbols in Ada-like languages.

Much of work has been directed towards (1) and (2), e.g., [LeRS 74], [JaWa 75], [Boch 76], [Watt 77], [PoJa 78], [Kast 80] and [Pohl 83] among others. Little work has been done on (3) [Watt 80] and [JoMa 80], and work reported on (4) in the attribute grammarian circles is very little, though natural language processing experts have shown a greater interest in this area.

6.2.1 Methods for evaluation of attributes

The semantic rules given in attribute grammars are intended to evaluate the attributes of each node in the derivation tree of a sentence. An attribute grammar is said to be well defined if the attributes associated with nonterminals at any node of any derivation tree can be evaluated using the semantic rules for the productions which make up the tree. Each semantic rule of a production indicates a dependency among the attribute occurrences of a production, that is if $a=f(b_1,...,b_m)$, then a is dependent on $b_1,...,b_m$, and must be evaluated after $b_1,...,b_m$ are evaluated. This order of evaluation is represented by a directed graph. A dependency graph for a production consists of a node for each attribute occurrence in the production and an arc from a to b iff evaluation of b depends on a. A dependency network of a derivation tree is obtained by piecing together the dependency graphs associated with individual productions. An attribute grammar is well defined iff no dependency network of a derivation tree of the grammar has an oriented cycle [Knut 68]. Circularity implies that there is no order in which all the attributes can be evaluated. Jazayeri [JaOR 75] has shown that the problem of deciding for non-circularity condition intrinsically has exponential complexity. In the remainder of this thesis it is assumed that attribute grammars under consideration are well defined (non-circular).

More than a decade of research has produced a variety of attribute evaluators for non-circular AG's. Given an attribute grammar for a language, an evaluator is an algorithm which can evaluate all the attributes of any sentence in the language. If the attribute grammar defines the language in terms of another language, the evaluator is a translator (or a compiler if the target language is some machine language). An evaluator-generator is an algorithm that given an attribute grammar as input, produces an evaluator for it. In the words of Jazayeri [JaPo 81], an evaluator-generator theoretically may be considered to be a complete compiler writing system, although in practice it would probably be a part of it. A number of research works have been directed in the area of complier writing systems, notably Helsinki HLP system [RSST 78] and Karlsruhe GAG system [KaHZ 82]. Presently, the latter system is available as a commercial product.

Essentially the evaluators may be divided into three categories:

1. Tree walk evaluators.
2. 'On the fly' evaluators.
3. DAG evaluators.

6.2.1.1 Tree walk evaluators

Several classes of attribute evaluators are based on a number of passes over the derivation tree. These evaluators assume that a syntax tree has been created and labeled with inherited attribute of the start symbol and synthesized attributes of the terminal symbols. The evaluator traverses the syntax tree in some order, until all the attributes are evaluated. Among this class of evaluators are m-pass and 1-pass left to right evaluators of Bochmann [Boch 76]. m-pass evaluators use m depth first, left to right traverse of the syntax tree to evaluate the attributes. Right to left evaluators can be defined similarly.

An attribute grammar that allows evaluation of attributes to be carried out in one left to right pass over the syntax tree are termed L-attributed grammars. Basically in an L-attributed grammar, for each production the following conditions are satisfied:

(i) Each inherited attribute of a right hand side symbol depends only on the inherited attributes of the left hand side symbol and arbitrary attributes of the symbols to the left of the symbol.

(ii) Each synthesized attribute of the LHS symbol depends only on the inherited attributes of that symbol and arbitrary attributes of the RHS symbols.

Clearly such an attribute dependency allows evaluation to be carried out in one left to right depth first traversal.

The m-alternating pass evaluators defined by Jazayeri and Walter [JaWa 75] employ m alternating left-right and right-left passes over the syntax tree to evaluate all the attributes.

S-attributed grammars, which can be considered as a subclass of L-attributed grammars, have only synthesized attributes. Thus, a one pass over the syntax tree in a bottom-up fashion suffices to evaluate all the attributes in these grammars.

The classes of tree-walk evaluators discussed above are known as the pass-oriented evaluators. In all of these evaluators the order of evaluation is fixed by the constructor, independently of any particular syntax tree. Jazayeri's alternating method can be considered to be the most general strategy among the pass-oriented evaluators.

By contrast, there are some evaluation methods in which the evaluation order is dependent on the particular syntax tree. These are general enough to accept any non-circular attribute grammar. Ordered attribute grammars [Kast 80] may be regarded as the most general subclass of well-defined attribute grammars that can be evaluated in this fashion. An attributed grammar is ordered if for each symbol, a partial order over its associated attributes can be given such that in any context of the symbol, the attributes are evaluated in an order which includes that partial order. Kastens [Kast 80] shows that it is decidable whether an attribute grammar is ordered. The time needed for that decision and for the computation of the order depends polynomially on the size of grammar. It can be shown that for pass-oriented grammars such an order exists. The evaluator computes visit-sequences for each node in the syntax tree from the attribute dependencies given by an ordered attribute grammar. They describe the control flow of an algorithm for attribute evaluation. The computation of visit-sequences may be combined with a parser. This method is used in GAG: the Karlsruhe compiler generator [KaHZ 82].

6.2.1.2 On the fly evaluators

Most often computational linguists are satisfied with building a syntax tree for the input sentence and having it decorated with features or attributes afterwards. On the other hand many compiler writers, specially one-pass compiler producers have resisted the idea of constructing parse trees. Parse trees for programs are usually large, and each node of a tree usually has several attributes. Therefore, a serious problem in evaluation is storage space. On the fly evaluators that evaluate attributes in conjunction with a parse rather than after it, solve the storage problem by throwing away the attribute values after they are no longer needed. Only the attribute values of the left context that are needed in the evaluation are kept on a stack. One could observe that if the underlying context-free grammar in an L-attributed grammar is LL(k), then attribute evaluation can be meshed with LL-parsing producing a subclass of L-attributed grammars, namely LL(k) L-attributed grammars. Similarly, attribute evaluation in an S-attributed grammar can be carried out in conjunction with LR(k) parsing if the underlying grammar is LR(k). This latter subclass of S-attributed grammars are called LR(k) S-attributed grammars. These two classes of attribute grammars are considered to be good simple models of one-pass syntax-directed translators.

6.2.1.3 DAG evaluators

As was seen, LR(k) parsing is suitable for S-attributed grammars, and inherited attributes create problem with LR(k) parsing as they cannot be evaluated on the fly as parse of the input sentence proceeds. Even in less complex affix grammars, inherited affixes cause problem [Watt 77]. To cope with this problem, DAG evaluators were proposed in [LeRS 74] and developed by O.L. Madsen [Mads 80].

The basic idea in DAG evaluators is that they do not store the parse tree at all, instead known attributes, i.e., those that can be evaluated during the parse, are computed on the fly. For unknown attributes an expression DAG is created during the parsing. This has at most one node for each attribute of a node of the parse tree. Let a be an attribute of some node of the parse tree. Then there is an expression $ex(a_1,\ldots,a_n)$ giving the value of a in terms of the attributes of the other nodes. Node a in the DAG will be labeled with ex and there will be an ordered sequence of arcs from a to $a_1,\ldots,a_n$. This graph will be acyclic for any parse tree, since the attributed grammar is assumed to be non-circular. After the parse is finished unknown attributes can be evaluated in one-pass depth first traversal of the DAG.

Recently, Pohlmann [Pohl 83] has presented a method for LR parsing of affix grammars that handles inherited affixes. Essentially Pohlmann's method establishes a pointer to an entry on the parse stack where an inherited affix of current node can be found. Since affix grammars are a special case of the attribute grammars which only involve copy rules[1], this mechanism works for L-affixed grammars. However, the method cannot be extended to L-attributed grammars in general.

Watt [Watt 77] also introduced a method for LR parsing of affix grammars which involves a mapping of AG rules into context-free productions, the so-called head grammar, for the given affix grammar. It can be shown that Watt's method is weaker than Pohlmann's. Even an affix grammar built around an LR(k) skeleton not necessarily transmits the LR(k) property to its head grammar.

[1] One could consider affix grammars as extended attribute grammars in which every attribute expression is either a constant attribute or an attribute variable.

6.2.2 Attribute flow and evaluation in LRRL grammars:

If an arbitrary flow of attributes is allowed in an LRRL grammar, then for the evaluation of attributes probably one needs to use a post-parsing evaluator. So, the problem in this case is not any different from the evaluation of attributes in general context-free grammars. However, it is believed that a combination of Kastens' ordered attributed grammars [Kast 80] and storage management method of Jazayeri [JaPo 81] will produce a more general and space efficient evaluators.

But, a main thrust of LRRL parsers like any other deterministic parser is to avoid building of parse trees. Therefore, it is of interest to investigate what sort of attributes can be evaluated in conjunction with LRRL parsing. Obviously, S-attributed grammars, i.e., those with only synthesized attributes, do not pose any difficulty in LRRL parsing. So the question is only concerned with inherited attributes.

Knuth [Knut 68] rightly observed that "the synthesized attributes alone are in principle sufficient to define any function of a derivation tree". However, "the importance of inherited attributes is that they arise naturally in practice". There are languages for which restriction of attributes to synthesized ones "leads to a very awkward and unnatural definition of semantics". Therefore, it is attempted here to extend the on the fly evaluations, at least in a limited form, to some inherited attributes during LRRL parsing. The followings are some possibilities:

6.2.2.1 Scheme 1: A subclass of L-attributed grammars

The scheme described below handles a special class of L-attributed grammars in which the inherited attributes are classified as 'known' during the parsing process. First the scheme is considered in conjunction with LR parsing. Then, modification of the method which makes it applicable to LRRL parsing is discussed. In the following, it will be assumed that a grammar is given in the extended attributed form. In this way, the issue of explicit constraints can be avoided.

From the remarks of Knuth, it is rather obvious that in an L-attributed grammar, the inherited attributes of a symbol are functions of the synthesized attributes of its left context and the inherited attributes of the start symbol (if it is allowed to carry one). Thus, on entry to a state of LR parser, say s with the basis $\{I_i:(A_i \to \alpha_i.B_i\beta_i,L_i)|i=1,...,m\}$, the inherited attributes of each B_i are potentially available on the stack.

If one assumes that all the attributes of α_i are attached to its symbols on the parse stack (or equivalently, they may reside on a separate semantic stack), and the inherited attributes of A_i are known (later it will be described where these might be). Then, under the following condition the inherited attributes of each

B_i can be computed. The required condition is that no $B_i = B_j$ for $i \neq j$. Or, given that $B_{i}, \ldots, B_{i} = B$, IA(B) can be described by a unique expression in terms of the attributes of A_i 's and α_i 's.

Furthermore, suppose that a condition is imposed on the grammar such that when the closure of the basis of s is obtained, for every symbol C that appears in an item like $(C \rightarrow .\gamma, L)$, its inherited attributes can be given as a unique closed expression in terms of the attributes of A_i 's and α_i 's, $i=1,\ldots,m$. (Note that C may be derived from two different B_i 's and can be predicted from the same B_i on different derivation paths). Then, by a simple deduction one can prove the availability of the inherited attributes of any symbol during LR parsing.

