FILE COMPRESSION USING

PROBABILISTIC GRAMMARS AND LR PARSING

BY

ADIL M.M. AL-HUSSAINI, B.Sc., M.Sc.

A Doctoral Thesis

submitted in partial fulfilment of the requirements

for the award of Doctor of Philosophy of the

Loughborough University of Technology

November, 1982

Supervisor: DR. R.G. STONE
Department of Computer Studies

DECLARATION

The work contained in this thesis (except where otherwise stated) is original research by the author and has not been submitted in full or part to this or any other institution for degree purposes.

ADIL M.M. AL-HUSSAINI

To my Parents

whom I owe more than

I can possibly express

CONTENTS

ABSTRACT

Data compression, the reduction in size of the physical representation of data being stored or transmitted, has long been of interest both as a research topic and as a practical technique. Different methods are used for encoding different classes of data files. The purpose of this research is to compress a class of highly redundant data files whose contents are partially described by a context-free grammar (i.e. text files containing computer programs).

An encoding technique is developed for the removal of structural dependancy due to the context-free structure of such files. The technique depends on a type of LR parsing method called LALR(K) (Lookahead LR(K)). The encoder also pays particular attention to the encoding of editing characters, comments, names and constants.

The encoded data maintains the exact information content of the original data. Hence, a decoding technique (depending on the same parsing method) is developed to recover the original information from its compressed representation.

The technique is demonstrated by compressing Pascal programs. An optimal coding scheme (based on Huffman codes) is used to encode the parsing alternatives in each parsing state. The decoder uses these codes during the decoding phase. Also Huffman codes, based on the probability of the symbols concerned, are used when coding editing characters, comments, names and constants. The sizes of the parsing tables (and subsequently the encoding tables) were considerably reduced by splitting them into a number of sub-tables.

The minimum and the average code length of the average program are derived from two different matrices. These matrices are constructed from a probabilistic grammar, and the language generated by this grammar. Finally, various comparisons are made with a related encoding method by using a simple context-free language.

ACKNOWLEDGEMENTS

I wish to thank Professor D.J. EVANS, Director of Research, for his encouragement and help throughout my study.

My sincere thanks to my Supervisor, Dr. R.G. STONE, for his guidance, encouragement and invaluable suggestions throughout the course of this thesis.

I am grateful to Mr. S. BEDI, Systems Manager, for his help during my practical work on the computer.

I am greatly indebted to my family for their moral and financial support which never came to an end. Their support and encouragement have given me the opportunity to continue my studies.

DETAILED CONTENTS

CHAPTER 1

INTRODUCTION

Computers are used because of their accuracy in getting the right results, the speed at which they accomplish the job, and their capacity for storing information. Of course, there is a limit (restriction) to each of the above facilities. The obvious restriction is the insufficient size of the storage space. To overcome this, it is necessary either to extend the size of the secondary storage which is costly, or to find efficient algorithms for compressing and restoring (decompressing) data which allow the storage that is available to be better utilized. This study considers, for a given string of symbols, the problem of finding a shorter string that uniquely determines the original string. It must always be possible to recover the original string from the short string. The algorithm for transforming a string into a shorter string is called data compression, and the algorithm for recovering the original string is called data decompression.

This study assumes that the input string consists of a finite set of symbols with some sort of structure from a context-free language. This structure produces redundancy in the language which is described by a context-free grammar. In addition, the string includes characters which lead to a more readable string. These characters are called editing characters.

The data compression model (i.e. an encoder) is designed to accept the above input stream, check for its correctness from a syntactical point of view (parsing the data), and then generate the required codes. The whole operation requires a finite set of steps (states) to be completed. The encoded data, which is supposed to occupy less storage space than the

original string, maintains the exact information content of the original data and should uniquely represent the original data.

To recover the original data, a decompression model (i.e. a decoder) is designed to accept the coded data, check for any syntax error, and then output the required symbols. There must be an agreement between the encoder and the decoder as to the way of parsing the data and the class of codes used (by the encoder) to represent different aspects of the encoding process. The decoded string must be exactly the same as the original string.

A particular encoder (and corresponding decoder) are developed to compress data written in the Pascal language. The parsing of the input depends on a technique called LR(K) parsing. The codes used are of variable-length; they are constructed according to the optimal Huffman coding method using the probabilities attached to the symbols, grammar rules, and choices in each state. Samples of Pascal programs have been collected, and a frequency program was written to find the frequency of different elements from those samples.

A matrix called the expectation matrix is constructed from the probabilistic grammar, which will help to obtain the average size of an input string, the average number of steps required to parse the string, and the average size of the encoded string.

The overall structure of the presentation is as follows:

Chapter 2 explains, in general, the construction of codes for a sequence of letters. It also explains some general properties of codes and specifies in particular a type of code called an instantaneous code. Two classes of codes are available, fixed-length codes, and variable-length codes. The attention is concentrated on the way of constructing variable

length codes, especially Huffman method, because such codes will be used in the encoder and the decoder programs, and can produce an average code length nearer to the minimum code length (i.e. the entropy).

Chapter 3 defines and explains a structured language called context-free language which would be generated from a context-free grammar. It also illustrates the way of deriving a string of symbols from the set of grammar rules (productions) using leftmost derivation and rightmost derivation techniques. By including a probability with each production, the grammar becomesaprobabilistic grammar which can generate a probabilistic language. Different methods used for compressing and decompressing strings from a probabilistic language will be described.

In the encoding method, the input must be (parsed) checked for any possible syntax error before the actual encoding procedure starts. So, Chapter 4 explains the techniques used for parsing a string from a language. Most emphasis is placed on a parsing method called LR(K) method. This includes the construction of the states and the parsing tables. Because of the large size of the parsing tables, different techniques are used to minimize the tables into a reasonable size. A parser generator called YACC is described in this chapter.

Chapter 5 describes the encoder model which accepts a program written in a context-free language as input and generates a corresponding encoded file. The encoder is an LR(K) parser generating Huffman code output. The codes represent the user names, constants, editing characters, comments and the parsing actions. The encoder program requires tables for holding the necessary codes which will be used during the encoding process. The codes

are constructed according to the frequencies of different actions, and symbols which were found by using a special program (frequency program).

The coded file has to be decoded in order to obtain the original file. The decoder model is described in Chapter 6. The structure of the decoder follows the structure of the encoder very closely. It includes the decoding of user names, constants, editing characters and comments. The decoder requires information to recognize the codes. This information is stored in decoding tables.

Chapter 7 discusses the properties of a probabilistic grammar which can generate a probabilistic language. Finding the properties depends on constructing a matrix called the expectation matrix from the grammar rules. It is possible to find out the average size of the input file, the average number of steps for recognizing a string of symbols, and the average code length (average length of a coded file). For rightmost derivations, the probability distribution of each state, and the average number of states required to parse a string of symbols is also discussed.

Finally, Chapter 8 concludes the overall work.

CHAPTER 2

OPTIMAL CODE LENGTH PER LETTER

This chapter basically explains the problem of representing letters from a source alphabet in terms of another set of letters. This representation is referred to as a code. Some properties of a code are explained in Section 2.1. For a set of source letters, the code consists of a finite number of code words. These words have either a fixed length or different lengths (Section 2.2). Section 2.3 shows how to represent code words by building up a tree. This representation helps when the source letters are retrieved from a sequence of code letters. For certain types of codes known as instantaneous codes, there is a formula (Kraft's inequality) in which it is possible to prove the existence of such codes for a given set of code word lengths. This inequality is explained in Section 2.4. Section 2.5 shows how to find the minimum average code length per source letter which is equal to the entropy. The average code length for a source letter is explained in Section 2.6. In Section 2.7, it is shown that an optimal variable-length code can be constructed from a well known method called Huffman's method. Finally, an extension to Huffman's method which led to a reduction in both the longest code word and the total number of digits, is illustrated in Section 2.8.

2.1 SOME PROPERTIES OF A CODE

Let $S=(s_1,s_2,\ldots,s_N)$ be a source alphabet consisting of N source letters. These source letters can be represented by a sequence of different letters called code letters from another set $C=(c_1,c_2,\ldots,c_M)$, such that, for each $s_i \in S$, $i=1,\ldots,N$ there is a sequence of $c_j \in C$, $j=1,\ldots,M$ (repetitions are allowed) representing s_i. For example, consider the representation of 4 source letters, using binary digits (0,1) as code letters, defined by Table 2.1.

Source Letters	Binary Representation
s_1	00
s_2	01
s_3	10
s_4	11

TABLE 2.1: Binary Representation of Source Letters

So, there are 4 binary sequences called code words, and each source letter corresponds to one code word. The correspondence of binary sequences to source letters is an example of a code. Using the code in Table 2.1, it is possible to obtain a sequence of binary digits for any sequence of source letters. For example, suppose that the sequence, $s_1s_2s_1s_4$ of source letters is required to be coded, the corresponding sequence of binary digits is

00 01 00 11

Conversely, it is possible (with the help of Table 2.1) to obtain the same sequence of source letters (i.e. $s_1s_2s_1s_4$) from the above sequence of binary digits. To discuss properties of codes (Abramson 63), it is

necessary first to give a formal definition.

Let $S=(s_1,s_2,\ldots,s_N)$ be a set of source letters. Then a code is defined as a mapping of all possible sequences of letters of S into sequences of letters (code letters) of some other alphabet $C=(c_1,c_2,\ldots,c_M)$. S is called the source alphabet, and C is the code alphabet.

The definition of a code, as mentioned above, is general. Therefore, it is necessary to investigate some of its conditions and try to give a clear idea of what a code looks like.

1. The procedure of transforming a source letter into a corresponding sequence of code letters is called an encoding, and the processor is called an encoder. Hence, for each source letter in the source alphabet, there is a corresponding code word. This enables the encoder to generate the right code word during the encoding process. For example, in Table 2.1, there are 4 source letters, and each one has its own code word. s_1 can be encoded as 00, s_2 as 01, s_1s_2 as 0001,... and so on. Since there is a fixed number of source letters, then the number of code words is fixed as well. A code satisfying this condition is called a block code. All code words contain either the same number of code letters (fixed-length) or different number of code letters (variable-length).

2. All code words of a block code should be distinct, that is, no two code words have the same sequence of code letters. In Table 2.1, for example, all code words are different. But in Table 2.2 the code words of both s_3 and s_4 are the same (101). A block code in which all the code words are distinct is called non-singular.

Source	Code Word
s_1	00
s_2	01
s_3	101
s_4	101

TABLE 2.2: A binary code

The process of retrieving the source letters from a sequence of code words is called a decoding process, and the processor is called a decoder. Without the distinction of code words, the decoder can not obtain the exact source letters. Given the following sequence of code words

00 101 01

the decoder could generate (by using Table 2.2) either

$$s_1 s_3 s_2$$

or

$$s_1 s_4 s_2.$$

3. Although a code should be non-singular (Table 2.3), it is possible to have a sequence of code letters which does not represent a unique sequence of source letters. Suppose that the sequence of binary digits

001001

Source	Code
s_1	0
s_2	01
s_3	001
s_4	111

TABLE 2.3: A binary code

is given, then the decoder (using Table 2.3 as a dictionary) could generate one of the following sequences of source letters.

$s_3 s_3$;

$s_1 s_2 s_3$;

$s_3 s_1 s_2$;

or $s_1 s_2 s_1 s_2$.

Therefore, code words should be uniquely decodable. The code in Table 2.4 satisfies this condition.

4. For a sequence of uniquely decodable code words, the decoder ought to be able to decode each code word as it arrives without checking the succeeding code letters. This can be achieved (Hamming 80) when no code word is the prefix of another code word. A code in which no code word is a prefix of *any* other code word is called an instantaneous code, or a prefix condition code.

Source	Code
s_1	0
s_2	01
s_3	011
s_4	0111

TABLE 2.4: Uniquely decodable code

Suppose that, a sequence of binary digits composed of code words from the code in Table 2.4 is given, and the decoder has already received the first binary digit (0), then it can not decide whether that digit is the code word of s_1, or it is a prefix of a code word representing s_2, or s_3 or s_4; unless a further check on the next digit

is made. Thus the code in Table 2.4 is not an instantaneous code. An example of an instantaneous code is given in Table 2.1. The advantage of an instantaneous code is that the decoding can be accomplished without delay, because the end of a code word can be recognized immediately and subsequent code letters do not have to be observed before decoding is commenced.

2.2 CLASSES OF CODES

As has been mentioned in the previous section, a code consists of a fixed number of code words. The number of code letters in a code word is called the length of the word. If every code word has the same length, then the code is called a fixed-length code. In contrast, if the code words are not all of the same length, then the code is called a variable-length code (Johns, 79).

Section 2.2.1 explains the construction of a fixed-length code, including the length of the code word. Variable-length codes are discussed in Section 2.2.2.

2.2.1 Fixed-Length Code

Let 0,1,2,...,9 be a source alphabet. Then there are 10 different ways of selecting only one letter from the source alphabet. For selecting 2 consecutive source letters, there are $(100=10^2)$ different ways. So, the number of selections of a sequence of letters depends on the number of source letters and the length of the source sequence.

In general, suppose that $s_1,s_2,\ldots,s_N$ be a source alphabet. Let k be the number of selections from the source alphabet. Then there are N^k different source sequences of length k that might be emitted from the source. Suppose that $c_1,c_2,\ldots,c_M$ is a code alphabet. Let the length of a code word be L. Since all code words have the same length, then the number of different code words is M^L.

From Section 2.1, each source sequence of length k must correspond to a separate code word. This is not possible unless there are at least

as many code words as there are source sequences (Johns, 79; Gallager, 68). So, to find the length of a code word (i.e. L) the following condition is satisfied:

$$M^L \geq N^K$$

$$L \log M \geq K \log N$$

$$L \geq K \frac{\log N}{\log M}$$

For K=1, the minimum length of a code word is $\frac{\log N}{\log M}$. For example, suppose that N=4, and at each time only one source letter (K=1) is encoded into a sequence of binary digits (M=2). Then:

$$L \geq \frac{\log 4}{\log 2} \doteq \log_2 4 = 2 \text{ binary digits}$$

So, all code words must be at least of length 2. See Table 2.1. If N=6 then

$$L \geq \log_2 6 \geq 2.58 \text{ binary digits}$$

L must be an integer number, so the minimum length of a code word is 3.

Encoding and decoding of source sequences using fixed-length codes are trivial. Both procedures require a dictionary of all source letters and their corresponding code words to be consulted. Almost all current computer systems use a fixed-length code for transforming or storing characters. Nevertheless, this class of codes does not, in general, provide a minimum average code word length per source letter. This will be explained in Section 2.5.

2.2.2 Variable-Length Code

In a variable-length code, the length of a code word for a source letter may be different from that of the code word for another source

letter. Choosing different lengths for the code words represents a statistical point of view, that is the source letters of a source alphabet are all used with different frequencies (i.e. have different probabilities). Consequently, a code word with a short length should be assigned to a high frequency source letter, and a long length code word assigned to a low frequency source letter.

If the source letters are used with about the same probability, little extra compression will be achieved by using a variable-length code rather than by a fixed-length code (section 2.2.1) (Holborow, McNemar and Stoneburner, 76). Hence a fixed-length code may be regarded as a method for encoding source letters which have a uniform probability distribution. However, if the statistics describing the usage of source letters are known accurately, the use of a correctly chosen variable-length code will produce a total code length much less than that obtained by a fixed-length code.

Before discussing the ways of constructing a variable-length code, it is important to mention that the code, which will be implemented in this study, must satisfy the properties in Section 2.1 (i.e. an instantaneous code). A necessary condition imposed on an instantaneous code is that no code word is the prefix of any other code word. This condition is called a prefix condition.

2.2.2.1 How to Construct a Variable-Length Code - Method 1

Let S be a set of N source letters $\{s_1, s_2, \ldots, s_N\}$. Each s_i has a propability $p(s_i)$ $1 \leq i \leq N$. Let M be the number of code letters in the code

alphabet C. Divide the source letters into M subsets making the probability of each subset as close to $\frac{1}{M}$ as possible (Johns, 79; Gallager, 68). Assign a different code letter to each of these subsets. If a subset has only one source letter, then the process on that subset will terminate. Divide each subset into M approximately equiprobable subsets, and assign to each new subset a different code letter. Continue in this process until each subset contains only one source letter.

As an example, suppose that $S=\{s_1,s_2,s_3,s_4\}$ and the probability of each source letter as shown in Table 2.5. Let C=(0,1) i.e. M=2. Then 2 subsets $\{\{s_1\},\{s_2,s_3,s_4\}\}$ are obtained each with probability $\frac{1}{2}$. Assign 0

Source	Prob.	Step 1	Step 2	Step 3	Code	$p(s_i)=\frac{1}{M^{\ell_i}}$
s_1	$\frac{1}{2}$	0			0	$\frac{1}{2}$
s_2	$\frac{1}{4}$	1 0			10	$\frac{1}{2^2}$
s_3	$\frac{1}{8}$	1 1	1 0		110	$\frac{1}{2^3}$
s_4	$\frac{1}{8}$	1 1	1 1	1 1	111	$\frac{1}{2^3}$

TABLE 2.5: An instantaneous code

to the first subset, and 1 to the second subset (step 1). Since the first subset has only one source letter, i.e. s_1, then the process is terminated, and s_1 gets code word 0. Divide the second subset into two subsets $\{\{s_2\},\{s_3,s_4\}\}$, each with probability $\frac{1}{4}$. Assign 0 to the first subset, and 1 to the second subset (step 2). Since s_2 is the only source

letter in the subset, then it gets code word 10. Finally, divide the second subset into two subsets $\{\{s_3\},\{s_4\}\}$, each with probability $\frac{1}{8}$. Assign 0 to the first subset, and 1 to the second subset (step 3). The process is terminated, s_3 gets code word 110, and s_4 gets code word 111 (see Table 2.5). The code satisfies the condition in Section 2.1; therefore it is an instantaneous code.

If the division can be achieved such that all subsets are equally probable at each step, then a relation can be established between the probability of a source letter and the code word length. That is:

$$p(s_i) = \frac{1}{M^{\ell_i}}$$

where ℓ_i is the code word length of the source letter s_i.

2.2.2.2 How to Construct a Variable-Length Code - Method 2

For M=2, there is another way of constructing an instantaneous code (Abramson, 63). That is by assigning 0 to the first source letter and 1 to the remaining source letters. The first source letter gets the code word 0. Select one source letter from the remaining letters and assign 0 to its code word which becomes 10. Assign 1 to the remaining source letters. Continue with this process until no more selections can be made. For example, suppose that there are 4 source letters s_1,s_2,s_3,s_4. Let M=2 i.e. C=(0,1), then assign 0 to s_1, and 1 to s_2,s_3 and s_4 i.e.

s_1	0
s_2	1
s_3	1
s_4	1

The code word for s_1 is 0. Select s_2 and assign 0 to it. Assign 1 to s_3 and s_4, i.e.

s_1	0
s_2	10
s_3	11
s_4	11

So, the code word of s_2 is 10. In the last selection, assign 0 to s_3 and 1 to s_4, i.e.

s_1	0
s_2	10
s_3	110
s_4	111

s_3 gets the code word 110, and s_4 gets 111.

For four source letters, an instantaneous code consists of four code words which can be obtained. As mentioned above, the shortest code word is assigned to the highest frequency source letter and assign the longest code word to the lowest frequency source letter.

Although methods 1 and 2 can construct an instantaneous code, they cannot always generate optimal codes. A well known method used to generate an optimal variable-length code is called Huffman's method. It is explained in Section 2.7. This method will be applied to the encoder and the decoder programs.

2.3 TREE REPRESENTATION OF CODE WORDS

Another way of *describing* a set of code words for an instantaneous code is by building up a tree (Johns, 79; Gallager, 68). A tree (sometimes called a rooted tree) is a finite set of points (nodes) connected by lines (branches) which satisfies the following properties (Page and Wilson, 73; Hopcroft and Ullman, 69).

1. Any two nodes in a tree are connected by a unique path (sequence of branches). The branch leaves one node and enters another node.
2. There is exactly one node which no branch enters. This node is called the root.
3. Exactly one branch enters every node except the root.

A node *with* at least one branch leaving it, is called a branch node (or non-terminal node). A node with no branch leaving is called a terminal node. For example, in Figure 2.1, the tree is a rooted tree (node 1 is the root). It has 8 nodes and 7 branches. Nodes (1,2,3) are non-terminal nodes, and nodes (4,5,6,7,8) are terminal nodes.

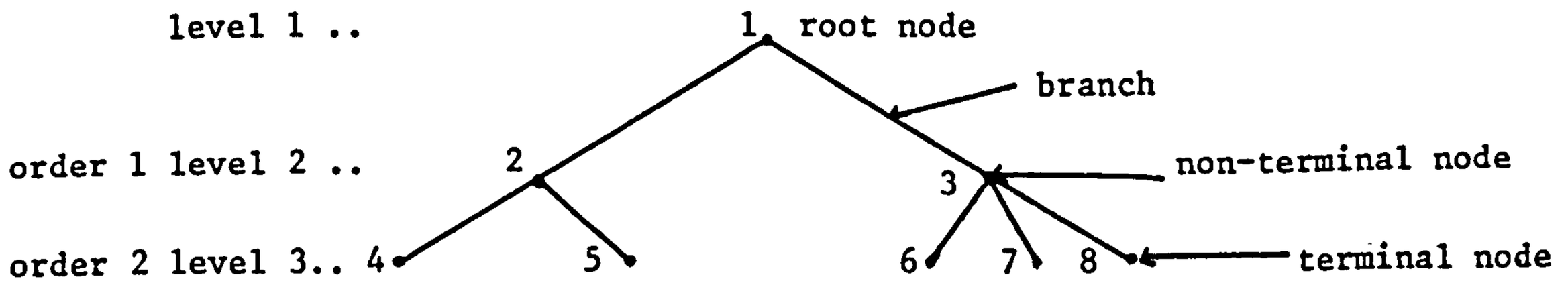


FIGURE 2.1: A rooted tree

The level of a node in a tree is the number of nodes passed through

on the path from the root to that node (inclusive of both the root and the node). For instance, node 2 is at level 2 because on its path there are only 2 nodes (node 1 and node 2). The order of a tree is the number of levels excluding level 1 (the root) which is assumed to be of order 0. For a particular order, the number of nodes is equal to the number of branches coming from each node in the previous level. The set of all nodes n, such that there is a branch leaving a given node m and entering n, is called the set of direct descendants of m. A node is called a descendant of node m if there is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that $n_1=m$, $n_k=n$, and for each i, n_{i+1} is a direct descendant of n_i.

So far, a general illustration of a tree has been given. A special case of a tree in which each node has exactly zero or two leaving branches is called a binary tree (Figure 2.2). A non-terminal node has two leaving branches, and a terminal node has zero leaving branches.

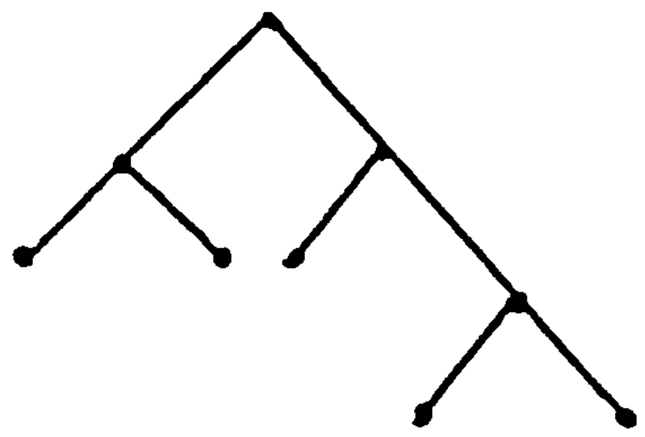


FIGURE 2.2: A binary tree

Suppose that the code letters are binary digits (0,1), then a binary tree is required to be constructed. To construct the code in Table 2.5, start from the root (level 1) of the tree. Two branches corresponding to the choice between 0 and 1 exist which lead to the second level (order 1) of the tree (Figure 2.3). In this level, one node

becomes a code word (0), and the second node represents the first code letter of the following code words. Another selection between 0 and 1 is made from the second node leads to the third level of the tree.

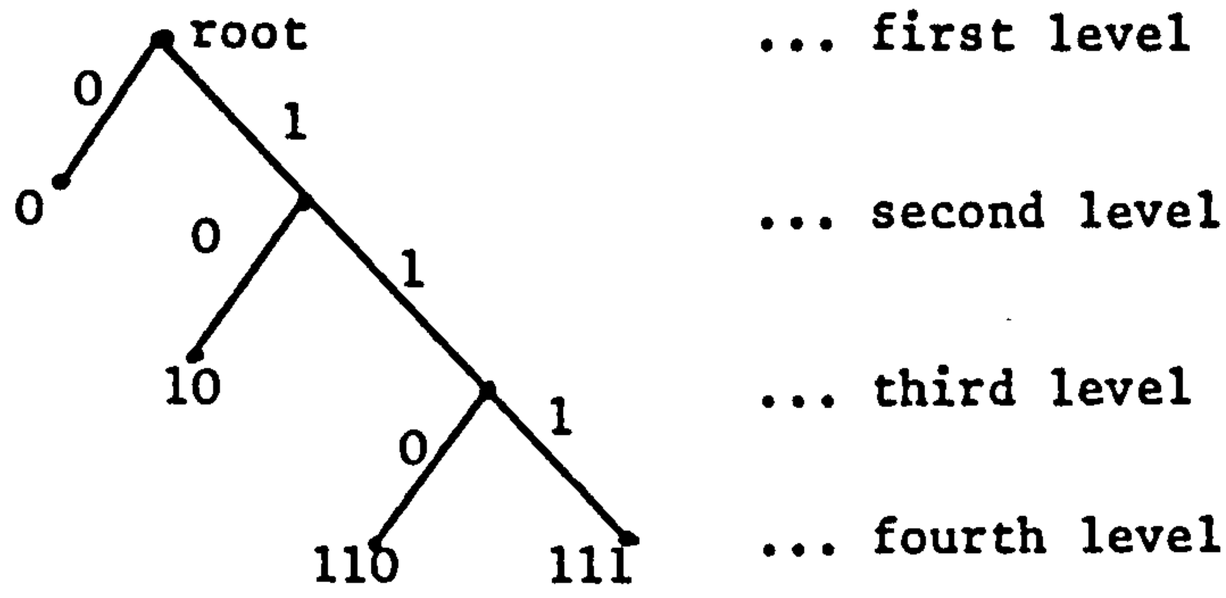


FIGURE 2.3: A tree for an instantaneous code

Again, two nodes exist, one node becomes a code word (10), and the second node represents the next code letter of the following code words. Similarly, the fourth level of the tree is obtained from the previous level. This level has 2 nodes representing 2 code words (110 and 111). Generally, by starting from the root, the successive letters leading to a terminal node represent a code word of a source letter.

The process of branching from one level to another, away from the root, can be done on any node (except terminal nodes). If all nodes at one level have either zero or two leaving branches (Fig. 2.4); this will lead to a full tree.

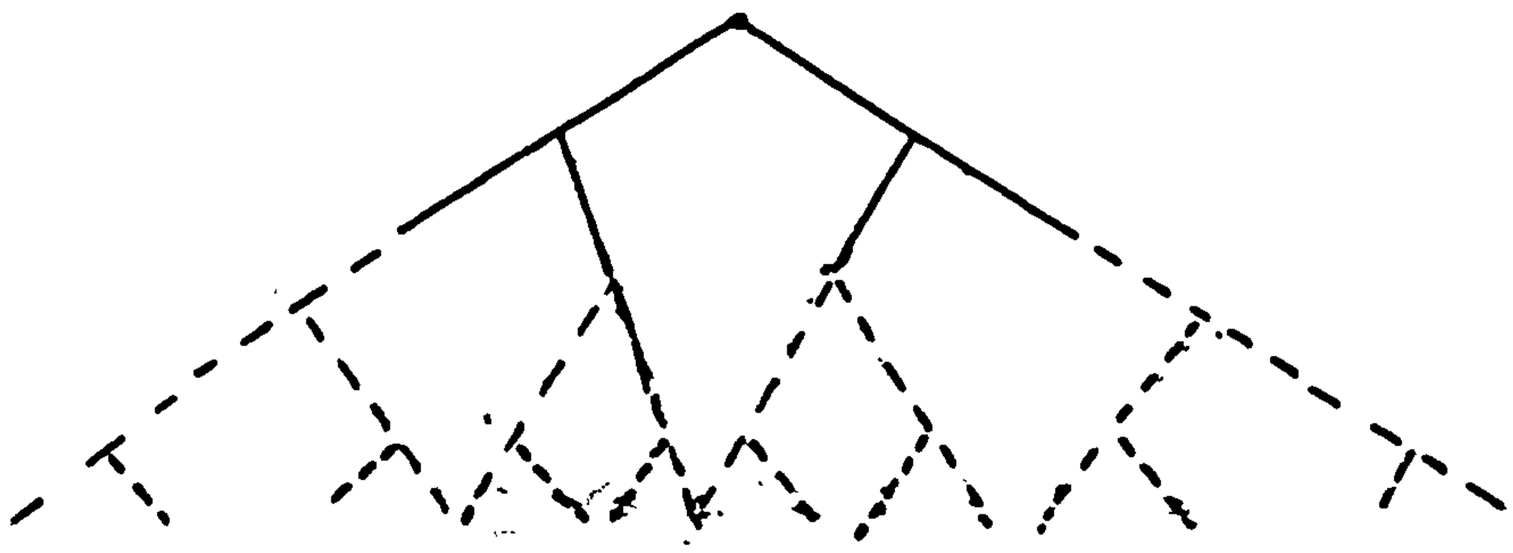


FIGURE 2.4: A full tree

2.4 THE KRAFT INEQUALITY

In Section 2.1, some constraints on the code word lengths of a prefix condition code have been discussed. Those constraints concern the quality of code words. It is possible to express the constraints in a quantitative fashion. The expression is provided by the following theorem.

Theorem (Kraft): A prefix condition code exists for code words of lengths $\ell_1,\ell_2,\ldots,\ell_N$ if, and only if,

$$\sum_{k=1}^{N} \frac{1}{M^{\ell_k}} \leq 1 \tag{2.1}$$

where M is the number of different letters in the code alphabet.

Proof 1: (Abramson, 63; Gallager, 68):

Part i): Sufficient condition.

Let $\ell_1,\ell_2,\ldots,\ell_N$ be code word lengths satisfying the inequality

$$\sum_{k=1}^{N} \frac{1}{M^{\ell_k}} \leq 1$$

These lengths may or may not be all distinct. Consider all code words of the same length at one time. Therefore, let n_1 be the number of code words of length 1; n_2 be the number of code words of length 2; etc. If the largest of the $\ell_i=\ell$ then

$$\sum_{i=1}^{\ell} n_i = N$$

The summation of (2.1) contains n_1 terms of $\frac{1}{M}$; n_2 terms of $\frac{1}{M^2}$; etc. It may then be written as

$$\sum_{i=1}^{\ell} \frac{n_i}{M^i} \leq 1 \tag{2.2}$$

On multiplying (2.2) by M^{ℓ}

$$\sum_{i=1}^{\ell} \frac{n_i M^{\ell}}{M^i} \leq M^{\ell}$$

or

$$\sum_{i=1}^{\ell} n_i M^{\ell-i} \leq M^{\ell}$$

$$n_1 M^{\ell-1} + n_2 M^{\ell-2} + n_3 M^{\ell-3} + \ldots + n_{\ell} \leq M^{\ell}$$

$$n_{\ell} \leq M^{\ell} - n_1 M^{\ell-1} - n_2 M^{\ell-2} - \ldots - n_{\ell-1} M$$

By dropping the term n_{ℓ} and dividing by M

$$n_{\ell-1} \leq M^{\ell-1} - n_1 M^{\ell-2} - n_2 M^{\ell-3} - \ldots - n_{\ell-2} M$$

Continue dropping the subsequent terms and dividing by M each time,

$$n_3 \leq M^3 - n_1 M^2 - n_2 M$$

$$n_2 \leq M^2 - n_1 M$$

$$n_1 \leq M$$

For n_1 (the number of code words of length 1), M possible such words can be formed using a code alphabet of M code letters. Since $n_1 \leq M$, select n_1 code words arbitrarily. Then $M-n_1$ code letters were not used as code words. They are prefixes of length 1. By adding one letter to the end of each of these permissible prefixes, a number of code words of length 2 could be formed i.e.

$$(M-n_1)M = M^2-n_1M$$

From the inequality above it is possible to select n_2 code words arbitrarily from among M^2-n_1M choices; then

$$(M^2-n_1M) - n_2$$

were not used as code words. By adding one code letter, there are

$$(M^2-n_1M-n_2)M = M^3-n_1M^2-n_2M$$

permissible prefixes of length 3. It is certain according to the above inequality that no more than this number is needed. So, n_3 code words may be selected arbitrarily. Proceed in this way until all code words have been formed.

Part ii): Necessary condition:

To prove that equation (2.1) is a necessary condition, the arguments already used are reversed.

End of Proof 1.

Proof 2: (Johns, 79):

Draw a full tree which has M branches coming from each node (Figure 2.6). There are M nodes of order 1, M^2 of order 2,...,M^k of order k, etc.

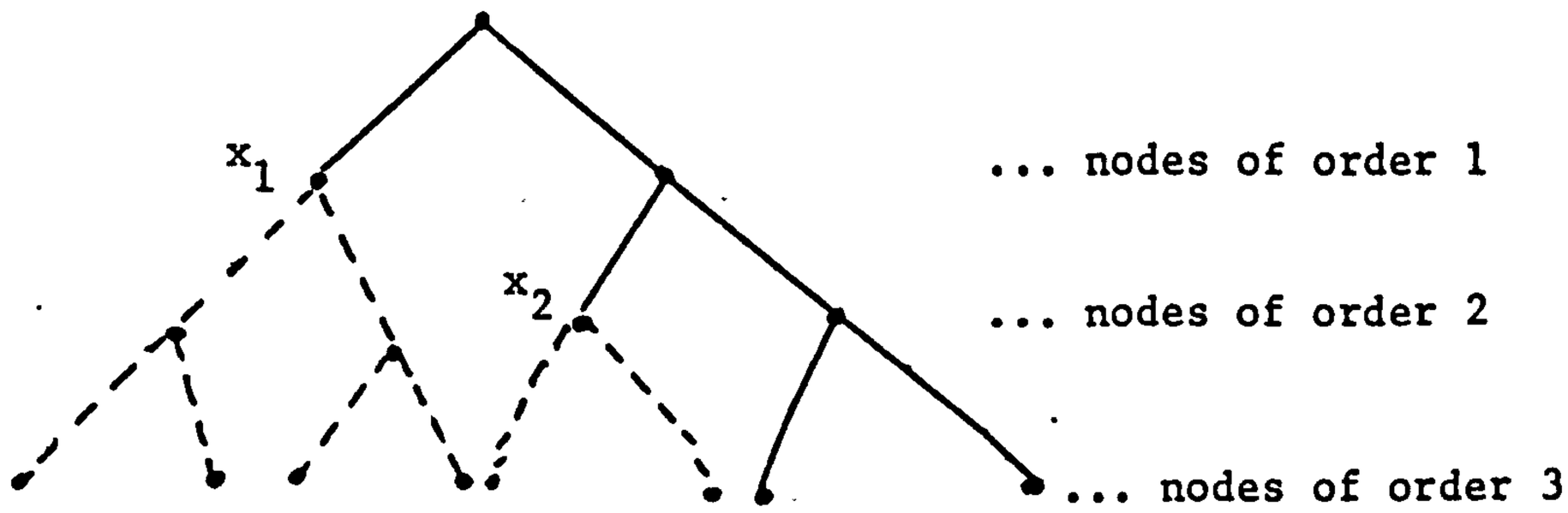


FIGURE 2.6: The full tree of order 3 where M=2

Each node gives rise to a code word. M code words of length 1 are available at order 1, M^2 code words of length 2 are available at order 2, and so forth.

Let $\ell_1,\ell_2,\ldots,\ell_N$ satisfy (2.1). If ℓ is the largest of ℓ_i $1\leq i\leq N$, then the full tree would be of order ℓ, and the tree representing the code will be embedded in it (in Figure 2.6 the tree (solid) embedded in the full tree (dashed)).

Arrange the lengths in an ascending order $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_N$, choose any node of order ℓ_1, say x_1, in the full tree as the first code word. Eliminate all the branches leaving x_1. All nodes on the full tree of each order greater than or equal to ℓ_1 are still available for use as code words except for the fraction $\frac{1}{M^{\ell_1}}$ that stem from node x_1. Choose any available node of order ℓ_2, say x_2, as the second code word. Eliminate all the branches leaving x_2. All nodes on the full tree of each order greater than or equal to ℓ_2 are still available for use as code words except for the fraction $\frac{1}{M^{\ell_1}} + \frac{1}{M^{\ell_2}}$. Repeating this process will lead to the situation that after choosing x_k $(k<N)$ as the k^{th} code word, all nodes in the full tree of each order greater than or equal to ℓ_k are still available except for the fraction $\sum_{i=1}^{k} \frac{1}{M^{\ell_k}}$ stemming from x_1 to x_k. From (2.1) this fraction is always less than 1 and so nodes are still available for further code words. Therefore the procedure can be taken as far as x_N.