In the presence of left recursive rules like $C \rightarrow C\gamma$, the above condition requires that if $\underline{f}$ is a function that computes the inherited attributes of the right hand side C, i.e., the attributed rule is $<C\downarrow\underline{a}\uparrow\ldots> \rightarrow <C\downarrow\underline{f}(\underline{a})\uparrow\ldots>$, then $\underline{f}$ must be an identity function or copy rule.

Notice that in the state s, for all i and j, α_i in the basis is a suffix of α_j or vice versa. Therefore, one only needs to access the attributes of longest α_i and the inherited attributes of A_i, $i=1,\ldots,m$. Since at the entry into the state s, no actual node would be created for any of the A_i 's, the set of inherited attributes of A_i 's may be attached to state s on the state stack. These and the attributes of α_i's will be called the **defining** or **known** attributes at entry to the state s.

In order to use the above scheme in LRRL parsing, one of two possible modifications could be considered. The simpler one is that on entry to a state s with conflicting items, the lookahead symbols do not inherit any attribute value from the questionable phrases or their dominators. In the more general scheme, the inherited attributes of lookaheads may depend on the attributes of the questionable phrases, but it is required that they are given as unique expressions in terms of the known attributes of the state s.

The following is a simple example that illustrates this scheme.

Example:

Consider the LRRL(2) grammar G_1 examined in the earlier chapters.

G_1:

(0) S' → S

(1) S → d

(2) S → A S B

(3) A → a

(4) S → a S

(5) B → b

Suppose for each d in a sentence of $L(G_1)$, one wishes to compute the number of d's seen so far when scanning from the left end of the sentence. The following extended attribute grammar is suitable for this purpose. In the EAG, every construct that derives a "d" has an inherited attribute that gives the number of d's seen before parsing of the construct. A synthesized attribute provides the number of the d's after parse of the construct.

EAG:

(0) <S' ↓0 ↑No> → <S ↓0 ↑No>

(1) <S ↓No ↑No+1> → d

(2) <S ↓No ↑No2> → <A ↓No ↑No1> <S ↓No1 ↑No2> B

(3) <A ↓No ↑No> → a

(4) <A ↓No ↑No1> → a <S ↓No ↑No1>

(5) B → b

The state 4 of the CFSM for G_1 (Diag. 3.2 of Section 3.2.5) is depicted in Figure 6.1. One may observe that on entry to this state, the inherited attribute of the LHS symbol in the concealed items is available from the previous states. Thus, A↓No is classified as a known attribute of the state 4. Furthermore, the inherited attributes of lookaheads can be expressed uniquely as the identity functions of this attribute. Therefore, it suffices to consider two auxiliary attributed rules:

(Aux 1) <SUBGOAL-RED(3) ↓No> → <S ↓No ↑No1> B

(Aux 2) <SUBGOAL-SHIFT ↓No> → <S ↓No ↑No1> <S ↓No1 ↑No2>

where SUBGOAL-RED(3)↓No=SUBGOAL-SHIFT↓No is equal to A↓No, i.e., the known attribute of the state 4, at each entry to this state. Note that the inherited attributes of symbols on the LHS of the items obtained by the closure operation are also available and are equal to the above attribute. Therefore, every attributed can be computed during the LRRL parsing of a sentence in L(G_1).

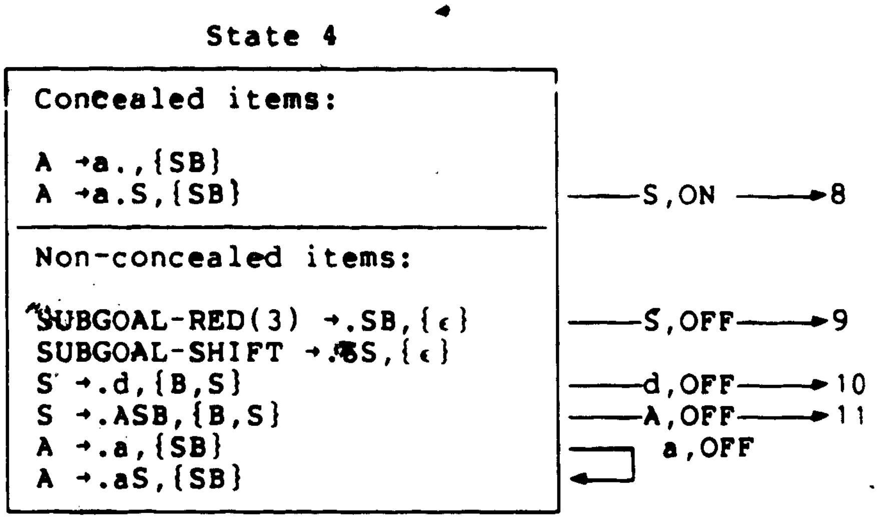


Fig. 6.1
State 4 of the CFSM for G_1

To establish that an inherited attribute is uniquely expressible in terms of the known attributes of a state, a number of expression DAG's with the attribute in question as their source node could be created. The sink node of ~~these~~ DAG's are the known attributes in the state. The attribute could be classified as known (i.e., it could be computed on the fly) if all the DAG's express the same function. Note that use of the DAG's in this approach is different from their use in DAG evaluators. Here, the DAG's are created in CFSM construction time in order to determine whether an attribute is computable on the fly. In DAG evaluators, the

DAG's are created at parsing time and they are used for the evaluation of unknown attributes by a post-parse mechanism.

The approach outlined here can be considered as a generalization of Pohlmann's method [Pohl 83] for handling of affixes in LR parsing. The scheme is also influenced by [Watt 80], and basically the intuitive methods that are used in hand-written semantic routines for compiler construction in which formal attributes are not used.

The method does not recognize the maximal set of known of attributes which theoretically is possible. With the exception of copy rules (i.e., identity functions), the DAG approach does not use any semantics of the functions. A maximal set can be identified if one exploits the properties of the evaluation functions, such as their commutativity, or the commutativity of compositions of the functions. Nevertheless, the method solves some subtle problems regarding inherited attributes in bottom-up parsing, without introducing global attributes which are much criticized by Räihä and other designers of Helsinki system. For example, by using this scheme symbol table information in programming languages, and discourse information in natural language can be passed to lower constructs before creating their parent nodes in a bottom-up parser.

6.2.2.2 Scheme 2: Dependency on the attributes of right siblings

Suppose an inherited attribute of a node A depends on the synthesized attributes of its right and left siblings in the higher construct, and the value of this attribute is not propagated down to the subtree of A beyond its immediate successors. Naturally, the attributes of the left siblings of A will be available on the parse stack. If the reduction of A is in question in an inadequate state and its right siblings are included in the corresponding lookahead set, then after the right context is parsed by the LRRL algorithm, the attributes of the right siblings will also become available on the buffer, and one can use them in the computation of the inherited attributes of A.

Otherwise, in the absence of local syntactic ambiguity, still one might enforce the "wait and see policy" voluntarily to obtain the parse of the right siblings and thus the value of their synthesized attributes.

6.2.2.3 Scheme 3: Dependency on the attributes of fully reduced right contexts

A general form of the scheme 2 can be also considered. In this scheme, the inherited attributes of a node A may depend in a unique way on the attributes of its k-symbol fully reduced context. Here again, one could proceed to parse the lookaheads before reducing the current construct and evaluating its attributes. For asserting that the attributes of A can be expressed uniquely in terms of the attributes of its lookaheads, the DAG technique may be used here too.

The applications of these schemes will be shown in the next sections. In brief, Section 6.2.2 has proposed two attribute evaluators. One is a post-parse method based on the combination of Kastens' ordered grammars and Jazayeri's space management method. Kastens' method by computing visit sequences for the nodes at the parse time will contribute to the time efficiency of the evaluator. While, Jazayeri's method in which lifetime of the attributes are analyzed, (much like live variable analysis in optimizing compilers), will benefit the evaluator in terms of storage use by discarding unwanted attribute values.

The second is an on the fly evaluator which computes the known attributes during the parsing phase. Obviously, in a general attributed grammar, if some of the inherited attributes cannot be evaluated by this technique alone, then one may resort to partial DAG evaluation.

6.3 Attribute-directed parsing

Sometimes the context-free description of a language is such that either the context-free grammar is ambiguous, or some local conflict cannot be resolved by means of limited syntactic context. However, the semantic information present at the time may resolve the syntactic ambiguity. Consider a fragment of the context~~-free~~ grammar given in the Pascal report [JeWi 74], for the syntax of ~~Pascal~~.

Actual parameter → Expression

Actual Parameter → Variable

Expression → Simple expression

Simple expression → Term

Term → Factor

Factor → Variable

When parsing the actual parameters, a variable can be reduced directly to an actual parameter, or can be reduced first to a factor, term, simple expression and then expression and finally to actual parameter. Therefore, the context-free grammar is ambiguous. In fact, reducing a VAR parameter to an expression is incorrect, while VALUE parameters first should be reduced to expressions.

An unambiguous attributed grammar can be given for the above situation by the following:

```
<Actual parameter ↓ENV ↓PARM> → <Expression ↓ENV ↑TYPE>
     where is-value(PARM) and
           TYPE=type(PARM)
<Actual parameter ↓ENV ↓PARM> → <Variable ↓ENV ↑TYPE>
     where is-var(PARM) and
           TYPE=type(PARM)
<Expression ↓ENV ↑TYPE> → <Simple expression ↓ENV ↑TYPE>
```

<Term ↓ENV ↑TYPE> → <Factor ↓ENV ↑TYPE>

<Factor ↓ENV ↑TYPE> → <Variable ↓ENV ↑TYPE>

Similar situations also arise in the context-free syntax of Ada [Ledg 81], where the non-terminal NAME is most troublesome symbol [Weth 81].

The LRRL parsing algorithms could be modified to let the attributes influence the parsing decisions in such situations. In an inadequate state of the CFSM say s with items $(A_1 \rightarrow \alpha_1.,L_1)$ and $(A_2 \rightarrow \alpha_2.,L_2)$ ~~and~~ known attributes $\{a_1,...,a_n\}$, one would allow these attributes to decide on which production to be reduced. The set of known attributes may include the attributes of right context. Thus even if $L_1 \cap L_2 \neq \emptyset$, i.e., when the underlying context-free grammar is not LRRL (the reduced right context syntactically does not indicate a resolution), one would parse head the right context if its attributes could resolve the conflict in the state s. Such a scheme also was indicated in Example III of Section 2.1, in which the conflict is between two items: $(A_1 \rightarrow \alpha.\beta,\{\epsilon\})$ and $(A_2 \rightarrow \alpha.,\{\beta\})$. Though the conflict cannot be resolved on a syntactic basis, it is resolvable on the basis of the attribute values of the reduced lookahead β.

In order to incorporate the attributes in parsing decisions, one needs to introduce the notion of predicates on the known attributes of the states of the CFSM. Correspondingly, the entries of the parse table where decisions are based on the attributes, instead of being a single action pattern will have a SELECT clause in the form of:

$$\mathrm{TABLE}(state,symbol,\{flag\}) = \left[\begin{array}{l} \mathrm{SELECT} \\ \quad P_1(a_1,\ldots,a_n) \rightarrow \langle \mathrm{Action}\ 1\rangle \\ \quad \ldots \\ \quad \ldots \\ \quad P_{m-1}(a_1,\ldots,a_n) \rightarrow \langle \mathrm{Action}\ m-1\rangle \\ \quad \mathrm{Otherwise} \rightarrow \langle \mathrm{Action}\ m\rangle \\ \mathrm{end\ SELECT}\end{array}\right.$$

where $P_1,\ldots,P_{m-1}$ are predicates on the known attributes of the state, and <Action i> is the action pattern that the parser proceeds according to it when the predicate P_i is true. Obviously, either one should have the predicates mutually exclusive, or the predicates can be ordered in a way such that the parser takes the action corresponding to the first predicate which is true.

The attribute-directed parsing in conjunction with LRRL grammars seems to be the most general scheme that formally has been proposed. Because in LRRL parsing the parse of the right context is possible, makes this scheme superior to other methods such as rule splitting [Watt 80] suggested in conjunction with LR parsing. The method is also readily applicable in error correction situations.

Notice that the above scheme is capable of handling unambiguous attributed grammars that their underlying context-free grammar is not LRRL or even not unambiguous.

Attribute-directed parsing clearly demonstrates the superiority of the on the fly evaluators in practical situations. Without such evaluation methods deterministic attributed parsing would not be possible.

6.4 Semantic disambiguation with the use of wait and see policy

In some languages it happens that some of the terminal vocabulary do not carry a single value for one or more of their synthesized attributes. Similarly, an attribute of a nonterminal may be such that it cannot be computed uniquely in terms of its defining attributes, that is when the relation f from defining attributes to applied attributes is not a function but a one-to-many relation. However, it may be the case that a constraint involving the syntax or the attributes of right context will determine the value of the attribute uniquely. In such cases, if one wishes not to carry around or propagate the multiple values or postpone the attribute evaluation to a post-parsing process, then one could simply impose the parsing of lookahead strings voluntarily, though syntactically there may be no need to do so. More formally, in a state s of the CFSM with only a single reduction item $(A \rightarrow \alpha.,L)$, if attributes of A cannot be determined uniquely, then one proceeds to parse the lookahead strings, i.e., items of the form SUBGOAL-RED(p) $\rightarrow.\gamma,\{\epsilon\}$; $\gamma \in L$ would be added to the state, provided that lookaheads can be parsed in LRRL manner. To accommodate such a scheme in an LRRL parser, a predicate P will be introduced in the state s and the corresponding parse table entry will be of the form:

```
                           ┌
                           │SELECT
TABLE(state,symbol,{flag})=│ P  → <reduce A →α >
                           │ Otherwise  → <goto s'>
                           │end SELECT
                           └
```

where predicate P determines if attributes of A are uniquely valued. s' will be the successor state of s under the symbol when reduction is delayed.

The followings are few examples in which such a method could be employed.

Example 1:

Consider the two sentences:

(1) The fish is swimming in the tank.

(2) The fish are swimming in the tank.

To assign a singular/plural feature or attribute to the noun fish, one needs to parse the verb phrase before reduction of the noun phrase in a deterministic parser.

Example 2:

An important application of the method arises in conjunction with the semantic disambiguation of overloaded symbols in polymorphic programming languages. Ada allows overloading of subprograms and enumeration literals [Ichb 79]. When an overloaded identifier appears at a given point of the text, the identifier itself does not provide enough information to determine a unique meaning. In such cases contextual information must be used to select the unique meaning. Thus, in a single module two subprograms with the same designator or identifier may be defined provided that parameters and result types provide enough information for unique identification. For example, the two procedures:

```
    (1) Procedure P(A: in integer; B: out boolean)
             is
                begin
                 ...
                end P;
```

and

```
    (2) Procedure P(A: in integer; B: out real)
             is
                begin
                 ...
                end P;
```

can be defined in the same scope. At the time of procedure call, the only thing that differentiates the procedure (1) from (2) is the type of actual parameters. In LRRL parsing of Ada, in order to obtain the unique meaning of P, one postpones the reduction of a rule like: **Procname → Identifier**, so that its right context is parsed first. The right context, i.e., the actual parameter list provides enough information to disambiguate the meaning of P.

It is interesting to note that since the introduction of Ada, a number of papers appeared on the subject of overloaded resolution algorithms. The algorithm originally offered by the Ada language design team [Ichb 79] suggested an indefinite number of passes over the syntax tree. Subsequent algorithms reduced the number of passes to four, three and two. Finally, Baker [Bake 82] suggested a one-pass algorithm over the syntax tree for overload resolution in Ada.

All the above solutions assumed existence of a parse tree or at least a partial one in the form of expression tree. However, the one-pass deterministic algorithm suggested here does not rely on the existence of a parse tree. Moreover, unlike the previous algorithms, it rarely would propagate multiple values for attributes because of the advantage of the parsing of the lookaheads.

6.5 Syntactic errors repair

Basically in compiler writting, two methods are used for repairing of syntactic errors. One is by introducing error productions in the grammar that specifies the syntax of the language. Usually, the error productions are added to the grammar to guarantee the correct treatment of a rather common error. Specially, when other correction methods fail, because errors are discovered too late. An example of this kind was considered in Section 2.1. A considerable care is required in this method to ensure that the grammar is still parsable after the addition of error productions.

In the second method, which is known as least-distance or least-cost technique, one tries by a series of insertions, deletions and replacements of terminal symbols to correct the input sentence. Since there is usually more than one possible repair for any error, a cost is associated with any repair operation such as insertion. A cost function (in the simplest case accumulated cost of single operations) guides the algorithm in chosing the least-cost repair. A number of error-correction algorithms are defined in this way, e.g., [PeDe 78], [Tai 78], [GrHJ 79], [FiMQ 80], [MaFi 82] and [SiSo 83]. Most of these algorithms are incorporated into the LR or LL parsers.

It could be well argued that LRRL parsing provides a more robust technique with both of the above schemes. As it was noted in Section 2.1 a grammar augmented with error productions not always is parsable by an LR method. While, LRRL methods because of their generality give more freedom to the designer of a error-corrector parser in adding error productions.

The corrections in the least-cost methods based on LL or LR parsers are usually local. In chosing the best repair only the prefix of sentence, i.e., the left context, and the next terminal input that signals the error are taken into consideration. Tai [Tai 78], Mauney and Fischer [MaFi 82] consider k terminal symbols of the remainder in their algorithms, but still their methods are considered locally least-cost. Pennello and DeRemer [PeDe 78] discuss a forward move algorithm for LR error recovery in which corrections are made by considering some nonterminal lookaheads. However, in their method parsing of the right context is independent of the left context. Also, the action of such algorithms may be limited by the presence of a second error in the input.

The globally least-cost methods are very expensive and require $O(n^3)$ time [AhPe 72]. Thus generally, use of some right context which allows regionally least-cost repairs is very desirable. Such a method can be considered as a middle ground between locally and globally least-cost methods. However, implementation of forward moves and restarting of the parser in LR methods have not been without problems and difficulties [SeSo 83]. In the following, it will be shown that the LRRL methods can be used as regionally least-cost error-corrector parsers with a great advantage. Similar to [FiMQ 80], only insert-correctable errors will be illustrated. Deletions and replacements (i.e., combinations of a deletion followed by an insertion) can be handled in a like manner.

In an LRRL parser, a k-symbol reduced right context can be used to contribute in making the best repair. One may observe that the problem of forward moves for LR grammar is readily solved by the LRRL technique. In fact, the method is such that the suggested correction in some cases is unique and behaves as good as a global correction method. For the purpose of illustration, consider the two Pascal program fragments:

(1) ... ; *A:=B C[I,J,...]+D;* ...

(2) ... ; *A:=B C[I,J,...]:=D;* ...

An LR-based parser upon reading C would indicate an error. The appropriate repairs are insertion of an operator, e.g., a plus in the first fragment and a semi-colon in the second. However, a locally least-cost method would repair both fragments in the same way, resulting a cascade of new errors in one of the fragments. But, an LRRL parser would delay the correction and parse ahead the reduced context (i.e., either an expression or a statement) and would choose the best repair. More formally and without loss of generality, suppose that there are only two items in a state s that have terminal symbols next to the right of dot, i.e., $[A \rightarrow \alpha_1.a\alpha_2,L_1]$ and $[B \rightarrow \beta_1.b\beta_2,L_2]$. The legal moves of the parser in this state includes shift operation on terminals 'a' and 'b'. If one desires to repair erroneous input at this state by inserting an 'a' or a 'b', then one can add auxiliary items $\{[\text{SUBGOAL(Insert a)} \rightarrow .\gamma, \{\epsilon\}] \mid \gamma \in (\{\alpha_2\} \underset{k}{+} L_1)\}$ and $\{[\text{SUBGOAL(Insert b)} \rightarrow .\sigma, \{\epsilon\}] \mid \sigma \in (\{\beta_2\} \underset{k}{+} L_2)\}$ to the state s. If $(\{\alpha_2\} \underset{k}{+} L_1) \cap (\{\beta_2 \underset{k}{+} L_2) = \emptyset$ and the lookaheads are parsable by the LRRL(k) method, then this policy would lead to a unique repair. Otherwise, one can associate a cost factor with each auxiliary item, and when two auxiliary items exhibit a conflict the tie could be broken in favour of the item with the least cost. Notice that introduction of subgoal symbols such as SUBGOAL(Insert a) does not add a new mechanism to the LRRL parsing. In fact, SUBGOAL(Insert a) can be considered as SUBGOAL-RED(Nonterminal-a $\rightarrow\epsilon$) where Nonterminal-a $\rightarrow\epsilon$ is the error production in Aho and Peterson model [AhPe 72] indicating a missing terminal symbol 'a'.

The scheme described above considers only the syntactic structure of the reduced lookaheads in choosing the best repair. Obviously the method can be used recursively to repair additional errors that may lie ahead in the lookaheads. However, there are cases in which a good repair cannot be made without using the semantics of the lookaheads. In such cases, one needs to employ an attributed LRRL parser, or at least combine error correction with attribute-directed parsing as discussed in Section 6.3. Consider the following two fragments from Pascal programs.

(1) *A:=B F(X);* (2) *A:=B P(X);*

Suppose F is declared as a function, but P as a procedure.

Thus, in the first fragment insertion of an operator is a suitable repair, while in the second fragment insertion of a semi-colon is more appropriate. However, an LRRL parser cannot differentiate the procedure call from the function application in the presence of error, unless it refers to the type of the identifiers. Such a scheme in which attributes resolve syntactic ambiguities was discussed in Section 6.3.

6.6 Transformations in the Marcus parser

So far, this thesis has ignored the transformational capability of the Marcus parser, and nothing has been mentioned about the way the transformational rules will be handled in LRRL parsing. Fortunately, transformations in the Marcus parser are very limited. In fact the parser carries out a *surface structure* analysis, and the transformational rules either involve traces or interchanging of constructs at a rule level.