Conversely, the tree representing any prefix condition code can be embedded in a full tree whose order is the largest of the code word lengths. A terminal node of order ℓ_k, in the tree representing the code, has stemming from it a fraction $\frac{1}{M^{\ell_k}}$ of the terminal nodes in the full tree. But the sets of terminal nodes in the full tree stemming from different terminal nodes in the tree are disjoint on account of the prefix condition. Hence these fractions can sum to at most 1 which yields the equation (2.1).

End of Proof 2.

To show whether a given sequence of code word lengths is acceptable as the lengths of the code words for an instantaneous code; examine the following sets of code lengths (Table 2.6).

Source	Code a	Code b	Code c	Code d
s_1	00	0	0	0
s_2	01	01	10	10
s_3	10	011	110	110
s_4	11	0111	11	1110

TABLE 2.6: Sets of code lengths

In binary code letters, the inequality becomes

$$\sum_{k=1}^{N} \frac{1}{2^{\ell_k}} \leq 1$$

For code a

$$\sum_{k=1}^{4} \frac{1}{2^{\ell_k}} = \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = 1$$

which satisfies the Kraft's inequality. This means that there is an instantaneous binary code with four code words each of length 2. For code b

$$\sum_{k=1}^{4} \frac{1}{2^{\ell_k}} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = \frac{15}{16}$$

which satisfies the Kraft's inequality. For code c

$$\sum_{k=1}^{4} \frac{1}{2^{\ell_k}} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{4} = \frac{9}{8}$$

Here, the lengths do not satisfy the Kraft's inequality and therefore it could not possibly be an instantaneous code.

Kraft's inequality can help to find a code word length for a set of

words having the same length. For example, suppose that there are 4 code words, then

$$\sum_{k=1}^{4} \frac{1}{2^{\ell_k}} = 4 * \frac{1}{2^{\ell}} \leq 1$$

or $\ell \geq 2$.

Therefore, the length should at least be equal to 2 in order to satisfy Kraft's inequality (see Table 2.6 code a).

Kraft's theorem provides a sufficient condition on the word lengths of a code by showing that it is possible to construct a prefix condition code with the prescribed word lengths. However, it does not say that any code satisfying the inequality is a prefix condition code. For example, in Table 2.6, code b is not a prefix condition code. Nevertheless it satisfies Kraft's inequality. So, it is possible to construct a prefix condition code with the prescribed word lengths (see code d).

The relation between a uniquely decodable code and the Kraft's inequality is provided by the following theorem (Johns, 79).

Theorem (McMillan): If a code is uniquely decodable with code words of lengths $\ell_1, \ell_2, \ldots, \ell_N$ then the inequality of Kraft's theorem is satisfied.

Proof: Let n be any arbitrary positive integer, then

$$\left(\sum_{k=1}^{N} \frac{1}{M^{\ell_k}} \right)^n = \sum_{k_1=1}^{N} \frac{1}{M^{\ell_{k_1}}} \sum_{k_2=1}^{N} \frac{1}{M^{\ell_{k_2}}} \cdots \sum_{k_n=1}^{N} \frac{1}{M^{\ell_{k_n}}}$$

$$= \sum_{k_1=1}^{N} \sum_{k_2=1}^{N} \cdots \sum_{k_n=1}^{N} \frac{1}{M^{\ell_{k_1} + \ell_{k_2} + \cdots + \ell_{k_n}}}$$

Now $\ell_{k_1} + \ell_{k_2} + \ldots + \ell_{k_n}$ is the number of code letters in a sequence of n code words. Let r_i be the number of sequences of n code words which contain i code letters. Let ℓ_{max} be the largest of $\ell_1, \ell_2, \ldots, \ell_N$. The

value of i could not be less than 1 letter nor more than $n\ell_{max}$. Hence

$$\left(\sum_{k=1}^{N} \frac{1}{M^{\ell_k}}\right)^n = \sum_{i=1}^{n\ell_{max}} \frac{r_i}{M^i}$$

If the code is uniquely decodable, then all code words with a length of i code letters are distinct. Thus

$$r_i \leq M^i$$

i.e. r_i can not exceed the maximum number of different sequences of i code letters which is M^i. Therefore

$$\left(\sum_{k=1}^{N} \frac{1}{M^{\ell_k}}\right)^n \leq \sum_{i=1}^{n\ell_{max}} \frac{M^i}{M^i}$$

$$\leq \sum_{i=1}^{n\ell_{max}} 1$$

$$\leq n\ell_{max}$$

$$\sum_{k=1}^{N} \frac{1}{M^{\ell_k}} \leq (n\ell_{max})^{1/n}$$

By allowing $n \to \infty$, the right-hand side tends to unity. Therefore,

$$\sum_{k=1}^{N} \frac{1}{M^{\ell_k}} \leq 1$$

which satisfies Kraft's inequality.

2.5 ENTROPY

It is mentioned, in Section 2.2, that for a code alphabet M there are M^L equally likely words in which each word contains a number L of separate code letters (not necessarily all different). Each of these words can be assigned to a different letter from the source alphabet S. Thus the amount of information gained when a source letter is encoded is represented by L code letters. Hence it is possible to measure the amount of information per source letter.

Let D be the number of different words, let M=2 i.e. (0,1). Then $D=2^L$. To measure the information, the logarithmic method is used (Young, 71; Gallager, 68)

$$\log_2 D = L \log_2 2$$

since $\log_2 2=1$

then $$L = \log_2 D$$

Thus L is equal to the logarithm to the base 2 of the number D of different equally likely words. The probability p_i of any one of the D different equally likely words is $\frac{1}{D}$. So

$$L = -\log_2 \frac{1}{D}$$

$$= -\log_2 p_i$$

which means that, the amount of information obtained from a source letter s_i is equal to $-\log_2(p_i)$. In general, let $s_1, s_2, \ldots, s_N$ be a sequence of N different source letters. Each letter has a probability $p(s_i)=p_i$, with $0 \leq p_i \leq 1$ and $\sum_{i=1}^{N} p_i = 1$ then the self-information of the letter s_i is defined as (Johns, 79)

$$I(s_i) = -\log_2 p_i$$

The base for the logarithm fixes the unit of information. Namely, it determines the numerical scale used to measure information. With base 2, the self-information is measured in bits (an abbreviation of binary digits). Since $0 \leq p_i \leq 1$, then $I(s_i)$ is always positive and its value depends on the probability of the letter concerned. That is $I(s_i)$ increases when p_i decreases, and vice versa. For example, suppose that $p_i = \frac{1}{2}$ then

$$I(s_i) = -\log_2(\tfrac{1}{2})$$
$$= 1 \text{ bit}$$

one bit is the amount of information obtained when one of two possible equally likely letters is received. Let $p_i = \frac{1}{4}$, then

$$I(s_i) = -\log_2(\tfrac{1}{4})$$
$$= 2 \text{ bits}$$

Two bits are obtained when one letter is chosen at random from 4 different letters. Note that, when the probability is decreased, the self-information is increased.

The average amount of information obtained per letter from a source S, or the average of the self-information, is called the entropy of S (Johns, 79) i.e.

$$H(S) = \sum_{i=1}^{N} p_i I(s_i)$$

or

$$H(S) = -\sum_{i=1}^{N} p_i \log_2 p_i \quad \text{bits}$$

As an example, consider the source $S=(s_1,s_2,s_3,s_4)$ with $p_1=\frac{1}{2}, p_2=\frac{1}{4}, p_3=p_4=\frac{1}{8}$. Then, the average amount of information obtained per source letter is

$$H(S) = -\sum_{i=1}^{4} p_i \log_2 p_i$$
$$= -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{8}\log_2\frac{1}{8} - \frac{1}{8}\log_2\frac{1}{8}$$
$$= \frac{1}{2} + \frac{1}{2} + \frac{3}{8} + \frac{3}{8}$$
$$= 1\frac{3}{4} \text{ bits.}$$

If H(S) is the entropy of a source letter, then a sequence of source letters can not be represented by a sequence of bits using fewer than H(S) bits per source letter on the average (Gallager, 68). However, it can be represented by a sequence of bits close to H(S) bits per source letter on the average. It is mentioned that the self-information of a letter increases when the uncertainty of that letter grows (the probability of the letter decreases). Hence, the entropy may be regarded as an average amount of uncertainty.

From the definition of the entropy, $\log_2 p_i \leq 0$ for all $0 \leq p_i \leq 1$, thus it can never be negative. Let one letter s_j have probability one ($p_j = 1$) and the remaining letters have zero probabilities in a source S of N letters. Then

$$H(S) = -\sum_{i=1}^{N} p_i \log_2 p_i$$

$$= -(0+0+\ldots+1 \log_2(1)+0+\ldots+0)$$

since $\log_2(1) = 0$

then $H(S) = 0$

i.e. the amount of uncertainty is zero; namely it is certain that the letter s_j is received.

As well as a lower bound of zero there is an upper bound.

which the entropy will never exceed. This limit is $\log_2 N$ (Abramson, 63). Consider the quantity

$$\log_2 N - H(S) = \log_2 N + \sum_{i=1}^{N} p_i \log_2 p_i$$

$$= \sum_{i=1}^{N} p_i \log_2 N + \sum_{i=1}^{N} p_i \log_2 p_i$$

$$= \sum_{i=1}^{N} p_i \log_2 N p_i$$

since $\log_2 Np_i = \frac{\ell nNP_i}{\ell n2}$

and $\log_2 e = \frac{1}{\ell n2}$

therefore $\log_2 Np_i = \ell nNP_i \log_2 e$

Thus $$\log_2 N - H(S) = \log_2 e \sum_{i=1} p_i \ell n Np_i \qquad (2.3)$$

From the relation between the natural logarithm of a variable x and the value (x-1), it is found that

$$\ell n\, x \lesssim x-1 \qquad (2.4)$$

with equality if and only if x=1.

By multiplying (2.4) by (-1)

$$\ell n \frac{1}{x} \gtrsim 1-x \qquad (2.5)$$

Assume $$\frac{1}{x} = Np_i .$$

$$x = \frac{1}{Np_i}$$

From (2.5) $$\ell n\, Np_i \gtrsim 1 - \frac{1}{Np_i} \qquad (2.6)$$

From (2.3) and (2.6)

$$\log_2 N - H(S) \gtrsim \log_2 e \sum_{i=1}^{N} p_i (1 - \frac{1}{Np_i})$$

$$\gtrsim \log_2 e (\sum_{i=1}^{N} p_i - \frac{1}{N} \sum_{i=1}^{N} 1)$$

$$\gtrsim \log_2 e (1-1)$$

$$\gtrsim 0$$

From (2.4), the equality obtains when $\frac{1}{Np_i} = 1$ for all i. Therefore $H(S) = \log_2 N$ only when $p_i = \frac{1}{N}$ for all i. Hence the maximum value of the entropy is exactly $\log_2 N$ if and only if all the source letters have equal probabilities. For example, the entropy of 4 source letters

s_1, s_2, s_3, s_4; each having a probability equal to $\frac{1}{4}$ is

$$H(S) = -\sum_{i=1}^{4} p_i \log_2 p_i$$

$$= -4 * \frac{1}{4} \log_2 \frac{1}{4}$$

$$= 2 \text{ bits}$$

$$= \log_2 4$$

2.6 AVERAGE LENGTH OF A CODE WORD

Let $S=\{s_1,s_2,\ldots,s_N\}$ be a sequence of source letters with their corresponding probabilities $p_1,p_2,\ldots,p_N$. Let $c_1,c_2,\ldots,c_N$ be a sequence of code words; such that each s_i can be transformed into a code word c_i, $1\leq i\leq N$. Let $\ell_1,\ell_2,\ldots,\ell_N$ be the lengths of the code words. Then the average length of a code word ℓ_{av} is defined as:

$$\ell_{av} = \sum_{i=1}^{N} p_i \ell_i$$

The relationship between the average code length (ℓ_{av}) and the entropy ($H(S)$) can be obtained as follows (Abramson, 63; Hamming, 80). From the Kraft inequality (Section 2.4), let M=2,

$$y = \sum_{i=1}^{N} \frac{1}{2^{\ell_i}} \leq 1$$

$$x_i = \frac{2^{-\ell_i}}{y}$$

be regarded as a probability distribution where

$$\sum_{i=1}^{N} x_i = 1$$

Consider the expression involving two probability distributions x_i and p_i

$$\sum_{i=1}^{N} p_i \log_2\left(\frac{x_i}{p_i}\right) = \frac{1}{\ell n 2} \sum_{i=1}^{N} p_i \ell n \left(\frac{x_i}{p_i}\right)$$

From the relation (2.4)

$$\begin{aligned}
\sum_{i=1}^{N} p_i \log_2\left(\frac{x_i}{p_i}\right) &\leq \frac{1}{\ell n 2} \sum_{i=1}^{N} p_i \left(\frac{x_i}{p_i} - 1\right) \\
&\leq \frac{1}{\ell n 2} \sum_{i=1}^{N} (x_i - p_i) \\
&\leq \frac{1}{\ell n 2} \left(\sum_{i=1}^{N} x_i - \sum_{i=1}^{N} p_i\right) \\
&\leq 0
\end{aligned}$$

$$\text{or} \quad \sum_{i=1}^{N} p_i \log_2 \frac{1}{p_i} \leq \sum_{i=1}^{N} p_i \log_2 \left(\frac{1}{x_i}\right)$$

$$\text{since} \qquad H(S) = \sum_{i=1}^{N} p_i \log_2 \left(\frac{1}{p_i}\right)$$

$$\text{Then} \qquad H(S) \leq \sum_{i=1}^{N} p_i \log_2 \left(\frac{1}{x_i}\right)$$

$$\leq \sum_{i=1}^{N} p_i \log_2 \left(\frac{y}{2^{-\ell_i}}\right)$$

$$\leq \sum_{i=1}^{N} p_i \left(\log_2 y - \log_2 2^{-\ell_i}\right)$$

$$\leq \log_2 y + \sum_{i=1}^{N} p_i \ell_i \log_2 2$$

since $y \leq 1$, then $\log_2 y \leq 0$.

$$\text{Therefore } H(S) \leq \sum_{i=1}^{N} p_i \ell_i \text{ or } H(S) \leq \ell_{av}. \qquad (2.7)$$

The necessary conditions for the equality of (2.7) are

$$y = 1$$

and $$p_i = x_i \text{ , for all } i$$

$$= \frac{2^{-\ell_i}}{y}$$

$$= 2^{-\ell_i}$$

By taking logarithms to the base 2 of both sides

$$\log_2 p_i = -\ell_i$$

or $$-\log_2 p_i = \ell_i$$

Thus, for an instantaneous code, ℓ_{av} must be greater than or equal to the entropy. Furthermore, ℓ_{av} can achieve the equality if and only if $\ell_i = -\log_2 p_i$ for all i.

Given a sequence of source letters and their corresponding probabilities,

a coding technique known as (Shannon-Fano coding) can be applied to obtain the code word length for each source letter directly from the corresponding probability, such that

$$-\log_2 p_i \leq \ell_i < -\log_2 p_i + 1$$

The implementation of this method is trivial and satisfies (2.7). However, it does not generate optimal codes as it will be found in section 2.7.

Let s_1, s_2, s_3, s_4 be a sequence of source letters, let the probability of each source letter $p_i = \frac{1}{4}$, then

$$H(S) = -\sum_{i=1}^{4} p_i \log_2 p_i$$

$$= -4 * \frac{1}{4} \log_2 \frac{1}{4}$$

$$= 2 \text{ bits}$$

since $\ell_i \geq -\log_2 p_i$

$$\geq -\log_2 \frac{1}{4}$$

$$\geq 2 \text{ bits}$$

then the minimum value that ℓ_i can get is 2

$$\ell_{av} = \sum_{i=1}^{4} p_i \ell_i$$

$$= 4 * \frac{1}{4} * 2$$

$$= 2 \text{ bits}$$

Therefore $H(S) = \ell_{av}$

Suppose that the probabilities of s_1, s_2, s_3, s_4 are $\frac{1}{2}, \frac{1}{3}, \frac{1}{12}, \frac{1}{12}$ respectively. Then

$$H(S) = -(\frac{1}{2} \log_2 \frac{1}{2} + \frac{1}{3} \log_2 \frac{1}{3} + \frac{2}{12} \log_2 \frac{1}{12})$$

$$= 1.623 \text{ bits}$$

$$\ell_1 \geq -\log_2 \frac{1}{2}$$

$$\geq 1 \text{ bit}$$

$$\ell_2 \geq -\log_2 \frac{1}{3}$$

$$\geq 1.58$$

The closest integer to 1.58 is 2, so ℓ_2=2 bits.

$$\ell_3 \geq -\log_2 \frac{1}{12}$$

$$\geq 3.58$$

so $\ell_3 = \ell_4 = 4$ bits

$$\ell_{av} = \sum_{i=1}^{4} p_i \ell_i$$

$$= \frac{1}{2} + 1 + \frac{1}{3} * 2 + \frac{1}{12} * 4 + \frac{1}{12} * 4$$

$$= 1.833 \text{ bits}$$

Therefore $H(S) \leq \ell_{av}$.

The bounds of ℓ_{av} are formally provided by the following theorem (Johns, 79; Gallager, 68).

Theorem: For any uniquely decodable code

$$\ell_{av} \geq \frac{H(u)}{\log M}.$$

Code words can always be chosen to satisfy the prefix condition and

$$\ell_{av} < \frac{H(u)}{\log M} + 1$$

where u is a set of letters with their probabilities.

Proof: Let $p_1, p_2, \ldots, p_N$ be the probabilities of the source letters, and let $\ell_1, \ell_2, \ldots, \ell_N$ be the code word lengths.

$$H(u) - \ell_{av} \log M = \sum_{i=1}^{N} p_i \log \frac{1}{p_i} - \sum_{i=1}^{N} p_i \ell_i \log M$$

$$= \sum_{i=1}^{N} p_i \log \frac{1}{p_i} + \sum_{i=1}^{N} p_i \log \frac{1}{M^{\ell_i}}$$

$$= \sum_{i=1}^{N} p_i \log \frac{1}{p_i M^{\ell_i}}$$

since $\ell n\ x = \frac{\log x}{\log e}$

$$\log x = \ell n\ x \log e$$

Using the inequality $\ell n\ x \leq x-1$ for $x>0$

or $\log x \leq (x-1) \log e$

$$H(u) - \ell_{av} \log M \leq \log e \sum_{i=1}^{N} p_i \left(\frac{1}{p_i M^{\ell i}} - 1\right)$$

$$\leq \log e \left(\sum_{i=1}^{N} \frac{1}{M^{\ell i}} - \sum_{i=1}^{N} p_i\right)$$

$$\leq \log e \left(\sum_{i=1}^{N} \frac{1}{M^{\ell i}} - 1\right)$$

since the Kraft's inequality

$$\sum_{i=1}^{N} \frac{1}{M^{\ell i}} \leq 1$$

is valid for any uniquely decodable code, then,

$$H(u) - \ell_{av} \log M \leq 0$$

or

$$\ell_{av} \geq \frac{H(u)}{\log M}$$

The equality occurs only when $p_i = \frac{1}{M^{\ell i}}$, $1 \leq i \leq N$.

In the second part of the theorem, only the probabilities p_i of the source are given and it has to be shown that lengths can be obtained for the code words of a code satisfying the stated condition. If the code word lengths did not have to be integers, then ℓ_i could be obtained to satisfy,

$$p_i = \frac{1}{M^{\ell i}} \quad , \ 1 \leq i \leq N.$$

However, by choosing ℓ_i to be the integer satisfying

$$\frac{1}{M^{\ell i}} \leq p_i < \frac{1}{M^{\ell i - 1}} \quad , \ 1 \leq i \leq N \qquad , \tag{2.8}$$

Summing over N, the left-hand side of (2.8) becomes

$$\sum_{i=1}^{N} \frac{1}{M^{\ell i}} \leq \sum_{i=1}^{N} p_i$$

$$\leq 1$$

which satisfies the Kraft's inequality. Therefore a prefix condition code exists with these lengths. Taking the logarithm of the right-hand side of (2.8)

$$\log p_i < \log \frac{1}{M^{\ell_i - 1}}$$

$$\log p_i < (1-\ell_i) \log M$$

$$-\log p_i > (\ell_i - 1) \log M$$

$$\ell_i < \frac{-\log p_i}{\log M} + 1$$

Multiplying the above by p_i and summing over N, then

$$\sum_{i=1}^{N} p_i \ell_i < - \sum_{i=1}^{N} \frac{p_i \log p_i}{\log M} + \sum_{i=1}^{N} p_i$$

$$\ell_{av} < \frac{H(u)}{\log M} + 1$$

which satisfies the stated condition.

From the second example in this section,

$$H(S) = 1.623 \text{ bits}$$

and $$\ell_{av} = 1.833 \text{ bits}$$

therefore $$1.623 \leq 1.833 < 1.623 + 1$$

which satisfies the conditions in the above theorem. So, it is possible to construct a prefix condition code from the specified lengths of the code words.

2.7 HUFFMAN CODES

In any code, the average code word length can not be less than the entropy of the code. But it can be very close to it. This can be achieved only when the lengths of the code words are variable. That is, by assigning short code words to highly probable source letters, and long code words to the least probable source letters (see Section 2.6). However, there is no guarantee that an optimal coding can be obtained from the above assignment. For example, consider the codes in Table 2.7. Both codes (a and b) are uniquely decodable, and satisfy the Kraft's inequality.

Source	p_i	$-\log_2 p_i$	ℓ_i	code a	code b
s_1	$\frac{1}{2}$	1	1	0	0
s_2	$\frac{1}{3}$	1.58	2	10	10
s_3	$\frac{1}{12}$	3.58	4	1100	110
s_4	$\frac{1}{12}$	3.58	4	1110	111

TABLE 2.7: Uniquely decodable codes

The entropy of the source is

$$H(S) = -\sum_{i=1}^{4} p_i \log p_i$$

$$= -(\frac{1}{2} * 1 + \frac{1}{3} * 1.58 + \frac{2}{12} * 3.58)$$

$$= 1.623 \text{ bits.}$$

The average length of the code a is

$$\ell_{av} = \sum_{i=1}^{4} p_i \ell_i$$

$$= \frac{1}{2} * 1 + \frac{1}{3} * 2 + \frac{1}{12} * 4 + \frac{1}{12} * 4$$

$$= 1.833 \text{ bits.}$$

It is bounded by $H(S) \leq \ell_{av} < H(S) + 1$

To find the average length of the code b

$$\bar{\ell}_{av} = \frac{1}{2} * 1 + \frac{1}{3} * 2 + \frac{1}{12} * 3 + \frac{1}{12} * 3 .$$

$$= 1.667 \text{ bits.}$$

Again, it is bounded by

$$H(S) \lesssim \bar{\ell}_{av} < H(S) + 1$$

Since the entropy H(S)=1.623 bits represents the minimum average length that can be achieved, then the nearer the average length of a code to the entropy, the more optimal a code would be. Therefore code b is more nearly optimal than code a. A well known optimal code is called the Huffman code (Huffman, 52; Maurer, 69; Wells, 72; Abramson, 63; Hamming, 80). The method of constructing Huffman codes is based on the construction of a probability tree (for simplicity, a binary tree is assumed).

Let $s_1, s_2, \ldots, s_N$ be a sequence of source letters, and $p_1, p_2, \ldots, p_N$ be a set of probabilities such that $p(s_i)=p_i$, and $\sum_{i=1}^{N} p_i = 1$. Arrange the probabilities in descending order, i.e.

$$p_1 \geq p_2 \geq \ldots \geq p_N$$

these will represent the leaves of the tree. Form a new node by grouping the two least probable nodes. Now, the new node has a probability equal to the sum of the probabilities of the nodes forming it. The remaining leaves and the new node will form a new set of nodes which contains one less node. The nodes should be rearranged to keep the probabilities in descending order. Form a node as above. Repeat this process until the tree is completed (i.e. until the last node (the root) has a probability equal to one). For a given N source letters the procedure is terminated after N-1 groupings. Assign the digits 0 and 1 to the branches at each node in an arbitrary way. The code of each source letter is determined

by listing the digits which lie between the root of the tree and the leaf that corresponds to the source letter. Any source letter may be reached from the root in one and only one way. Fig. 2.7 shows an example of a binary tree constructed for 6 source letters. By assigning the digits 0 and 1 to the branches of each node in the tree, a Huffman code is

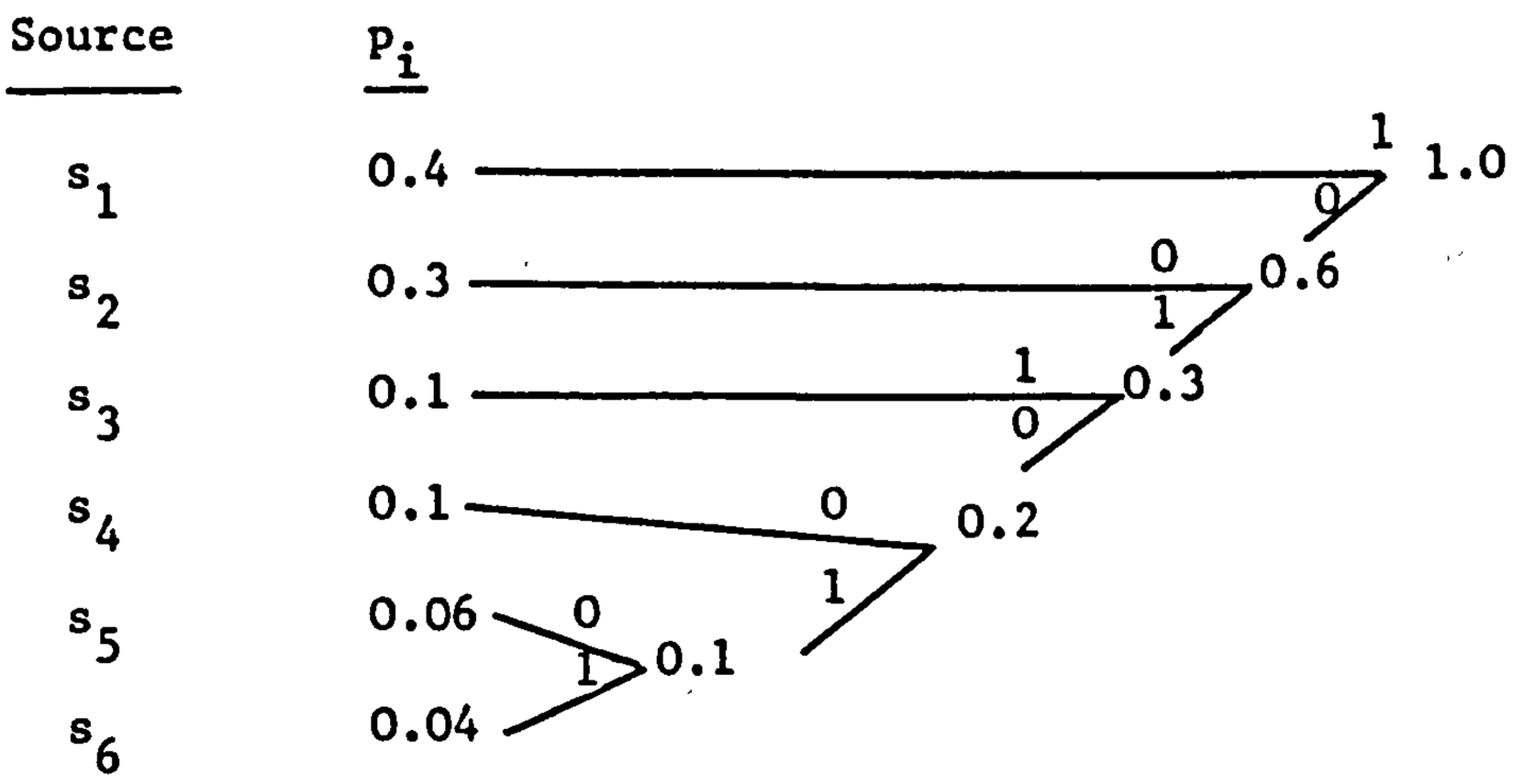


FIGURE 2.7: Generating a binary tree

generated for the specified source. Fig. 2.8 illustrates the code of each source letter obtained from the above binary tree.

Source	p_i	code
s_1	0.4	1
s_2	0.3	00
s_3	0.1	011
s_4	0.1	0100
s_5	0.06	01010
s_6	0.04	01011

$$\sum_{i=1}^{6} p_i \ell_i = 2.2 \text{ bits}$$

FIGURE 2.8: Huffman code

A Huffman code is a prefix condition code in which no code word is a prefix of any other code word. The shortest code word is assigned to the

most frequent source letter. That is when the probabilities are arranged in descending order, the lengths of code words come out in ascending order. The last two codes are identical except the last digit. Huffman code is a minimum redundancy code. That is the average number of bits required to encode a source letter is a minimum.

It is easy to construct manually the probability tree, and from it, the Huffman code for a sequence of source letters. Nevertheless, Schwartz and Kallick (Schwartz and Kallick, 64) described a computer program which generates an optimal code based upon Huffman's method. Generally, the program reads a set of frequencies of source letters, constructs a frequency tree and then assigns codes.

To prove that Huffman code is optimal, assume that there is a shorter code with average code length L' and

$$L' < L$$

where L is the length of Huffman code. Construct a coding tree for each code, and try to compare them. The two least probable nodes have identical codes except the last digit, which means that they have the same length. Suppose that the nodes are n_p and n_q with the probabilities p_p and p_q respectively. Assume that the code lengths are ℓ_p and ℓ_q. Then $\ell_p = \ell_q$ so the average code length of these nodes would be

$$\ell_p p_p + \ell_p p_q = \ell_p (p_p + p_q)$$

The common node (the new node occurs as a consequence of grouping n_p and n_q) which is in the higher level of the tree has a code length equal to $(\ell_p - 1)$ and a probability equal to $(p_p + p_q)$, so the average code length would be

$$(\ell_p - 1)(p_p + p_q) = \ell_p (p_p + p_q) - (p_p + p_q)$$

Therefore, as the tree is reduced, the code length is shortened by the amount:

$$p_p + p_q$$

This process can be done on the next two least probable nodes, ... and so on. By applying this to both the coding trees, it is easy to see that both are decreased by the same amount. Thus the amount of inequality between their lengths remains unchanged. Since in the Huffman code the code length of the last two nodes is 1; for the other, it must be less than 1, which is impossible. Therefore, the Huffman code is the shortest possible code.

2.8 MINIMIZING THE LONGEST CODE AND TOTAL NUMBER OF DIGITS

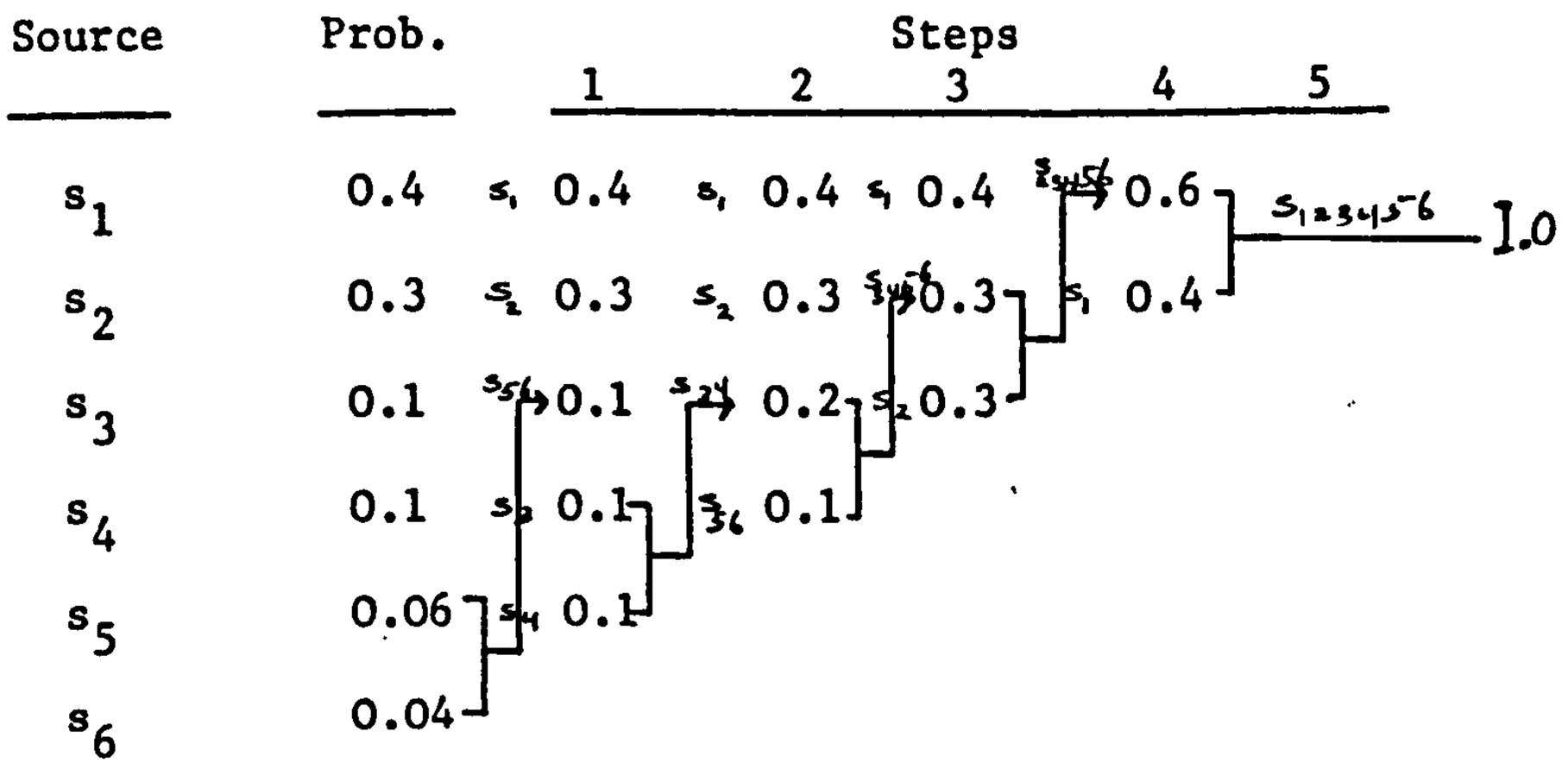
It has been shown, in Section 2.3, that the probability tree is constructed by merging two nodes which have the lowest probabilities, and this process of merging continues until only one node (the root) remains. Each branch of a node has assigned a digit. Therefore the code of a source letter is the sequence of digits along the path which starts from the root and terminates in the leaf. In other words, it is equal to the number of mergings on the path of a source letter. So, if it is possible to reduce the number of mergings on different paths in the tree, then it will minimize some code word lengths, and hence, the number of digits of all code words will be minimized.

In Huffman's method, the merge among equiprobable nodes (including the leaves) can be done by choosing any two nodes without affecting the average code length. That is, when two nodes are merged, and there exists a number of nodes which have probabilities equal to the probability of the new node, then it can immediately merge this new node with any other node of the same probability. Hence, a new digit is added to the code words of the specified source letters.

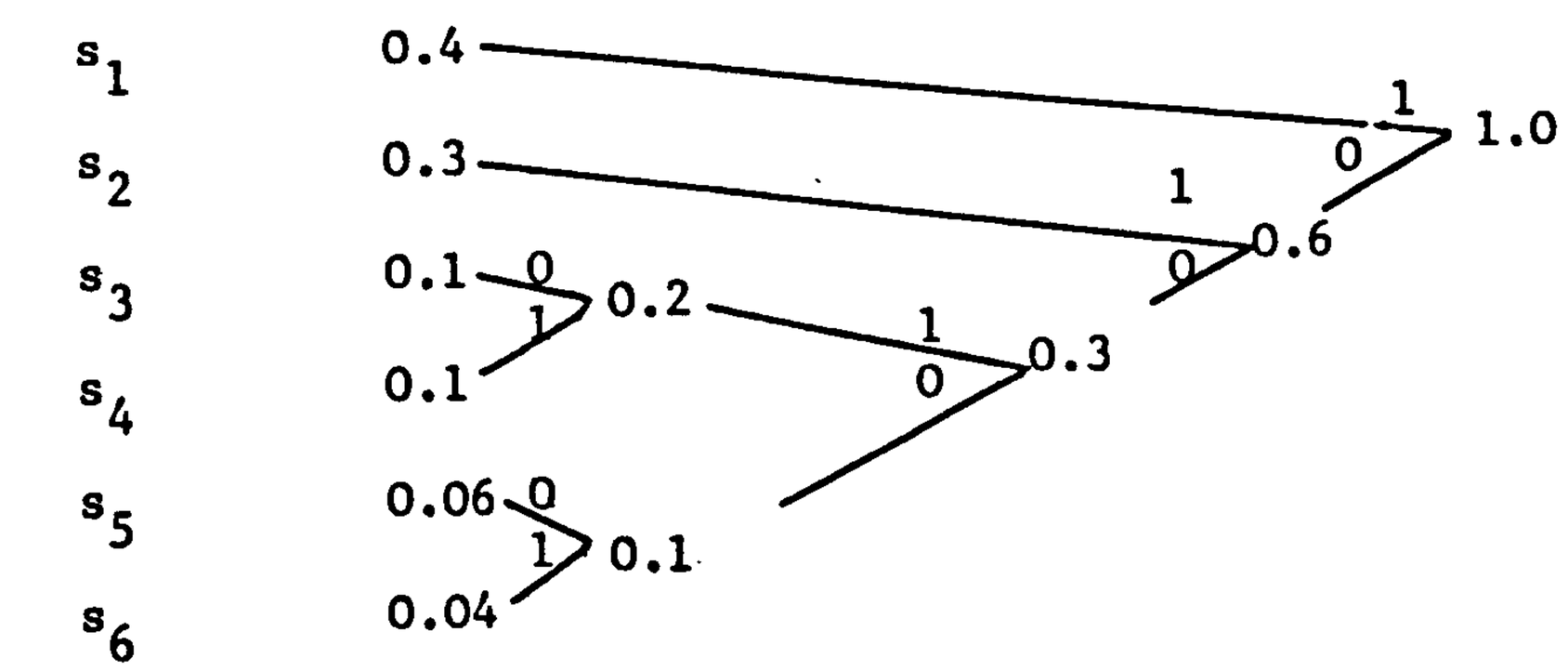
Schwartz (Schwartz, 64) showed a method of merging equiprobable nodes called bottom merge, such that the average code length remains unaffected, but minimisesthe longest code word and the total number of digits. The way is to place the new node at the top of the nodes which have equal probabilities (Fig. 2.9a). This will avoid, if possible, an immediate merging with another node, and therefore not assigning a new digit. For instance, consider the example in the previous section (Figures 2.7 and 2.8), It is easy to notice that, after merging s_5 and s_6, the new node

(with probability 0.1) is immediately merged with s_4. Consequently, a new digit is added to the codes of both s_6 and s_5. However, by applying a bottom merge, the new node will be placed above s_3, and the next merge will be between s_4 and s_3 (Fig. 2.9b). The new code words of the same source letters is mentioned in Fig. 2.10.