Basically, the trace theory in transformational grammars deals with displacement (*movement*) of constructs in an utterance. At the point where a construct is missing (a *gap*), a trace which is a non-terminal node that derives null string is added. When the actual displaced construct (a *filler*) is found, a pointer to that construct is placed in the trace. In this way, one obtains a *deep structure* of the sentence that can be used for semantic analysis.

Traces can be considered as null deriving nonterminals that have an attribute with pointer values, and thus they can be easily handled by an attributed LRRL parser. Marcus [Marc 80] handles the traces similarly with the use of special features named registers. A similar strategy may be employed to handle the slash categories in Gazdar-style context-free grammars [Gazd 82, ScPe 82] when they are parsed by an LRRL parser.

In addition to the traces that are used to handle long distance transformations such as wh-questions and topicalizations, the Marcus parser has explicit rules for inversion of short distance transformations. For example, an auxiliary-inversion rule is used to interchange the auxiliary and NP in yes/no questions. In a sentence like:

Has John scheduled the meeting?

after recognition of the auxiliary "has" and the noun phrase "John", the order of two constructs are changed (by attaching NP first) so that the auxiliary appears next to the main verb "scheduled".

In the Marcus parser such inversion takes place in the buffer. However in LRRL parsers normally they would be handled while the constructs are on the top of the parse stack. If one wishes to incorporate this inversion explicitly into the CFSM, it can be done by adding a new state with an ϵ-transition, i.e., new action pattern <interchange stack[i],stack[j]; goto s> will be added to the parse table, where s would be the successor state of the current state with an ϵ-transition.

6.7 Conclusion

This chapter has raised some of the issues regarding the attributes and attributed LRRL(k) parsing. It was shown that attributed LRRL(k) parsers could allow some non-S-attributed grammars to be parsed by this technique. The idea of semantic-directed parsing is extended to include the attributes of the right contexts. The attri[illegible] influenced decisions are embedded in the parse tabl[illegible] directs a parser. Applications of the wait and see p[illegible]n semantic disambiguation and syntactic error corrections are discussed. A scheme, which uses DAG's at the parser generation time, is proposed for on the fly evaluation of attributes that enables one to handle some inherited attributes during the parsing phase. However, there is a lot of room for the improvement of the scheme. By using the semantics of the evaluation functions and a rigorous use of fixpoint theory, one can enhance the scheme in a way that identifies a maximal set of known attributes that can be evaluated during the parse.

Though a complete algorithm for generation of attributed LRRL parsers is not presented here, one can implement such an algorithm in a more ambitious general language processing system that generates translators for different languages. Apart from the issues discussed in this chapter, other applications of attributed LRRL(k) parser such as backpatching in one-pass compilers for simple programming languages, and use of wait and see policy in programming languages editors could be studied.

CHAPTER 7

CONCLUSIONS

7.1 In summary...

This thesis has defined a family of classes of unambiguous context-free grammars that can be considered as non-canonical extensions of LR(k) grammars. The new family of grammars differ from the LR(k,t) grammars which were originally suggested by Knuth and subsequently developed by Szymanski. It is shown that the most general class in this family properly includes the LR(k,t) grammars. The LRRL(k) grammars provide a formal framework for the procedurally defined Marcus parser, and thus a capability for automatic generation of such parsers. Contrary to Berwick's conjecture, this research shows that Marcus-style parsers in principle cannot be characterized by LR(k,t) grammars. In contrast to Hunt's belief, it is shown that LR(k) parsing concepts could be generalized to deal with considerably complex lookaheads, while retaining the decidability of the membership in the generalized class.

The LRRL(k) grammars when augmented with attributes should have a significant impact on natural language processing, and the design and implementation of programming languages. It is hoped that this dissertation would help to establish parsing through the use of complex lookaheads as a viable technique.

7.2 Contributions of the research

The major contributions of this research may be listed as the followings:

(1) It provides a generalization of LR(k) parsing in the form of LRRL(k) parsers. LRRL(k) parsers employ non-terminal as well as terminal symbols in a generalized lookahead policy. The recursive parsability of the lookahead information by the method itself makes this method distinct from any other LR-based parser. Depending on the grammar, LRRL(k) parsers may postpone an unbounded number of parsing decisions, while in LR(k,t) parsers this number is limited to t.

(2) The theoretical issues concerning the LRRL(k) are studied and in particular it is shown that:

- The parsing algorithm is linear in the size of input sentence, and unlike LR-regular grammars no prescan of input sentence is required.

- Unlike LR-regular, FSPA(k) and LR(k,∞) grammars, the membership problem for LRRL(k) grammars (with a fixed k) is decidable.

In LR-regular grammars, properties are given in terms of the language of the lookaheads. Similarly, in a discussion by Hunt, certain undecidable conditions are defined on languages that lead to a belief that a generalization of LR(k) parsing, in which some arbitrary part of the remainder of a sentence is analyzed before making a parsing decision, must yield a grammar class with undecidable membership problem. In LRRL(k) parsers conditions are stated in terms of the grammar that generates these terminal segments. In fact, one may argue that dealing with properties of languages in general, and arbitrarily length lookahead terminal strings in particular (without a close consideration of their syntactic structures) often leads to undecidable problems. While retaining the structure of the lookaheads, one might consider 'almost any property' on their grammar short of unambiguity or any other condition that is equivalent to it. Consequently, a generalization of LRRL grammars, namely GLRRL grammars, was defined in this research. For a fixed value of parameter k, it was shown that this class of grammars is the largest proposed class that generalizes the concepts of LR(k) parsing while it still leaves the membership problem decidable.

The $LR(k,\infty)$ and FSPA(k) properties lead to undecidable situations because no restrictions are put on the forward move of the parser to the right of a questionable phrase. In LRRL(k) parsing, only the subtrees which are the descendants of the k fully reduced right context of a bypassed phrase are examined.

- The LRRL(k) property (except GLRRL(k) with bounded buffer) can be tested by a polynomial time algorithm in the size of input grammar.

(3) This research, apparently for the first time, embarks on the idea of recursive definitions of grammar classes. It is shown that such recursive definitions under certain conditions are valid and they do not lead into circular decision problems.

The notion of reduced context language of a grammar and its generating grammar are introduced to provide such definitions.

(4) The introduction of LRRL(k) grammars makes it possible to increase the power of deterministic precompiled table driven parsers to the cases where traditionally a backtracking method with non-finite buffering of input, or parallel processing of multiple paths is used.

(5) The type of the string set accepted by Marcus' parser is investigated, and it is shown this language can be recognized by a deterministic pushdown automaton. This confirms the determinism of the language parsed by the above parser, and justifies assuming of a context-free underlying

grammar in automatic generation of Marcus-type parsers.

In building a Marcus-style parser, one can assume either a context-free or a context-sensitive grammar (as a base grammar) which one feels is naturally suitable for describing the surface structures. However, if one chooses a context-sensitive grammar then one needs to make sure that it only generates a context-free language. It is proposed that for automatic generation a context-free grammar be used for describing the surface structures. The grammar could be augmented with syntactic features (e.g., person, tense, etc.) much like attributed grammars in compiler writing systems. An additional advantage with this scheme is that semantic attributes such as logical translations may also be added to the nodes without an extra effort.

(6) The LRRL(k) grammars provide a formalization of the Marcus parser. Thus the concepts in that parser can be given formally in the shape of a context-free base grammar.

The goal of this research in formalizing Marcus' parser is two fold. First, it provides a more viable basis for Marcus parsing in terms of the traditional parsing theory. Second, it supplies a tool for automatic generation of Marcus-style parsers. The easily obtained parsers facilitate investigating determinism hypotheses in natural languages.

(7) A few schemes for parsing attributed LRRL(k) grammars are suggested. It is shown that some inherited attributes

can be evaluated on the fly during a bottom-up parse. The applications of the attributed LRRL(k) parsing in such areas like semantic-directed parsing and semantic disambiguation are demonstrated.

(8) It is shown that some form of wait and see policy in parsing provides more flexibility in design and implementation of programming languages. It is argued that some features of programming languages are modified or sacrificed to make them conform to the traditional LR(k) or LL(k) parsers.

The LRRL(k) parsers can easily be used to handle many features of one-pass compilers in an efficient way. Among them are simple backpatchings (e.g., code generation for forward referencing goto's in simple languages like Basic) and resolution of overloaded symbols in Ada-like languages that traditionally are accommodated in such compilers by means of ad-hoc or expensive routines. An attributed LRRL(k) grammar could be used to induce such actions on the automatically generated compiler at no extra cost.

(9) The use of LRRL method as a syntactic error corrector parser is discussed. It is shown that both in cases where adding of error productions violate LR condition, and as a regionally least-cost error corrector, LRRL parsing provides a very robust technique.

7.3 Future research

It may be noted that an attempt was made here to cover nearly all the aspects of LRRL(k) grammars in one project. The reported research encompasses both the theoretical issues pertaining to LRRL(k) grammars, and the matters that arise in relation with their practical applications. However, there are two loose ends in this thesis that require further research. One is the type of languages for which GLRRL(k) grammars with unbounded buffer exist. The question was left as an open problem in Section 5.3.3. The other is concerned with the enhancements to the schemes described in the sections 6.2.2.1-6.2.2.3. In Section 6.7 it was suggested that methods could be improved to identify a maximal set of attributes that can be evaluated on the fly.

Furthermore, there are few areas that research in them would complement the present thesis. Investigating a lattice of relations on context-free languages and a general theory of predictors for context-free grammars [Tai 80] are such problems. In the simplest canonical form, the concept of predictors formalizes the heuristic idea of important terminal symbols in context-free grammars (e.g., the reserved words in programming languages). It is believed that LRRL techniques can be used to contribute toward a general theory that considers non-canonical and phrase level predictors.

It is also possible to develop an incremental LRRL parser constructor. Given a parse table corresponding to a grammar and a set of rules to be inserted into or deleted from the grammar, such a method modifies the parse table in a way that corresponds to the new grammar.

An interesting problem is to investigate a set of generalized phrase structure rules [Gazd 82, ScPe 82] for a subset of natural language that can be supported by an attributed LRRL parser. Generators for LRRL parsers similar to YACC [John 75] or LINGUIST-86 [F[illegible]82] can be written very easily to support such research. However, a disadvantage with those systems and majority of others is that they do not support any language as an interface between the user and the system for exchange of linguistic information. A good exception is the embedding of ALADIN (A Language for Attributed Definitions) in the GAG System [KaHZ 82] which has contributed to the success of that system. Nevertheless the GAG system and ALADIN are intended for the use of compiler construction community. Therefore, to facilitate the investigation of linguistic problems such as above one, the need for a higher level programming language for processing linguistic information is very apparent. Shieber [Shie 84] discusses the design of such a language, i.e., PATR-II. Pereira and Shieber [PeSh 84] provide the denotational semantics of the language.

It is believed that the design and implementation of a general programming language based on attributed grammars that can be used as a tool in compiler writing and natural language problems would be an ambitious but very fruitful project. Hehner and Silverberg [HeSi 83] have suggested use of attributed grammars as a high-level programming language. A preliminary study in [PaKo 86] shows possible use of such a language as a knowledge engineering tool.

REFERENCES

[AhJU 75] A.V. Aho, S.C. Johnson and J.D. Ullman. Deterministic parsing of ambiguous grammars. CACM, vol. 18, no. 8, pp. 441-452. 1975.

[AhPe 72] A.V. Aho and T.G. Peterson. A minimum distance error correcting parsing for context-free languages. SIAM Journal of Computing, vol. 1, no. 4, pp. 305-312. December 1972.

[AhUl 72a] A.V. Aho and J.D. Ullman. The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing. Prentice Hall, Englewood Cliffs, N.J. 1972.

[AhUl 72b] A.V. Aho and J.D. Ullman. Optimization of LR(k) grammars. Journal of Computer and System Sciences, vol. 6, pp. 573-602. 1972.

[AhUl 73a] A.V. Aho and J.D. Ullman. The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling. Prentice Hall, Englewood Cliffs, N.J. 1973.

[AhUl 73b] A.V. Aho and J.D. Ullman. A technique for speeding up LR(k) parser. SIAM Journal of Computing, vol. 2, pp. 106-127. 1973.

[AnDG 82] M. Ancona, G. Dodero and V. Gianuzzi. Building collections of LR(k) items with partial expansion of lookahead strings. ACM Sigplan Notices, vol. 17, no. 5, pp. 24-28. 1982.

[Bake 82] T.P. Baker. A one-pass algorithm for overload resolution in Ada. ACM Transactions on Programming Languages and Systems, vol 4, no. 4, pp. 601-614. Oct. 1982.

[Berw 81] R.C. Berwick. Computational complexity and lexical functional grammars. Proceedings of the 19th Meeting of the Association for Computational Linguistics, pp. 7-12. Stanford University, CA. 1981.

[Berw 83] R.C. Berwick. A deterministic parser with broader coverage. IJCAI 83, Proceedings of the 8th International Joint Conference on Artificial

Intelligence, pp. 710-712. August 1983.

[Berw 84] R.C. Berwick. Bounded context parsing and easy learnability. Proceedings of COLING 84, 10th International Conference on Computational Linguistics, pp. 20-23. Stanford University, CA. July 1984.

[BeWe 82] R.C. Berwick and K. Wexler. Parsing efficiency, binding, and c-command. Proceedings of the First West Coast Conference on Formal Linguistics, ed. D.P. Flickinger, M. Macken and N. Wiegard, pp. 29-34. Stanford University. January 1982.

[Boch 76] G.V. Bochmann. Semantic evaluation from left to right. CACM, vol. 19, no. 2. February 1976.

[Bris 83] E.J. Briscoe. Determinism and its implementation in PARSIFAL. Automatic Natural Language Parsing, ed. K. Spark Jones and Y. Wilks. Ellis Horwood, Chichester, England. 1983.

[Cele 81] A. Celentano. An LR parsing technique for extended context-free grammars. Computer Languages, vol. 6, pp. 95-107. 1981.

[Chap 84] N.P. Chapman. LALR(1,1) parser generation for regular right part grammars. Acta Informatica, vol. 21, pp. 29-45. 1984.

[Char 83] E. Charniak. A parser with something for every one. Parsing Natural Language, ed. M. King, pp. 117-149. Academic Press, London. 1983.

[Chur 80] K.w. Church. On memory limitations in natural language processing. MIT 1980. Reproduced by Indiana University Linguistic Club. January 1982.

[CoHK 82] J. Cohen, T. Hickey and J. Katcoff. Upper bounds for speedup in parallel parsing. JACM, vol. 29, no. 2, pp. 408-428. April 1982.

[CuCo 73] K. Culik II and R. Cohen. LR-regular grammars: an extension of LR(k) grammars. Journal of Computer and System Sciences, vol. 7, pp. 66-96. 1973.

[Deme 75] A.J. Demers. Elimination of single productions and merging non-terminal symbols of LR(1) grammars. Computer Languages, vol. 1, pp. 105-119. 1975.

[Dere 71] F.L. DeRemers. Simple LR(k) grammars. CACM, vol. 14, pp. 453-460. 1971.

[Earl 70] J. Earley. An efficient context-free parsing algorithm. CACM, vol. 13, no. 2, pp. 94-102. February 1970.

[Farr 82] R. Farrow. Linguist-86: Yet another translator writing system based on attributed grammars. Proceedings of the Sigplan' 82 Symposium on Compiler Construction, Boston, MA. ACM Sigplan Notices, vol. 17, no. 6, pp. 160-171. June 1982.

[FiMQ 80] C.N. Fischer, D.R. Milton and S.B. Quiring. Efficient LL(1) error correction and recovery using only insertions. Acta Informatica, vol. 13, no. 2, pp. 141-154. 1980.

[Fisc 75] C.N. Fischer. On parsing context-free languages in parallel environments. Ph.D. thesis, Department of Computer Science, Cornell University. 1975.

[Floy 61] R.W. Floyd. A descriptive language for symbolic manipulations. JACM, vol. 8, no. 4. 1961.

[Gazd 82] G. Gazdar. Phrase structure grammar. The Nature of Syntactic Representation, ed. P. Jacobson and G.K. Pullum, pp. 131-186. Reidel. 1982.

[GeHa 77a] M.M. Geller and M.A. Harrison. Characteristic parsing: a framework for producing compact deterministic parsers, I. Journal of Computer and System Sciences, vol. 14, pp. 265-317. 1977.

[GeHa 77b] M.M. Geller and M.A. Harrison. Characteristic parsing: a framework for producing compact deterministic parsers, II. Journal of Computer and System Sciences, vol. 14, pp. 318-343. 1977.

[GrHJ 79] S.L. Graham, C.B. Haley and W.N. Joy. Practical LR error recovery. Proceeding of the Sigplan Symposium on Compiler Construction, Denvor, Collorado, ACM Sigplan Notices, vol. 14, no. 8, pp. 168-175. August 1979.

[GrHR 80] S.L. Graham, M.A. Harrison and W.L. Ruzzo. An improved context-free recognizer. ACM Transactions on Programming Languages and Systems, vol. 2, no. 3, pp. 415-462. July 1980.

[Heil 81] S. Heilbrunner. A parsing automata approach to LR theory. Theoretical Computer Science, vol. 15, pp. 117-157. 1981.

[HeSi 83] E.C. Hehner and B.A. Silverberg.

Programming with grammars: an exercise in methodology-directed language design. The Computer Journal, vol. 26, no. 3, pp. 277-281. 1983.

[Hunt 82] H.B. Hunt, III. On the decidability of grammar problems. JACM, vol. 29, no. 2, pp. 429-447. April 1982.

[Hunt 84] H.B. Hunt, III. Terminating Turing machine computations and the complexity and/or decidability of correspondence problems, grammars and program schemes. JACM, vol. 31, no. 2, pp. 229-318. April 1984.

[HuSU 74] H.B. Hunt, III, T.G. Szymanski and J.D. Ullman. Operations on sparse relations and efficient algorithms for grammar problems. Conference Record of IEEE 15th Annual Symp. on Switching and Automata Theory, pp. 127-132. 1974.

[HuSU 75] H.B. Hunt, III, T.G. Szymanski and J.D. Ullman. On the complexity of LR(k) testing. CACM, vol. 18, no. 12, pp. 707-716. December 1975.

[HuSU 77] H.B. Hunt, III, T.G. Szymanski and J.D. Ullman. Operations on sparse relations. CACM, vol. 20, pp. 171-176. 1977.

[Ichb 79] J.D. Ichbiah et al. Rationale for the design of Ada programming language. ACM Sigplan Notices, vol. 14, no. 6, part B. June 1979.

[Jack 77] R. Jackendoff. $\bar{X}$ Syntax: A Study of Phrase Structures. MIT Press, Cambridge, MA. 1977.

[JaOR 75] M. Jazayeri, W.F. Ogden and W.C. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. CACM, vol. 18, no. 12; pp. 697-706. Dec. 1975.

[JaPo 81] M. Jazayeri and D. Pozefsky. Space-efficient storage management in an attribute grammar evaluator. ACM Transactions on Programming Languages and Systems, vol. 3, no. 4, pp. 388-404. October 1981.

[JaWa 75] M. Jazayeri and K.G. Walter. Alternating semantic evaluators. Proceedings of the ACM 75 Annual Conference, pp. 230-234. 1975.

[JeWi 74] K. Jensen and N. Wirth. Pascal User Manual and Report, second edition. Springer-Verlag, New York. 1974.

[John 75] S.C. Johnson. YACC: yet another compiler-compiler. Technical Report 32, Bell Laboratories, Murray Hill, N.J. 1975. Also reproduced in Unix Programmer's Manual, vol. 2.

[JoMa 80] N.D. Jones and M. Madsen. Attribute-influenced LR parsing. Proceedings of the Workshop on Semantic-Directed Compiler Generation, Aarhus, Denmark, January 1980. Lecture Notes in Computer Science, no. 94, ed. N.D. Jones. Sringer-Verlag. 1980.

[Josh 81] A.K. Joshi. Factoring recursion and dependencies. 21st Conference on Computational Linguistics. 1981.

[KaHZ 82] U. Kastens, B. Hutt and E. Zimmermann. GAG: A Practical Compiler Generator. Lecture Notes in Computer Science, no. 141. Springer-Verlag. 1982.

[Kast 80] U. Kastens. Ordered attributed grammars. Acta Informatica, vol. 13, pp. 229-256. 1980.

[Knut 65] D.E. Knuth. On the translation of languages from left to right. Information and Control, vol. 8, pp. 607-639. 1965.

[Knut 68] D.E. Knuth. Semantics of context-free languages. Mathematical Systems Theory, vol. 2, no. 2, pp 127-145. 1968.

[Kost 71] C.H.A. Koster. Affix grammars. Algol 68 Implementation, ed. J.E. Peck, pp. 95-109. North-Holland, Amsterdam. 1971.

[Lalo 77] W.R. Lalonde. Regular right part grammars and their parsers. CACM, vol. 20, pp. 731-741. 1977.

[Lalo 79] W.R. Lalonde. Constructing LR parsers for regular right part grammars. Acta Informatica, vol. 11, pp. 177-193. 1979.

[Ledg 81] H. Ledgard. Ada: Introduction and Ada Reference Manual. Sringer-Verlag, New York. 1981.

[Ledg 84] H. Ledgard. American Pascal Standard. Springer-Verlag, New York. 1984.

[LeRS 74] P.M. Lewis, D.J. Rosenkrantz and R.E. Stearns. Attributed translations. Journal of Computer and System Sciences, vol. 9, pp. 279-307. 1974.

[Mads.80] O.L. Madsen. On defining semantics by means of attribute grammars. Proceedings of the Workshop on Semantic-Directed Compiler Generation, Aarhus, Denmark, January 80. Lecture Notes in Computer Science, no. 94, ed. N.D. Jones. Springer-Verlag. 1980.

[MaFi 82] J. Mauney and C.N. Fischer. A forward move algorithm for LL and LR parsers. Proceedings of the Sigplan' 82 Symposium on Compiler Construction, Boston, MA., ACM Sigplan Notices, vol. 17, no. 6, pp. 79-87. June 1982.