By comparing the codes in Figures (2.8 and 2.10), it is found that the codes of both s_5 and s_6 are reduced from 5 bits to 4 bits. The total number of digits of the code in Fig. 2.8 is 20, whereas, in Fig. 2.10 it is 19. Notice that the average code length is the same in both codes.



a. An illustration of bottom merging



b. A probability tree (binary tree)

FIGURE 2.9: Generating a probability tree by using bottom merging

Source	Prob.	Code
s_1	0.4	1
s_2	0.3	01
s_3	0.1	0010
s_4	0.1	0011
s_5	0.06	0000
s_6	0.04	0001

$$\sum_{i=1}^{6} p_i \ell_i = 2.2 \text{ bits}$$

FIGURE 2.10: A Huffman code

CHAPTER 3

ENCODING PROBABILISTIC CONTEXT-FREE LANGUAGES

A computer program (data file) usually contains data causing redundancy such as spaces, zeros, keywords, common words, and comments, which occupy a considerable space compared with the overall storage used by the program. It can be transformed into another file which reflects the same information as in the original file. The transformed file can occupy less storage, and is called a compressed file. The transformation method is called a data compression method. Different techniques are used to compress data files (Martin, 76). Each technique used depends on the nature of the file, whether it contains a lot of spaces and zeros, or contains many common words ... and so forth. This chapter illustrates a special type of data file in which the data is partially generated by a context-free grammar, and tries to explain different compression methods implemented on such files.

In Section 3.1, some definitions are given concerning the language and grammars. The derivation of a string of symbols from a grammar is explained in Section 3.2. Section 3.3 explains the rightmost derivations. In Section 3.4, the probability of a string and hence a language generated from a probabilistic grammar is illustrated. Compression and decompression of data is introduced in Section 3.5. Ways of encoding a data file character by character, are explained in Section 3.6. Sometimes instead of encoding one character at a time a string of characters (word) is encoded. This is explained in Section 3.7. In Section 3.8, an encoding of a structured data file is shown. Finally, different ways of evaluating the encoding methods are explained in Section 3.9.

3.1 DEFINITIONS

An alphabet of a language is any finite set (T) of symbols. From this set, strings of finite lengths can be composed. Each string is called a sentence. A language over a set (T) of terminal symbols is a subset of all strings (sentences) over T. Usually, these symbols are not all of equal importance, and hence we can apply a measure on each one of them (Booth and Thompson, 73). If each measure is bounded by zero and one, and the total is equal to one, then it is called a probabilistic measure of the symbol. Let T be a finite set. A language L over a set (T) is a probabilistic language if there exists a probability measure p(x) for each $x \in L$ such that $0 \leq p(x) \leq 1$ and $\sum_{x \in L} p(x) = 1$ (Thompson and Booth, 71; Thompson, 74). $p(x)=0$ means that x will never occur. If x is certain to occur then $p(x)=1$.

Although any subset of strings over T is a language, the emphasis will be placed on a structured language generated by a type of grammar called context-free grammars. McGettrick (McGettrick, 80) explains in detail the relations between the languages and the grammars.

A context-free grammar G is a four-tuple G=(N,T,R,S) where:

$N=\{v_1, v_2, \ldots, v_k\}$ is a finite set of non-terminal symbols;

$T=\{a_1, a_2, \ldots, a_m\}$ is a finite set of terminal symbols;

$R=\{r_1, r_2, \ldots, r_n\}$ is a finite set of productions of the form

$$v_i := \alpha_j \ , \quad v_i \in N, \ \alpha_j \in (N \cup T)^*$$

where (N U T) is a finite non-empty set of grammar symbols; (N U T)* is either non-empty set or empty;

S is an initial symbol.

From now on, every grammar mentioned is considered to be a context-free grammar. The following notation will be used:

{A,B,C,...,Y,Z} to denote non-terminal symbols;

{a,b,c,...,y,z} to denote terminal symbols; and

{α,β,γ,...,ψ,ω} to denote sets of grammar symbols.

The set of strings of terminal symbols generated by a context-free grammar is called a context-free language. Another definition of a context-free language is mentioned in the next section.

For example, let v={E,F}; T={i,d,+,*,(,)}; S=E; and the set of productions

E:= E + F

E:= E * F

E:= F

F:= (E)

F:= i

F:= d

then the grammar is a context-free grammar. The strings i+d, (d*i), i+d*d,... are subsets of the language generated by the above grammar.

If each production in a context-free grammar is assigned a probability then the grammar is a stochastic (probabilistic) context-free grammar (Hutchins, 72a; Thompson and Booth, 71; Thompson, 74) which is a five tuple G=(N,T,R,P,S) where:

$N=\{v_1,v_2,\ldots,v_k\}$ is a finite set of non-terminal symbols;

$T=\{a_1,a_2,\ldots,a_m\}$ is a finite set of terminal symbols;

$R=\{r_1,r_2,\ldots,r_n\}$ is a finite set of productions of the form

$$v_i := \alpha_j \ , \quad v_i \in N, \quad \alpha_j \in (N \cup T)^*$$

For each non-terminal symbol, there is a group of productions R_i, i=1,2,...,k such that all productions in each group have the same v_i.

$$p = \{p_1, p_2, \ldots, p_n\}$$

is a finite set of probabilities, p_j is the probability that r_j is chosen; $S=v_1$ is the initial symbol.

A probabilistic grammar is said to be normalized (proper) (Huang and Fu, 71; Thompson, 74) if and only if

$$\sum_{R_i} p_j = 1$$

for all productions which have the same left-hand side symbol. For example, the grammar

E:= E+F 0.3

E:= E*F 0.2

E:= F 0.5

F:= (E) 0.2

F:= i 0.4

F:= d 0.4

is a proper probabilistic grammar.

3.2 DERIVATIONS AND DERIVATION TREES

In a context-free grammar G, a production of the form

$$A := \alpha$$

means that at certain step in the parsing process the non-terminal symbol (A) can be substituted by a set of grammar symbols (α). This substitution is called a derivation of α from A, and is written as $A \Rightarrow \alpha$ where $A:=\alpha$ is a production in G. So $\alpha A \beta \Rightarrow \alpha\gamma\beta$ means that the string $\alpha A \beta$ directly derives the string $\alpha\gamma\beta$ if $A:=\gamma$ is a production in G. If there are a sequence of derivations, i.e.

$$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 ,\ldots,\Rightarrow \alpha_n$$

this means that α_1 indirectly derives α_n, and can be written in a short form as

$$\alpha_1 \overset{*}{\Rightarrow} \alpha_n$$

If a derivation always occurs on the first non-terminal symbol in a string of grammar symbols, i.e. $\alpha A \beta \Rightarrow \alpha\gamma\beta$ where $A:=\gamma$ is a production and α is a string of terminal symbols or empty, the derivation is called a left most derivation (Aho and Ullman, 77). Top-down parsing methods implement this type of derivation. Details of top-down methods are explained in the next chapter. If the derivation always occurs on the last non-terminal symbol of a string of grammar symbols, i.e. $\alpha A \beta \Rightarrow \alpha\gamma\beta$ where $A:=\gamma$ is a production and β is a string of terminal symbols or empty, then it is called a right-most derivation. Bottom-up parsing methods implement right-most derivations which are explained in detail in the next chapter.

As an example, consider the following grammar:

S:=AA

A:=aA

A:=b

To derive the string aabb from (S) using left-most derivations, use the first production,

$$S \Rightarrow AA$$

The first (A) is a non-terminal symbol, it can derive the string (aA). So $S \Rightarrow AA \Rightarrow aAA$. Again, the first (A) can derive the string (aA); i.e. $S \Rightarrow AA \Rightarrow aAA \Rightarrow aaAA$. Now the first (A) can derive the string (b) by using the last production. Then $S \Rightarrow AA \Rightarrow aAA \Rightarrow aaAA \Rightarrow aabA$. Do the same thing to the last (A):

$$S \Rightarrow AA \Rightarrow aAA \Rightarrow aaAA \Rightarrow aabA \Rightarrow aabb.$$

It can be expressed as

$$S \overset{*}{\Rightarrow} aabb$$

To derive the same string by using right-most derivations, the sequence of derivations would be:

$$S \Rightarrow AA \Rightarrow Ab \Rightarrow aAb \Rightarrow aaAb \Rightarrow aabb$$

A graphical description of a derivation can be expressed in the form of a tree (see Section 2.3) called a parse (derivation) tree. This tree shows the hierarchical syntax structure of sentences that is implied by the grammar (Aho and Ullman, 77).

Let G=(N,T,R,S) be a context-free grammar. A tree is a derivation tree for G if (Hopcroft and Ullman, 69):

1. Every node has a label which is a symbol of either N or T.
2. The label of the root is S.
3. If a node n has at least one branch leaving it, and has label A, then A must be in N.
4. If nodes $n_1, n_2, \ldots, n_k$ are the direct descendents of node A in order from left to right with labels $A_1, A_2, \ldots, A_k$ respectively, then

$$A := A_1 A_2, \ldots, A_k$$

must be a production in R.

To construct a parse tree, let $\alpha_1 \Rightarrow^* \alpha_n$ be realized by $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ where α_1 is the root of the tree. Below α_1 place a list of nodes equal to the number of symbols in α_2. Each node is labelled by a symbol in α_2. Connect the root by a directed line to each new node. Assume the tree has been constructed until α_{i-1}. α_i is derived from α_{i-1} by applying a specific production to a non-terminal symbol (A) in α_{i-1}. Now, below the node labelled (A), list nodes labelled by the right hand side of that specific production, and draw directed lines from (A) to each node in the list. Fig. 3.1 shows the steps of construction a parse tree for the string aabb using left-most derivations.

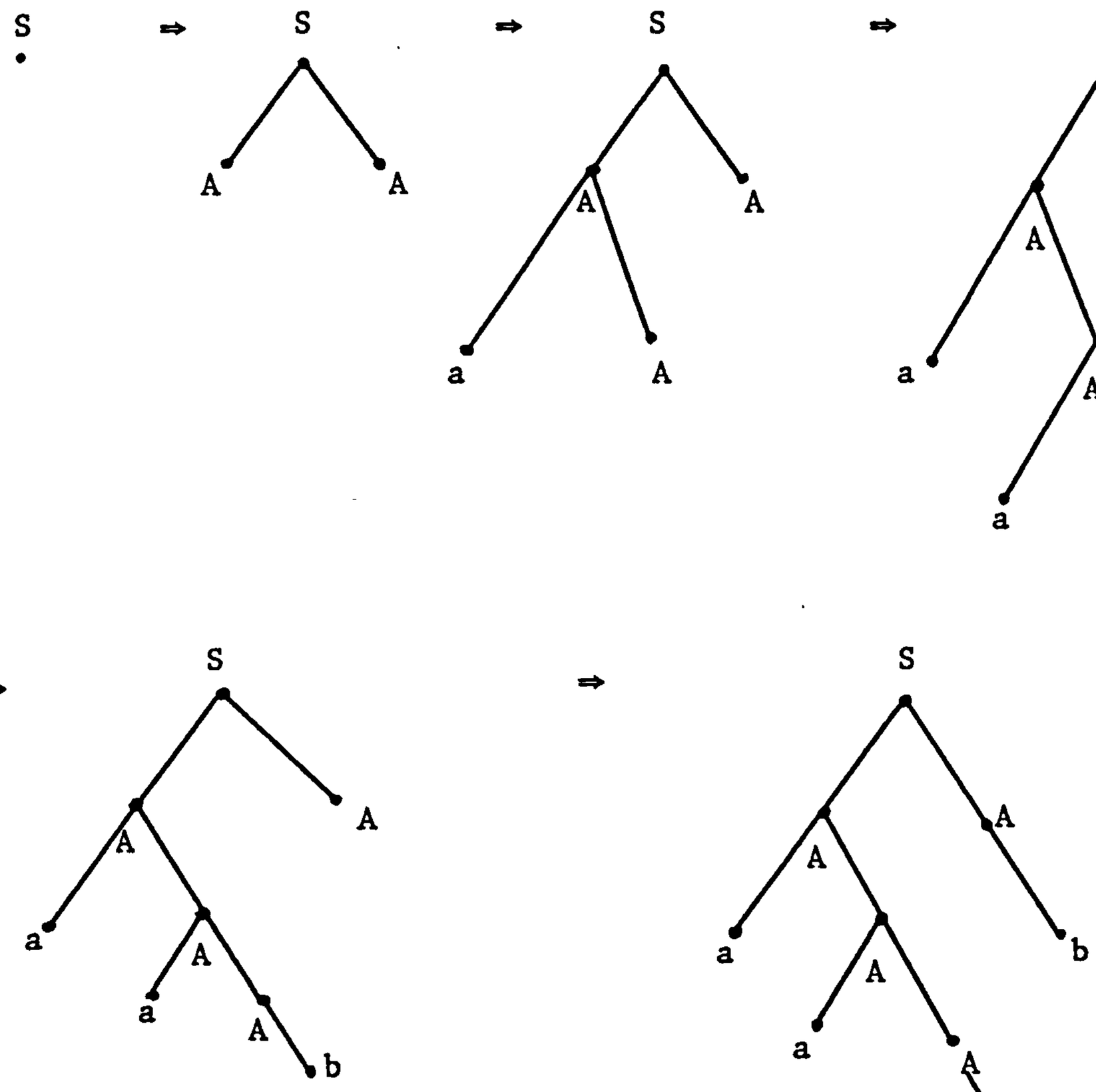


FIGURE 3.1: Building a parse tree

Any sequence of grammar symbols produced as a consequence of a derivation is called a sentential form of the grammar G. If a sentential form has only terminal symbols, then it is called a sentence generated by the grammar. The set of sentences generated from a grammar is called a language. So, a language generated by a context-free grammar G can be defined as $L(G)=\{\alpha \in T^* \mid S \overset{*}{\Rightarrow} \alpha\}$. That is the set of strings of terminal symbols which can be derived from the initial symbol S. If for each $\alpha \in T^*$ there exists a probability $p(\alpha)$ then the language is called a probabilistic language (see Section 3.4).

There is a connection between the probabilities of the productions in the grammar and the probabilities of the sentences in the language which is exploited in Chapter 7.

3.3 HANDLES

At each step in the right-most derivation, the non-terminal symbol is replaced by the right-hand side of the production concerned. For a right sentential form, the right-hand side of that particular production is called a handle (Aho and Ullman, 77; Lewis II, Rosenkrantz, Stearns, 76) which is very important in bottom-up parsing (see next chapter); that is right-most derivations in reverse. For example, consider the derivations

$$S \overset{*}{\Rightarrow} \alpha A \gamma \Rightarrow \alpha \beta \gamma$$

where α is a string of grammar symbols, γ is a string of terminal symbols, and A:=β is a production. Then β is a handle of the right sentential form $\alpha\beta\gamma$ and can be replaced by the symbol (A) to produce the previous right sentential form $\alpha A \gamma$. The production A:=β is called a handle production.

In general, a handle of a right sentential form is the replacement of the right-hand side of the last production applied in a right-most derivation of the right sentential form. The last production applied in a right-most derivation of a right sentential form is a handle production. If a right sentential form can have at most one handle and one handle production, then the grammar is unambiguous.

Consider the grammar in the previous section which derives the string aabb using right-most derivations. The handle, and the handle production of each right sentential form is

Derivations	Handle	Handle Production
S ⇒ AA	AA	S:=AA
⇒ Ab	b	A:=b
⇒ aAb	aA	A:=aA
⇒ aaAb	aA	A:=aA
⇒ aabb	b	A:=b

In bottom-up parsing, the above derivations will occur in reverse, i.e.

Right sentential form	Handle	Handle Production
aabb	b	A:=b
aaAb	aA	A:=aA
aAb	aA	A:=aA
Ab	b	A:=b
AA	AA	S:=AA
S		

This can be interpreted as pruning the derivation tree. The tree leaves corresponding to the right-hand side of the production would be deleted, and the node, labelled by the left-hand side of the production, in which the deleted leaves are the direct descendents, becomes the leaf of the new tree.

3.4 PROBABILISTIC CONTEXT-FREE LANGUAGES

To find the probability of a string α of symbols in a language L generated by a probabilistic grammar, consider the sequence of derivations, i.e.

$$S \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha$$

where S is the start (initial) symbol, α_1 derives from S if $S:=\alpha_1$ is a production with a probability p_1. Now, α_1 occurs with probability p_1. The second production is applied to one non-terminal symbol in α_1, say (A), where $A:=\beta$ is a production with p_2 as its probability. If $\alpha_1=\gamma A\phi$, where γ and ϕ are sets of terminal and grammar symbols respectively, then $\alpha_2=\gamma\beta\phi$ with probability p_1p_2 (Booth and Thompson,73; Huang and Fu 71). The third production is applied to another non-terminal symbol (usually the left-most) from α_2 to result in α_3 with probability $p_1p_2p_3$; and so forth.

The probabilities associated with the productions are assumed to be independent. If k productions are required to derive α, it follows from the independence of the productions that the probability of generating α by means of one of the N derivations is equal to the product of the probabilities of the sequence of the productions used in the derivation, i.e.

$$p(\alpha) = p_1p_2,\ldots,p_k$$

$$= \prod_{i=1}^{k} p_i$$

For an unambiguous grammar, the probability of all strings $\alpha \in L$ would be

$$\sum_{\alpha \in L} p(\alpha) = \sum \prod_{i=1}^{k} p_i$$

If $\sum_{\alpha \in L} p(\alpha) = 1$ for all $\alpha \in L$ of finite length then the production probabilities are said to be consistent, and the grammar is said to be consistent. More discussion of consistent grammars can be found in Chapter 7.

3.5 COMPRESSION AND DECOMPRESSION PHASES

In a compression phase, Fig. 3.2(a), symbols are input to a program called compressor (or encoder) which produces as an output a sequence of code symbols. The codes are output either from some computational transformation or from a table. In the latter case, characters, words, or strings of characters from the input are selected and replaced by code words generally of shorter length than the original elements. The sequence of code symbols is the compressed form of the input symbols. To recover the original information from compressed data, a decompression, Fig. 3.2(b), must be performed. The program (decoder) uses the same technique as the compressor program. If a computational transformation was used during the encoding process, then the decompressor uses the same process but in reverse. If a table is used in the encoding process, then with a related table the decompressor can restore the original input symbols.

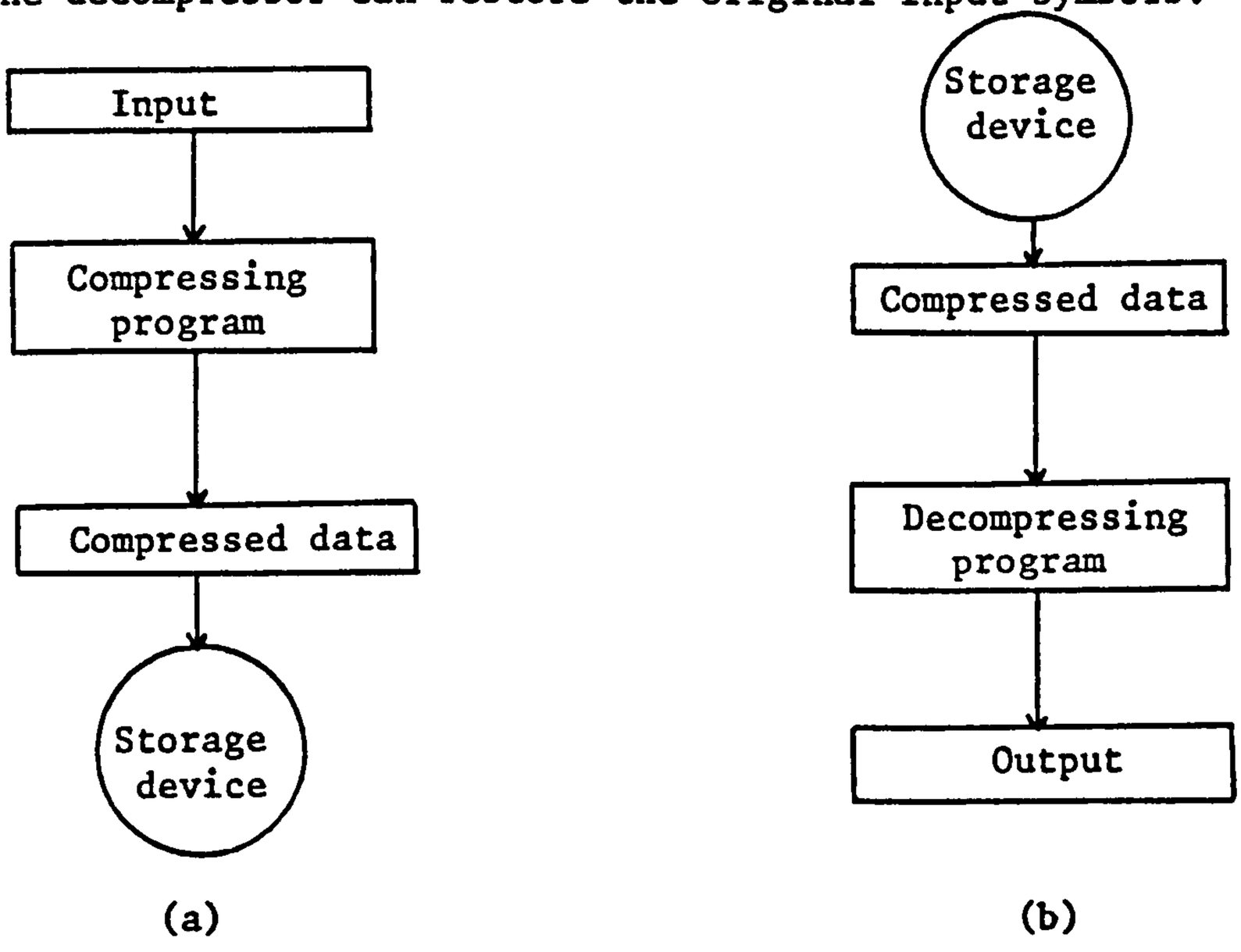


FIGURE 3.2: Compression and decompression phases

The decompression methods can be divided into two different classes called reversible or irreversible (Schuegraf,76). If the output of the decompression program is not an exact copy of the original input, then the method is called irreversible. If the output produced by a decompression program is the exact copy of the input, then the method is called reversible. Usually with the latter method a table is used.

The table used by both encoding and decoding methods must be determined before starting the actual encoding and decoding of symbols. The construction of such tables depends on the language elements and the statistical analysis of those elements.

In the following discussions concerning the decoding of files, only reversible methods will be explained because the decoder must provide an output file exactly the same as the original one.

3.6 CHARACTER ENCODING

In a character encoding H of a finite set T, each character a_i in T maps onto a code c_i in H. So, the encoder reads one character at a time and generates the corresponding code. This process continues until there are no more characters to be coded. If there is a sequence of characters $a_1a_2a_3,\ldots a_n$ in a probabilistic language, then the sequence of codes $u_1u_2,\ldots,u_n$ corresponding to those characters is in the code language. The properties of the code language are the same as those of probabilistic language (Thompson and Booth, 71), that is the code language is a probabilistic language, and if the source language is context-free then the code language is context-free as well.

Different techniques are used for encoding characters. Nevertheless the most popular technique is Huffman method (explained in Section 2.7). Hahn (Hahn, 74) explains a method of encoding a sequence of characters after squeezing off the leading and trailing blanks, the remaining characters are encoded in groups of a fixed length as unique fixed point numbers. The characters are encoded according to their positions in a dictionary comprising all those characters. The unique fixed point number representing a group of characters is constructed from:

$$p_1B^{N-1} + p_2B^{N-2} + \ldots + p_{N-1}B + p_n$$

where $p_1,p_2,\ldots,p_N$ are the positions of characters in the dictionary. B is the number of characters in the dictionary; and N is the length of each group. For example, suppose that B=10, N=4, and the symbols to be encoded have the positions 7,5,8,9,4,2,6 and 3 in the dictionary. These symbols would be encoded in two groups. The first group having the value $7 * 10^3 + 5 * 10^2 + 8 * 10 + 9 = 7589$;

and the second group having the value

$$4 * 10^3 + 2 * 10^2 + 6 * 10 + 3 = 4263$$

The value of B can be less than the actual number of elements in the dictionary. The first B-1 elements comprise the primary dictionary, the B^{th} position is used as an escape character and is coded as 0. This allows the dictionary to extend beyond B. So, more characters can have positions in the range B+1 to 2B-1, and so on. So, a character with position 12 is encoded as 02 (B=10). For example, to encode the symbols having the positions 7,12,2 would be

$$7 * 10^3 + 0 * 10^2 + 2 * 10 + 2 = 7022$$

The way of storing the encoded data is to store the number of leading blanks followed by the number of characters encoded followed by the codes.

The problem with this method is when a character is encountered which has not already been in the dictionary. It must be added to the dictionary before the start of the encoding process. Later when decoding takes place, the same character positions in the dictionary will be used to produce the original sequence of characters because the dictionary is written as the first record of the encoded file.

There is another technique concerning identical characters especially blanks and zeros (Smith, 76); that is instead of generating a code for each character, the encoder counts them and generates the number of occurrences followed by a code of one item only. For example, five zeros could be encoded as 50.

When Huffman code is applied, there are no delimiters between the sequence of codes. So the decoder must know when to consider the received sequence of code symbols as a complete coding for a character. But since Huffman codes are uniquely decodable, then if the first k

code symbols received are not a coding for any character in the set T, then the decoder must read another code symbol and check again. Once the sequence of code symbols matches one of the coding of characters then the corresponding character is output and the next received code symbol is considered as the first symbol of the coding of the subsequent character. The exact sequence of characters will be obtained during a decoding process. For the encoder and the decoder programs, Huffman code will be implemented to encode and decode characters.

3.7 WORD ENCODING

Instead of encoding character by character, here a group of characters (word) is encoded at a time. So for a set of words in a language, there is a code corresponding to each word such that the encoder does not output the code until all the word has been read. Huffman code (explained in Section 2.7) is used to find the codes. However, Huffman code can only be constructed over a finite set of words. So if a probabilistic language is not finite, it may be approximated (Thompson and Booth, 71). That is by ordering the words $x_i \in L$ in decreasing order of their probabilities $p(x_i)$ and then selecting the words in order until

$$\sum_{i=1}^{n} p(x_i) = (1-\varepsilon)$$

Now, the new probabilistic language L contains n words plus one word (dummy) which has a probability ε. If a code is constructed for each word in $\bar{L}$, then the encoder outputs a code for each word in L which is in $\bar{L}$. However, for a word in L which is not in $\bar{L}$, the encoder might report an error or generates the code of the dummy word.

3.8 PARSING ENCODING

The input generated from a context-free grammar should be parsed before allowing the encoder to generate any code (Thompson and Booth, 71). A code is constructed over a set of productions which have the same left-hand side (say A_i) i.e. a set of productions belonging to a non-terminal symbol (A_i). This set belongs to a probabilistic grammar which generates the probabilistic language to be encoded. Each production in the set is assigned a code. Obviously the total probabilities of the productions in one set is equal to one. Then for optimal code Huffman method is applied to generate a suitable unique code for each production in the set. The method is applied to all sets in the probabilistic grammar, and hence the sets have independent codes. It is possible that more than one set has the same code. However, this does not cause any problem to the encoder program because during the parsing process, the program recognizes the exact production of the set and then generates its code. For example, consider the following grammar:

Productions	Probabilities	Huffman Codes
1. E:=E+F	0.3	00
2. E:=E*F	0.2	01
3. E:=F	0.5	1
4. F:=(E)	0.2	01
5. F:=i	0.4	00
6. F:=d	0.4	1

There are two sets: the first set has the productions (1-3), and the second set has the productions (4-6). The codes of both sets are exactly the same. However, although the productions,

E:=E+F

and F:=i

have got the same code (00), they are treated completely different by the encoder program. The same argument applies to the decoder program.

The way used for encoding a string of symbols (Hutchins, 72a) is to parse the symbols, list the productions used in the parse in the order in which they appear in the left-most derivation, concatenate the code words corresponding to the productions in the list†. This will form the coding of the string. The parser does not need to know all the productions before outputting the codes. It can generate a code as soon as a production has been recognized. Note that, although a set might have only one production, the encoder generates its code when the production is recognized by the parser.

The decoder must translate a string of code symbols from the input stream into a string of productions which can be used to construct a parse tree. The decoder contains a stack, a code table holding the productions with their codes, and the productions of the grammar. The decoding procedure would be:

1. Begin with the initial symbol of the grammar on the stack.
2. By examining the top of the stack and checking the code table, determine the next code word.
3. The word taken from the input determines the next production. Apply the production to the stack and remove the code word from the input.

---

†*Note that there is a code for every production in the list, even though some productions are certain to occur, i.e. they do not need any code to be generated.*

4. If any terminal symbols on the top of the stack, output them.
5. If the stack is not empty, then go to step (2); otherwise a complete string has been decoded.

The decoder can be decomposed into two operations, the first segmenting the input stream, and the second operating the stack to reconstruct the string. The problem with the decoding process is that the code words are variable length codes (Huffman codes). So care must be taken when reading a code word.

3.9 MEASURES OF DATA COMPRESSION

Before discussing the design and the implementation of the encoder, it is necessary to explain the measures used for evaluating different encoding techniques. The first measure expresses data compression results in terms of the average number of binary digits that are required to encode a given character (Martin, 76). The second measure is to compare the entropy (i.e. the theoretical minimum length), and the average length of the compressed data (Schuegraf, 76). That is

$$E = \frac{\text{Theoretical minimum length of compressed data}}{\text{Average Length of compressed data}}$$

$$= \frac{-\sum_{i=1}^{N} p_i \log_2(p_i)}{\sum_{i=1}^{N} p_i \ell_i}$$

The values of E are always less than or equal to one, and the maximum of one is obtained only when

$$\ell_i = -\log_2(p_i)$$

In other words, E is equal to one only when the average length of the compressed data is equal to the entropy. The last type of data compression measure is to find the ratio of the size of the compressed data to the size of the data in its original form, i.e.

$$S = \frac{\text{Length of compressed data}}{\text{Length of original data}}$$

An encoding method is said to be optimal under some specific condition if the average length of an encoded string is less than that for any other encoding method under the same condition (Thompson, 71).

CHAPTER 4

LR PARSING

This chapter discusses the main methods for checking the syntactic structure of an input generated from a context-free grammar and proving its validity. These methods are called parsing methods, and the programs are called parsers. The parser tries, during its process to construct a parse tree for the specified input. Accordingly, parsers fall into two main classes called top-down parsers and bottom-up parsers (Gries, 71; Aho and Ullman, 77). One type of top-down parsing method is called Recursive-Descent. An example of bottom-up parsing is called LR(K) parsing which is the most attractive method among the same class of parsing methods for practical context-free grammars. L stands for reading the input from left to right, R for producing a right parse, and K for the number of Lookahead symbols. In practice, K is always 0 or 1.

The LR parsing method was originally described by Knuth (Knuth, 65). The algorithm explains how to construct the set of states from the grammar; and how the parser works with the help of a stack. However, the method was not practically efficient because of the waste of space and time. A simple method called SLR(K) parsing is explained in DeRemer, 71; Bornat, 79: and Aho and Ullman, 77. However, for some grammars, it failed to produce parsers. More general methods called LR(1) and LALR(1) are used to construct LR parsers (Pager, 77; Aho and Ullman, 77; Bornat, 79). A general survey of LR parsing including the construction of the set of states and also the parsing tables is contained in Aho and Johnson, 74.

Different optimization techniques are used to reduce the size of the parser, and also to speed-up its execution. These techniques are explained in detail in Aho and Ullman, 72; Anderson, Eve and Horning, 73; Aho and Ullman, 73; Demers, 75; Joliat, 76.

Attempts have been made to generate automatically LR parsers from a set of productions. One such generator is called YACC (Johnson, 78).

Section 4.1 explains briefly the two classes of parsing methods. A Recursive-Descent parsing method is explained in Section 4.2. In Section 4.3, the general construction of LR parsers is shown. The algorithm of LR parsing is illustrated in Section 4.4. The way of constructing the items and hence, the set of states is explained in Section 4.5. Section 4.6 illustrates how to construct the parsing tables from the set of states. The construction of SLR(K) parsers is mentioned in Section 4.7. The construction of LR(1) parsers and LR(1) parsing tables is illustrated in Sections 4.8 and 4.9 respectively. Sections 4.10 and 4.11 are respectively concerned with the construction of LALR(1) parsers and LALR(1) parsing tables. Section 4.12 shows some techniques used to optimize the parsing tables. Finally, an explanation of the parser generator called YACC is given in Section 4.13.

4.1 PARSING METHODS

During the validation process, the parser tries to build up a syntax tree (or parse tree) for the specified input string, according to the sequence of productions used. The completion of the tree means that the input is syntactically correct and no error is reported. Referring to the way in which the syntax trees are built, the parsing methods can be divided into two categories, top-down and bottom-up.

4.1.1 Top-Down Parsing Method

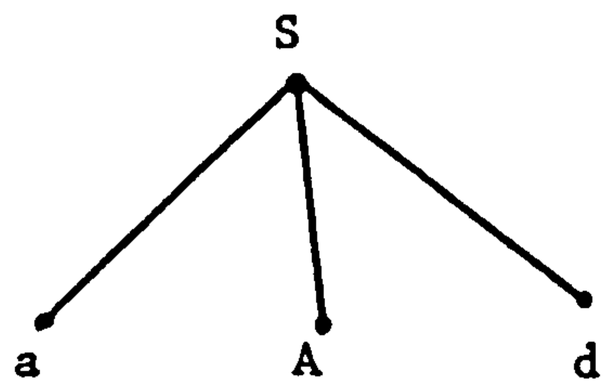
In this method, the parser tries to find a left most derivation for an input string. Equivalently, the parser attempts to build a parse tree by starting from the root and working down to the leaves. The leaves represent terminal symbols, and the remaining nodes (including the root) represent non-terminal symbols (i.e. the left-hand side of the productions). For example, consider the grammar

S:=aAd

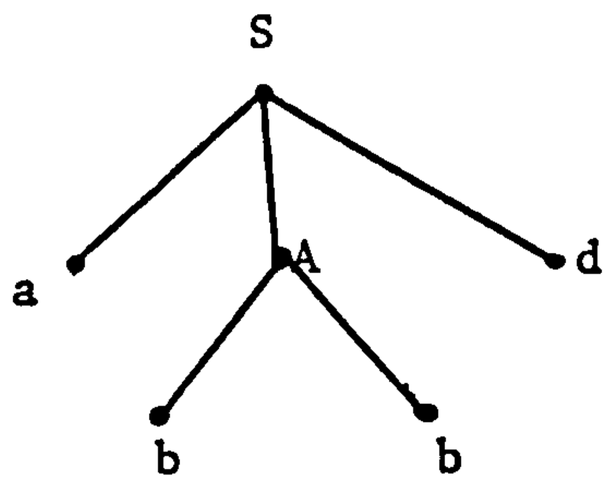
A:=bb

A:=c

and the input abbd. To build up a parse tree for the input, create a tree consisting of only one node labelled S. Since the first input character is a, then use the first production to expand the tree, i.e.



The left-most leaf, labelled a, matches the current input character. The next input character is b which becomes the current input character. Since the next leaf is a non-terminal symbol, then it is possible to expand it by using the second production. The tree becomes:



Now, the leaf labelled b matches the current input character. The next input character is b which matches with the next leaf labelled b. The next input character is d which matches with the last leaf labelled d. The tree is completed without any error. Hence the input abbd is syntactically correct.