[MaKr 76] O.L. Madsen and B.B. Kristensen. LR-parsing of extended context-free grammars. Acta Informatica, vol. 7, pp. 61-73. 1976.

[Marc 75] M.P. Marcus. Diagnosis as a notion of grammar. Theoretical Issues in Natural Language Processing, ed. R. Schank and B. Nash-Weber, pp. 6-10. 1975.

[Marc 76] M.P. Marcus. A design for a parser for English. Proceedings of ACM Annual Conference, pp. 62-68. 1976.

[Marc 78] M.P. Marcus. A computational account of some constraints on language. Theoretical Issues in Natural Language Processing-2, pp. 236-246. University of Illinois, Urbana-Champion. July 1978. Also appeared in Elements of Discourse Understanding, ed. A.K. Joshi, B.L. Webber and I.A. Sag. Cambridge University Press, Cambridge. 1981.

[Marc 80] M. Marcus. A Theory of Syntactic Recognition for Natural Language. MIT Press, Cambridge, MA. 1980.

[MiSc 78] M.D. Mickunas and R.M. Schell. Parallel compilation in multiprocessor environment. Proceedings of ACM Annual Conference, pp. 241-246. 1978.

[Nijh 80] A. Nijholt. Context-Free Grammars, Covers, Normal Forms and Parsing. Lecture Notes in Computer Science, no. 93. Springer-Verlag. 1980.

[Nijh 83] A. Nijholt. Deterministic Top-Down and Bottom-Up Parsing: Historical Notes and Bibliographies. Mathematical Centre, Amsterdam. 1983.

[Nozo 86] R. Nozohoor-Farshi. On formalizations of Marcus' parser. To appear in Proceedings of COLING 86, 11th International Conference on Computational Linguistics. University of Bonn, West Germany. August 1986.

[Page 77a] D. Pager. A practical general method for constructing LR(k) parser. Acta Informatica, vol. 7, pp. 249-268. 1977.

[Page 77b] D. Pager. Eliminating unit productions from LR parsers. Acta Informatica, vol. 9, pp. 31-59. 1977.

[PaKo 86] G. Papakonstantinou and J. Kontos. Knowledge representation with attribute grammars. The Computer Journal, vol. 29, no. 3, pp. 241-245. 1986.

[PeDe 78] T.J. Pennello and F. DeRemer. A forward move algorithm for LR error recovery. Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, pp 241-254. Tucson, Arizona. 1978.

[PeSh 84] F.C.N. Pereira and S.M. Shieber. The semantics of grammar formalisms seen as computer languages. Proceedings of COLING 84, 10th International Conference on Computational Linguistics, pp. 123-129. Stanford University, CA. July 1984.

[Pohl 83] W. Pohlmann. LR parsing for affix grammars. Acta Informatica, vol. 20, pp. 283-300. 1983.

[PoJa 78] D. Pozefsky and M. Jazayeri. A family of pass-oriented attribute grammar evaluators. Proceeding of ACM 78 Annual Conference, pp. 261-270. 1978.

[Ritc 83] G.D. Ritchie. The implementation of a PIDGIN interpreter. Automatic Natural Language Parsing, ed. k. Spark Jones and Y. Wilks, pp. 69-80. Ellis horwood, Chichester, England. 1983.

[RoLe 70] D.J. Rosenkrantz and P.M. Lewis, II. Deterministic left corner parsing. Conference Record of the IEEE 11th Annual Symp. on Switching and Automata Theory, pp. 139-152. 1970.

[RSST 78] K. Räihä, M. Saarinen, E. Soisalon-Soininen and M. Tienari. The compiler writing system HLP (Helsinki language processor). Technical Report A-1978-2, Department of Computer Science, University of Helsinki, Finland. March 1978.

[Samp 83] G.R. Sampson. Deterministic parsing. Parsing Natural Language, ed. M. King, pp. 91-116. Academic Press. 1983.

[Schu 83] L.K. Schubert. Private discussion. 1983.

[Schu 84] L.K. Schubert. On parsing preferences. Proceeding of COLING 84, 10th International Conference on Computational Linguistics, pp. 247-250. Stanford University, CA. July 1984.

[ScPe 82] L.K. Schubert and F.J. Pelletier. From English to logic: context-free computation of 'conventional' logic translation. American Journal of Computational Linguistics, vol. 8, no. 1, pp. 26-44. January-March 1982.

[Shie 83] S.M. Shieber. Sentence disambiguation by a shift-reduce parsing technique. Proceeding of the 21st Annual Meeting of the Association for Computational Linguistics, pp. 113-118. MIT, MA. June 1983.

[Shie 84] S.M. Shieber. The design of a computer language for linguistic information. Proceedings of COLING 84, 10th International Conference on Computational Linguistics, pp. 362-366. Stanford University, CA. July 1984.

[ShMa 79] D.W. Shipman and M.P. Marcus. Towards minimal data structures for deterministic parsing. IJCAI 79, Advanced Papers of the 6th International Joint Conference on Artificial Intelligence, pp. 815-817. 1979

[SiSo 83] S. Sippu and E. Soisalon-Soininen. A syntax-error handling technique and its experimental analysis. ACM Transactions on Programming Languages and Systems, vol. 5, no. 4, pp. 656-679. Oct. 1983.

[SiSU 83] S. Sippu, E. Soisalon-Soininen and E. Ukkonen. The complexity of LALR(k) testing. JACM, vol. 30, no. 2, pp. 259-270. April 1983.

[Sois 80] E. Soisalon-Soininen [illegible] the space optimizing effect of eliminati[illegible] productions from LR parsers. Acta Info[illegible], 14, pp. 157-174. 1980.

[Sois 82] E. Soisalon-Soinine[illegible] nessential error entries and their use in LR parser optimization. ACM Transactions on Programming Languages and Systems, vol. 4, no. 2. April 1982.

[Spec 81] D. Spector. Full LR(1) parser generation. ACM Sigplan Notices, vol. 16, no. 8, pp. 58-66. August 1981.

[SzWi 76] T.G. Szymanski and J.H. Williams. Non-canonical extensions of bottom-up parsing techniques. SIAM Journal of Computing, vol. 5, no. 2. June 1976.

[Szym 73] T.G. Szymanski. Generalized bottom-up parsing, Ph.D. thesis, Cornell University. 1973.

[Tai 78] K-C. Tai. Syntactic error correction in programming languages. IEEE Transactions on Software Engineering, vol. SE-4, no. 5, pp. 414-425. September 1978.

[Tai 79] K-C. Tai. Non-canonical SLR(1) grammars. ACM Transactions on Programming Languages and Systems, vol. 1, pp. 295-320. 1979.

[Tai 80] K-C. Tai. Predictors of context-free grammars. SIAM Journal of Computing, vol. 9, no. 3, pp. 653-664. August 1980.

[Tien 80] M. Tienari. On the definition of attribute grammar. Proceedings of the Workshop on Semantic-Directed Compiler Generation, Aarhus, Denmark, January 80. Lecture Notes in Computer Science, no. 94, ed. N.D. Jones. Springer-Verlag. 1980.

[Tomi 84] M. Tomita. LR parsers for natural languages. Proceedings of COLING 84, 10th International Conference on Computational Linguistics, pp. 354-357. Stanford University, CA. July 1984.

[Tomi 85] M. Tomita. An efficient context-free parsing algorithm for natural languages. IJCAI 85, Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 756-764. University of California, Los Angeles, CA. 1985.

[Vali 75] L. Valiant. General context-free recognition in less than cubic time. Journal of Computer and System Sciences, vol. 10, pp. 308-315. 1975.

[Walt 70] D.A. Walters. Deterministic context-sensitive languages. Information and Control, vol. 17, pp. 14-61. 1970.

[WaMa 83] D.A. Watt and O.L. Madsen. Extended attribute grammars. The Computer Journal, vol. 26, no. 2, pp. 142-153. 1983.

[Watt 77] D.A. Watt. The parsing problem for affix grammars. Acta Informatica, vol. 8, pp. 1-20. 1977.

[Watt 80] D.A. Watt. Rule splitting and attribute-directed parsing. Proceedings of the Workshop on Semantic-Directed Compiler Generation, Aarhus, Denmark, January 80. Lecture Notes in Computer Science, no. 94, ed. N.D. Jones. Springer-Verlag. 1980.

[WeSH 77] J. Welsh, W.J. Sneeringer and C.A.R. Hoare. Ambiguities and insecurities in Pascal. Software- Practice and Experience, vol. 7, pp. 685-696. 1977.

[Weth 81] C.S. Wetherell. Problems with the Ada reference grammar. ACM Sigplan Notices, vol. 16, no. 9, pp. 90-104. Sept. 1981.

[Will 75] J.H. Williams. Bounded context parsable grammars. Information and Control, vol. 28, pp. 314-334. 1975.

[Wino 72] T. Winograd. Understanding Natural Language. Academic.Press, New York. 1972.

[Youn 67] D.H. Younger. Recognition and parsing of context-free languages in time n^3. Information and Control, vol. 10, pp. 189-208. 1967.

APPENDIX I

Context-freeness of the language accepted by Marcus' parser

While Marcus does not use phrase structure rules as base grammar in his parser, he points out some correspondence between the use of a base rule and the way packets are activated to parse a construct [Marc 80]. Charniak [Char 83] has also assumed some phrase structure base grammar in a partial reimplementation of Marcus' parser that handles some ungrammatical situations. However, neither has suggested a type for such a grammar or the language accepted by Marcus' parser. In this appendix, it is shown that the set of sentences accepted by a Marcus type parser is a context-free language. This provides a justification for assuming a context-free underlying grammar in in automatic generation of such parsers.

The proof for context-freeness of the string set is based on simulating a simplified version of the parser by a pushdown automaton. Then some modifications of the PDA are suggested in order to ascertain that Marcus' parser, regardless of the structures it puts on the input sentences, accepts a context-free set of sentences. Furthermore, since the resulting PDA is a deterministic one, it confirms the determinism of the language parsed by this parser.

I.1 Further elaboration of the parser operation and assumption of a bounded buffer.

The basic operations of Marcus' parser were discussed in Section 2.2.1. Here, "attention shift" operations and the role of the buffer are examined further.

An "attention shift" operation moves a window of size k=3 to a given position on a buffer of size b=5. This occurs in parsing of some NP's, in particular when a buffer node other than the first indicates start of an NP. "Restore buffer" restores the window to its previous position before the last "attention shift".

Marcus suggests that the movements of the window can be achieved by employing a stack of displacements from the beginning of the buffer, and in general he suggests that the buffer could be unbounded on the right. But in practice, he notes that he has not found a need for more than five cells, and PARSIFAL does not use a stack to implement the window or virtual buffer.