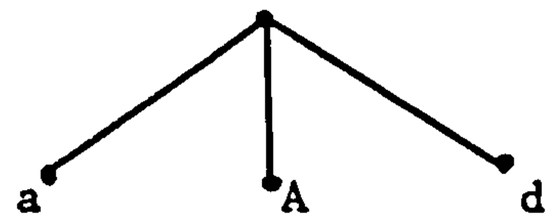
The important factor when writing a top-down parser is to prepare a grammar which is suitable for top-down parsing. Once this has been done, it is easy to write a parser. The main problems which have to be overcome when preparing a grammar are backtracking and left recursion. The problem of backtracking is that at certain state of parsing, the parser discovers that the way used is not the proper one and it would fail to parse the remaining input characters. Thus it has to backtrack to a state in which an alternative way can be used. For example, consider the grammar

S:=aAd

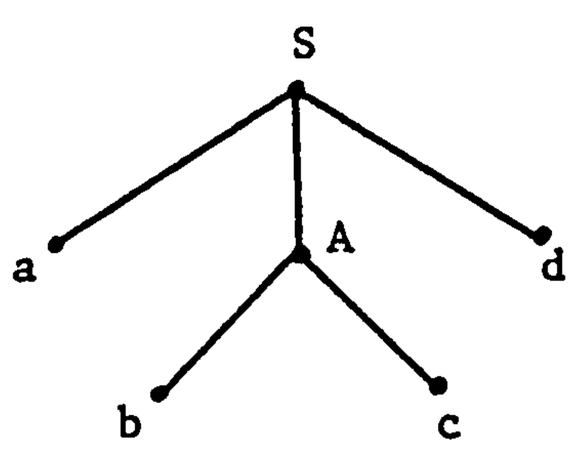
A:=bc

A:=bb

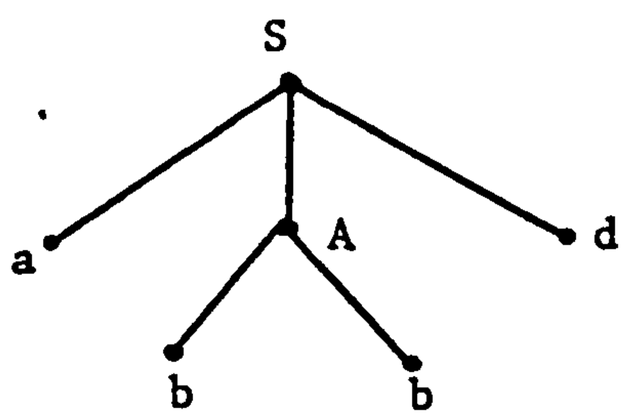
and the input is abbd. To construct a parse tree for the input, create a tree consisting of only one node labelled S. The first input character is a, use the first production to expand the tree:



The current input character matches the left-most leaf, i.e. a. The next current character is b. Since the next leaf is a non-terminal node, then it is possible to expand it by applying the first alternative for A. The tree becomes



Now, the leaf labelled b matches the current input character. The next current character is b, and the next leaf labelled c do not match. Hence, the parser could not carry on its job and it has to go back to the node A to see if there is another alternative for A that has not yet been tried which might produce a match. In going back, the current input character should be the one when the node A was firstly expanded, that is the character b. By trying the second alternative for A, the tree becomes,



The leaf b matches the current character b, and the remaining input characters (i.e. b and d) match the last two leaves. Hence, there is a parse tree for the input, and the input is said to be syntactically correct.

The next problem is a grammar which contains a left recursive production (simple recursion) i.e. a production in which the left hand symbol appears at the left end of the right hand side of the production. For example, consider the productions

S:=S,a

S:=a

and assume that each non-terminal symbol is represented by a procedure in the parser. Then from the first alternative production, the procedure S will call itself an infinite number of times.

To overcome the above problems, the grammar should be modified in such a way that the new grammar is structurally equivalent to the original one, but the input is recognized without backtracking and left recursion. To eliminate backtracking, try to factor out the common portions at the left end of each alternative. This action enables the parser to check these portions only once. Parentheses are used for this purpose as syntax notations. For example, the grammar

S:=aAd

A:=bc

A:=bb

which has a backtracking, could be rearranged as

S:=aAd

A:=b(c|b)

To remove the left recursion from a production, a better way is to iterate the sequence of elements zero or more times. For this purpose assume that brackets { and } are used. So, the productions

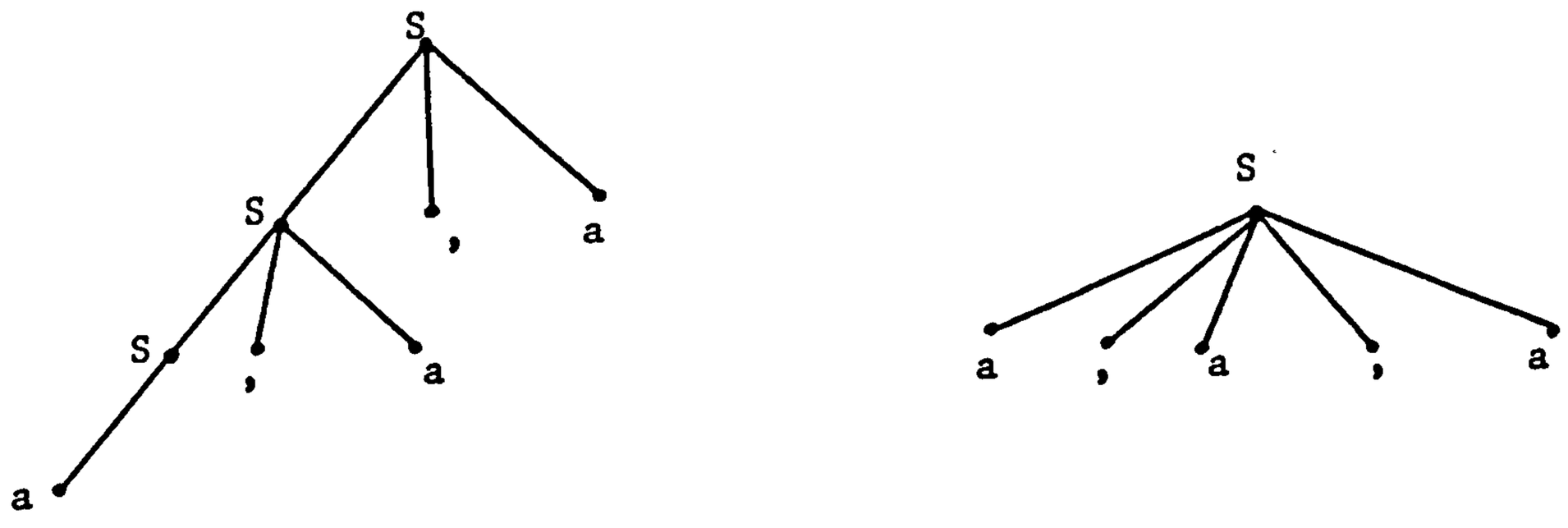
S:=S,a

S:=a

cause the repetition of (,a) zero or more times. This can be arranged as

S:=a{,a}

Fig. 4.1 shows the parsing trees of both the recursion and the iteration for the input a,a,a. Both trees are treated as equivalent.



recursion iteration

FIGURE 4.1: Parsing trees using recursion and iteration

An example of a top-down parsing method is called Recursive-Descent which is explained in Section 4.2.

4.1.2 Bottom-up Parsing Method

In this method, the parser tries to build up a parse tree for a given input, by starting from the terminal nodes (leaves) and building to the root. That is, it starts with the terminal string and replaces a substring of symbols by a non-terminal symbol from which the substring can be derived by one application of a production of the grammar. Then,

using the resulting string, the process of replacing a substring of symbols by a non-terminal symbol is repeated until the start symbol S is obtained. For example consider the following grammar

S:=aAd

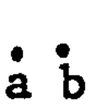
A:=bb

A:=c

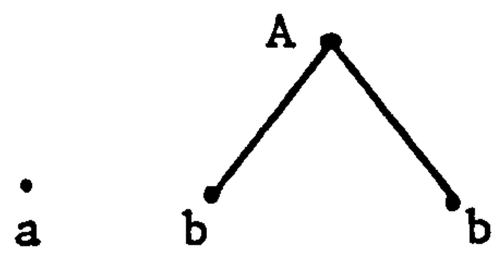
and the input abbd. The parser reads the input symbol (a) and constructs the tree:

then reads the next input symbol (b) and creates a single node, i.e.

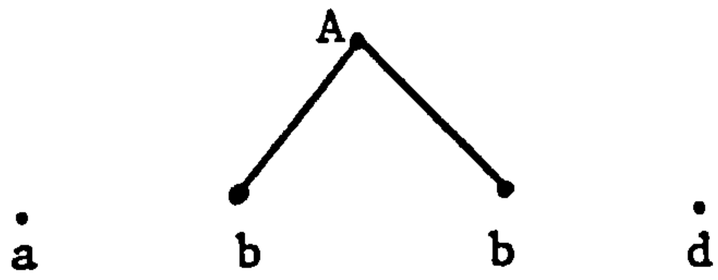
the parser reads another input symbol (b), and adds a new node, i.e.



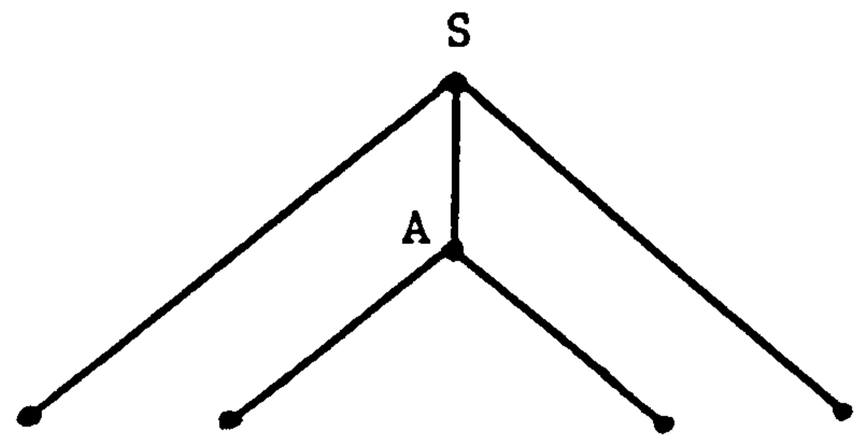
By using the second production, the string (bb) can be reduced to (A). So a new node is created labelled (A) from the leaves (b) and (b).



Now, the parser reads the last input symbol (d) and adds a new node to the tree



By using the first production, the string (aAd) can be reduced to (S). So a new node labelled (S) is created from the nodes a,A, and d.



Node S is called the root of the tree. At this point, all the input has been read, and the parsing is completed. An example of a bottom-up parsing method is called LR method which is explained in detail in Sections 4.3-4.13.

4.2 RECURSIVE-DESCENT METHOD - A DETAILED EXAMPLE

An easy way to implement top-down parsing is to create one (possibly recursive) procedure for each non-terminal symbol, which parses the input derived from that non-terminal symbol. The procedure is told where in the program to begin looking for its input. This can be found by using the right-hand side of the productions for the non-terminal symbol. During this process other procedures might be called.

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a Recursive-Descent parser. The recursive procedures can be quite easy to write and fairly efficient if written in a programming language that implements procedure calls efficiently. If the programming language has not the ability to call procedures recursively, then a stack could be created and maintained by the parser (this would be a LIFO or a push-down stack).

As an example, consider the following grammar rules (productions)

E:=E+F|E*F|E-F|E/F|F
F:=i|c|(E)

It is assumed that all arithmetic operators have equal precedence. Since the grammar suffers from left recursion problem, then it can be rearranged as

E:=F{+F|-F|*F|/F }
F:=i|c|(E)

There are two recursive procedures (E and F) involved recognizing the input. In addition assume that SCAN( ) is a procedure which reads an input character and stores its type in a variable location called token. From the first production, the procedure E( ) immediately calls the

procedure F( ), see Fig. 4.2, and then whenever there is an arithmetic operator SCAN is called to advance to a new token and the procedure F( ) is called. Similarly, F( ) is coded directly from the production F. Note that in the programming language C the symbol (==) is used to test for equality, and the symbol (!!) is used to mean a logical (or) operator.

```
E( )
{F( );
   while (token =='+'!! token =='-'!! token == '*'!! token =='/')
        {SCAN( ); F( );}
}
F( )
 {If(token ==i!! token ==c) SCAN( );
     Else if (token =='(')
        {SCAN( ); E( );
        If (token ==')') SCAN( );
        Else error( );
     }
   Else error( );
}
```

FIGURE 4.2: Mutually recursive procedures written in C language

4.3 LR PARSERS

LR(K) parsers are considered to be one of the more efficient types of bottom-up parsers. They can recognize most context-free languages. Syntax errors can be detected as soon as they occur. The input string is parsed in a time which is proportional to the length of the string. No backtracking is required. The function of the parser is divided into a finite sequence of steps called states. In each state, all possible actions that can be taken by the parser are provided. The construction of these states is described in Section 4.5.

The parser consists of a driver routine, a parsing table which governs its operation, an input stream, and a stack (Fig. 4.3). The driver routine is the same for all LR parsers which reflect the parsing algorithm mentioned in Section 4.4. The input contains only terminal symbols and is read from

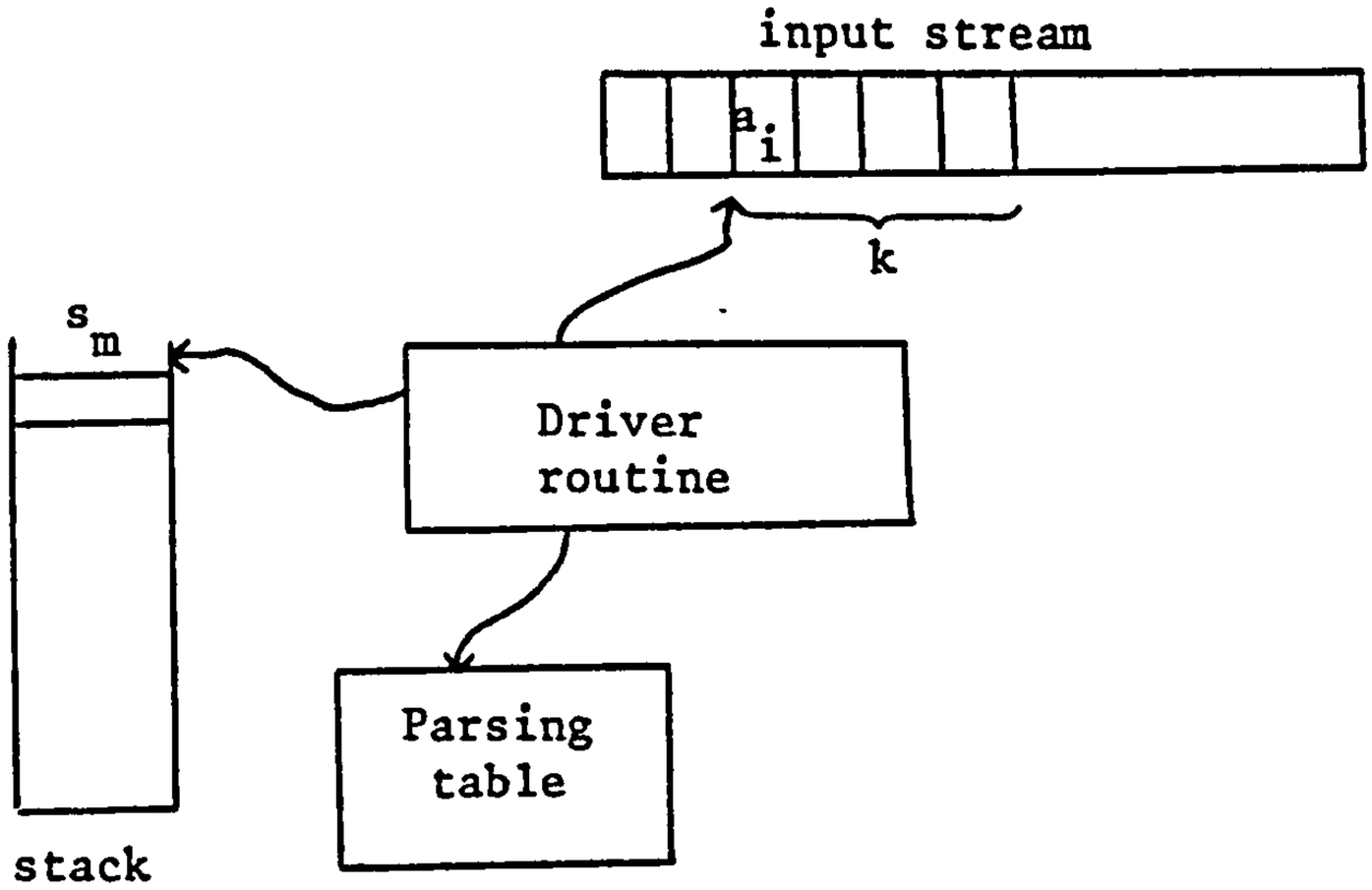


FIGURE 4.3: LR parser

left to right, one symbol at a time. The stack contains a string of symbols called states. The parsing table consists of two parts; the ACTION table and GOTO table. The ACTION table specifies which action is

going to be taken by the parser with respect to the current state and the next input symbol (see Section 4.4). There are four different actions:

1. Shift the input symbol and change to a new state;
2. Reduce by the production
   $A:=\alpha$, and goto a new state;
3. Accept the input;
4. Error

The GOTO table specifies the next state as a new current state after each reduction.

4.4 LR PARSING ALGORITHM

For a given input, the parser starts from the initial state, parsing the input by consulting the ACTION table until an accept or an error action is encountered.

Let $\{s_0, s_1, \ldots, s_m\}$ be a set of states stored on the stack, where s_m is the current state on the top of the stack. Let a_i, $a_{i+1}, \ldots, a_n$, \$ be the remaining input symbols (\$ is the end of input marker), a_i is the next input to be expected by the parser. By consulting the ACTION table, the algorithm would be:

1. If ACTION [current state, next input]= shift s, then the parser shifts a_i from the input and enters state s. The stack becomes $s_0, s_1, \ldots, s_m, s$. s becomes the current state, and the next input symbol is a_{i+1}. Go to step 1.
2. If ACTION [current state, next input]= reduce by the production

   $A:=\alpha$.

   Suppose α is a string of grammar symbols of length r. The parser has found the handle of the above production, and can now do the reduce action. It will remove, by starting from the top of the stack, a number of elements equal to the length of α which is r. Now, s_{m-r} on the top of the stack. To find the next current state, consult the GOTO table, i.e. GOTO $[s_{m-r}, A]=s$. Push s onto stack. Since no shift action has been made on the input symbol, it remains as the current input symbol. Go to step 1.
3. If ACTION [current state, next input]= accept, then the input has been successfully parsed. Here \$ is the next input.
4. If ACTION [current state, next input]= error, then a syntax error has been discovered.

4.5 CONSTRUCTING THE SET OF STATES

Before generating both the ACTION table and GOTO table, it is necessary to construct the set of states for a particular grammar. Each state represents the position of the parser and the range of possible next actions.

An item is defined as a production of a grammar G with a marker (say dot) at some position in the right side of the production. The position of the dot indicates that the parser has already recognized the string derivable from the grammar symbols before the dot of this particular production, and expecting to see the string derivable from the grammar symbols after the dot before making any reduction by the same production. For example, consider the production

$S:=\alpha\beta$

Then three items can be obtained

$S:=.\alpha\beta$

$S:=\alpha.\beta$

$S:=\alpha\beta.$

The first item indicates that a string derivable from $\alpha\beta$ is expected next on the input. The second item indicates that a string derived from α has already been seen and a string derivable from β is expected next. The last item indicates that a string derivable from $\alpha\beta$ has been seen and a reduction by the production

$S:=\alpha\beta$

is possible.

To indicate to the parser when it should stop parsing and announce acceptance of the input, a new start symbol P is added with the production

P:=S

to the grammar, because if the parser reaches a point where the marker (.) is at the right-most of the item

P:=S.

then the input has been accepted.

The construction of the collection sets of items starts from the augmented production

P:=S

The first set of items must contain the item

P:=.S

If the marker is placed immediately before a non-terminal symbol, include an item with a marker in first position for each of the productions which define that non-terminal. Continue to apply this process until no more items can be added to the set of items. The included set of items is called the closure set. The first item with its closure set represents the first state (initial state). The successor states are computed by starting from the initial state. If a state contains items in which the marker is positioned immediately before a particular symbol in their productions (i.e. the marker is not at the right most of the items), create a new state which contains only those items such that the marker is positioned immediately after that symbol. Now find the closure set of items of the new state as mentioned above. As an example, consider the grammar

1. S:=AA
2. A:=aA
3. A:=b

First, add to the grammar the following production

0. P:=S

Next, to construct the initial state (s_0), it must contain the item:

P:=.S

since the dot is immediately before the non-terminal symbol S, then the closure set must be obtained. The idea of finding the closure items is that the parser does not expect to find S as the next input, but a string of input symbols derivable from S. Thus s_0 must also contain

S:=.AA

The dot is before A which is a non-terminal symbol so the following items should be included in s_0:

A:=.aA

A:=.b

So, s_0 has four items. To find the successor states, choose the item

P:=.S

place the dot after the symbol S, i.e.

P:=S.

since the position of the dot is at the right-most, then the closure set of items can not be obtained. Hence the new state (s_1) has only one item. s_1 is called the final state. From the item

S:=.AA

two states can be generated (s_2 and s_3)

s_2: S:=A.A
A:=.aA
A:=.b

s_3: S:=AA.

continue in this process until no more states can be created. Fig. 4.4 shows a complete set of states generated from the above grammar. Fig. 4.5 illustrates the relations between different states according to the grammar symbols. For instance, if the current state is s_0 and the current grammar symbol is S, then s_1 would be the new current state.

s_0: P:=. S
S:=.AA
A:=.aA
A:=.b

s_1: P:= S.

s_2: S:=A.A
A:=.aA
A:=.b

s_3: S:=AA.

s_4: A:=a.A
A:=.aA
A:=.b

s_5: A:=aA.

s_6: A:=b.

FIGURE 4.4: A set of LR(0) states

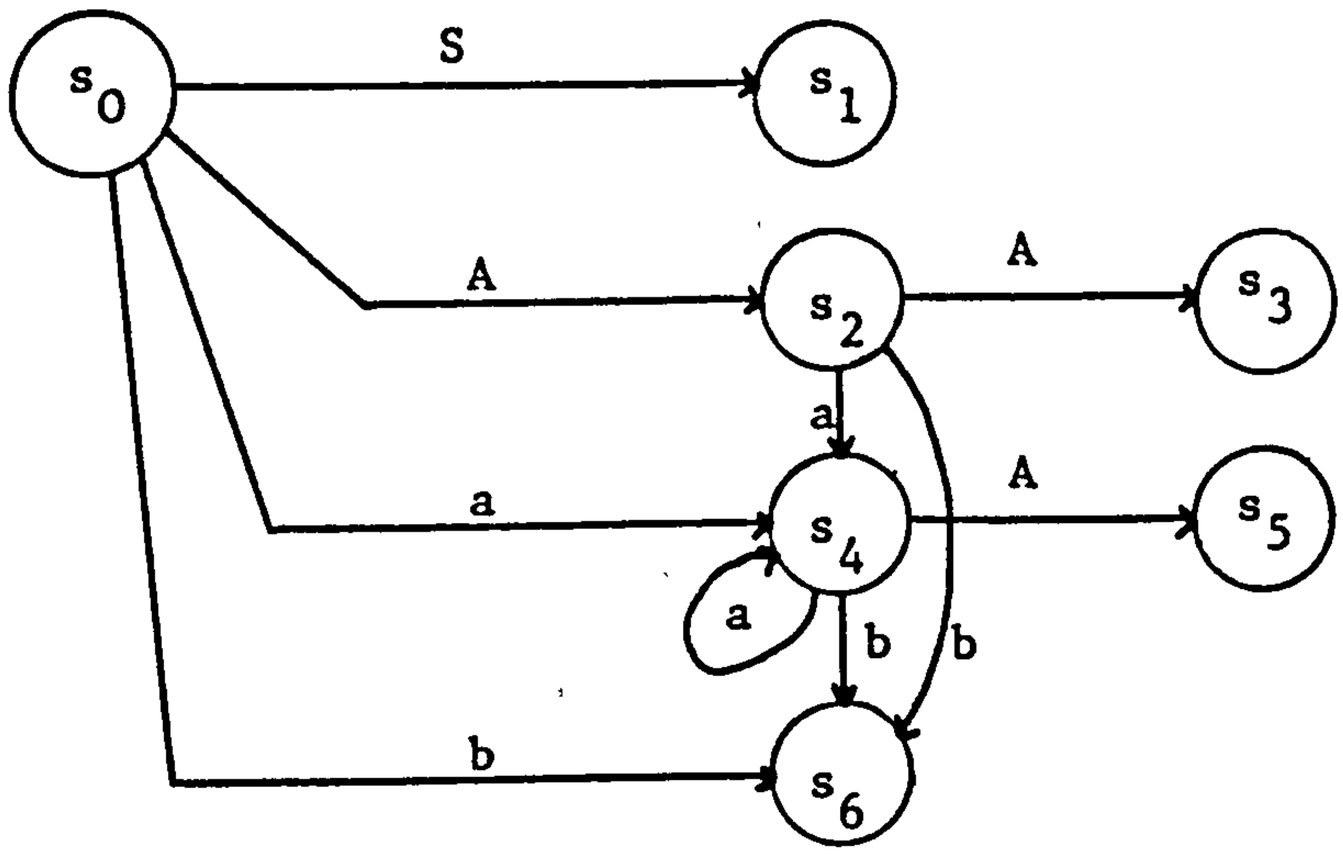


FIGURE 4.5: A graph showing the relationships between the states

4.6 CONSTRUCTING LR PARSING TABLES

This section shows how to construct the LR parsing ACTION and GOTO tables from a set of states (described in the previous section) generated from an augmented grammar G' (assuming that the original grammar is G).

Let $s_0, s_1, \ldots, s_n$ be a set of states, the elements of the ACTION table are determined as follows:

1. If A:=x.az is in s_i and the successor state is s_j, then set ACTION[i,a] to shift j. a is a terminal symbol.
2. If A:=x. is in s_i, then set ACTION[i,a] to reduce by the production

   A:=x
3. If P:=S. is in s_i, then set ACTION[i,$] to accept. $ is the end of input marker.
4. The remaining undefined elements are set to error.

The elements of GOTO table are obtained as follows:

1. If A:=.Xy is in s_i and A:=X.y is in s_j, then set GOTO[i,X] to j.

   X is a non-terminal symbol, and y is a grammar symbol or empty.
2. The remaining undefined elements in the GOTO table are set to error.

The representation of both the ACTION table and GOTO table depends on the number of states and the way of accessing a particular element. If the number of states is relatively small, then the parsing actions for each state can be represented by a sequence of programming language statements, and GOTO table can be represented by a sequence of programming language statements for each non-terminal symbol. For example, consider the construction of both the ACTION table and GOTO table from the set of states mentioned in Fig. 4.4. The ACTION table would be:

```
0:      If (input =='a') shift 4;
        Else if (input =='b') shift 6;
        Else error;

1:      If (input =='$') accept;
        Else error;

2:      If (input =='a') shift 4;
        Else if (input =='b') shift 6;
        Else error;

3:      reduce 1;

4:      If (input =='a') shift 4;
        Else if (input =='b') shift 6;
        Else error;

5:      reduce 2;

6:      reduce 3;
```

The GOTO table would be:

```
S:      If (state =='0') goto 1;

A:      If (state =='0') goto 2;
        If (state =='2') goto 3;
        If (state =='4') goto 5;
```

However, for a practical grammar where the number of states might reach several hundreds, the above method looks impractical because of the increase in the size of the parser.

The next method is to represent the ACTION table and GOTO table by two different matrices. For the ACTION table, each row represents a particular state and each column represents a terminal symbol. Each row of the GOTO table represents a particular state and each column represents a non-terminal symbol.

In what follows (s) denotes a shift; (r_i) denotes a reduction by the production number i; (a) denotes the accept action; a space denotes an error; and an integer denotes a state number. The ACTION table and GOTO table corresponding to the set of states in Fig. 4.4 is shown in Fig. 4.6.

state	a	b	$	S	A
0	s_4	s_6		1	2
1			a		
2	s_4	s_6			3
3	r_1	r_1	r_1		
4	s_4	s_6			5
5	r_2	r_2	r_2		
6	r_3	r_3	r_3		

FIGURE 4.6: Two matrices representing the parsing tables

Another way of constructing the ACTION table is to store the elements of each state separately, and try to link the states as required after each action. Fig. 4.7 shows the relations between the states mentioned in Fig. 4.4. Some states are connected to the GOTO table.

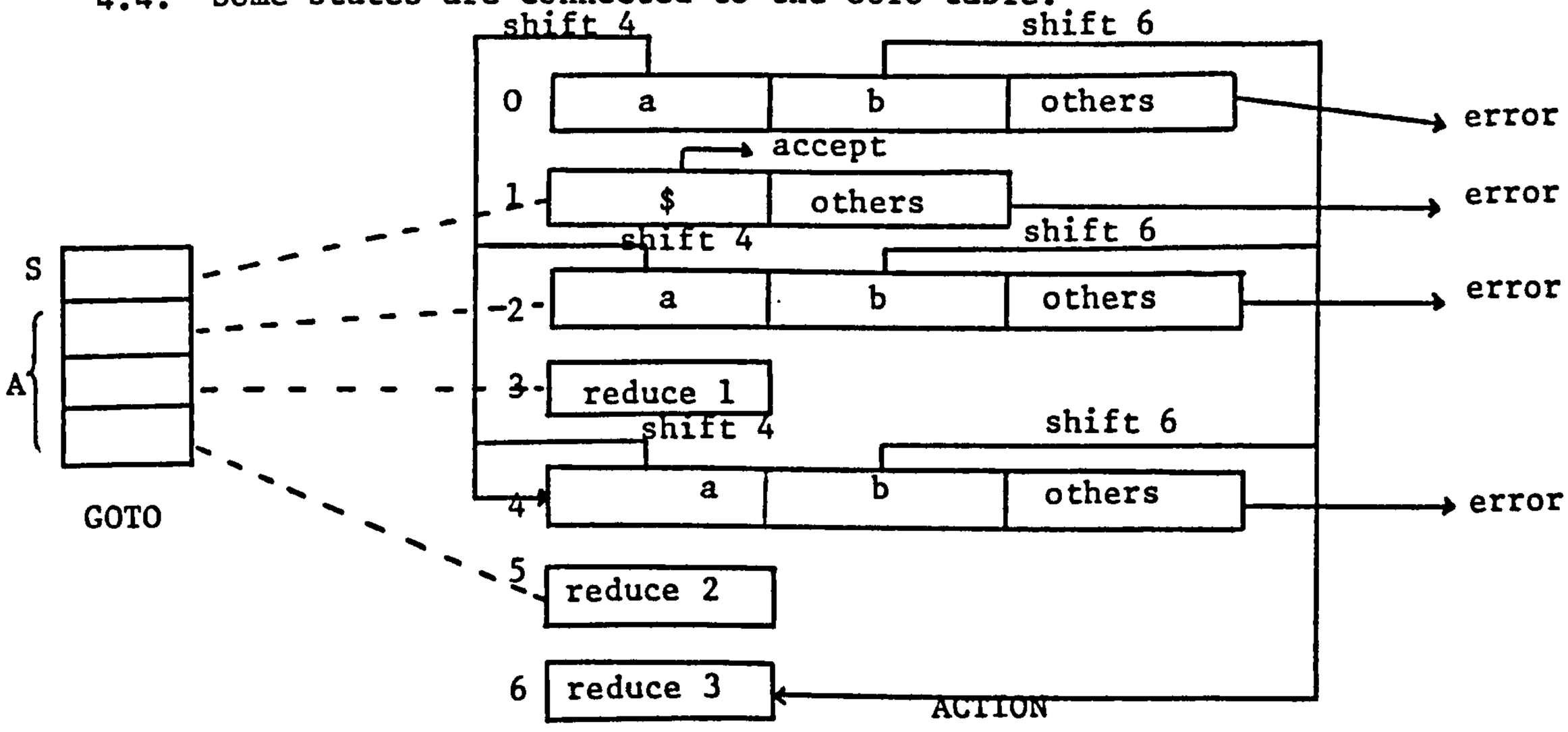


FIGURE 4.7: Constructing the parsing table using a pointer type structure

Using matrices for constructing a parsing table is more practical than others because any element can be obtained in one access. Furthermore, it is relatively easy to build and maintain the matrices. From now on, the parsing table will be represented by matrices unless otherwise mentioned.

4.7 SLR(K) PARSERS

For K=0, the parser begins scanning the input from left to right. It identifies a production when it gets to the right-most symbol derived from that production, and each handle can be detected without looking at any input symbols beyond the last input symbol derived from the handle. When a handle is found, the parser does the same reduce action regardless what the current input symbol is. Such parsers sometimes called simple LR(0) or SLR(0). Although LR(0) parsers can be constructed for different grammars, sometimes it is not possible because in certain states the parser can not decide whether to shift the input symbol or to reduce by a particular production without looking ahead to the next input symbol(s). This problem called a shift-reduce conflict. So, to solve this conflict, allow the parser to inspect at most K>0 input symbols ahead in order to make the right decision. For a practical reason K=1 is assumed. The parsing algorithm and the construction of the set of states of SLR(K) are explained in Sections 4.4 and 4.5.

To find the set of lookahead symbols for each non-terminal symbol in the grammar, it is required to discuss two functions called FIRST and FOLLOW. If α is a string of grammar symbols then FIRST(α) is the set of terminal symbols that begin strings derived from α. For example, consider the productions

A:=Bb

B:=(A)|b

then FIRST(B)={(,b}.

To find FIRST(A) for all grammar symbols A, apply the following rules until no more terminals or ε (empty) can be added to any FIRST set.

1. If A is a terminal symbol, then FIRST(A) is {A}.

2. If A is a non-terminal symbol and

$$A:=a\alpha$$

is a production, then add a to FIRST(A). a is a terminal symbol. If

$$A:=\varepsilon$$

is a production, then add ε to FIRST(A).

3. If $A:=B_1B_2,\ldots,B_n$

is a production, then for all i such that all of $B_1,B_2,\ldots,B_{i-1}$ are non-terminal symbols and FIRST(B_j) contains ε for j=1,...,i-1, add every non-ε symbol in FIRST(B_i) to FIRST(A). If ε is in FIRST(B_j) for all j=1,...,n then add ε to FIRST(A).

Let A be a non-terminal symbol, then FOLLOW(A) is the set of terminal symbols that can appear immediately to the right of A in some sentential forms. If A can be the right-most symbol in some sentential form, then add the end of input marker ($) to FOLLOW(A). For example, FOLLOW(A)={),$}.

To compute FOLLOW(A) for all non-terminal symbols A, apply the following rules until nothing can be added to any FOLLOW set.

1. If S is the start symbol, then the end of input marker ($) is in FOLLOW(S).

2. If there is a production

$$A:=\alpha B\beta \ , \ \beta\neq\varepsilon$$

then everything in FIRST(β) except ε is in FOLLOW(B).

3. If there is a production

$$A:=\alpha B$$

or $A:=\alpha B\beta$

where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following set of productions:

1. A:=CB

2. B:=+CB

3. B:=ε

4. C:=ED

5. D:=*ED

6. D:=ε

7. E:=(A)

8. E:=i

To compile the FIRST of each non-terminal symbol, according to rule 3, FIRST(A)=FIRST(C)=FIRST(E). From rule 2, FIRST(E)={(,i}, then

FIRST(A)=FIRST(C)=FIRST(E)={(,i}

From rule 2,

FIRST(B)={+,ε}

FIRST(D)={*,ε}

To find FOLLOW of each non-terminal symbol, from rules (1 and 2) and the productions (1 and 7).

FOLLOW (A)={),$},

FOLLOW(B) is equal to FOLLOW(A) according to rule 3 and the production 1, i.e.

FOLLOW(A)=FOLLOW(B)={),$}

From rule 2 and the production 1, every element (except ε) in FIRST(B) is in FOLLOW(C). Also according to rule 3 and the production 1, FIRST(B) contains ε, then every element in FOLLOW(A) is in FOLLOW(C). Therefore

FOLLOW(C)=FIRST(B) plus FOLLOW(A)

={+,),$}

From rule 3 and the production 4, FOLLOW(D)=FOLLOW(C).

To find FOLLOW(E), according to rule 2 and production 5 every element except ε in FIRST(D) is in FOLLOW(E). Since FIRST(D) contains ε, then from rule 3 and production 4, every element in FOLLOW(C) is in FOLLOW(E). Therefore

$$\text{FOLLOW(E)=FIRST(D)} \cup \text{FOLLOW(C)}$$
$$=\{*,+,),\$\}.$$

After having an idea of how to compute the FOLLOW set of characters, now consider constructing (Fig. 4.8) a set of states with FOLLOW sets included from the following augmented grammar:

1. P:=E
2. E:=E+T
3. E:=T
4. T:=T*F
5. T:=F
6. F:=(E)
7. F:=i

$\longrightarrow$ FOLLOW(P)={$}
FOLLOW(E)={+,),$}
FOLLOW(T)={*,+,),$}
FOLLOW(F)={*,+,),$}

s_0: P:=.E
E:=.E+T
E:=.T
T:=.T*F
T:=.F
F:=.(E)
F:=.i

s_1: P:=E. $
E:=E.+T

s_2: E:=E+.T
T:=.T*F
T:=.F
F:=.(E)
F:=.i

s_3:	E:=E+T.	),+,$
	T:=T.*F	
s_4:	E:=T.	),+,$
	T:=T.*F	
s_5:	T:=T*.F	
	F:=.(E)	
	F:=.i	
s_6:	T:=T*F.	),*,+,$
s_7:	T:=F.	),*,+,$
s_8:	F:=(.E)	
	E:=.E+T	
	E:=.T	
	T:=.T*F	
	T:=.F	
	F:=.(E)	
	F:=.i	
s_9:	F:=(E.)	
	E:=E.+T	
s_{10}:	F:=(E).	),*,+,$
s_{11}:	F:=i.	),*,+,$

FIGURE 4.8: A set of SLR(1) states

State 0 expects either the input symbol "(" in order to shift it and goto state 8, or the symbol "i" in which the action would be to shift the input symbol and goto state 11. In state 3 there are two actions (reduce when the first item is implemented, and shift when the second item is used), one of them must be chosen by the parser. For SLR(0) this case causes a shift-reduce conflict because it can not decide whether to make a reduction by the production number 2 or to shift and goto state 5. The same conflict

occurs in the state 4. Hence, it is not possible to construct a SLR(0) parser from the above grammar.

However, for K=1, the shift-reduce conflict mentioned above will disappear because the parser can check the next input symbol and accordingly decides which action should be done. For example, if the parser is in the state 3, it checks the next input symbol, for (*) a shift action is required, if the input symbol is either ")","+" or "$" then a reduce action is required otherwise an error has occurred. The same argument applies to the state 4. So, it is possible to construct a parsing table from the above set of states. The parser which has such parsing table is called an SLR(1) parser.

4.8 LR(1) PARSERS

It was mentioned in the previous section that SLR(1) method can solve some conflicts which the SLR(0) method can not handle. Nevertheless there are conflicts that can not be solved by looking at a symbol in the FOLLOW set. In such a case it is not possible to construct a SLR(1) parser, and the grammar is not SLR(1). For example, consider the following augmented grammar:

P:=S
S:=A=B
S:=B
A:=*B
A:=i
B:=A

The set of states together with their FOLLOW sets is shown in Fig. 4.9. In state 2, suppose that the next input symbol is (=), then the first item causes a shift action and goto state 3, whereas the second item causes a reduce action by the production

B:=A

because the symbol (=) is in the FOLLOW(B). This situation causes a shift-reduce conflict on the input symbol (=). So, the grammar is not SLR(1).

Another problem which causes a conflict is when a state has two or more completed items (a completed item is one in which the marker is at the right-most position in the right part of a production) with a common

s_0:	P:=.S	$
	S:=.A=B	$
	S:=.B	$
	A:=.i	=,$
	A:=.*B	=,$
	B:=.A	=,$

s_1:	P:=S.	$
s_2:	S:=A.=B	$
	B:=A.	=,$
s_3:	S:=A=.B	$
	B:=.A	=,$
	A:=.i	=,$
	A:=.*B	=,$
s_4:	S:=A=B.	$
s_5:	S:=B.	$
s_6:	A:=i.	=,$
s_7:	A:=*.B	=,$
	B:=.A	=,$
	A:=.i	=,$
	A:=.*B	=,$
s_8:	A:=*B.	=,$

FIGURE 4.9: Non SLR(1) states

input symbol in the FOLLOW sets of these items. With respect to this input symbol, if a reduce action is required then the parser can not decide by which production the reduction should be made. This type of conflict is called reduce-reduce conflict. For example, consider the following augmented grammar

P:=S
S:=V=E
V:=i
V:=R!E
E:=V
E:=R
R:=i

The set of states of the above grammar is shown in Fig. 4.10. State 5

shows a reduce-reduce conflict. (V) can be followed either by (=) or ($), (R) can be followed either by (!),(=), or ($) in some sentential form. So SLR(1) method can not solve this conflict because if the next input symbol is (=) then it can not decide whether to reduce by the production

V:=i

or by the production

R:=i

Hence the above grammar is not an SLR(1) grammar.

It is possible to make the parser choose arbitrarily between conflicting actions and continue in the presence of conflicts. For instance, the conflict in state 5 can be resolved by looking at the next input symbol, if it is (=) then reduce by the production

V:=i

and if it is (!) then reduce by the production

R:=i

Another way for solving conflicts is the inclusion of more information in the state. The information allows the parser to know exactly which input symbols can follow a handle for which there is a possible reduce action.

s_0:	P:=.S	$
	S:=.V=E	$
	V:=.i	=,$
	V:=.R!E	=,$
	R:=.i	!,=,$
s_1:	P:=S.	$
s_2:	S:=V.=E	$

s_3:	S:=V=.E	$
	E:=.V	=,$
	E:=.R	"
	V:=.i	"
	V:=.R!E	"
	R:=.i	!,=,$
s_4:	S:=V=E.	$
s_5:	V:=i.	=,$
	R:=i.	!,=,$
s_6:	V:=R.!E	=,$
s_7:	V:=R!.E	=,$
	E:=.V	"
	E:=.R	"
	V:=.i	"
	V:=.R!E	"
	R:=.i	!,=,$
s_8:	V:=R!E.	=,$

FIGURE 4.10: Non SLR(1) states

The extra information is incorporated into the state by redefining items to include one or more terminal symbols as a second component. The general form of an item becomes:

$$A:=\alpha.\beta,\ \ell_s$$

where $A:=\alpha\beta$ is a production, and (ℓ_s) is a set of terminal symbols (the set might include the end of input marker). If the expected number of terminal symbols from (ℓ_s) is 1, then the item is called LR(1) item. The first component is called the core of the item, and the second component is called the lookahead set of the item. If β is not empty, the lookahead set has no effect on the item. But an item of the form

$$A:=\alpha., \ell_s$$

calls for the reduction by the production

$$A:=\alpha$$

only when the next input symbol is in (ℓ_s).

The way of constructing LR(1) states is the same as mentioned in Section 4.5 except that the lookahead set of each item should be taken into consideration. Suppose that

$$A:=\alpha.B\beta, \quad \ell_s$$

is an item where B is a grammar symbol, and (ℓ_s) is a lookahead set, then the successor item on B is

$$A:=\alpha B.\beta \ ,\ell_s$$

the lookahead set remains unchanged. To find the closure set, suppose that a state contains the item

$$A:=\alpha.B\beta \ ,\ell_s$$

where B is a non-terminal symbol, include the items

$$B:= .\gamma \ , \ n\ell_s$$

for each production of the form

$$B:=\gamma$$

If β is empty, the new lookahead set ($n\ell_s$) will contain the set (ℓ_s), otherwise ($n\ell_s$) will contain FIRST($\beta\ell_s$). If a set of items in a state contains identical core, e.g.

$$A:=\alpha.\beta \quad , \ \ell_{s_1}$$
$$A:=\alpha.\beta \quad , \ \ell s_2$$

but different lookahead sets, then these items must be merged into a single item which has the union of the lookahead sets of all the items, e.g.

$$A:=\alpha.\beta \ , \quad \ell_{s_1} \cup \ell_{s_2}$$

As an example, consider the following augmented grammar:

P:=S

S:=AA

A:=aA

A:=b

The first item is

P:=.S ,$

since S is a non-terminal symbol, then include the item

S:=.AA

the input symbols following S are empty, so the lookahead set of the new item is ($), i.e.

S:=.AA ,$

(A) is a non-terminal symbol, includes the set of items with their lookahead set which is equal to FIRST(A). According to the rules mentioned in Section 4.7, FIRST(A) contains (a) and (b), then

A:=.aA ,a,b

A:=.b ,a,b

None of the new items has a non-terminal symbol immediately after the dot, therefore no more items can be added. So the initial state is

s_0: P:=.S $

S:=.AA $

A:=.aA a,b

A:=.b a,b

The lookahead set of the successor item remains unchanged then

s_1: P:=S. $

continue in this process until no more states can be added. The complete set of states is shown in Fig. 4.11. By comparing the set of LR(0) states in Figure 4.4 with the set of LR(1) states in Figure 4.11,

notice that the LR(0) states are identical to some LR(1) states (ignoring lookahead sets). The extra states are caused by the lookahead sets. For instance, in Fig. 4.11, states 4 and 7 are identical except for the lookahead set of each item. This situation does not happen in LR(0) states.

s_0:	P:=.S	$
	S:=.AA	$
	A:=.aA	a,b
	A:=.b	"
s_1:	P:=S.	$
s_2:	S:=A.A	$
	A:=.aA	$
	A:=.b	$
s_3:	S:=AA.	$
s_4:	A:=a.A	a,b
	A:=.aA	"
	A:=.b	"
s_5:	A:=aA.	a,b
s_6:	A:=b.	a,b
s_7:	A:=a.A	$
	A:=.aA	$
	A:=.b	$
s_8:	A:=aA.	$
s_9:	A:=b.	$

FIGURE 4.11: LR(1) states

4.9 CONSTRUCTING LR(1) PARSING TABLES

Let G be an augmented grammar. Let $\{s_0,s_1,\ldots,s_n\}$ be a set of states constructed as in the previous section. The elements of the ACTION table are determined as follows:

1. If $[A:=\alpha.a\beta,\ell_s]$ is in s_i and the successor state is s_j, then set ACTION[i,a] to shift j. (a) is a terminal symbol.
2. If $[A:=\alpha.,\ell_s]$ is in s_i, then for all (a) in (ℓ_s), set ACTION[i,a] to reduce by the production

   $$A:=\alpha$$

3. If $[P:=S.,\ell_s]$ is in s_i, and ($) in (ℓ_s), then set ACTION[i,$] to accept.
4. The remaining undefined elements are set to error.

The elements of GOTO table are obtained as follows:

1. If $[A:=.Xy,\ell_s]$ is in s_i and $[A:=X.y,\ell_s]$ is in s_j, then set GOTO[i,X] to j. X is a non-terminal symbol and y is a grammar symbol or empty.
2. All undefined elements are set to error.

The representation of both tables is the same as in Section 4.6.

4.10 LALR(1) PARSERS

It was mentioned in Section 4.8 that LR(1) method solves the conflicts encountered with LR(0) and SLR(1) by including a lookahead set of input symbols with each item. But this requires a much larger number of states. This section discusses a method which uses LR(1) algorithm for resolving conflicts but uses no more than the number of LR(0) states and is called LALR(1) (LookAhead LR).

The reason for the smaller number of states is the merge of all sets of items that have the same core into one set of items, and the new lookahead set will be the union of lookahead sets of the merged sets of items. The number of the new sets of items is exactly equal to the number of LR(0) sets of items (i.e. sets of states).

Consider the LR(1) states mentioned in Fig. 4.11, the cores of the items in the states 4 and 7 are identical. So, it is possible to merge them into one state (say s_{47}) i.e.,

s_{47}:	A:=a.A	a,b,$
	A:=.aA	a,b,$
	A:=.b	a,b,$

Actually the merge has no effect on the parser because there is no reduce action, and it is clear that if the next input symbol is neither a nor b, an error will occur. The set of states 5 and 8 are identical and can be merged into one state (say s_{58}), i.e.

s_{58}:	A:=aA.	a,b,$

The same situation occurs with the states 6 and 9. The complete set of LALR(1) states is shown in Fig. 4.12. The number of LALR(1) states is equal to the number of LR(0) states (see Fig. 4.4).

s_0:	P:=.S	$
	S:=.AA	$
	A:=.aA	a,b
	A:=.b	"
s_1:	P:=S.	$
s_2:	S:=A.A	$
	A:=.aA	$
	A:=.b	$
s_3:	S:=AA.	$
s_{47}:	A:=a.A	a,b,$
	A:=.aA	"
	A:=.b	"
s_{58}:	A:=aA.	"
s_{69}:	A:=b.	"

FIGURE 4.12: LALR(1) States

If a set of LR(1) states has no conflicts, and all states having the same core are merged into one state with a lookahead set equal to the union of all lookahead sets of the merged states, then it is possible that the new set of states will have a reduce-reduce conflict. For example, consider the following augmented grammar

P:=S

S:=aAd

S:=bBd

S:=aBe

S:=bAe

A:=c

B:=c

The set of LR(1) states (Fig. 4.13) has no conflict. So the grammar is LR(1) grammar. Notice that the cores of the states 6 and 7 are the same.

They can be merged into one state (say s_{67}) i.e.,

s_{67}:	A:=c.	d,e
	B:=c.	e,d

This state has a reduce-reduce conflict because with the input symbol (either d or e) the parser can not decide which reduce action should be performed. Hence, the grammar is not LALR(1). However such cases are rare

s_0:	P:=.S	$
	S:=.aAd	$
	S:=.bBd	$
	S:=.aBe	$
	S:=.bAe	$
s_1:	P:=S.	$
s_2:	S:=a.Ad	$
	S:=a.Be	$
	A:=.c	d
	B:=.c	e
s_3:	S:=aA.d	$
s_4:	S:=aAd.	$
s_5:	S:=b.Bd	$
	S:=b.Ae	$
	A:=.c	e
	B:=.c	d
s_6:	A:=c.	d
	B:=c.	e
s_7:	A:=c.	e
	B:=c.	d
s_8:	S:=bB.d	$
s_9:	S:=bBd.	$
s_{10}:	S:=aB.e	$

s_{11}: S:=aBe. $

s_{12}: S:=bA.e $

s_{13}: S:=bAe. $

FIGURE 4.13: LR(1) states

in real life grammars, and in practice, LALR(1) parsing method is considered to be the most practical method.

4.11 CONSTRUCTING LALR(1) PARSING TABLES

The general idea is to construct the set of LR(1) states and if no conflicts arise, merge the states in which all items have the same cores. The parsing table is constructed from the new set of states.

Let G be an augmented grammar, the algorithm of constructing the parsing table will be:

1. Construct the set of LR(1) states. Let this set be $s_0, s_1, \ldots, s_n$.
2. Merge all states in which the items have a same core into one state. The new lookahead sets will be the union of the lookahead sets of all the items merged.
3. The elements of the ACTION table can be constructed from the new set of states in the same way as mentioned in Section 4.9.

The elements of GOTO table can be constructed as follows:

1. Let $\{s_0, s_1, \ldots, s_n\}$ be a set of states having the same core and merged into one state (say s). Suppose that GOTO $[s_0, X] = y_0$, GOTO$[s_1, X] = y_1, \ldots,$ GOTO$[s_n, X] = y_n$. Then $y_0, y_1, \ldots, y_n$ have the same core and can be merged into one state (say y). Now, set GOTO[s,X] to y.
2. All undefined elements are set to error.

The representation of both tables is the same as in Section 4.6.

4.12 OPTIMIZING THE PARSING TABLE

There are two main factors that should be taken into consideration during the construction of the parser. They are the size and the speed of the parser. As mentioned in Section 4.3 the main part of an LR parser, which occupies a large amount of space, is the parsing table. So any reduction in the size of the parsing table will have an effect on the size of the parser as a whole.

As far as the ACTION table is concerned, some states have identical parsing actions. These states can be merged into one state. For example, in Fig. 4.14, states 0,2 and 47 are identical and can be merged into one row.

states	a	b	$	S	A
0	s_{47}	s_{69}		1	2
1			a		
2	s_{47}	s_{69}			3
3			r_1		
47	s_{47}	s_{69}			58
58	r_2	r_2	r_2		
69	r_3	r_3	r_3		

ACTION TABLE GOTO TABLE

FIGURE 4.14: LALR parsing table constructed from Fig. 4.12

The ACTION table becomes

	a	b	$
0,2,47	s_{47}	s_{69}	
1			a
3			r_1
58	r_2	r_2	r_2
69	r_3	r_3	r_3

similar merging could be done with the GOTO table.

It has been mentioned earlier that all undefined elements in GOTO table are set to error. However, in practice these entries will never be used because the function of GOTO table is just to specify the next state after a reduce action has been carried out. Moreover, any error will be caught while the parser[15] consulting the ACTION table. Hence, each row in GOTO table in which all elements are undefined can be erased. For example, GOTO table in Fig. 4.14 will be

	S	A
0	1	2
2		3
47		58

To reduce further the size of the parsing table and increase the speed of the parser, there are some productions (single productions) which are semantically insignificant and are of the form

A:=x

where A is a non-terminal symbol and x is a grammar symbol. The elimination of reductions by such productions will improve the parsing speed because it allows the parser to by-pass the eliminated productions during a parse. For example, consider the following augmented grammar:

P:=S
S:=S,B
S:=B
B:=a
B:=b

The set of LALR(1) states constructed from the above grammar is shown in Fig. 4.15.

s_0:	P:=.S	$
	S:=.S,B	,$
	S:=.B	"
	B:=.a	"
	B:=.b	"
s_1:	P:=S.	$
	S:=S.,B	,$
s_2:	S:=S,.B	,$
	B:=.a	"
	B:=.b	"
s_3:	S:=S,B.	,$
s_4:	S:=B.	"
s_5:	B:=a.	"
s_6:	B:=b.	"

FIGURE 4.15: A set of LALR(1) states

After recognizing an input symbol and reducing it to B, the parser at state 0 consults GOTO table to find the next current state, which is state 4. Here, the parser will do a reduce action by the single production S:=B.

Now, the current state is state 0. Again, the parser consults GOTO table to find the next state which is state 1. The last reduce by the production S:=B can be avoided by letting the parser go directly to state 1 rather than state 4. The size of the ACTION table is reduced by eliminating state 4.

4.13 AUTOMATIC GENERATION OF LR PARSERS

It has been shown that a lot of manual work is necessary to construct the set of states and subsequently both the ACTION table and GOTO table. If there is an automatic generation of such states and tables by a program which accepts a context-free grammar as an input and produces a set of states and parsing table as an output, it will save a lot of time. Fortunately, there exists such programs, such as the YACC program (Yet Another Compiler-Compiler) which is written in the programming language C and runs under UNIX. The user provides YACC with an input file, and YACC builds the LALR(1) parser. This includes the construction of the states. The input consists of three sections, the declarations, productions and programs. They are separated by '%%' marks, i.e.,

```
declarations section
%%
productions section
%%
programs section
```

The first and last sections are optional, and when they are omitted, the layout of the input looks like

```
%%
productions section
```

The productions section consists of one or more productions and each one has the following form,

```
A: BODY ;
```

where A is a non-terminal symbol (left-hand side of the production), and BODY is the right hand side of the same production. The colon and the semi-colon are YACC punctuations. If there are several productions with

the same left hand side, the mark '!' can be used to avoid rewriting the left-hand side. The semi-colon at the end of a production must be dropped before '!'. For example, the productions

```
A: aAb ;
A: bBc ;
A: cCd ;
```

can be written as

```
A: aAb !
   bBc !
   cCd ;
```

All non-terminal and terminal symbols must be known to YACC. This is done by declaring all terminal symbols in the declarations section. Any name not defined in the declarations section is assumed to represent a non-terminal symbol. The terminal symbols are defined as

```
%token name1,name2, ...
```

If there are no conflicts then the user need not supply anything more than the grammar. But when there are shift-reduce or reduce-reduce conflicts, YACC still produces a parser. It does this by selecting one of the valid choices as follows:

1. If there is a shift-reduce conflict, then a shift action is selected.
2. If there is a reduce-reduce conflict, then reduce by the production listed first in the original input.

If the user is satisfied with the default selections, provided by YACC, this will resolve the problem. However, it is possible for the user to provide more information to help YACC resolve the conflicts. This extra information is to specify the precedence and the associativity to the

terminal symbols in the declarations section. This is done by a series of lines beginning with a YACC keyword: %left, %right, or %nonassoc, followed by a list of terminal symbols. For example, consider the production,

```
E:=E '*'E
```

then with the input of the form E*E*E the parser can treat it either as (E*E)*E or E*(E*E) which causes a conflict. So if (*) is chosen as left associative, i.e.

```
%left '*'
```

then the input is treated only as (E*E)*E, whereas, if (*) is chosen as right associative, i.e.

```
%right '*'
```

then the input is treated only as E*(E*E). Therefore the conflict is resolved when the associativity of the symbol (*) is specified.

All of the terminal symbols on the same line are assumed to have the same precedence level and associativity. The lines are listed in order of increasing precedence. For example, in the following declarations

```
%left '+''-'
%left '*''/'
```

(+) and (-) are left associative, and have lower precedence than (*) and (/), which are also left associative. YACC checks the precedence of the terminal symbols, if they are the same then it can apply the associativity to them.

When the input is read by YACC program, an output file is produced. This can be compiled to get an executable parser program. An optional file can also be produced by YACC. This file contains a description concerning the set of states, and also details about the conflicts that might exist.

CHAPTER 5

THE ENCODER

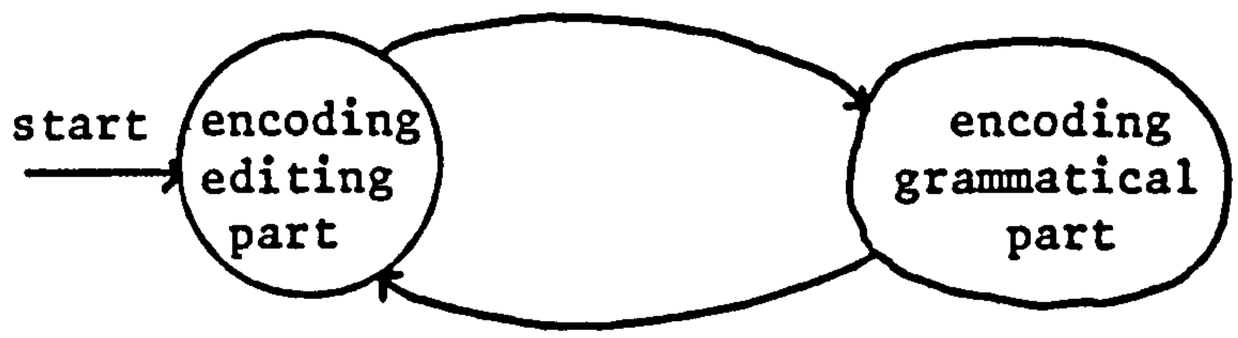
For a given probabilistic context-free grammar G, a design of an encoder program is discussed in which whenever a valid input sequence of symbols is in the language generated by G (together with editing characters and perhaps comments) a corresponding sequence of code words is generated by the encoder. In general, the function of the encoder is to start at a certain state (initial state), encode the input by following some intermediate states, and terminate at a certain state (final state). In any state, only one action is chosen, and when one action is chosen there is no way that, after a while, the encoder will go back and choose an alternative one. Thus there is no backtracking.

Section 5.1 illustrates the basic model of the encoder. Section 5.2 explains the encoding of the grammatical symbols of the input and illustrates its work. The construction of the encoder program is explained in Section 5.3. Different ways of encoding editing characters are explained in Section 5.4. The encoding of comments is illustrated in Section 5.5. Section 5.6 explains the encoding of names and numbers (identifiers). The encoding of strings is explained in Section 5.7. An optimization has been done on the size of the parsing table in order to reduce the total space of the encoder. This is mentioned in Section 5.8. Section 5.9 illustrates the construction of the encoding table which depends on the way of constructing the ACTION table. Section 5.10 illustrates the construction of a program in which the frequencies of all possible symbols, required to be coded, are obtained. Finally, some sample Pascal programs have been submitted to the encoder to find the size of the coded files, and the amount of space saved. These results are recorded in Section 5.11.

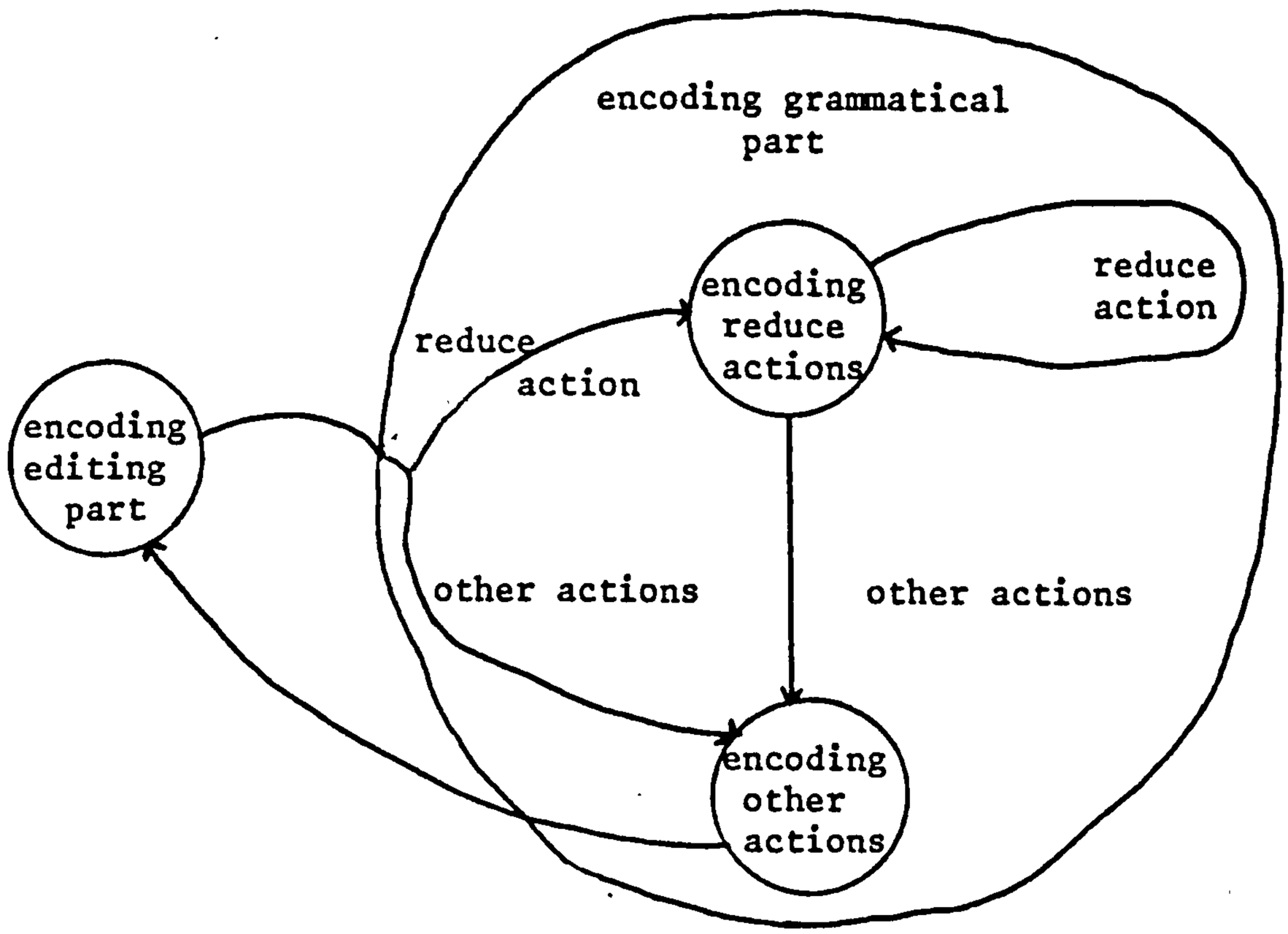
5.1 THE MODEL

The input file contains a mixture of language tokens and editing characters (including comments). So the data can be classified into two parts, the editing part which includes all editing characters and comments; and the grammatical part which includes keywords, identifiers, special symbols and strings.

The encoding procedure, in general, will alternate between the two parts; that is, once the encoding of elements of the first part is accomplished, the encoding of elements of the second part will start. Then returns to the first part, ... and so on. i.e.



However, for the grammatical part, the encoder needs to parse (an LR(K) parsing method is used) each element of it before generating any code. In each state, one of the following three actions (shift, reduce, and accept) will be chosen and a required code will be generated. In all cases, the encoder shifts to the editing part except when the action is reduce. In this action (i.e. reduce), no change on the input symbols will occur, and the current token must be a language token. So, as far as there is a reduce action which does not alter the input file, the encoder does not need to shift to the editing part. Diagrammatically, the relation would be:



Details of the encoding of each part are explained in the following sections in this chapter.

5.2 ENCODING THE GRAMMATICAL PART

Basically, the encoder consists of:

1. A finite set of states $S=\{s_0,s_1,\ldots,s_n\}$; s_0 is the initial state.
2. A sequence of input symbols (I) from a context-free language.
3. A sequence of code words represents an output (O) from a code (H).
4. A push-down stack holds the current state and a grammar symbol at the top.
5. A defining function: for a state s_i on top of the stack and an input symbol a_i from (I), the encoder transfers to a particular state s_j and, if required, generates a code word. The state s_j will be placed on the top of the stack, and the input symbol may or may not be removed. If s_j is not in the set S then the input a_i is not an element of the language. The following notation will be used to express the function:

   current state: (top of the stack, input stream, output stream) → (new top of the stack, remaining input, updated output).

6. There exist a final state s_ℓ in S, and an end of input marker ($), such that the encoder stops the processing when the current state is s_ℓ and the input is $.

Suppose that $a_1a_2,\ldots,\$$ is a string of input symbols (including the end of file marker $) which is required to be encoded. The encoder starts from the initial state s_0 on the top of the stack as the current state, and the output stream is empty (e). This can be expressed as:

$$s_0:(s_0,a_1a_2,\ldots,a_n\$,e) \rightarrow (a_1s_i,a_2a_3,\ldots,a_n\$,O_1)$$

Now, the new current state is s_i on the top of the stack, a_1 is shifted away from the input, and a new code O_1 is added to the output. Continue in this process until it reaches the final state. It can be expressed as:

$$s_j: (\alpha s_j, \$, 0_1 0_2, \ldots, 0_k) \rightarrow (Gs_\ell, \$, 0_1 0_2, \ldots, 0_{k+1})$$

where α is a grammar symbol and G is the initial symbol. To express the whole process:

$$s_0 s_1, \ldots, s_j : (s_0, a_1 a_2, \ldots, a_n \$, e) \overset{*}{\rightarrow} (Gs_\ell, \$, 0_1 0_2, \ldots, 0_{k+1})$$

where $0_1 0_2, \ldots, 0_{k+1}$ is the encoded data of the input $a_1 a_2, \ldots, a_n$ in the form of a sequence of codes. $s_0 s_1, \ldots, s_j$ are the states (some of them are repeated) used by the encoder which are a subset of S. Thus for each string of an input language, there is an equivalent sequence of code words generated by a unique sequence of states.

Transferring from one state to another is equivalent to going one step further to the right of the right-hand side of a particular production. This transfer can be represented by a sequence of code words. Apart from the states and terminal symbols, the stack holds non-terminal symbols. This occurs when a handle production has been found; the right-hand side of the production will be substituted by the left-hand side of the same production. This substitution (reduction) always occurs to the recent updated or read symbols (symbols which are on the top of the stack). So a right-most derivation is applied everytime a handle is found.

The sequence of processing the states can be seen as building a parse tree for a given input, where the input symbols represent the leaves of the tree, and the marker \$ equivalent to recognizing the root of the tree (G). At any intermediate state, the input symbols together with the grammar symbols on the stack are regarded as a sentential form. Since the building of the parse tree starts from the leaves upwards to the root, the set of states involved could be treated as steps of a bottom-up parse.

5.3 THE ENCODER PROGRAM

The function of the encoder program (Fig. 3.2(a)) is to compress partially structured data (a program) written in Pascal language (Appendix A contains the full Pascal syntax). It transforms a fixed length representation into a shorter, variable length representation. The program is sufficiently complex to make it difficult to understand it as a single entity. Therefore, it is preferable to divide the process into a number of small processes connected to each other. Hence, the encoder consists (Fig. 5.1) of two main parts, the parsing part and the encoding part. The operation of the encoder program begins in the parsing part. The data is read and checking will be done as to whether it is syntactically correct or not. During this checking the encoder generates, when required, an appropriate code concerning the parsing, names, constants, ... etc.

5.3.1 The Parsing Part

The parser uses the LR parsing method which has been explained in Chapter 4. At any stage, the parser depends on the current state and the current input symbol to decide the next state. The input symbols are divided into 3 classes: names, constants, and special symbols. The classification of the symbols is the work of a routine called scanner. It also recognizes the editing characters (spaces, tabs and new lines), and the comments which could immediately be encoded because they are not elements of the grammar rules. Names are either keywords such as (begin, end, if, ... etc.) or user names. Numbers are either integers or reals. When the scanner recognizes any element in each class, it passes information to the parser known as a token. The same token is returned

for all user names. Another three different tokens are assigned to the strings, integers and reals. Each remaining recognized symbol . . . has its own token. New names and constants are stored in a table called the symbol table. The algorithm for the scanner would be:

```
Begin
     If input is editing character  or comment
     Then encode the input;
          read new input
     Else If input is name
          Then If it is a keyword
               Then generate the appropriate token
               Else generate a token for a user name;
                    store the name in a symbol table if it is new
     Else If input is a constant
          Then If input is an integer
               Then generate a token and store the number (if new)
                    in the symbol table
               Else generate a token for a real number and store
                    it (if new) in a symbol table
     Else generate a specific token
End
```

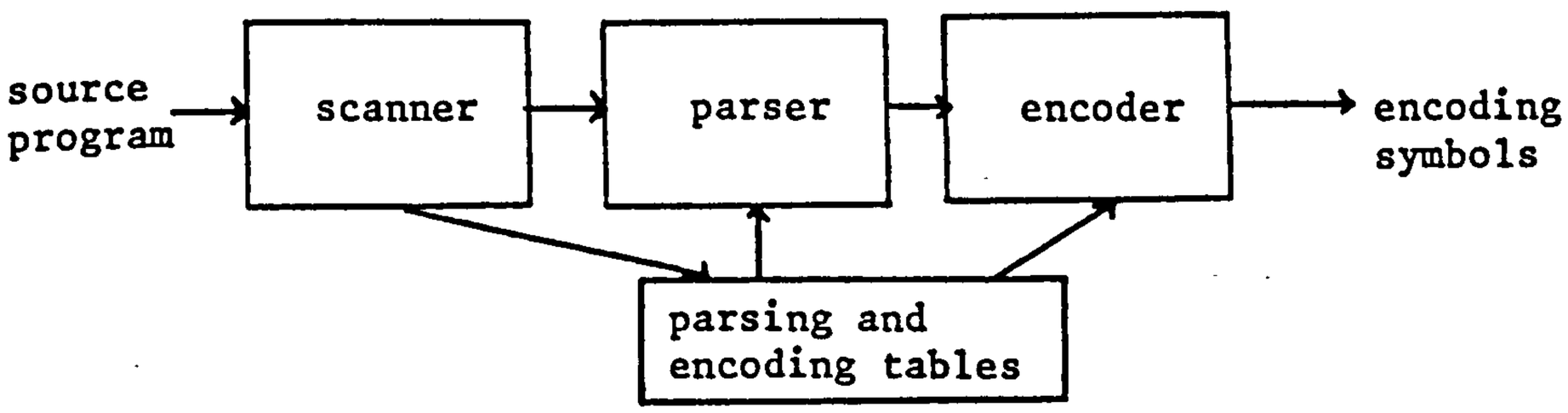


FIGURE 5.1: The encoder program

Assume that the initial state on the top of the stack is the current state, and the scanner provides the next token. The parser can consult the ACTION table to determine the next action. For a shift action, the next state will be the current state and is stored on top of the stack. A new token will be provided by the scanner routine. If a handle is found

then a reduce action is required. The parser removes elements from the top of the stack equal to the length of the right hand side of the handle production. The left hand side of the handle production and the state on top of the stack decide the next current state by consulting the GOTO table. After each action, a specific code could be selected from the encoding tables (see Section 5.9) and passed to the encoding part in order to be stored on the encoded file. Before performing a new action, the encoder checks the token as to whether it represents a user name or represents a constant. In both cases, another code is generated for each character, or a code representing the location of the identifier (name, or constant) in the symbol table would be generated (this is explained in detail in Section 5.6). For a state which has only one choice, the probability of that choice must be one. Therefore, if the state is the current one (in both the encoder and the decoder) then it is certain that the choice would be selected. Thus it is not required to generate any code. If the final state is reached and the next token is end of the input file, then an accept action occurs, and the processing will be stopped. Usually the data submitted to the encoder program is already syntactically correct and no syntax error is expected. Nevertheless, a simple error routine is included in the parser. The algorithm of the parser is:

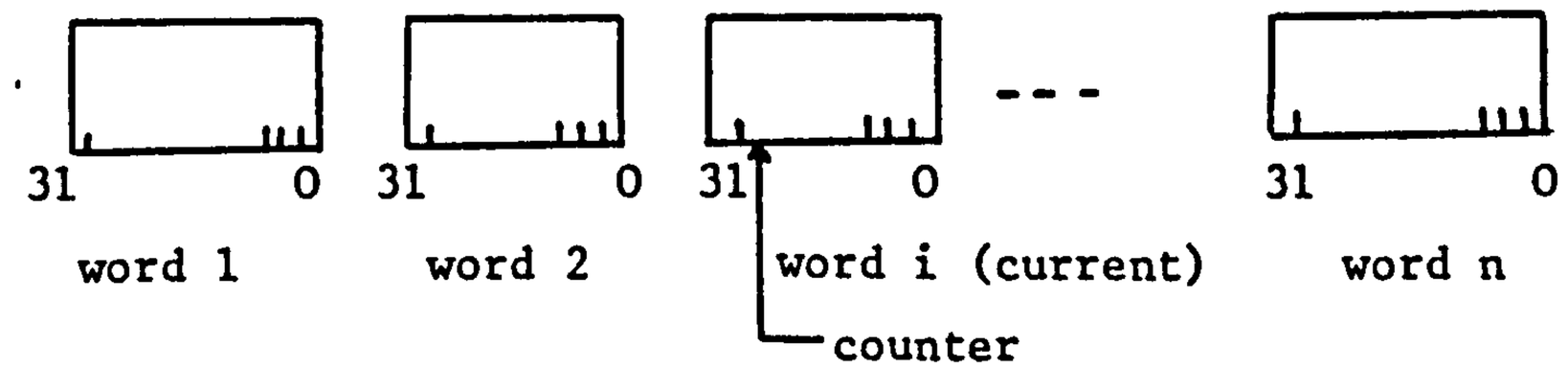
```
Begin
     while the action is reduce
     do pop off the stack elements equal to the length of the handle;
          find a new current state
          generate a code if required
     end do
     if the action is shift
     then push next current state on the stack
          generate a code if required
     else if the action is accept
          then stop parsing
     else error
End
```

The complete parsing part of the encoder program can be seen in Appendix B.