A comment regarding an infinite buffer is in place here. An unbounded buffer will give a parser with two stacks. Generally, such parsers characterize context-sensitive languages and are equivalent to linear bounded automata. They have also been used for parsing some context-free languages. In this role they may hide the non-determinism of a context-free language by storing an unbounded number of lookaheads. Furthermore, basing parsing decisions on the whole left contexts and k lookaheads in them, has often resulted in defining a class of context-free (context-sensitive) grammars with undecidable membership. LR-regular, LR(k,∞) and FSPA(k) are such classes. The class of GLRRL(k) grammars with unbounded buffer (defined in Chapter 5) seems to be the known exception in this category that has decidable membership. However, it is not known whether these grammars generate only the deterministic context-free languages.

The class of BCP(m,n), bounded context parsable grammars, is another class of context-free grammars for which deterministic two-stack parsers exist and yet the membership in the class is decidable [Will 75]. In these grammars at least one phrase in any sentential form is recognized by m symbols to the left of the phrase and n symbols to its right. Naturally, the parsers for these grammars forget the left context except the last m symbol of it. Thus, the BCP grammars are not considered to be in the above category. In fact, they are a proper subclass of FSPA(n) grammars that do not include LR grammars. They also generate some non-deterministic languages.

Walter [Walt 70] considers context-sensitive grammars with deterministic two-stack parses and shows the undecidability of the membership problem for the class of such grammars.

In this appendix, it will be assumed that the buffer in a Marcus style parser can only be of a finite size b (e.g., b=5 in Marcus' parser). The limitation on the size of the buffer has an important consequence. It allows a proof for the context-freeness of the language to be given in terms of a PDA. However, one should note that Marcus style parsers with unbounded buffer, similar to GLRRL parsers, can still be constructed for languages that are known to be context-free.

I.2 Simplified parser

A few restrictions on Marcus' parser will prove to be convenient in outlining a proof for context-freeness of the language accepted by it.

I.2.1 Prohibition of features

Marcus allows syntactic nodes to have features containing the grammatical properties of the constituents that they represent. For implementation purposes, the type of a node is also considered as a feature. However, here a distinction will be made between this feature and others.

The type of a node and node itself will be considered to convey the same concept (i.e., a non-terminal symbol). Any other feature is disallowed.

In Marcus' parser, the binding of traces is also implemented through the use of features. A trace is a null deriving non-terminal (e.g., an NP) that has a feature named register pointing to another node, i.e., the binding of the trace. At the outset it must be stressed that Marcus' parser outputs the annotated surface structure of an utterance and traces are intended to be used by the semantic component to recover the underlying predicate/argument structure of the utterance. (In this respect Marcus' parser differs from parsing of deep structures in classical transformational grammars, where fillers are supposed to be shifted from one place to another in a sentence. Thus Marcus' parser is more in line with current linguistic theories and the treatment of the traces are very similar to that in Gazdar's context-free approach). Therefore one could put aside the issue of trace registers without affecting any argument that deals with the strings accepted by the parser, i.e., frontiers of surface structures. However, the features will be reintroduced in the generalized form of PDA for the completeness of the simulation.

I.2.2 Non-accessibility of the parse tree

Although most of the information about the left context is captured through the use of packeting mechanism in Marcus' parser, he nevertheless allows limited access to the nodes of the partial parse tree (besides the current active node) in the action parts of the grammar rules. In some rules, after the initial pattern matches, conditional clauses test for some property of the parse tree. These tests are limited to the left daughters of the current active node and the last cyclic node (NP or S) on the stack and its dominants. It is plausible to eliminate tree accessibility entirely through adding new packets and/or simple flags. In the simplified parser, access to the partial parse tree is disallowed. However, by modifying the stack symbols of the PDA, it will be shown later that the proof of context-freeness carries over to the general parser (that tests limited nodes of parse tree).

Combination of access to partial tree and packeting mechanism degrades the quality of error detection in Marcus' parser. The sample grammar given in [Marc 80] seems to be very relaxed in detecting errors in the input sentence. The rules of any given packet are spread throughout the listing. Because of this and the use of priorities it is difficult to pinpoint a situation where the parser accepts an erroneous sentence. But a thorough test of an implementation of the sample grammar would most likely reveal that some errors are not detected. For example, there seem to be situations where a node A can have two descendents B and C. After parsing and attachment of B, a packet apparently could remain active which upon erroneous input could trigger parsing and attachment of a second B node.

I.2.3 Atomic actions

Action segments in Marcus' grammar rules may contain a series of basic operations. To simplify the simulation, it is assumed that in the simplified parser actions are atomic. Breakdown of a compound action into atomic actions can be achieved by keeping the first operation in the original rule and introducing new singleton packets containing a default pattern and a remaining operation in the action part. These packets will successively deactivate themselves and activate the next packet much like **"run <rule> next"**s in PIDGIN. The last packet will activate the first if the original rule leaves the packet still active. Therefore in the simplified parser action segments are one the following forms:

(1) Activate packets1; [deactivate packets2].

(2) Deactivate packets1; [activate packets2].

(3) Attach ith; [deactivate packets1]; [activate packets2].

(4) [Deactivate packets1]; create node; activate packets2.

(5) [Deactivate packets1]; cattach node; activate packets2.

(6) Drop; [deactivate packets1]; [activate packets2].

(7) Drop into buffer; [deactivate packets1]; [activate packets2].

(8) Attention shift (to ith cell); [deactivate packets1];

[activate packets2].

(9) Restore buffer; [deactivate packets1]; [deactivate packets2].

Note that forward attention shift has no explicit command in Marcus' rules. An "AS" prefix in the name of a rule implies the operation. Backward window move has an explicit command "restore buffer". The square brackets in the above forms indicate optional parts. Feature assignment operations are ignored for the obvious reason.

I.3 Simulation of the simplified parser

Given a simplified parser, one can construct a PDA that recognizes the same string set that is accepted by the parser. Roughly, the states of this PDA are symbolized by the contents of the parser's buffer, and its stack symbols are ordered pairs consisting of a non-terminal symbol (i.e., a stack symbol of the parser) and a set of packets associated with that symbol.

Let N be the set of non-terminal symbols, and Σ be the set of terminal symbols of the parser. Also, let S_0, a distinct element of N, denote the top S node, i.e., the root of a parse tree. One may also assume that a final packet is added to the grammar. When the parsing of a sentence is completed, the activation of this packet will cause the root node S_0 to be dropped into the buffer, rather than being left on the stack. Furthermore, let P denote the set of all packets of rules, and 2^P the powerset of P, and let $P, P_1, P_2, \ldots$ be elements of 2^P.

When a set of packets P is active, the pattern segments of the rules in these packets are compared with the current active node and contents of the virtual buffer (the window). Then the action segment of a rule with highest priority that matches is executed. In effect the operation of the parser can be characterized by a partial function M from active packets, current active node and contents of the window into atomic actions, i.e.,

$$M: 2^P \times N^{(1)} \times V^{(k)} \to \text{ACTIONS}$$

where $V=N \cup \Sigma$, $V^{(k)}=V^0+V^1+\ldots+V^k$ and ACTIONS is the set of atomic actions (1)-(9) discussed in the previous section.

The following constructs a PDA $A=\{Q,\Sigma,\Gamma,\delta,q_0,Z_0,f\}$ which is equivalent to the simplified parser.

Σ = the set of input symbols of A, is the set of terminal symbols in the simplified parser.

Γ = the set of stack symbols $[X,P]$, where $X \in N$ is a non-terminal symbol of the parser and P is a set of packets.

Q = the set of states of the PDA, each of the form $<P_1,P_2,\text{buffer}>$, where P_1 and P_2 are sets of packets. In general P_1 and P_2 are empty sets except for those states that represent dropping of a current active node in the parser. P_1 is the set of packets to be activated explicitly after the drop operation, and P_2 is the set of those packets that are deactivated. "buffer" is a string in the set $(|^{(1)}V)^{(n)}|V^{(b-n)}$, where $0\leq n\leq b-k$. The last vertical bar in "buffer" denotes the position of the current window in the parser and those on the left indicate former window positions.

q_0 = the initial state = $<\emptyset,\emptyset,|\epsilon>$.

f = the final state = $<\emptyset,\emptyset,|S_0>$. This state corresponds to the outcome of an activation of the final packet in the parser. In this way, i.e., by dropping the S_0 node into the buffer, one can show the acceptance of a sentence simultaneously by empty stack and by final state.

Z_0 = the start symbol = $[S_0,P_0]$, where P_0={initial packets}, e.g., {SS-Start, C-Pool} in Marcus' parser.

δ = the move function of the PDA, defined in the following way:

Let P denote a set of active packets, X an active node and, $W_1W_2\ldots W_l$, $l\leq k$, the content of a window. Let $\alpha|W_1W_2\ldots W_l\beta$ be a string (representing the buffer) such that:
$\alpha\in(|^{(1)}V)^{(b-k)}$ and $\beta\in V^*$ where Length($\alpha'W_1W_2\ldots W_l\beta$)$\leq$b, and α' is the string α in which vertical bars are erased.

Non-ϵ-moves:

The non-ϵ-moves of the PDA A correspond to bringing the input tokens into the buffer for examination by the parser. In Marcus' parser input tokens come to the attention of parser as they are needed. Therefore, one can assume that when a rule tests the contents of l cells in the window and there are fewer [illegible] the buffer, terminal symbols will be brought [illegible] buffer. More specifically, if $M(P,X,W_1\ldots W_l)$ [illegible] defined value (i.e., P contains a [illegible]t with a rule that has pattern segment $[X][W]..![W_l]$),

[illegible]$(<\emptyset,\emptyset,\alpha|W_1\ldots W_j>,W_{j+1},[X,P])=(<\emptyset,\emptyset,\alpha|W_1\ldots W_j W_{j+1}>,[X,P])$
for all α, and for $j=0,\ldots,l-1$ and $W_{j+1}\in\Sigma$.

ϵ-moves:

By ϵ-moves, the PDA mimics the actions of the parser on successf[illegible] matches. Thus the δ-function on ϵ input corresponding to each individual atomic action is determined according to one of the following cases.

Cases (1) & (2):

If $M(P,X,W_1 W_2\ldots W_l)$=activate P_1; deactivate P_2 (or deactivate P_2; activate P_1), then

$\delta(<\emptyset,\emptyset,\alpha|W_1 W_2\ldots W_l\beta>,\epsilon,[X,P])=$
$(<\emptyset,\emptyset,\alpha|W_1 W_2\ldots W_l\beta>,[X,(P\cup P_1)-P_2])$ for all α and β.

Case (3):

If $M(P,X,W_1 W_2\ldots W_i\ldots W_l)$= attach ith (normally i is [illegible] deactivate P_1; activate P_2, then

$\delta(<\emptyset,\emptyset,\alpha|W_1\ldots W_i\ldots W_l\beta>,\epsilon,[X,P])=$
$(<\emptyset,\emptyset,\alpha|W_1\ldots W_{i-1}W_{i+1}\ldots W_l\beta>,[X,(P\cup P_2)-P_1])$ for all α,β.

Cases (4) and (5):

If $\mathcal{M}(P,X,W_1 \ldots W_l)=$ deactivate P_1; create/cattach Y; activate P_2,

then

$\delta(<\emptyset,\emptyset,\alpha|W_1 \ldots W_l\,\beta>,\epsilon,[X,P])=$

$(<\emptyset,\emptyset,\alpha|W_1 \ldots W_l\,\beta>,\ [X,P-P_1][Y,P_2])$ for all α and β.