5.3.2 The Encoding Part

Once the input symbol has been parsed, an appropriate code, if any, is generated. The encoder picks up the code from the encoding table and stores it in an output buffer (in the program, the buffer length is two words). Since the code is a variable-length (Huffman code), more than one code could be stored in a buffer word. So, care must be taken in this case, especially when a code has to be stored across two words. Suppose that a code is required to be output, the algorithm would be:

The (binary) code symbols are stored in a buffer so as to make the final output to disc file more efficient. The buffer consists of a certain number of computer words and one of them is partially filled to a point indicated by counter, i.e.,




```
Begin
     If code length + counter ≤ word length
     Then shift the word to the left (code length) times;
          store the code at the right-most of the word without
               destroying the previous stored codes;
          update the counter
     Else shift the word to the left for the remaining unused
               bits in the word;
          store part of the code;
          If the word is the last in the buffer
          Then store buffer on a file
          Else prepare the next word
          clear the counter;
          store the remaining bits of the code in the word;
          update the counter.
End.
```

The full buffer is stored on the output file by redividing into 8 bit

bytes and outputting as if characters. When no more codes are generated, the buffer has to be flushed in order to save the significant code symbols in it.

As far as encoding editing characters, names, constants, and comments is concerned, these will be explained in Sections 5.4, 5.6 and 5.5 respectively. The listing of the encoding routine and its relation with the parsing routine is mentioned in Appendix B.

5.4 ENCODING EDITING CHARACTERS

Editing characters i.e. space, tab , and new line are mainly used to make a computer program more readable, well formatted, and also to separate keywords, identifiers, and numbers. Hence, in general, source programs consist of a high percentage of those characters. They reserve a considerable space compared with the total area occupied by the whole program. So, any attempt to compress these characters and reduce the space which they occupy will have a direct effect on the size of the program as a whole. Usually, editing characters can be used anywhere in the program. At each time one or more characters can be used. For the editing characters, there are different ways of encoding them; such as character encoding, using counters or arrays to encode groups of characters instead of individual characters. So three different methods of encoding editing characters will be presented in subsections 5.4.1, 5.4.2 and 5.4.3.

5.4.1 Character Encoding

The editing characters used in Pascal programs are: spaces, tabs and new line characters. Statistically, the most frequent editing characters are the spaces, followed by the tabs and then the new lines. By applying Huffman codes, the code of each of those characters would be:

space	0
tab	10
new line	11

Since editing characters can be placed anywhere in the program and also there exists a mutual knowledge between the encoder and the decoder, then the encoder must inform the decoder as to whether the next code

represents an editing character or not. This requires the generation of an indicator (one code symbol) before generating any code which may or may not represent an editing character.

Each time an editing character has been recognized, the encoder generates one bit of value 1 as mentioned above, and then generates the code of the editing character. The format would be:

Indicator Code

1	0	space
1	10	tab
1	11	new line
0		no editing character

For example, if there are three spaces and one tab then the sequence of code symbols will be

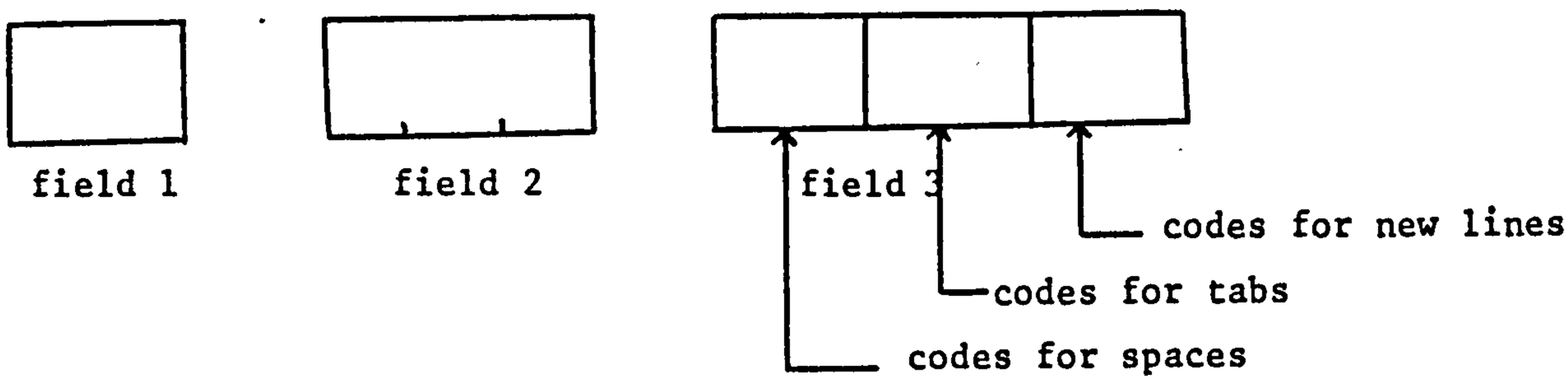
1 0 1 0 1 0 1 10

But, if there are no such characters, the encoder will generate only one code symbol (bit) of value 0. That is to inform the decoder that no editing character is expected next.

5.4.2 Using Counters

Instead of using one editing character at a time, it is possible to use a sequence of subsequent characters without affecting the structure of the program. Hence, before generating any code, assign a counter (accumulator) to each type of the editing characters. This allows the encoder to accumulate all subsequent identical characters until the next character is not an editing character. Then an indicator of value 1 followed by the total number of editing characters followed by codes of all editing characters will be generated. Theoretically, the number of editing

characters which can be used between names, numbers, ... etc. is unlimited. However, in practice it is limited; thus it is required to determine a coding scheme to indicate the total number of characters. The format would be:



where:

Field 1: an indicator;

Field 2: the number of characters;

Field 3: codes of the characters.

Since the number of editing characters which are expected at each time is variable, this will lead to the field 2 to be of a variable length as well. To simplify both the encoding and the decoding processes, a fixed length will be assigned to field 2. Accordingly, field 2 can not hold any number. So a set of ranges is provided; for each range there is a corresponding length assigned to field 2 such that any number within the range can be stored in field 2. The ranges are organized in a way such that the length of field 2 is equal to or multiple of a certain length. Let the initial length be 3 bits, then the set of ranges would be:

Length of field 2	The range of characters
3	1-6
6=3+3	7-62
9=3+3+3	63-510
12=3+3+3+3	511-
⋮	

So, if the number of editing characters is within the range 1-6, then only one field of length 3 bits is required. If the number of characters is within the range 7-62, then 2 fields each of length 3 bits are required. The first field must hold the maximum value (i.e. 111). Obviously, in most programs, the number of editing characters is within the first range. Hence, only 3 bits will be sufficient. The length of field 2 could be changed to 2 bits or multiple of 2 bits. Assume, for example, the number of characters is five, then according to the above format the output would be:

1 101 000010

In the case that the total number of editing characters exceeds six (if the length of field 2 is 3). The encoder must generate the value seven (3 bits of ones) and then subtract 7 from the accumulator. If the remaining number is less than seven then only three more bits containing the new number required to be generated; otherwise generate another three bits of ones (value 7) and carry on subtracting as above. For example, if the number of characters is 7 then the output codes would be:

1 111 000 000001011

The codes for 8 characters would be

1 111 001 0000001011

The idea of generating extra three bits of zeros when the number of characters is exactly 7 is to let the decoder know that the following sequence of code symbols are the codes of 7 editing characters and not more, because the first three bits are always (111) when the number of editing characters is greater than or equal to 7. Generally, the extra 3 bits of zeros will always be added when the number of characters is a multiple of 7.

However, this method can not be applied for any combination of editing characters especially when the encoded file must be reversible, because editing characters can be used in any suitable order. So, the encoding of some combinations of editing characters causes the encoded file to be regarded as irreversible. For example, if the combination of editing characters is two spaces, a new line and one space; then the encoder could be written to print either spaces characters first or new line characters first, i.e.

1 100 00011

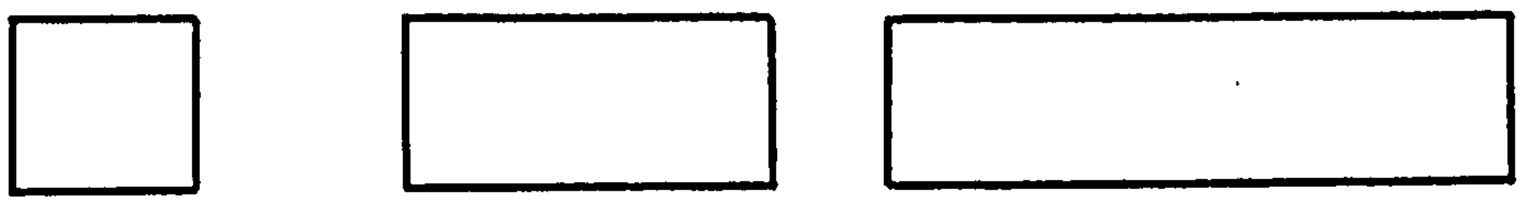
or 1 100 11000

To decode the above sequences, the result would be either three spaces and a new line, or a new line and three spaces; which are different from the original sequence of characters. One way to overcome the above problem is to generate a code after accumulating identical editing characters. Hence only one code representing the editing character will be generated rather than for each character. For instance, the code of the above example would be

1 010 0 1 001 11 1 001 0

5.4.3 Using an Array

The length of editing sequence is unknown, but in practical actual programs, is mostly less than, say x. So the idea is to save editing characters until the next character is not an editing character, or the limit x is reached. Then y editing characters have been collected $y \leq x$ and a code expressing y, followed by codes for each editing character are sent. Values of x=3 or x=7 have been tried in the encoder program. The output format would be:



Field 1 Field 2 Field 3

where:

Field 1: an indicator;

Field 2: a fixed size holds the number of characters;

Field 3: codes of the characters.

The encoder generates one bit of value 1, three bits contain the total number of characters, and then the codes of these characters. This way ensures that the encoded data is reversible. For instance the code of the above example would be

1 100 00110

If there are more than seven subsequent editing characters, then each group of seven characters would be treated independently. So, for each group, the decoder expects only three bits (field 2) containing the number of characters, which is in contrast with the method explained in the previous section.

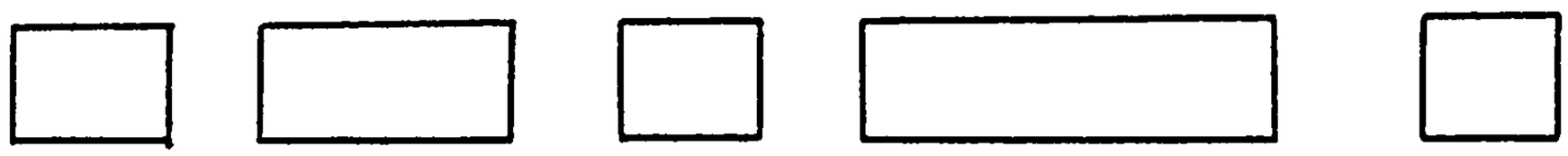
In the encoder program (as has been described in Section 5.3), the scanner routine usually checks for the existence of editing characters. It also stores them in an array. When the array is full or no more editing characters are read, the scanner calls another procedure to generate the required code. The algorithm of the code generator procedure will be:

```
Begin
     If number of characters is not zero
     Then generate a code (value 1) for the indicator;
          generate the length of the characters;
          generate a code for each character in the array;
          clear the field containing the number of characters
End
```

The coding of the above algorithm can be seen in Appendix B under the name (editproc).

5.5 ENCODING COMMENTS

A comment is usually a string of symbols enclosed by either "{ and }" or "(* and *)" (in Pascal). It can be embedded anywhere in the program; and can be of any length. It has no syntactic recognition. So, the parser does not require to check for its existence. The only routine dealing with comments is the scanner which treats them in the same way as the editing characters. When the scanner recognizes the start of a comment, it reads all the characters until the end of the comment. Hence, encoding a comment is part of the scanner's function. The format of the code will be an extention to the format of encoding editing characters mentioned in (5.4.3), i.e.



field 1 field 2 field 3 field 4 field 5

where:

Field 1: an indicator of value 1;

Field 2: contains zeros; (to distinguish from editing characters which have an entry greater than or equal to 1)

Field 3: has value 1 if delimiters are "(* and *)", and 0 if delimiters are "{ and }";

Field 4: codes of the characters;

Field 5: delimiter code.

Fields 1 and 2 are exactly the same as in the editing characters. The only difference is the value of field 2 which is zero in order to distinguish the comment from editing characters, because field 2 will never contain zero in the case of encoding editing characters. The codes

of the characters can be obtained from the character table mentioned in Section 5.9. The last field will contain either the code of "}", or a special code, representing "*)", depending on the type of delimiters used.

The routine which checks for a comment, and passes the necessary code to the encoding part would be:

```
Begin
     generate a code of value 1 (field 1);
     generate a code of value 0 (field 2);
     If input is "(*"
     Then generate a code of value 1 (field 3);
          read new input;
          while "*)" has not been found
               generate character code;
               read new input
          generate a delimiter code
     Else generate a code of value 0 (field 3);
          while input not "}"
                read new input;
                generate character code
     End
```

The above routine could be coded as a separate procedure or as part of the scanner procedure as it has been done in the encoding program (see Appendix B)

As an example, consider the following sequence of symbols together with their codes (Huffman codes)

Symbol	Probability	Code
a	0.32	01
b	0.20	10
{	0.04	1100
}	0.04	1101
*	0.16	001
(	0.08	111
)	0.08	0000
special code	0.08	0001

The codes of the following comment

{(a * b) * a}

would be

(a * b) * a }

1 00 0 111 01 001 10 0000 001 01 1101

For the following comment

(*(a*b)*a*)

The codes would be

(a * b) * a *)

1 00 1 111 01 001 10 0000 001 01 0001

There are two rules for using comments:

1. comment finishes at the first closing delimiter *) or } depending on the opening delimiter.
2. nested comments, that is, a comment can be part of another comment, such as

   {start first comment {second comment} end first comment}

   or

   (*start first comment(*second comment*)end first comment*).

   So after the first closing delimiter (which belongs to the second comment), the encoder must carry on encoding the following characters (because they are part of the first comment) until it reaches the second closing delimiter (which belongs to the first comment).

The encoder program uses the first rule for encoding comments. However, the second rule can be implemented with the condition that the delimiters of the first comment must be different from the inner comments, such as

{start first comment(*second comment*) end first comment}

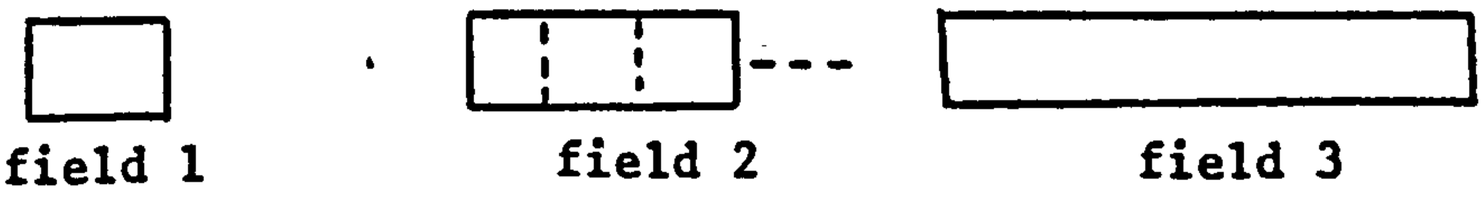
or (*start first comment {second comment} end first comment*).

5.6 ENCODING NAMES AND CONSTANTS

Non-keyword names usually consist of any combination of alphabetical and numerical characters including the hyphen symbol. Any unsigned integers or reals are considered to be constants. These names and constants (identifiers) are almost always repeated more than once in a source program. Therefore, it is necessary to pay some attention to the way these identifiers will be encoded.

Assume that Huffman codes are applied to the characters and symbols involved. The simplest method is to generate a code for each character of the identifier; preceded by the number of elements involved. Specifying the number of elements is important to the decoder because identifiers are usually of variable lengths. The code can be generated everytime the encoder recognizes an identifier despite that some of them could be repeated somewhere in the program. Hence, the same codes could be duplicated in the encoded file which is impractical especially for long identifiers.

The method applied by the encoder program distinguishes between new identifiers and already existing identifiers. For an already existing identifier, the encoder generates a sequence of code symbols different from the sequence of code symbols generated when the same identifier was firstly recognized as a new identifier. Hence, it considers two different formats for encoding such identifiers with the help of a lookup table called symbol table. The first format used for all new identifiers which are immediately stored on the symbol table, and the code is regarded as the sequence of codes of each character of the identifier. It consists of three fields:



where:

Field 1: one bit of value 0 which means that the identifier is a new one;

Field 2: 3 or multiple of 3 bits which is used to store the length of the identifier;

Field 3: the code of each character.

The way of constructing field 2 is exactly the same as explained in the construction of field 2 in Section (5.4.2). As an example, consider the encoding of the identifier (abcd) where the codes of a,b,c, and d are 00,01,10 and 11 respectively. According to the first format, the sequence of bits generated is

1 100 00011011

abcd will be stored in the next available element in the symbol table.

The second format is applied only when the encoder recognizes an identifier which already exists in the symbol table. So instead of generating a code for each character of the identifier, the encoder only needs to generate the location of that identifier in the symbol table. The format consists of two fields

where:

Field 1: one bit of value 1 which means that the identifier already exists on the symbol table;

Field 2: holds the position of the identifier in the symbol table.

For example, if the identifier (abcd) is required to be encoded again, taking into consideration that it is stored in the location zero in the symbol table, then the code would be

1 0

which is rather short compared with the previous code generated for the same identifier.

The problem here is to find the size of field 2. A simple way is to assign a fixed length depending on the maximum number of identifiers which can be stored in the symbol table. This maximum number is actually equal to the size of the symbol table. For instance, if the size of the symbol table is 256 (0-255) then a field of length 8 bits is sufficient to hold the maximum number, i.e. 256. However, it is possible to optimize the length of field 2 by assigning a variable length rather than fixed length. For instance if there is only one identifier in the symbol table, then a size of one bit would be enough. For three identifiers, two bits are necessary and sufficient to hold that number.

To seek a general way of recognizing the size of field 2, consider the following two ideas:

1. Table lookup: the relationship between the locations and the length of field 2 can be expressed in the following table:

Locations	Length of field 2
0-1	1 bit
2-3	2 bits
4-7	3 "
8-15	4 "
16-31	5 "
32-63	6 "
64-127	7 "
128-255	8 "
⋮	

Thus if the start location numbers which cause an increase in the size

of field 2 are stored in an array, then it would be possible to seek the correct length by finding the location of the smallest number in the array which is greater than the location of the current free element in the symbol table. For example, suppose that the size of the symbol table is 256, then from the above table, the length required is 9.

2. The length of field 2 can be expressed by means of logarithms to the base 2 of the current free location. Suppose that n is the current free location, then

   INTeger value of $\log_2(n)+1$ will be the size of field 2.

   For example, $INT(\log_2(4))=2$

   $2+1=3$ bits

   are required when the location is 4. For the location 9

   $INT(\log_2(9))=3$

   $3+1=4$ bits

   are required.

The implementation of the first method is a straight-forward process and it does not require a lot of calculations in order to obtain the field length. But the second method needs a special routine to deal with the logarithmic function and then find the integral part of the result. This required a considerable computing time especially when the process is repeated many times during the encoding procedure. Hence the first method is more economical than the second method and will be used by the encoder program.

Whatever method used to find the length of field 2, there must be an agreement between the encoder and the decoder on the way of recognizing new and old identifiers, and also on constructing the symbol table. That

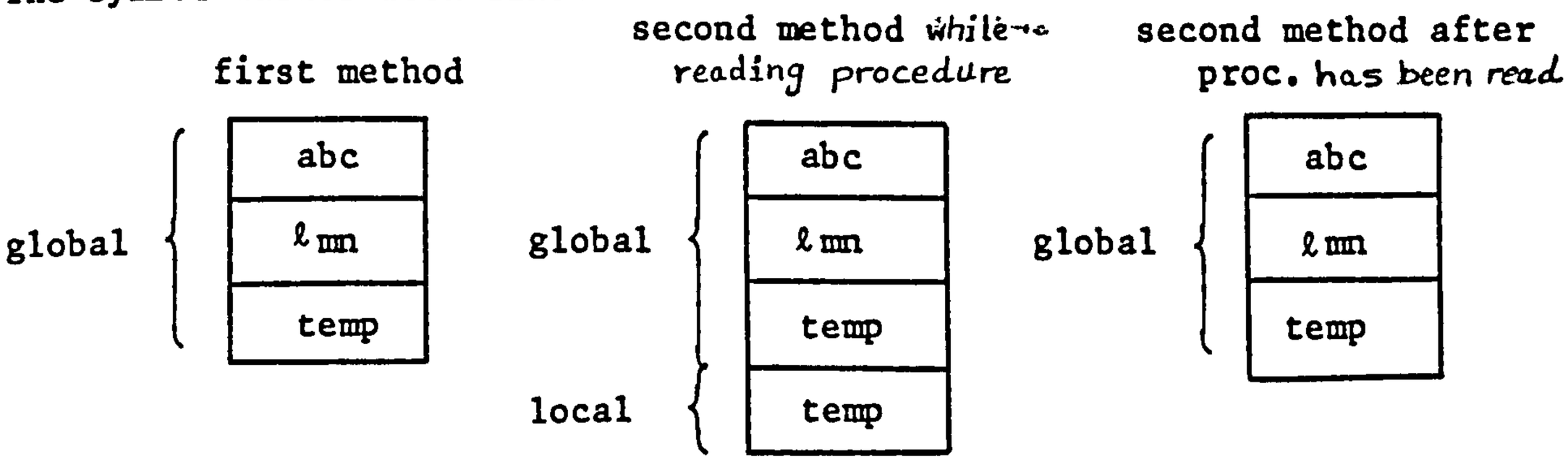
is when the encoder stores a new identifier in a particular location in the symbol table, the decoder, when it recognizes the code of the same identifier, must store it in the same location in its own symbol table. Thus the sequence of the identifiers in the symbol table must be the same in both the encoder and the decoder. This is important because when the encoder generates a code for an old identifier using the second format, the decoder must know (with the help of the indicator) the length of field 2, and also the correct identifier which has been stored in the symbol table.

In a block structured language, identifiers are defined either globally which means that they are accessible throughout the program, or locally which means that they are accessible only inside a part of the program (usually procedures). So, the construction of the symbol table in the simple way (as mentioned above) comes because of the assumption that all identifiers are considered to be global. For instance, the encoder treats both an identifier defined outside a procedure (i.e. global to the procedure), and another identifier which has the same name defined inside the procedure (i.e. local to the procedure) equally the same despite the fact that they are independent. The first identifier is considered by the encoder as a new one; whereas the latter is considered as an already existing one. Actually, this is preferable because the encoder does not need to generate a code for each character of the second identifier, instead it generates only its location in the symbol table.

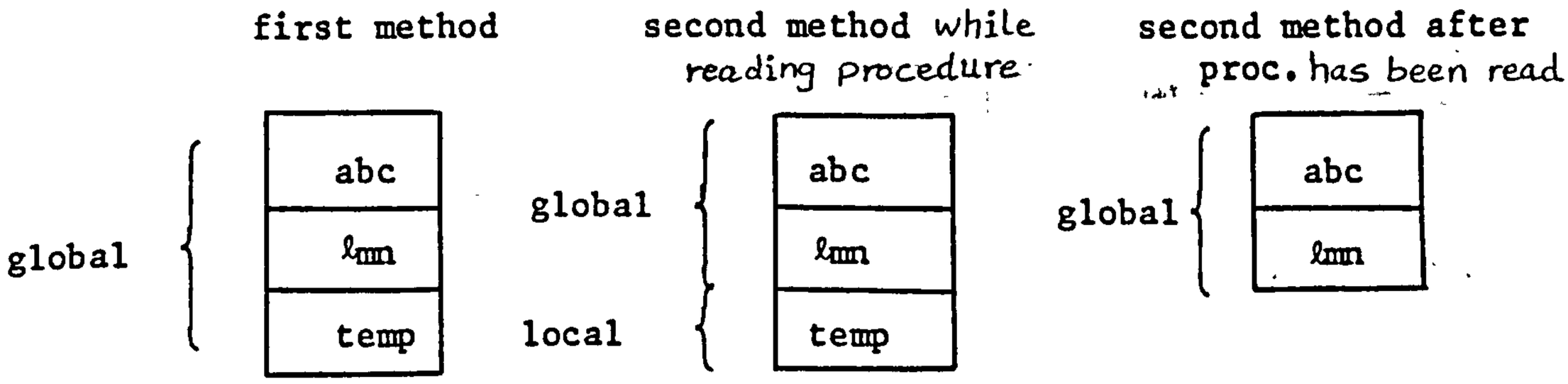
Another method can be used to construct a symbol table, that is when the encoder differentiates between global and local identifiers. It should generate a code for a local identifier as if it is a new one even if there

exists a global identifier which has the same name. When a local identifier becomes unaccessible (not valid outside the procedure in which it has been defined), the encoder should remove it from the symbol table.

As an example, consider (abc,ℓmn and temp) are three global identifiers, and (temp) is a local identifier defined inside a procedure. The symbol tables look like:



As a second example, reconsider the first example but assume that the global identifier (temp) does not exist. The symbol tables would be:



The first method is very simple and does not require much calculation to find any identifier. The size of the symbol table might be bigger or smaller than the size of the symbol table used by the second method depending on the structure of the Pascal program. The first method has been implemented by the encoder for simplicity and because the second method is not uniformly superior.

Usually the scanner recognizes all the identifiers, then searches the symbol table for such identifiers. The search routine would be:

```
Begin
     For all identifiers in symbol table
          If the input identifier exists
          Then save the location;
               return
          Store the identifier  in the symbol table;
          Increment the indicator
     End
```

After the parsing process, a code must be generated representing the input identifier. The routine would be

```
Begin
     If old identifier
     Then begin generate a code of value 1;
                find the length of field 2;
                output the location of the identifier in the
                     symbol table in required length
          end
     Else begin generate a code of value 0;
                output the length of the identifier;
                generate a code for each character
          end
End
```

The coding of the above routines can be found in Appendix B, under the names (lookup) and (check) respectively.

5.7 ENCODING STRINGS

A string is any finite sequence of symbols enclosed by the symbols "' and '", such as

'this is a string'

It is considered as an element of the grammatical part, and hence it is a language token. The symbol "'" can be used inside the string, but it has to be doubled in order to discriminate it from the end of the string, so this allows strings to be nested.

The encoding of a string is almost straightforward; as soon as the scanner recognizes the first delimiter ('), it passes a token to the parser in order to check its syntax, and then generates the grammatical code. Now, the encoder starts generating the codes of all symbols belonging to the string until the last delimiter. In the encoding process the character encoding table will be consulted. At the end of the encoding, no code will be generated for the end delimiter, instead, a special code representing the end of the string will be generated. This allows the decoder to recognize the end of the string; otherwise it can not be sure whether the symbol (') is the end delimiter or it is a symbol within the string. Assume the first symbol is available to the encoder, the algorithm of generating the code of each symbol would be:

```
Begin while the symbol is not "'"
            (generate its code;
             read new symbol
            )
      read new symbol;
      If the symbol is "'"
      Then generate twice the code of "'";
           read new symbol;
           start the algorithm again
      Else generate a code for the end of string
End.
```

5.8 OPTIMIZING THE PARSING TABLES

It has been mentioned in Chapter 4, that the parsing table (ACTION and GOTO tables) is constructed from a set of states of a particular grammar. For a practical grammar in which the number of states might reach several hundreds, this will lead to a large parsing table which is difficult to implement on a computer system. One technique (see Section 4.12) used to reduce the size of the ACTION table is to merge all identical states into one state. This section discusses another way of reducing the size of the ACTION table which depends to a certain extent on the elements of the table. That is, a table in which different rows have elements doing the same function. For example, consider the following table

row no.	1	2	3	4	5	6	7	8	
1	x	x	x						
2	x								
3			x	x			x	x	
4	x	x	x						
5	x	x	x						9 * 8 = 72
6			x	x			x	x	
7	x	x	x						
8			x	x			x	x	
9	x								

where the elements marked by (x) are all different in their functions within each row, and the blank elements are doing the same function within each row. Then, it is possible to construct an equivalent table to the original one but with all blank elements replaced by only one element. The new table now has rows with different lengths. So, it can be divided into a number of subtables, each has rows of equal lengths.

The above table could be divided into three subtables as follows:

	1	2	3	4			1	2	3	4	5			1	2		
1	x	x	x	a		3	x	x	x	x	a		2	x	a		
4	x	x	x	a		6	x	x	x	x	a		9	x	a		
5	x	x	x	a		8	x	x	x	x	a						Total
7	x	x	x	a	4*4=16							3*5=15				2*2=4	35

(a) represents a blank element in the original row.

This way of reducing the size of a table could be implemented on the ACTION table. It has been assumed that the data file submitted to the encoder program is already syntactically correct. Nevertheless, an error could be detected during the parsing process. So any error might occur, the error routine generates a trivial message. Since there is a significant proportion of error elements in the ACTION table (the same thing happens with reduce actions), therefore it is worth applying the above technique in order to reduce its size. Generally, the construction of the subtables could be done as follows:

1. Put all identical states (the states in which all expected inputs are identical) into a subtable.
2. Add the state in which all the expected inputs are part of the expected inputs of a state belonging to any subtable, to that subtable.
3. Put all unique states into one subtable.
4. Substitute all columns which have only an error action by one column doing the same action.
5. In step (2), if a state can be part of more than one subtable, then choose a subtable which has less number of columns.

6. If each row in a subtable has only identical reduce actions, then substitute each row by a new row having only one reduce action (i.e. one column).

The above technique is implemented on the ACTION table used by the encoder program. The size of the ACTION table is

312 * 63 = 19656 memory units

because there are 312 states generated by the YACC program, and 63 tokens (including the end of file marker). This table is divided into 12 different subtables as follows

Table number	Size	Total
0	84*1	84
1	22*3	66
2	16*3	48
3	12*5	60
4	14*5	70
5	7*3	21
6	29*22	638
7	50*17	850
8	12*11	132
9	7*12	84
10	13*9	117
11	6*16	96
		2266
	mapping tables	1378
		3644

Obviously, reconstructing the ACTION table into a new form requires two different mapping tables. The function of the first table is to locate the position of a state in a particular subtable. So each row has information concerning one particular state, the information

determines the subtable containing this state, and also the row number within the subtable. Diagrammatically, it is illustrated in Fig. 5.2.

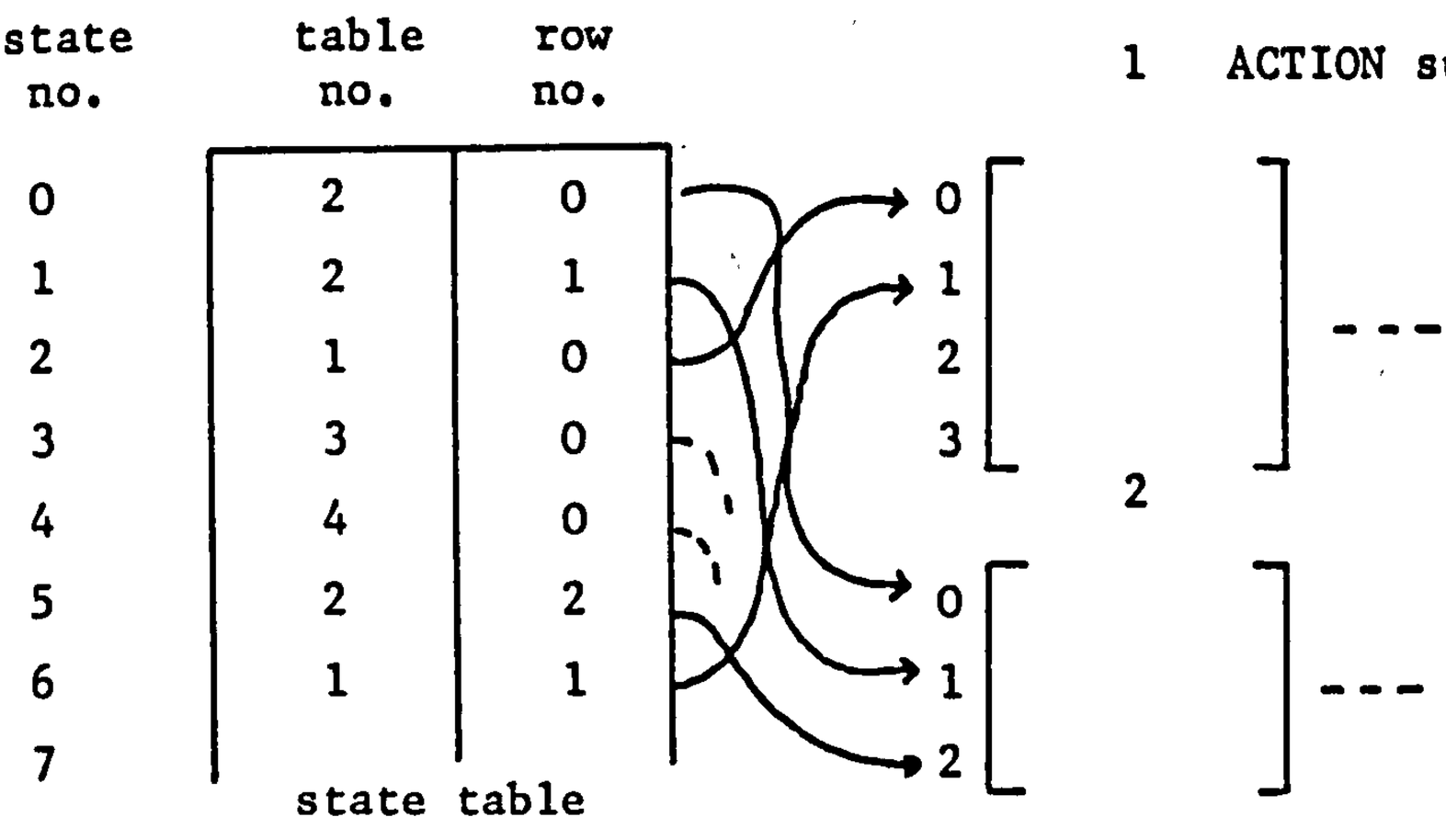


FIGURE 5.2: The state table

As far as the second table is concerned, its function is to renumber the tokens. Within the ACTION table, tokens must have unique numbers. This is the same within each subtable. But a token might have or have not the same token number in different subtables. Each row number in the mapping table represents a token number passed by the scanner routine. The row has 12 elements (because there are 12 subtables), each element has a value representing the new token number for the specified token in the appropriate subtable. This is illustrated in Fig. 5.3. Both these tables require extra space equal to 1.378 memory units. Therefore a tremendous amount of space has been saved by implementing the splitting method on the ACTION table.

To access any element in the ACTION table, it is required to access the mapping tables to find the exact subtable, the row number, and the column number. Then the specified element can be found easily. Thus

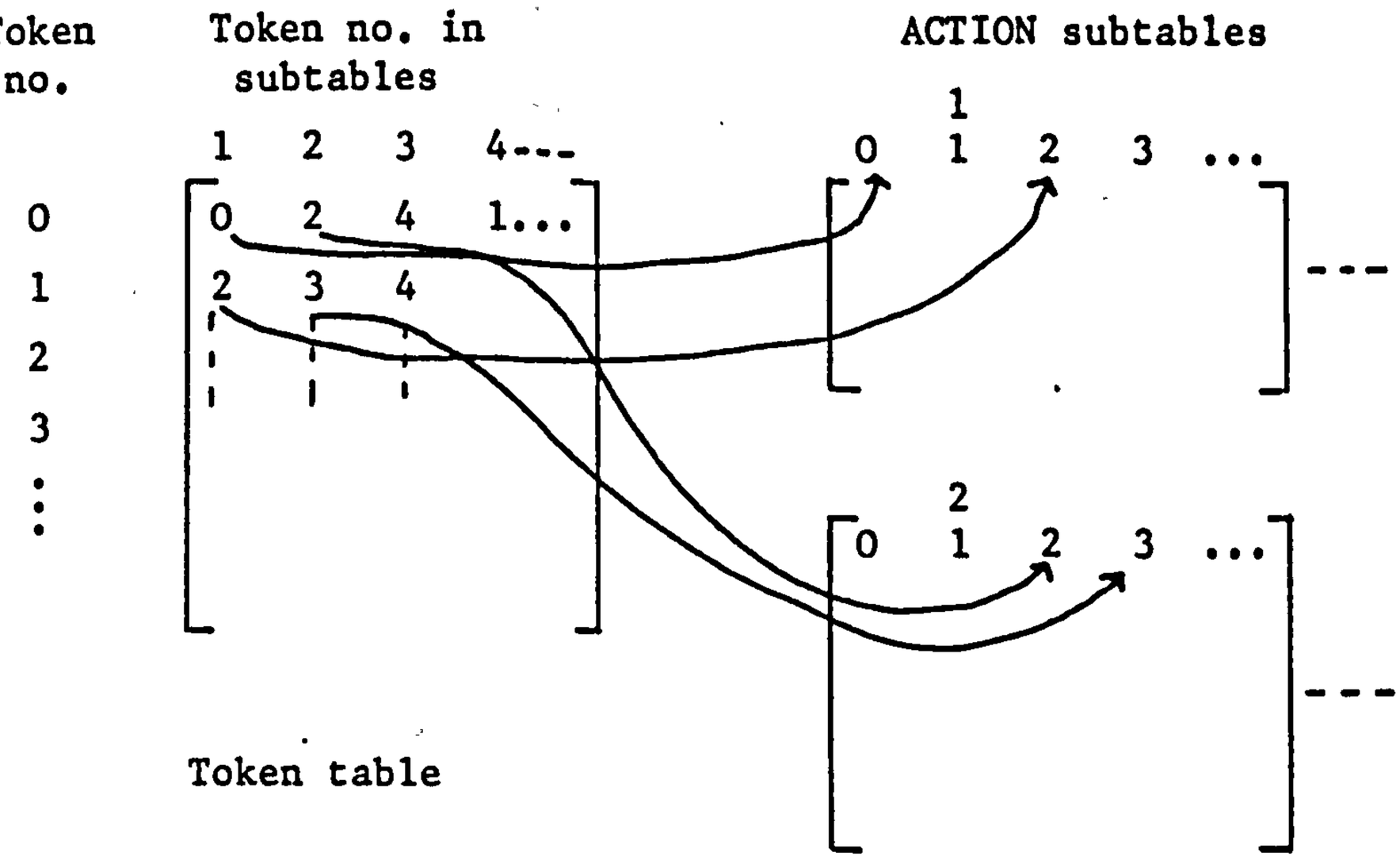


FIGURE 5.3: The token table

there are calculations before the element can be accessed. This will inevitably slow the parsing process. Hence there is a trade-off between the space and the speed.

As far as the GOTO table is concerned, assume that the optimization methods mentioned in Chapter 4 are applied. Since any syntax error could be discovered earlier when the parser consults the ACTION table, and there is no chance of an error in the GOTO table; therefore it is possible to reduce the size of the GOTO table further by exploiting the blank elements in each row. This could be done as follows:

1. Start from the right-most column in the table (say j);
2. If there is a column i such that for all rows r, GOTO[r,i] is a blank element and GOTO[r,j] is non-blank then move GOTO[r,j] to GOTO[r,i]. Else go to step 4.
3. Eliminate the right-most column (i.e. j); go to step 1.
4. Stop.

Instead of choosing the right-most column each time, it is possible to start from the second column and look for the possibility of moving it to the left. It must be done for each column until the right-most column. To illustrate the reconstruction of the GOTO table, consider the following table:

0	1	2	3	4	5	6
x_0	x_1	x_2		x_4		
	x_1	x_2	x_3			x_6
	x_1		x_3		x_5	x_6
		x_2				
		x_2				x_6

where x_i means a state number in column i. The elements of column 6 can be moved to their corresponding elements in column 0. So any reference to an element in column 6 must be diverted to column 0. Column 5 can be moved to column 2; and column 4 moves to column 3. The new table would be:

0	1	2	3
x_0	x_1	x_2	x_4
x_6	x_1	x_2	x_3
x_6	x_1	x_5	x_3
		x_2	
x_6		x_2	

A mapping table is required to direct the parsing program to the new column number, i.e.

0	1	2	3	4	5	6
0	1	2	3	3	2	0

Obviously, another mapping table is required because of the previous optimizations (Section 4.12) which directs the parsing program to a new row in the table.

In the encoder program, the size of the GOTO table before reducing the number of columns is

39 rows * 50 columns = 1950

whereas after the reduction, becomes

39 rows * 22 columns = 858

plus the size of the mapping table which is 50, i.e.

858 + 50 = 908

Finally, to understand practically the construction of both the ACTION table and the GOTO table of the encoder program, a simple example will be given below. Suppose that a context-free grammar is given as follows:

1. S:=E
2. E:=O
3. E:=EPO
4. P:=+
5. P:=-
6. P:=*
7. P:=/
8. O:=V
9. O:=c
10. O;=(E)
11. V:=i
12. V:=i!O

Then the ACTION table generated from a set of states would be

	c	(	i	)	+	-	*	/	!	$
1	s_7	s_{11}	s_8							
2					s_{14}	s_{15}	s_{16}	s_{17}		a
3	R_2	R_2	R_2	R_2	R_2	R_2	R_2	R_2	R_2	R_2
4	s_7	s_{11}	s_8							
5	R_3	R_3	R_3	R_3	R_3	R_3	R_3	R_3	R_3	R_3
6	R_8	R_8	R_8	R_8	R_8	R_8	R_8	R_8	R_8	R_8
7	R_9	R_9	R_9	R_9	R_9	R_9	R_9	R_9	R_9	R_9
8	R_{11}	R_{11}	R_{11}	R_{11}	R_{11}	R_{11}	R_{11}	R_{11}	s_9	R_{11}
9	s_7	s_{11}	s_8							
10	R_{12}	R_{12}	R_{12}	R_{12}	R_{12}	R_{12}	R_{12}	R_{12}	R_{12}	R_{12}
11	s_7	s_{11}	s_8							
12				s_{13}	s_{14}	s_{15}	s_{16}	s_{17}		
13	R_{10}	R_{10}	R_{10}	R_{10}	R_{10}	R_{10}	R_{10}	R_{10}	R_{10}	R_{10}
14	R_4	R_4	R_4	R_4	R_4	R_4	R_4	R_4	R_4	R_4
15	R_5	R_5	R_5	R_5	R_5	R_5	R_5	R_5	R_5	R_5
16	R_6	R_6	R_6	R_6	R_6	R_6	R_6	R_6	R_6	R_6
17	R_7	R_7	R_7	R_7	R_7	R_7	R_7	R_7	R_7	R_7

where s_i=shift and goto state i, R_j=reduce by the production number j, a=accept, empty places mean errors, and $=end of program.

The states 1,4,9 and 11 are identical and can be grouped in one subtable. The states 2,8 and 12 are unique. Finally, the states 3,5-7, 10, 13-17 are identical in their actions and can be grouped in one subtable. Therefore 3 subtables can be constructed:

Subtable 1:

	c	(	i	others
1	s_7	s_{11}	s_8	E
4	s_7	s_{11}	s_8	E
9	s_7	s_{11}	s_8	E
11	s_7	s_{11}	s_8	E

The size is

4 * 4 = 16

where E=error.

Subtable 2:

	)	+	-	*	/	$	!	others
2	E	s_{14}	s_{15}	s_{16}	s_{17}	a	E	E
8	R_{11}	R_{11}	R_{11}	R_{11}	R_{11}	R_{11}	s_9	R_{11}
12	s_{13}	s_{14}	s_{15}	s_{16}	s_{17}	E	E	E

the size is
3 * 8 = 24

Subtable 3:

	others
3	R_2
5	R_3
6	R_8
7	R_9
10	R_{12}
13	R_{10}
14	R_4
15	R_5
16	R_6
17	R_7

the size is
10 * 1 = 10

Two mapping tables are required to specify the exact element. These are state table and token table.

	Table no.	Row no.
1	1	1
2	2	1
3	3	1
4	1	2
5	3	2
6	3	3
7	3	4
8	2	2
9	1	3
10	3	5
11	1	4
12	2	3
13	3	6
14	3	7
15	3	8
16	3	9
17	3	10

state table

	col.no. subtable 1	2	3
c	1	8	1
(	2	8	1
i	3	8	1
)	4	1	1
+	4	2	1
-	4	3	1
*	4	4	1
/	4	5	1
!	4	7	1
$	4	6	1

the size is
17 * 2 = 34

token table

the size is
10 * 3 = 30

The size of the original ACTION table is

$$17 * 10 = 170$$

The total space required by the subtables and the mapping tables is

$$50 + 64 = 114$$

After removing the blank rows, the GOTO table would be:

	1	2	3	4
1	2		3	6
2		4		
4			5	6
9			10	6
11	12		3	6
12		4		

the size is

$6 * 4 = 24$

To move the columns to the left in order to reduce further the size of the table; it becomes

	1	2	3
1	2	6	3
2		4	
4		6	5
9		6	10
11	12	6	3
12		4	

the size is

$6 * 3 = 18$

Again, a mapping table is required to specify any element in the table:

column no.

1	2	3	4
1	2	3	2

the size is

$4 * 1 = 4$

So, the total space required is 22.

5.9 CONSTRUCTING THE ENCODING TABLES

Codes required to be available to the encoder program can be expressed in two groups. The first group comprises codes representing the parsing routes. So each state has its own codes (Huffman codes), and when the parsing action is completed, the necessary code (if required) should be generated by passing the sequence of code symbols to the encoding part, and then stored on the encoded file. Hence, the codes are organized in a set of tables corresponding to the ACTION subtables which have been organized in Section (5.8). Each element of the table consists of two fields (Fig. 5.4), the first one holds the length of the code, and the

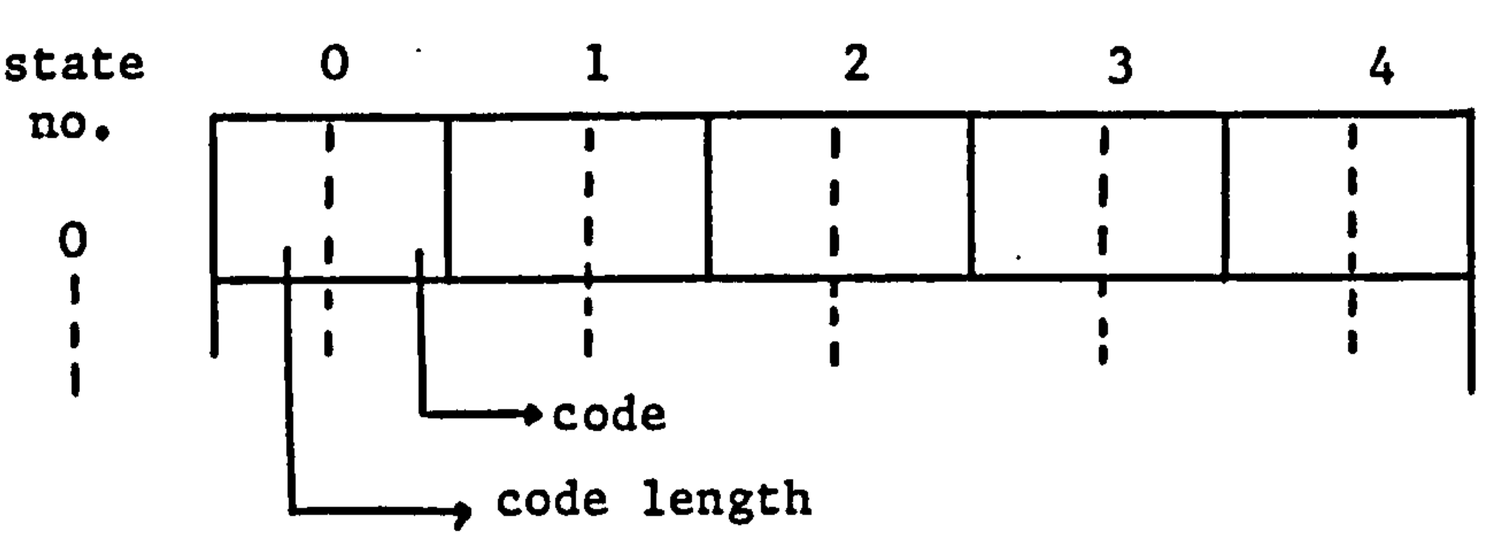


FIGURE 5.4: A row of an encoding subtable

second field holds the code itself. The mapping function will be exactly the same as the mapping function of the ACTION table; hence only one function will be sufficient for both. It might happen that all the elements in a table have zero length codes. This occurs when each state in the corresponding ACTION subtable has only one choice. So, the encoding subtable can be removed.

The second group consists of codes (Huffman codes) of all possible characters that might be used in the program which need to be encoded.

A table of size equal to the number of characters involved is constructed. Each element of the table consists of two fields (Fig. 5.5). The length of the code, and the code. The table is organized in ascending order according to the order of characters recognized by the computer.

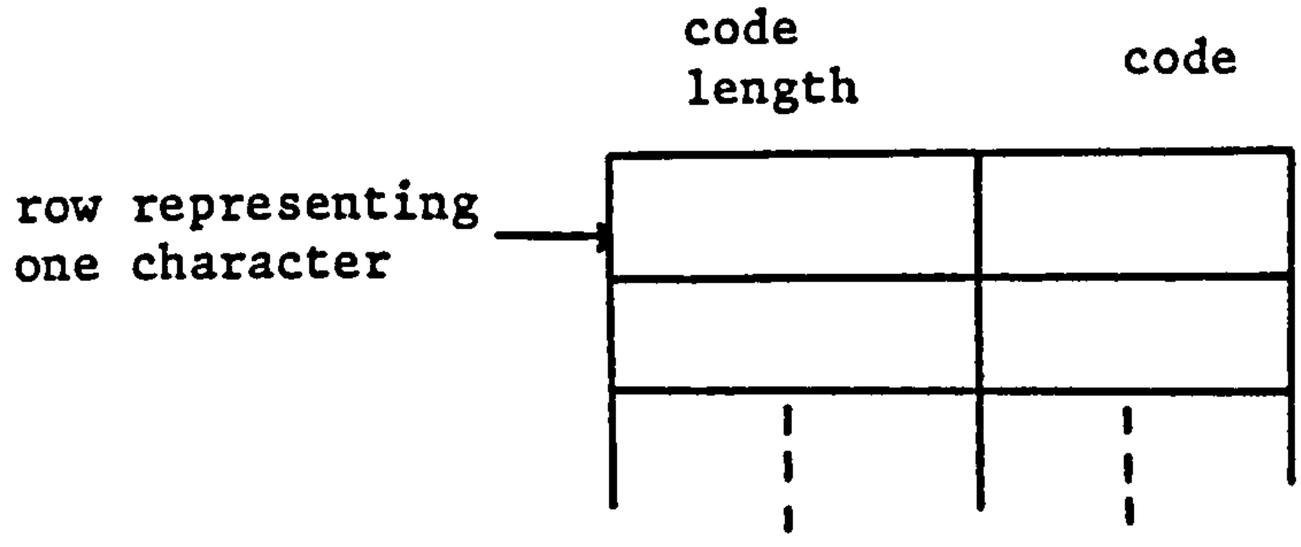


FIGURE 5.5: Character table

5.10 THE FREQUENCY PROGRAM

The function of the frequency program is to provide information needed for constructing the necessary codes which will be used by the encoder program. The information is regarded as the frequencies of different symbols, and also all possible options acquired in each state. Thus, the program can be divided into two different phases; a parsing phase, and a statistical phase. The first phase checks the syntactic structure of the input data before finding the frequencies. The parsing method used is LALR(1) as explained in Chapter 4; and the construction of the parsing table (both the ACTION table and GOTO table) is exactly as explained in Section (5.8). The second phase counts the frequency of each option in each state, and also the frequency of each character involved in identifiers (names and constants), strings and comments. Editing characters are independently treated, and hence a space character (for example) inside a comment is considered different from a space character treated as an editing character. The way of storing the frequencies of each state is to construct a set of tables equal to the set of ACTION tables. Each element of a frequency table is a counter of a particular action in a specific state. Obviously, for a state which has only one option, no counting is required because the encoder does not need to generate any code for an action which is certain to occur in a state. Another case occurs which does not need to find the frequency, that is when a state has only two options. Then whatever the frequency of each option, assign a code of value 1 (one bit only) to one option, and a code of value 0 to the other.

A set of Pascal programs of different sizes has been submitted to the

frequency program in order to get the occurrences of all possible characters, and also the frequencies of each action in all the states. For each state, Huffman method is applied to find its codes. Also the method used to provide codes for each character. These codes are used by the encoder program (Section 5.3), and stored in the encoding tables (Section 5.9). The frequencies were computed over 21 Pascal programs containing a total of 115,119 characters.

5.11 EXAMPLE

Samples of Pascal programs have been chosen to be encoded to see the size of the encoded file of each program. (Table 5.1).

Program size	Encoder 1	Encoder 2	Encoder 3
1231	504	472	464
378	172	168	160
461	220	220	208
822	280	256	256
137	60	60	60
141	72	68	68
148	72	72	68
4266	1620	1560	1528

TABLE 5.1: Sample programs

All the three encoder programs which output the encoded files are the same except in the way of encoding editing characters (see Section 5.4). The first file produced by an encoder program using a counter for each type of the editing characters. The second file produced when the encoder uses an array of size 7; the size of field 2 will be 3. The last file produced by an encoder uses an array of size 3, and the size of field 2 is 2. The last encoded file is the optimal among the others because the number of editing characters between terminal symbols is, in practice, one or two characters. Hence the field 2 of size 2 instead of 3 would be enough.

Table 5.2 illustrates in more detail the sizes in bits of different elements of the program, and their corresponding encoded version. The size of the data file in bits equals the number of characters multiplied by 8 bits (the number of bits for one character). Mostly the size of the

	Editing chars.	Comments	Strings	Identifiers	Others	Size	Waste
original	224	0	0	576	296	1096	
encoded	130	0	0	270	60	460	20
	232	0	0	472	424	1128	
	158	0	0	271	93	522	22
	248	0	0	472	464	1184	
	166	0	0	269	94	529	15
	4248	2848	2056	16992	7984	34128	
	2970	1662	1094	4294	2173	12193	31
	2176	464	312	4152	2744	9848	
	1242	286	220	1280	673	3704	8
	552	320	24	1104	1024	3024	
	374	230	10	457	187	1258	22
	1696	0	152	2712	2016	6576	
	913	0	76	632	403	2024	24
	584	0	240	1208	1656	3688	
	515	0	114	621	407	1657	7

TABLE 5.2: Encoding sample programs

encoded file in characters does not represent the actual file size because some bits have to be added to the last output buffer in order to make it full before storing it. (Waste in table above). The number of bits added to the buffer depends on the size of the buffer, and its range would be between zero and the size of the buffer minus one.

As far as the speed is concerned, the execution time of the encoder has been compared with the compilation time of Pascal programs (Tables 5.3 and 5.4). This is because both programs (i.e. the encoder and the compiler) have parts of their jobs in common such as scanning the input and also the parsing which is the main phase of both programs. The real time is the interval time between the execution command and the end of the execution response. The user time is the actual execution time of

the program. The system time is the time spent by the system to be ready for the execution of the program.

program size	real	user	system
1231	20.0	4.3	2.5
378	19.0	3.2	2.3
461	20.0	2.6	2.7
822	19.0	3.4	2.6
137	20.0	3.0	2.5
141	19.0	3.0	2.6
148	18.0	3.1	2.4
4266	23.0	6.5	2.4

TABLE 5.3: Compilation time

program size	real	user	system
1231	10.0	0.7	1.6
378	10.0	0.3	1.2
461	10.0	0.3	1.4
822	9.0	0.5	1.3
137	9.0	0.1	1.5
141	9.0	0.1	1.1
148	9.0	0.2	1.2
4266	14.0	3.2	1.9

TABLE 5.4: Encoding time

CHAPTER 6

THE DECODER

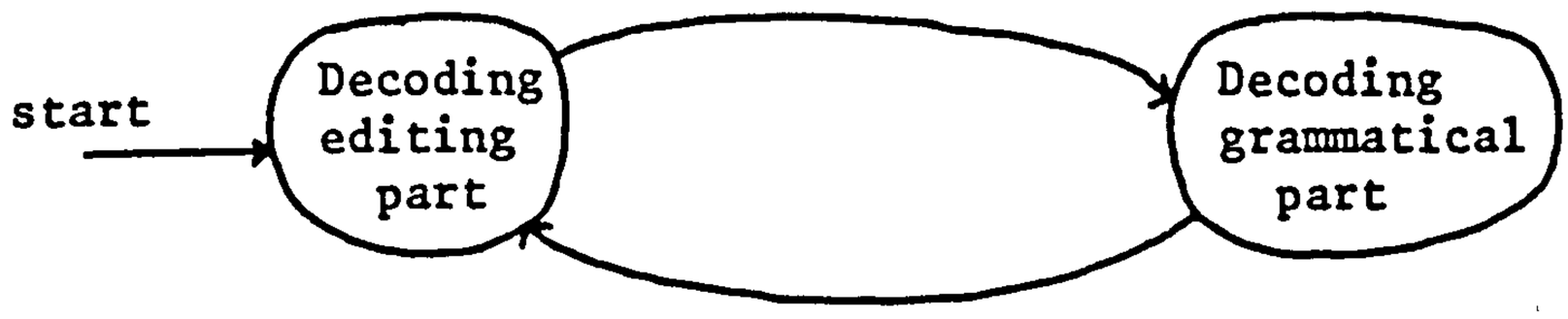
It has been mentioned in the previous chapters that the coded file should be reversible, and accordingly the encoder was designed with this in mind. This chapter discusses the construction of the decoder (Fig. 3.2(b)) which accepts the coded file as an input and produces as an output a file identical to the original one.

The method used for constructing the decoder depends to a certain extent on the encoder, and the strategy used for constructing the codes. So some subjects which have been explained in the previous chapter will not be explained again in this chapter, and only a reference will be made in the appropriate place.

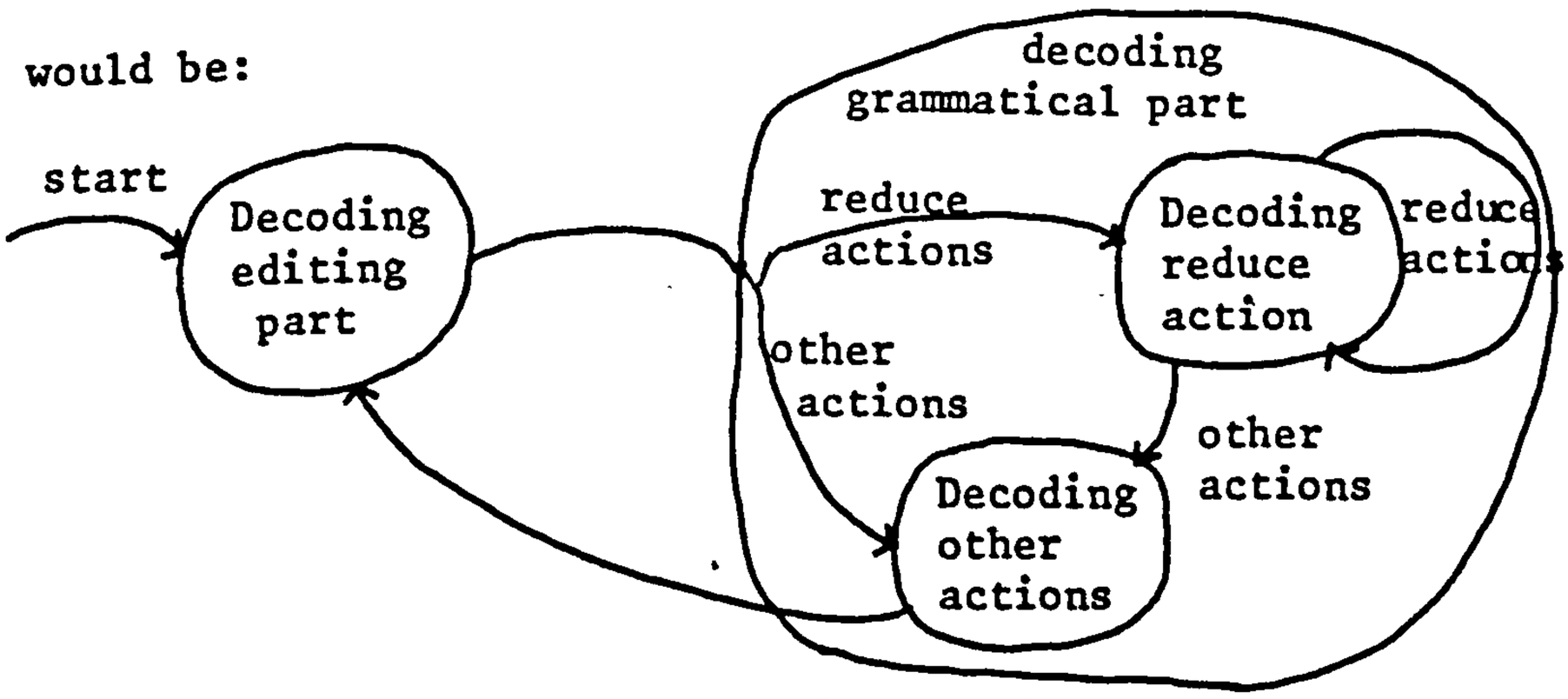
Section 6.1 introduces the basic model of the decoder. Section 6.2 defines the decoding of the grammatical symbols and illustrates its work. Section 6.3 illustrates the construction of the decoder program. The way of decoding editing characters and comments is explained in Section 6.4. Section 6.5 explains the decoding of identifiers (names and numbers). Section 6.6 explains the decoding of strings. The codes and other information required by the decoder are stored in tables which are explained in Section 6.7. Finally, some coded files have been decoded and compared with the original files. This is mentioned in Section 6.8. Also, the speed of the program is mentioned.

6.1 THE MODEL

The coded file, which is the input to the decoding program consists of a sequence of codes representing both the editing characters (including comments) and the grammatical symbols of a context-free language. So, the codes can be separated into two parts; in the first part, the codes represent the editing characters; the codes of the second part represent the grammatical symbols. The decoding model, accordingly, consists of two sections. The first section deals with the decoding of editing characters; whereas the second section deals with the decoding of the grammatical symbols. The decoding process will alternate between these sections, i.e.



For decoding the grammatical part, the same parsing method (LR(K)) used by the encoder (Section 5.1) will be used to select the exact codes and output the required symbols. The decoder needs to parse each code before generating any output. So in the reduce action, no more codes will be selected and the decoding process does not change to the decoding of editing characters. This will be repeated until no more successive reduce actions occur. Diagrammatically, the relation between the two sections of the decoder would be:



Details of decoding each part will be discussed in the following sections of this chapter.

6.2 DECODING THE GRAMMATICAL PART

The decoder consists of

1. A finite set of states $S=\{s_0,s_1,\ldots,s_n\}$; s_0 is the initial state.
2. A sequence of code words from the code alphabet C; stored on an input file.
3. A push-down stack holds the current state and a grammar symbol at the top.
4. A sequence of symbols (characters, numbers and special symbols) represents the output file.
5. A defining function: for a state s_i as the current state on the top of the stack; and a code word c_k from the input file, the decoder transfers to a particular state s_j and, if required, generates the appropriate output symbols. s_j will be on the top of the stack. If s_j is not in S, then an error has occurred. The code word may or may not be removed from the input file.
6. There exists a final state s_ℓ in S, such that the decoder stops the decoding process when the current state is s_ℓ.

The input file is actually the encoded file generated by the encoder. Hence the output file generated by the decoder should be the same as the original file. The decoder works in a similar way to the encoder (Section 5.2) except that the input and the output files of the encoder become the output and the input files of the decoder. Briefly, the work of the decoder could be arranged in steps as:

1. Start from the initial state as the current state.
2. If necessary, find the next allowable code word from the input file.
3. The current state and the word recognized in step 2 determine the next current state.

4. If any terminal symbols have been recognized, write them on the output file.
5. If the final state has not been reached, then goto step 2; otherwise stop and the input file has been decoded.

6.3 THE DECODER PROGRAM

The program consists of two parts (Fig. 6.1). The decoding part and the parsing part.

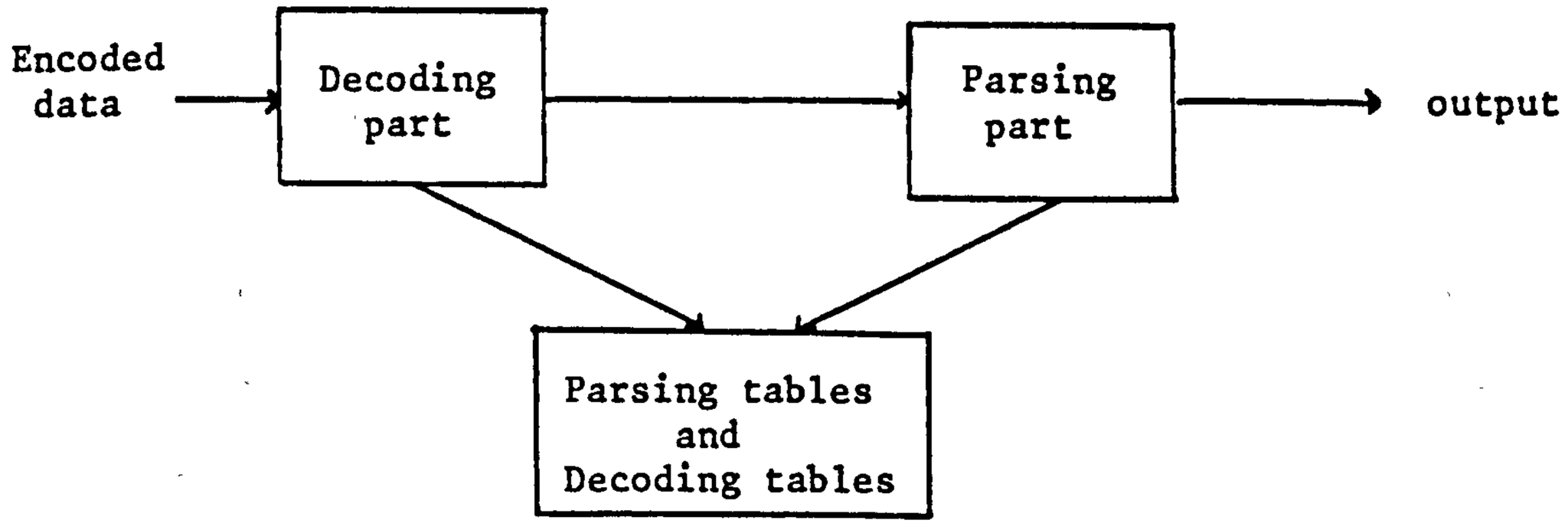


FIGURE 6.1: The decoder program

The way of constructing the parsing part and its function is exactly the same as explained in Section (5.3.1) except that instead of the scanner routine which reads the input and passes tokens to the parser, a decoding part is constructed which supplies the parser with the necessary information such as tokens, and codes for reduce actions to decide which action is going to be the next. It includes routines for decoding editing characters, comments (Section 6.4), and identifiers (Section 6.5). For the decoding part, suppose that the coded input file already exists, and the decoding tables (Tables 1 and 2 described later in Section 6.7) have been constructed. The information required to make the next decoding step is obtained either directly from Table 1, or from the input file. The choice depends on the actions listed for the current parsing state. If only one action is listed, the decoding part can provide it without reading any code from the file. If more than one action exists in a state, then the decoding part should

get code symbols from the input file, and with the help of Table 2, it is possible to find the exact action and then pass it to the parser. The problem with the input file is that the code words are all of variable-lengths (Huffman codes), and the same code word could be used for more than one token or one token might have more than one code word. The decoder, nevertheless, recognizes this problem; so at each state there is a unique code for each action, and the codes of each state are uniquely decodable. The routine for recognizing all tokens and the reduce action would be:

```
Begin
     Find the element of the current state from Table 1;
     If the element is a reduce action
     Then return to the parser
     Else if it is a token number
          Then pass it to the parser
     Else if it is an address to a location in Table 2
          Then identify the next code
          Else error
End
```

The routine of identifying the next code would be

```
Begin
     Get one bit from the input file;
     While the element of Table 2 is a location of a new row
     Do find the new row;
        Get next bit from the file
     If the element is not a reduce action
     Then pass it as a token
End
```

When the parser recognizes a token which causes a shift action, it will check for the type of the token. If it is a keyword, the parser will recognize which keyword it should be, and then output it. In the case of an identifier the routine mentioned in Section (6.5) will be called. If the current token represents a string then the output of the sequence of characters involved will be the same as outputting the characters of a comment. Hence the routine mentioned in Section (6.4) which is used for a

comment would be used to output a string. Additional checking should be added to secure the end of a string, and then output the closing delimiter.

It is assumed that the encoded file should be syntactically correct. Nevertheless, the decoder program checks for any error which might occur as a consequence of corrupted data. The program listings of the parsing part and the decoding part together with other related routines can be found in Appendix C.

6.4 DECODING EDITING CHARACTERS AND COMMENTS

The methods used for decoding editing characters reflect the encoding methods of those characters which have been mentioned in Section 5.4. So as soon as the decoder recognizes the value of the indicator as one, it will definitely know that the following sequence of bits represents a string of characters starting with at least one editing character or a comment (Section 5.5). The codes used by the encoder are variable-length (Huffman codes), and no code is a prefix of another code. Hence, for encoding character by character, the decoder can recognize a character without any doubt that the code might be a prefix of another code which represents another character. Since the code in this case is very simple, i.e.

space 0

tab 10

new line 11

then the decoder does not need to build a decoding table, instead of simple routine will do the job, i.e.

```
Begin
     Get a bit;
     If the bit is zero
     Then output a space
     Else get next bit
          If the bit is zero
          Then output a tab
          Else output a new line
End
```

In the case of using counters in the encoding process, the decoder needs to know the number of characters expected next. Suppose that the length of field 2 is 2 bits (i.e. counts maximum 3 characters), the decoder reads these 2 bits and checks with the number 3; if the number is greater than or equal to 3 then reads the next 2 bits and checks again as above.

In such case the decoder should accumulate the number in order to find the total number of characters. The above routine will be repeated as many times as the total number of characters. If an array was used in the encoding process, the decoder will behave in the same way as above except that only one field 2 exists.

Decoding a comment involves decoding the delimiters and the string of symbols bounded by them. Each symbol can be decoded by searching a tree which recognizes all possible symbols. The tree can be constructed as a 2-dimensional table (Fig. 6.2) in which row has 2 elements. The value of each element could either be a character, or an address to another row in the table. For a binary digit, the first and second elements can be

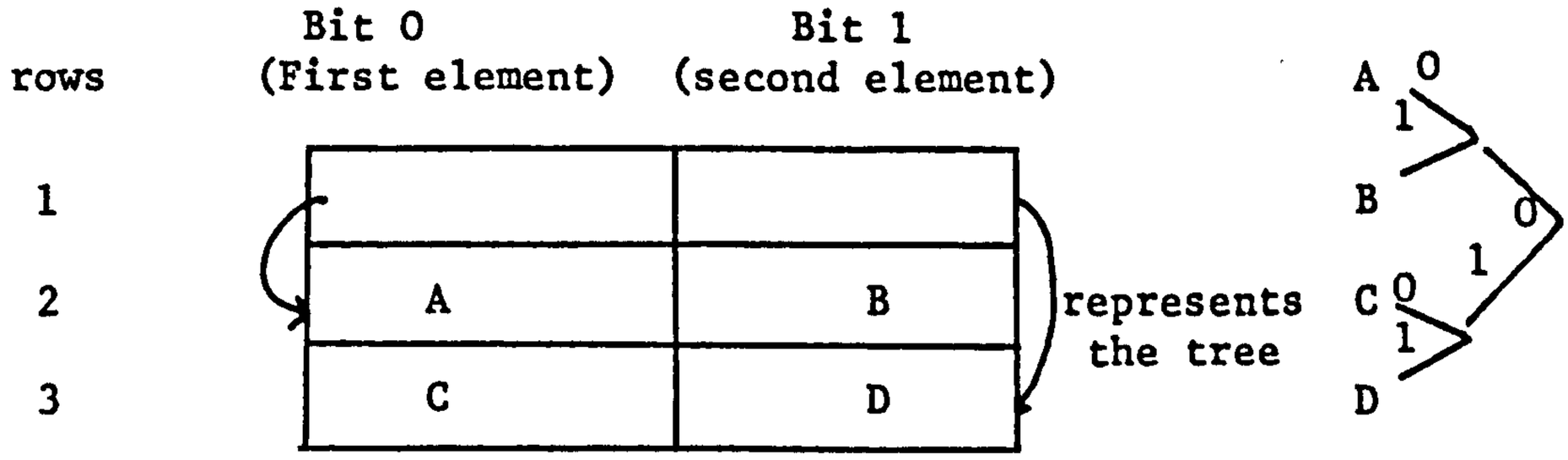


FIGURE 6.2: An example of a character table consists of 4 characters

accessed when the code symbols are respectively zero and one. For example, from Fig. 6.2, by starting from row 1, if the next bit is zero, the value of the first element is an address points to the second row. Suppose that the next bit is 1, then the value of the second element represents the character B.