Case (6):

If $\mathcal{M}(P,X,W_1 \ldots W_l)=$ drop; deactivate P_1; activate P_2, then $\delta(<\emptyset,\emptyset,\alpha|W_1 \ldots W_l\,\beta>,\epsilon,[X,P])=(<P_2,P_1,\alpha|W_1 \ldots W_l\,\beta>,\epsilon)$ for all α and β, and furthermore

$\delta(<P_2,P_1,\alpha|W_1 \ldots W_l\,\beta>,\epsilon,[Y,P'])=$

$(<\emptyset,\emptyset,\ \alpha|W_1 \ldots W_l\,\beta>,\ [Y,(P'\cup P_2)-P_1])$ for all α and β, and $P'\in 2^{\mathcal{P}}$, $Y\in N$. The latter move corresponds to the deactivation of the packets P_1 and activation of the packets P_2 that follow the dropping of a current active node.

Case (7):

If $\mathcal{M}(P,X,W_1 \ldots W_l)=$drop into buffer; deactivate P_1; activate P_2 (where $l<k$), then

$\delta(<\emptyset,\emptyset,\alpha|W_1 \ldots W_l\,\beta>,\epsilon,[X,P])=(<P_2,P_1,\alpha|XW_1 \ldots W_l\,\beta>,\epsilon)$ for all α and β, and furthermore

$\delta(<P_2,P_1,\alpha|XW_1 \ldots W_l\,\beta>,\epsilon,[Y,P'])=$

$(<\emptyset,\emptyset,\alpha|XW_1 \ldots W_l\,\beta>,\ [Y,(P'\cup P_2)-P_1])$ for all α and β, and for all $P'\in 2^{\mathcal{P}}$ and $Y\in N$.

Case (8):

If $\mathcal{M}(P,X,W_1 \ldots W_i \ldots W_l)=$ shift attention to ith cell; deactivate P_1; activate P_2, then

$\delta(<\emptyset,\emptyset,\alpha|W_1 \ldots W_i \ldots W_l\,\beta>,\ \epsilon,[X,P])=$

$(<\emptyset,\emptyset,\alpha|W_1 \ldots|W_i \ldots W_l\,\beta>,\ [X,(P\cup P_2)-P_1])$ for all α and β.

Case (9):

If $M(P,X,W_1 \ldots W_1) =$ restore buffer; deactivate P_1; activate P_2,

then

$\delta(<\emptyset,\emptyset,\alpha_1|\alpha_2|W_1 \ldots W_1\ \beta>,\epsilon,[X,P]) =$

$(<\emptyset,\emptyset,\alpha_1|\alpha_2 W_1 \ldots W_1\ \beta>,\ [X,(P \cup P_2)-P_1])$ for all α_1,α_2 and β such that α_2 contains no vertical bar.

Now from the construction of the PDA, it is obvious that A accepts those strings of terminals that are parsed successfully by the simplified parser. The reader may note that the value of δ is undefined for the cases in which $M(X,P,W_1 \ldots W_1)$ has multiple values. This accounts for the fact that Marcus' parser behaves in a deterministic way. Furthermore, many of the states of A are unreachable. This is due to the way in which the PDA was constructed by considering activation of all subsets of *P* with any active node and any lookahead window.

I.4 Simulation of the general parser

It is possible to lift the restrictions on the simplified parser by modifying the PDA. The following sections describe how Marcus' parser can be simulated by a generalized form of the PDA.

I.4.1 Non-atomic actions

The behaviour of the Marcus parser with non-atomic actions can be described in terms of $M \in M^*$, a sequence of compositions of M, which in turn can be specified by δ a sequence in δ^*.

I.4.2 Accessibility of descendants of current active node, and current cyclic node

What parts of the partial parse tree are accessible in Marcus' parser seems to be a moot point. Marcus' statement in this regard [Marc 80] does not help to clarify the situation. He states "the parser can modify or directly examine exactly two nodes in the active node stack... the current active node and S or NP node closest to the bottom of stack... called the dominating cyclic node... or... current cyclic node... The parser is also free to examine the descendants of these two nodes..., although the parser cannot modify them. It does this by specifying the exact path to the descendant it wishes to examine."

The problem is that whether by descendants of these two nodes, one means the immediate daughters or descendants at arbitrary levels. It seems plausible that accessibility of immediate descendants is sufficient. To explore this idea, one needs to examine the reason behind partial tree accesses in Marcus' parser. It could be argued that tree accessibility serves two purposes:

(1) Examining what daughters are attached to the current

active node considerably reduces the number of packet rules one needs to write.

(2) Examining the current cyclic node and its daughters serves the purpose of binding traces. Since transformations are applied in each transformational cycle to a single cyclic node, it seems unnecessary to examine descendants of a cyclic node at arbitrarily lower levels.

If Marcus' parser indeed accesses only the immediate daughters (a brief examination of the sample grammar does not seem to contradict this),,then the accessible part of the a parse tree can be represented by a pair of nodes and their daughters. Moreover, the set of such pairs of height-one trees are finite in a grammar. Furthermore, if one extends the access to the descendants of these two nodes down to a finite fixed depth (which, in fact seems to have a supporting evidence from $\bar{X}$ theory and C-command), one will still be able to represent the accessible parts, ~~of~~ parse trees with a finite set of finite sequences of fixed height trees.

A second interpretation of Marcus' statement is that descendants of the current cyclic node and current active node at arbitrarily lower levels are accessible to the parser. However, in the presence of non-cyclic recursive constructs, the notion of giving an exact path to a descendant of the current active or current cyclic node would not make a sense; in fact one can argue that in such a situation parsing cannot be achieved through a finite number of rule packets. The reader is reminded here that PIDGIN (unlike most programming languages) does not have iterative or recursive constructs to test the conditions that are needed under the latter interpretation.

Thus, a meaningful assumption in the second case is to consider every recursive node to be cyclic, and to limit accessibility to the subtree dominated by the current cyclic node in which branches are pruned at the lower cyclic nodes. In general, [illegible] may also include cyclic nodes at fixed recursion depths, but again branches of a cyclic node beyond that must be pruned. In this manner, one ends up with a finite number of finite sequences (hereafter called forests) of finite trees representing the accessible segments of partial parse trees. The information describing a partial parse tree is in the form of a forest, rather than a single tree, because some nodes may not yet be attached to a dominating node.

The conclusion is that at each stage of parsing the accessible segment of a parse tree, regardless of how Marcus' statement is to be interpreted, can be represented by a finite forest. In the PDA simulating the general parser, the set of stack symbols Γ would be the set of those finite forests, where in addition each node is paired with its associated packets. The states of this PDA will be of the form <X,P$_1$,P$_2$,buffer>. The last three elements are the same as before. The first entry is usually ϵ except that under the first/second interpretation when the current active/cyclic node is dropped, this element is changed to that node. For example, under the assumption that the pair of height-one trees rooted at current cyclic node and current active node is accessible to the parser, the definition of δ function would include the following statement among others:

If $\underline{M}(P,\underset{\underline{\wedge}}{X},W_1 \ldots W_l,\underset{\underline{\wedge}}{Y})=$ drop; deactivate P_1; activate P_2(where trees rooted at X and y represent current active and current cyclic nodes), then

$\delta(<\epsilon,\emptyset,\emptyset,\alpha|W_1 \ldots W_l\beta>, \epsilon,[(\underset{\underline{\wedge}}{Y},P'),(\underset{\underline{\wedge}}{X},P)])=$

$(<X,P_2,P_1,\alpha|W_1 \ldots W\beta>,\epsilon)$ for all α and β, and for all $P' \in 2^P$.

Furthermore, $\delta(<X,P_2,P_1,\alpha|W_1 \ldots W_l\beta>, \epsilon,[(\underset{\underline{\wedge}}{Y},P'),(\underset{\wedge_{\ldots x}}{Z},P'')])=$

$(<\epsilon,\emptyset,\emptyset,\alpha|W_1 \ldots W_l\beta>, [(\underset{\underline{\wedge}}{Y},P'),(\underset{\wedge_{\ldots x}}{Z},(P''\cup P_2)-P_1)])$ for all

$(Z,P'') \in N \times 2^P$ such that Z has X as a daughter.

This mechanism is devised to convey feature information to the higher level when the current active node is dropped. More specifically, there would be [illegible] associated with each symbol. When the node X is dropped, its associated features would be copied to the X symbol appearing in the state of the PDA (via first δ-move). The second δ-move allows these features to be copied from the X symbol in the state to the X node dominated by the node Z.

I.4.3 Accommodation of features:

The features used in Marcus' parser are syntactic in nature and have finite domains. Therefore the set of attributed symbols in that parser constitute a finite set. Hence syntactic features can be accommodated in the construction of the PDA by allowing complex non-terminal symbols, i.e., attributed symbols instead of simple ones.

Feature assignments can be simulated by replacing the top stack symbol in the PDA. For example, under the previous assumption that two height-one trees rooted at current active node and current cyclic node are accessible to the parser, the definition of δ function will include the following statement:

If $\underline{M}(P,\underset{\wedge}{X}:A,W_1\ldots W_l,\underset{\wedge}{Y}:B)=$ assign features A' to current active node; assign features B' to current cyclic node; deactivate P_1; activate P_2 (where A,A',B and B' are sets of features), then

$$\delta(\langle\epsilon,\emptyset,\emptyset,\alpha|W_1\ldots W_l\beta\rangle,\ \epsilon,\ [(\underset{\wedge}{Y}:B,P'),(\underset{\wedge}{X}:A,P)])=$$

$$(\langle\epsilon,\emptyset,\emptyset,\alpha|W_1\ldots W_l\beta\rangle,\ [(\underset{\wedge}{Y}:B\cup B',P'),(\underset{\wedge}{X}:A\cup A',(P\cup P_2)-P_1)])$$

for all α and β, and $P'\in 2^P$.

Now, by lifting all three restrictions introduced on the simplified parser, it is possible to conclude that Marcus' parser can be simulated by a pushdown automaton, and thus accepts a context-free set of strings. Moreover, since this pushdown automaton is a deterministic one, the determinism of the language parsed by Marcus' mechanism is confirmed.

It is easy to obtain (through a standard procedure) an LR(1) grammar describing the language accepted by the generalized PDA. Although this grammar will be equivalent to Marcus' PIDGIN grammar (minus any semantic considerations), and it will be a right cover for any underlying surface grammar which may be assumed in constructing the Marcus parser, it will suffer from being an unnatural description of the language. Not only may the resulting structures be hardly usable by any reasonable semantic/pragmatics component, but also parsing would be inefficient because of the huge number of non-terminals and productions.

I.5 Conclusions

It was shown that the information examined or modified during Marcus parsing (i.e., segments of partial parse trees, contents of the buffer and active packets) for a PIDGIN grammar is a finite set. By encoding this information in the stack symbols and the states of a deterministic pushdown automaton, one can show that the resulting PDA is equivalent to the Marcus parser. In this way, one proves that the surface sentences accepted by this parser is a context-free set.