The existence of a comment comes from checking the field 2 length for the value of zero. The next step is to check for the type of delimiters used. If the value of the next bit is one, then the delimiters are "(* and

*)"; otherwise they are "{ and }". The routine for outputting a comment would be:

```
Begin
     If the field 2 length is zero
     Then If the next bit is one
          Then output "(*"
          Else output "{"
               decode characters until the next delimiter
End
```

The routine of finding each character would be

```
Begin
     Start from the beginning of the character table;
     get a new bit;
     While the element is an address
     Do find the next row;
        get a new bit;
     If character represents "*)"
     Then output "*)"
     Else If character is "}" and the opening delimiter was "{"
     Then output "}"
          Else output the character;
              start the routine again
End
```

The coding of the above routines can be found as two separate procedures in Appendix C called (edit proc), and (rd chars). After decoding editing characters or comments, there is a chance that the next code symbols represent new editing characters, or another comment. In this case the same routines will be repeated.

6.5 DECODING NAMES AND CONSTANTS

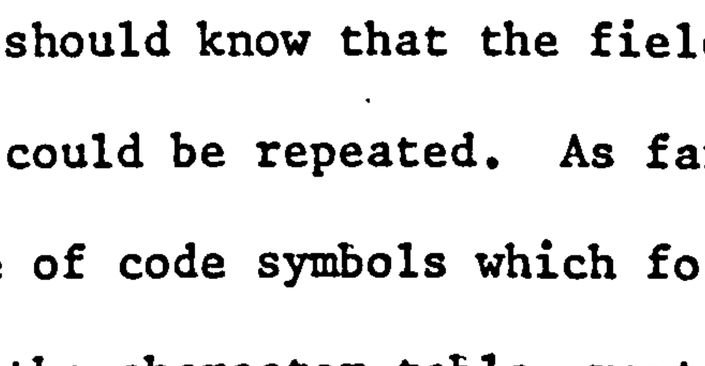
The method used to decode names and constants (i.e. identifiers) should be influenced by the way of encoding such identifiers. Otherwise the decoder would fail to provide a decoded file. As it has already been mentioned in Section (5.6) 2 formats have been used for encoding new and old identifiers with the help of a symbol table. The decoding method has to recognize these 2 formats in order to decide whether the next identifier is a new one or it already exists in the symbol table.

If a new identifier (the first bit has value 0) is expected next, the decoder should (according to the first format) find the length (number of characters) of this identifier and then output each character of it by recognizing their codes in the coded file. The new identifier must be stored in the current free location in the symbol table just in case the same identifier may occur again. The location of the identifier in the symbol table will be exactly the same location of the identifier when it was first encoded.

The decoder should know that the field 2 is of fixed number of bits (3 bits), and it could be repeated. As far as recognizing each character from the sequence of code symbols which follows field 2 is concerned, the decoder consults the character table, mentioned in Section (6.4), to output the exact characters of the identifiers. The identifier will later be stored in the symbol table. For example, suppose that

0 011 010011

is a sequence of code symbols represents an identifier, and the character table is



Then the first bit indicates that the identifier is a new one. The next 3 bits have value equal to 3 (the number of characters). Then according to the character table, (01) represents b, (00) represents a, and (11) represents d.

When an old identifier (the first bit has value 1) has been recognized, it must be in the symbol table, and the following sequence of bits in the coded file represents the location of the identifier in the symbol table. The decoder must find the exact number of bits concerned. It will apply the same method used by the encoder (see Section 5.6) which decides the size of this field. As soon as the location is recognized, the decoder can easily output the identifier. The routine for decoding identifiers would be

```
Begin
     If the next bit is one
     Then find the location in the symbol table;
          output the identifier
     Else find number of characters;
          store identifier in the symbol table;
          output the identifier;
          increment the symbol table pointer
End
```

The method of finding the location of an identifier in the symbol table reflects the method used by the encoder. It checks the length of the field 2 and then finds its value. The routine of finding the number of characters is

```
     Begin
          get 3 bits;
          accummulate the number;
          If the number equal to 7
          Then repeat the routine
     End
```

6.6 DECODING STRINGS

The decoding of each symbol in a string is similar to decoding the elements of a comment (Section 6.4). As soon as the parser recognizes a string (i.e. recognizes a code which is representing a string token), the decoder starts decoding and outputting all symbols belonging to the string until it reaches the delimiter code (a special code indicating the end of the string which is generated by the encoder). Obviously, the delimiters "› and ›" are included in the output.

The routine for finding each character is a modified version of the routine mentioned in Section 6.4. It becomes:

```
Begin
     start from the beginning of the character table;
     get a new bit;
     while the element is an address
     do find the next row;
        get a new bit;
     if character represents end of string
     then output the symbol "›"; stop;
     Else if character represents "*)"
     Then output "*)"; stop;
     Else If character is "}" and the opening delimiter was "{"
     Then output "}"; stop;
     Else output the character;
     Start the routine again;
End
```

6.7 CONSTRUCTING THE DECODING TABLES

The information in the decoding tables is classified into 2 groups. The first group helps to recognize characters and symbols of comments, strings, and identifiers. The construction of this group has been mentioned in Section (6.4). The second group, with the help of the coded file, directs the parser to the next state, and if required, outputs the appropriate words such as keywords or special symbols. Thus each state has required some information to deal with it. This depends on the type and number of actions permissible in the state. A state might require only one action such as shift or reduce; or one of many possible actions is required. If there is only one choice, the decoding table can pass it directly to the parser without reading any code symbols from the coded file. This is true because the encoder in such cases does not need to generate any code (i.e. an action which is certain to occur). For a state which has different choices of actions, a tree is constructed. The decoder needs only to read code symbols from the coded file and follows the appropriate path to decide the exact action. Hence, two different decoding tables are required (Fig. 6.3).

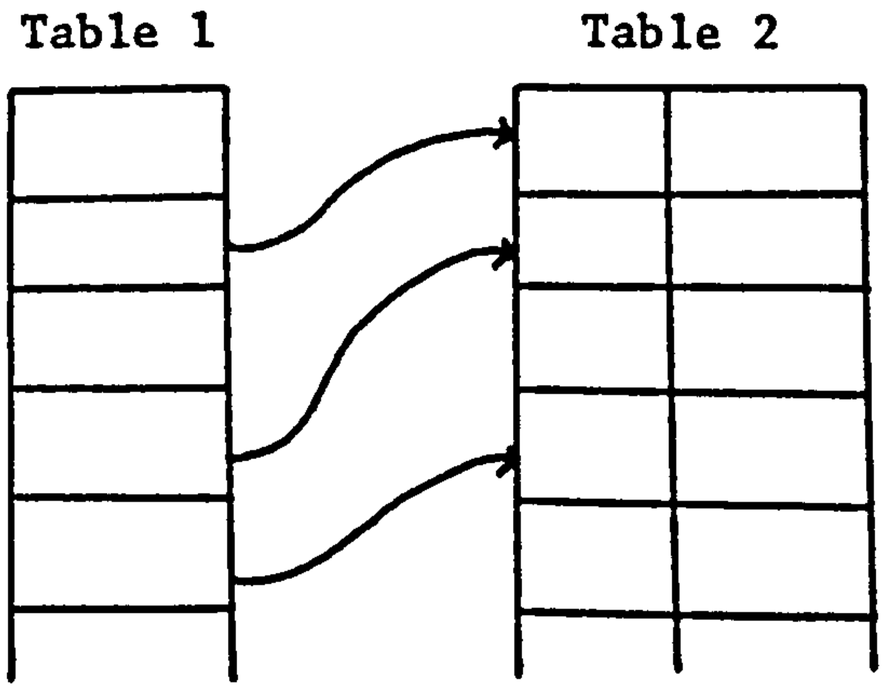


FIGURE 6.3: Decoding tables

Table 1 has elements equal to the number of states used by the parser.

This table should be consulted first before reading any codes. Each element has one of three different values:

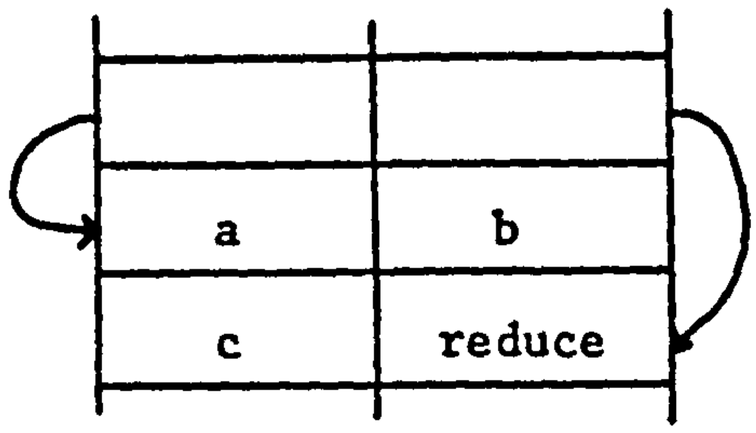
1. A value representing a token number,
2. A value representing a reduce action,
3. A value representing a pointer to a specific location in table 2.

The first and the third values are within two separate ranges so that the decoder can distinguish between them. For example, the token numbers are within the range (1000-1062), and the pointers are within the range (0-683). The first and second values occur when a state has exactly one action. Table 2 is regarded as the concatenation of different tables which have the same format. Each individual table represents a code tree for a specific state, and the construction of these tables is exactly as the construction of the character table which has been mentioned in Section (6.4). The value of each element could either be an address, or a token number, or a value represents a reduce action. The concatenation of those tables into one large table necessitates storing their start locations in table 1.

In the case of a state which has different reduce actions depending on the next input, each action has its own code. Table 2 uses these different codes to inform the parsing part of the decoder which production to reduce by. For example, consider the following state:

```
state i
        a shift j     00
        b shift k     01
        c reduce x    10
     others reduce y  11
```

part of Table 2 would be



For the code (10), the symbol (c) will be passed to the decoder, and from the ACTION table, the action will be a reduce by the production x.

6.8 EXAMPLE

The same samples of Pascal programs mentioned in Section (5.11) which have been encoded by the encoder program are applied to the decoder program (Table 6.1). The size of each file produced by the decoder program is exactly the same as the size of the original file. So, there is no loss in characters and more important, the layout of the new file is exactly similar

Program number	Original Size	Encoded Size	Decoded Size
1	1231	464	1231
2	378	160	378
3	461	208	461
4	822	256	822
5	137	60	137
6	141	68	141
7	148	68	148
8	4266	1528	4266

TABLE 6.1: Decoding samples of Pascal programs

to the original file, i.e. the encoded file is reversible.

Again, the execution time of the decoder program (Table 6.2) has been compared with the compilation time mentioned in Table (5.3).

Program number	real	user	system
1	7.0	1.1	0.9
2	5.0	0.4	0.7
3	6.0	0.5	0.9
4	6.0	0.6	0.8
5	6.0	0.1	0.8
6	6.0	0.1	0.9
7	6.0	0.1	0.9
8	10.0	3.8	0.9

TABLE 6.2: The execution times of the decoder

The execution time of the decoder is similar to the execution time of the encoder. But it takes less time to decode a file than compiling the original file. The explanations of different times (real, user, system) are mentioned in Section (5.11).

CHAPTER 7

CONSISTENT GRAMMARS AND

THE PROPERTIES OF A LANGUAGE

Probabilistic languages and grammars have been defined and the relation between them has been studied in Chapter 3. This chapter discusses a probabilistic grammar that can generate a probabilistic language. The grammar is called a consistent grammar. By examining the structure of a consistent grammar, it is possible to discover interesting properties such as the average length of a string[†] (average program size). Each string has to be parsed before encoding it; this requires the derivation of the string from the grammar rules. So from the grammar, the average number of derivations required to parse a string of symbols from a probabilistic context-free language can be obtained. Another important subject of this chapter is to find out the average length of a coded file generated by the LR encoder. This will be used as the basis for testing the efficiency of the method.

Section (7.1) provides some definitions and notation concerning matrices which will be used throughout the chapter. In Section (7.2), the expectation matrix is constructed from a probabilistic grammar. From this matrix, it is possible to prove the consistency of the grammar. The method of obtaining the average length of a string of symbols is described in Section (7.3). Any string has to be derived from grammar rules (productions). The average number of derivations is explained in Section (7.4). In right-most derivations, each syntactically correct string can be represented by a set of states. The average number of states is explained

---

[†]*Note in this chapter "string" refers to a complete sentence in a context-free language.*

in Section (7.5). Section (7.6) illustrates the probability distribution of each state. The minimum length of a coded file is explained in Section (7.7). In Section (7.8), the average code length of a coded file is illustrated. Finally, a comparison, through an example, between two different encoding methods using the parsing encoding technique is explained in Section (7.9).

7.1 NOTATION AND DEFINITIONS

Some notation and definitions are given in this section which will be used in the following sections of this chapter. These fundamental definitions can be found in most books which deal with matrices such as Campbell, 65; Wilkinson, 65; and Jennings, 77. Denotations:

A = (n*n) matrix

A^{-1} = The inverse of the matrix A

A^{T} = The transpose of the matrix A

det(A) = The determinant of the matrix A

$|N|$ = Length of the vector $\underline{N}$

I = Identity matrix.

An eigenvalue and corresponding eigenvector of a matrix satisfy the property that the eigenvector multiplied by the matrix yields a vector proportional to itself. The fundamental algebraic eigen-problem is the determination of those values of λ for which the set of n homogeneous linear equations in n unknowns

$$A\underline{x} = \lambda\underline{x}$$

or

$$(A-\lambda I)\underline{x} = 0 \qquad (1)$$

has a non-trivial solution. λ can be obtained as

$$\det(A-\lambda I) = 0$$

The values of λ are called the eigenvalues of the matrix A. Corresponding to any eigenvalue λ, the set of equations (1) has at least one non-trivial solution $\underline{x}$. Such a solution is called an eigenvector corresponding to the eigenvalue. If $\underline{x}$ is a solution of (1), then $k\underline{x}$ is also a solution for any value of k. It is convenient to choose k so that the eigenvector has some

desirable numerical property, and such vectors are called normalized vectors.

The eigenvalues of A^T are, by definition, those values of λ for which the set of equations

$$A^T\underline{y} = \lambda\underline{y}$$

has a non-trivial solution. These are the values for which

$$\det(A^T-\lambda I) = 0$$

and since the determinant of a matrix is equal to that of its transpose, the eigenvalues of A^T are the same as those of A.

If pairs of rows and corresponding columns of a matrix are interchanged, the eigenvalues remain the same.

Given a matrix A and a scalar c, then cA is a matrix of the same size as A, in which every element in it is the result of multiplying every entry of the matrix A by the scalar c.

7.2 CONSISTENT GRAMMARS

The aim in this section is to determine necessary and sufficient conditions on a probabilistic context-free grammar to ensure that the summation of probabilities over all strings of a context-free language (L) must total one, i.e. for all α in L

$$\sum_{\alpha \in L} p(\alpha) = 1 \qquad (1)$$

This can be achieved by examining the probabilistic context-free grammar which generates (L). The probabilistic grammar which satisfies equation (1) is called consistent grammar (Wetherell, 80). Therefore, it is necessary to study the conditions under which the grammar is consistent.

For a context-free grammar, all the productions have the form

A:=α

where α may contain either zero, one, or a number of non-terminal symbols. For example, the production

A:=aABcC

has got one A, one B and one C. So during the derivation process, if (A) is substituted in a sentential form, this rule will produce another (A), one B and one C. The occurrence of each non-terminal symbol can be defined in the following matrix (Wetherell, 80).

Definition: The matrix C has $|R|$ rows indexed by the productions and $|V_N|$ columns indexed by the non-terminal symbols. Element c_{ij} is the number of occurrences of non-terminal symbol v_j on the consequence of the production R_i, i.e.

$$C = \begin{array}{c} \\ R_1 \\ R_2 \\ R_3 \\ \vdots \\ R_k \end{array} \begin{array}{c} \begin{array}{cccc} V_1 & V_2 & V_3 & \cdots\cdots V_N \end{array} \\ \begin{bmatrix} c_{11} & c_{12} & c_{13} & \cdots\cdots c_{1N} \\ c_{21} & c_{22} & c_{23} & \cdots\cdots c_{2N} \\ & & & \\ & & & \\ c_{k1} & c_{k2} & c_{k3} & \cdots\cdots c_{kN} \end{bmatrix} \end{array}$$

Each production has a probability attached to it; the probabilities can be arranged in a matrix defined as follows:

Definition: The matrix Q has $|V_N|$ rows indexed by non-terminal symbols and $|R|$ columns indexed by productions. An element q_{ij} is the probability of the production R_j which has the non-terminal V_i as its premise and zero otherwise, i.e.

$$Q = \begin{array}{c} \\ V_1 \\ V_2 \\ \vdots \\ V_N \end{array} \begin{array}{c} \begin{array}{cccc} R_1 & R_2 & R_3 & \cdots\cdots R_k \end{array} \\ \begin{bmatrix} q_{11} & q_{12} & q_{13} & \cdots\cdots q_{1k} \\ q_{21} & q_{22} & q_{23} & \cdots\cdots q_{2k} \\ & & & \\ & & & \\ q_{N1} & q_{N2} & q_{N3} & \cdots\cdots q_{Nk} \end{bmatrix} \end{array}$$

Multiplying the matrix Q by the matrix C produces a square matrix (E) in which each element e_{ij} represents the average number of V_js expected each time V_i is re-written. In other words, e_{ij} is the expected number of V_js in the consequences of the productions in which V_i is the premise of

those productions. The matrix (E) is called the expectation matrix, or the first moment matrix. For example consider the following productions,

	Probability	Productions
R_1	0.5	A:=a
R_2	0.5	A:=AB
R_3	0.8	B:=cb
R_4	0.2	B:=bA

Then the matrices C and Q would be

$$C = \begin{array}{c} \\ R_1 \\ R_2 \\ R_3 \\ R_4 \end{array} \begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ A \\ B \end{array} \begin{array}{c} \begin{array}{cccc} R_1 & R_2 & R_3 & R_4 \end{array} \\ \begin{bmatrix} 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.8 & 0.2 \end{bmatrix} \end{array}$$

The expectation matrix E is

$$E = Q*C$$

$$= \begin{bmatrix} 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.8 & 0.2 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{array}{c} \\ A \\ B \end{array} \begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{bmatrix} 0.5 & 0.5 \\ 0.2 & 0 \end{bmatrix} \end{array}$$

So this means that:

An A generates an A $\frac{1}{2}$ the time

a B $\frac{1}{2}$ the time

B generates an A $\frac{1}{5}$ of the time

and a B never.

The expectation matrix provides useful means to prove the consistency of a probabilistic grammar, and also to find the average word length of a language (Section 7.3), and the average derivation length (Section 7.4).

For a square matrix (E) the modulus of the largest eigenvalue of (E) is called the spectral radius $\rho(E)$. If (E) is the expectation matrix computed from a probabilistic grammar, and $\rho(E)<1$, then the probabilistic grammar is consistent (Wetherell, 80) or strongly consistent (Booth and Thompson, 73). For instance, the above (E) matrix has eigenvalues 0.6532, and -0.1532. Both of them are less than one. So the grammar is consistent. Another way which can be used to show that $\rho(E)<1$ is as follows:

1. Set x=E
2. For each row of x, sum the absolute values of the elements of the row. If all the row sums are less than one, halt and the answer is $\rho(E)<1$
3. Otherwise, set x=x*x and go back to step 2.

For example, let

$$x = \begin{bmatrix} 0.5 & 0.5 \\ 0.2 & 0 \end{bmatrix}$$

The first row sum is 1 and the second row sum is 0.2. Since the first row is not less than 1, then perform the third step, i.e.

$$x^2 = \begin{bmatrix} 0.35 & 0.25 \\ 0.10 & 0.10 \end{bmatrix}$$

From step 2, the row sums of x^2 are 0.60 and 0.20. Both are less than 1. So $\rho(E)<1$. Hence the grammar is consistent.

Each element of the expectation matrix (E) (i.e. e_{ij}) represents the average number of V_js expected when V_i is rewritten exactly once. In other words, (E) is a matrix of averages for one-step derivations; (E^2) is the expectation matrix for two-step derivations;... and so on. In a zero-step derivation, a non-terminal derives exactly and only itself. Thus (E^0) is equal to the identity matrix (I). Now, for derivations of all lengths, the expectation matrix would be

$$E^\infty = \sum_{i=0}^{\infty} E^1 .$$

so, e^∞_{ij} is the average number of V_js to expect after an arbitrary derivation beginning with V_i. To simplify E^∞,

$$\begin{aligned} E^\infty &= \sum_{i=0}^{\infty} E^i \\ &= I + E^1 + E^2 + \ldots \\ &= \frac{I}{I-E} \\ &= (I-E)^{-1} \end{aligned}$$

The sum converges whenever E is small enough. Fortunately E is small enough exactly when the spectral radius $\rho(E)$ is less than one. For a consistant grammar, $\rho(E)$ is less than one, and hence

$$E^\infty = (I-E)^{-1}$$

E^∞ is called the non-terminal expectation matrix.

7.3 AVERAGE WORD LENGTH (AWL)

For a probabilistic language L, the average word length (AWL) is the average number of terminal symbols in a string obtained from the language[†]. Suppose that s is a string in L, and $\ell(s)$ is the length of the string, then the (AWL) of the language is defined (Booth and Thompson, 73) as:

$$AWL = \sum_{s \in L} \ell(s)p(s)$$

where p(s) is the probability of the string s.

Finding the (AWL) from the language is difficult because of the problem of finding all possible strings of the language. However, from the relationship between the probabilistic languages and the probabilisitic grammars generating them, it is possible to find the AWL from the grammar rules (productions). Let the productions of a probabilistic context-free grammar be specified as follows:

$$\begin{aligned}
&P_{11}: \quad A_1 := \beta_{11} \\
&P_{12}: \quad A_1 := \beta_{12} \\
&\quad \vdots \\
&P_{1k_1}: \quad A_1 := \beta_{1k} \\
&P_{21}: \quad A_2 := \beta_{21} \\
&\quad \vdots \\
&P_{n1}: \quad A_n := \beta_{n1} \\
&\quad \vdots \\
&P_{nk_n}: \quad A_n := \beta_{nk}
\end{aligned}$$

where, in general, a non-terminal A_i is re-written by the string β_{ij} with

---

[†]*Note that the average word length is not the average length of individual words but the average length of a string measured in words i.e. average string length. AWL is used here for compatibility with other research.*

probability p_{ij}. The string β_{ij} is a combination of terminal and non-terminal symbols. So for each production, the average number of terminal symbols can be expressed as the number of terminal symbols plus the average number of terminal symbols obtained from the non-terminal symbols. Since there are usually several productions which have the same premises, then the average number of terminal symbols generated by the grammar with A_i as the initial symbol is equal to the average number of symbols of all productions which rewrite A_i. For example, consider the following probabilistic context free grammar G=(T,N,R,P,S) where

T={a,b,c}; N={A,B}; S={A}; P and R =

R_1 0.5: A:=a

R_2 0.5: A:=AB

R_3 0.8: B:=cb

R_4 0.2: B:=bA

Then the average length of all symbols generated by G with A as the initial symbol is

$$n_A = 0.5 * 1 + 0.5\ (n_A+n_B)$$

and for B

$$n_B = 0.8 * 2 + 0.2 * (1 + n_A)$$

To express formally the value of n, let:

$m(\beta_{ij})$ = number of terminal symbols in β_{ij};

n_i = average length of all words generated by G with A_i as the initial symbol; and

$q_i(\beta_{ij})$ = number of occurrences of A_i in the consequence β_{ij}.

Then
$$n_i = \sum_{j=1}^{k_i} P_{ij}\{m(\beta_{ij}) + \sum_{x=1}^{n} n_x q_x(\beta_{ij})\}$$

The AWL of the language generated by G, can be obtained by the following theorem (Booth and Thompson, 73).

Theorem: Let L(G) be generated by a strongly consistent context-free probabilistic grammar G. Then the AWL of L(G) is

$$AWL = [100 \ldots 0](I-E)^{-1}\underline{T}$$

where E = expectation matrix (first moment matrix)

$\underline{T} = [t_i]$ column vector; such that

$$t_i = \sum_{j=1}^{k_i} P_{ij}\, m(\beta_{ij})$$

= average number of terminal symbols generated when A_i is rewritten;

I = identity matrix.

Proof: For any i

$$n_i = \sum_{j=1}^{k_i} P_{ij}\{m(\beta_{ij}) + \sum_{x=1}^{n} n_x q_x(\beta_{ij})\}$$

$$= \sum_{j=1}^{k_i} P_{ij}\, m(\beta_{ij}) + \sum_{j=1}^{k_i} P_{ij} \sum_{x=1}^{n} n_x q_x(\beta_{ij})$$

$$= \sum_{j=1}^{k_i} P_{ij}\, m(\beta_{ij}) + \sum_{x=1}^{n}\left[\sum_{j=1}^{k_i} P_{ij} q_x(\beta_{ij})\right] n_x$$

Then for the entire grammar

$$\begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{k_1} p_{1j}q_1(\beta_{1j}) & \sum_{j=1}^{k_1} p_{1j}q_2(\beta_{1j}) & \cdots \\ \sum_{j=1}^{k_2} p_{2j}q_1(\beta_{2j}) & \sum_{j=1}^{k_2} p_{2j}q_2(\beta_{2j}) & \cdots \\ \vdots & & \\ \sum_{j=1}^{k_n} p_{nj}q_1(\beta_{nj}) & \sum_{j=1}^{k_n} p_{nj}q_2(\beta_{nj}) & \cdots \end{bmatrix} \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_n \end{bmatrix} + \begin{bmatrix} \sum_{j=1}^{k_1} p_{1j}m(\beta_{1j}) \\ \sum_{j=1}^{k_2} p_{2j}m(\beta_{2j}) \\ \vdots \\ \sum_{j=1}^{k_n} p_{nj}m(\beta_{nj}) \end{bmatrix}$$

The square matrix is the expectation matrix E. Let $\underline{N}$ denote the column vector $[n_i]$, then

$$\underline{N} = E\underline{N} + \underline{T}$$

solving for N gives

$$\underline{N} = (I-E)^{-1}\underline{T}$$

where $(I-E)^{-1}$ exists if E has no eigenvalues with unit magnitude. This condition is guaranteed if G is a strongly consistent context-free grammar. The average word length for the overall grammar = n_1 because A_1 is the start symbol, so

$$AWL = [100 \ldots 0]\underline{N}$$

$$= [100 \ldots 0](I-E)^{-1}\underline{T}$$

End of proof.

From the above example $E = \begin{bmatrix} 0.5 & 0.5 \\ 0.2 & 0.0 \end{bmatrix}$

the eigenvalues are 0.65, and -0.15. Thus the grammar is strongly consistent.

$$\underline{T} = \begin{bmatrix} 0.5*1 + 0.5*0 \\ 0.8*2 + 0.2*1 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1.8 \end{bmatrix}$$

$$(I-E)^{-1} = \begin{bmatrix} 0.5 & -0.5 \\ -0.2 & 1.0 \end{bmatrix}^{-1}$$

$$= \begin{bmatrix} 2.5 & 1.25 \\ 0.5 & 1.25 \end{bmatrix}$$

Therefore

$$AWL = [10] \begin{bmatrix} 2.5 & 1.25 \\ 0.5 & 1.25 \end{bmatrix} \begin{bmatrix} 0.5 \\ 1.8 \end{bmatrix}$$

= 3.50 terminal symbols per string

Wetherell (Wetherell, 80) explains another way of finding the AWL

which leads to almost the same equation as mentioned above. The method is to find the expected number of each terminal symbol after one rewriting of each non-terminal symbol. The resulting matrix S will eventually be multiplied by the non-terminal expectation matrix E^{∞} (see Section 7.2) to yield matrix W. If A_1 is the initial symbol then the sum of the elements of the A_1th row of W is the AWL, i.e.

$$W = E^{\infty}.S$$

AWL = sum of the elements of the first row of W

First row of W = $[100 \ldots 0]E^{\infty}S$

$= [100 \ldots 0](I-E)^{-1}S$

To find the elements of the matrix S,

$$S = Q.D$$

where Q is a matrix which has $|N|$ rows indexed by non-terminal symbols and $|R|$ columns indexed by productions. The element Q_{ij} has value P_{ij} if production R_j has non-terminal A_i on its left and value zero otherwise. The matrix D has $|R|$ rows indexed by productions and $|T|$ columns indexed by terminal symbols. The element D_{ij} has as value the number of times terminal T_j occurs on the right-hand side of the production R_i. From the example above:

$$Q = \begin{bmatrix} 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.8 & 0.2 \end{bmatrix}$$

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$S = Q.D$$

$$S = \begin{bmatrix} 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 0.8 & 0.2 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 1.0 & 0.8 \end{bmatrix}$$

Therefore

$$\text{First row of } W = [10] \begin{bmatrix} 2.5 & 1.25 \\ 0.5 & 1.25 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 1.0 & 0.8 \end{bmatrix}$$

$$= [10] \begin{bmatrix} 1.25 & 1.25 & 1.0 \\ 0.25 & 1.25 & 1.0 \end{bmatrix}$$

$$= [1.25 \quad 1.25 \quad 1.0]$$

$$AWL = 1.25 + 1.25 + 1.0$$

$$= 3.50 \text{ terminal symbols per string}$$

7.4 AVERAGE DERIVATION LENGTH (ADL)

It has been mentioned in Section (3.2) that a string of symbols from a language L can be derived from the set of productions of a context-free grammar which generates L. In each derivation, a particular production is used, which is considered as one step in completing the derivation of the string. So the derivation length of a string by starting from a non-terminal symbol represents the number of steps required to complete the derivation. Suppose that S is the initial state, then the average derivation length ADL(S) of a derivation beginning with the non-terminal S is the expected number of steps in a derivation beginning in S and ending with a terminal string (Wetherell, 80). Since the length of a derivation is exactly equal to the number of non-terminals introduced (each non-terminal requires one application of a production rule to be replaced), then from the non-terminal expectation matrix E^{∞}, each element represents the average number of occurrence of the particular non-terminal symbol produced during the derivation of a string by the first non-terminal symbol. Assume that A_1 is the initial symbol, then the sum of the elements of the A_1th row of E^{∞} is the ADL (A_1). From the example in Section (7.3)

$$E^{\infty} = (I-E)^{-1} = \begin{bmatrix} 2.5 & 1.25 \\ 0.5 & 1.25 \end{bmatrix}$$

$$\begin{aligned} ADL(A) &= 2.5 + 1.25 \\ &= 3.75 \end{aligned}$$

This means that the average derivation will have under four production applications before a terminal string is reached.

To find formally the ADL, let $G=(\{S=v_1,v_2,\ldots,v_k\},T,R,P,S)$ be a

probabilistic context-free grammar. The following definitions are made (Hutchins, 72b; Feller, 57).

Definition: d_i, i=1,2,...,k are discrete random variables representing the number of steps necessary to complete a left-most derivation of a terminal string from v_i.

Definition: $q_i(n)=p(d_i=n)$, i=1,2,...,k.

Definition: For each i=1,2,...,k, $F_i(s)=\sum_{n=0}^{\infty} q_i(n)s^n$. $F_i(s)$ is the generating function for the sequence $\{q_i(n)\}_{n=0}^{\infty}$, and

$$\sum_{n=0}^{\infty} q_i(n) = F_i(1) = 1 \ , \ i=1,2,\ldots,k.$$

Definition: Let $\{a_i\}$ and $\{b_i\}$ be any two sequences of the same length. The new sequence $\{c_i\}$ is called the convolution of $\{a_i\}$ and $\{b_i\}$ and will be denoted by

$$\{c_i\} = \{a_i\} * \{b_i\}$$

where $c_r = a_0b_r+a_1b_{r-1}+ \ldots +a_{r-1}b_1+a_rb_0$.

Since v_1 is the initial symbol, d_1 is the random variable for left-most derivation length of strings in the language L generated by G, and $q_1(n)$ is the probability that left-most derivation requires n steps for completion. So $\sum_{n=1}^{\infty} q_1(n) = F_1(1)$ is the probability of completing a derivation. The average derivation length is equal to $\sum_{n=0}^{\infty} nq_1(n) = F_1'(1)$. Assume $D_1=ADL(V_1)$.

Consider R_i, the set of productions applicable to V_i, R_i =

$$\begin{array}{lll}
r_{j_1}: V_i & \xrightarrow{P_{j1}} & \alpha_1^{j_1} v_1^{j_1} \alpha_2^{j_1} v_2^{j_1} \ \ldots \ \alpha_{k(j_1)}^{j_1} v_{k(j_1)}^{j_1} \\
r_{j_2}: V_i & \xrightarrow{P_{j2}} & \alpha_1^{j_2} v_1^{j_2} \alpha_2^{j_2} v_2^{j_2} \ \ldots \ \alpha_{k(j_2)}^{j_2} v_{k(j_2)}^{j_2} \\
\vdots & & \\
r_{j_m}: V_i & \xrightarrow{P_{jm}} & \alpha_1^{j_m} v_1^{j_m} \alpha_2^{j_m} v_2^{j_m} \ \ldots \ \alpha_{k(j_m)}^{j_m} v_{k(j_m)}^{j_m}
\end{array}$$

where $\alpha_t^{j_k}$ is an arbitrary string of terminal symbols. If n steps are required to derive a string of terminal symbols from V_i, then (n-1) steps are required to generate the terminal substring from $v_1^{j_k} v_2^{j_k} \ldots v_{k(j_k)}^{j_k}$ by assuming that r_{j_k} production is used with probability p_{j_k}. So the probability that n steps are required for completing a string of terminal symbols from V_i is

$$q_i(n) = p_{j_1} \left[\sum_{n_1+n_2+\ldots+n_k(j_1)=n-1} q_1^{j_1}(n_1) q_2^{j_1}(n_2) \ldots q_{k(j_1)}^{j_1}(n_k(j_1))\right]$$

$$+p_{j_2} \left[\sum_{n_1+n_2+\ldots+n_k(j_2)=n-1} q_1^{j_2}(n_1) q_2^{j_2}(n_2) \ldots q_{k(j_2)}^{j_2}(n_k(j_2))\right]$$

$$\vdots$$

$$+p_{j_m} \left[\sum_{n_1+n_2+\ldots+n_k(j_2)=n-1} q_1^{j_m}(n_1) q_2^{j_m}(n_2) \ldots q_{k(j_m)}^{j_m}(n_k(j_m))\right]$$

From the definition above

$$q_i(n) = p_{j_1}(q_1^{j_1} * q_2^{j_1} * \ldots * q_{k(j_1)}^{j_1}(n-1))$$

$$+ p_{j_2}(q_1^{j_2} * q_2^{j_2} * \ldots * q_{k(j_2)}^{j_1}(n-1))$$

$$\vdots$$

$$+ p_{j_m}(q_1^{j_m} * q_2^{j_m} * \ldots * q_{k(j_m)}^{j_m}(n-1))$$

From (Feller, 57, and Hutchins, 72b) the generation function for the convolution of two sets is the product of their generating functions. If F(s) is the generating function of $\{q(n)\}_{n=0}^{\infty}$, then the generating function for $\{q(n+1)\}_{n=0}^{\infty}$ is sF(s).

$$F_i(s) = p_{j_1} s(F_1^{j_1}(s) F_2^{j_1}(s) \ldots F_{k(j_1)}^{j_1}(s))$$

$$+ p_{j_2} s(F_1^{j_2}(s) F_2^{j_2}(s) \ldots F_{k(j_2)}^{j_2}(s))$$

$$\vdots$$

$$+ p_{j_m} s(F_1^{j_m}(s)F_2^{j_m}(s) \ldots F_{k(j_m)}^{j_m}(s))$$

$$= s \sum_{r_j \in R_i} p_j \prod_{t=1}^{k(j)} F_t^j(s)$$

Since i=1,2,...,k, then there are k equations relating to the k unknown generating functions. The equations are non-linear in the $F_i(s)$. By differentiating the above equation

$$F_i'(s) = \frac{F_i(s)}{s} + s \sum_{r_j \in R_i} p_j \left(\sum_{t=1}^{k(j)} F_t'^j(s) \prod_{k \neq t} F_k^j(s) \right)$$

By letting $D_i = F_i'(1)$, $F_i(1)=1$, i=1,2,...,k, then,

$$D_i = 1 + \sum_{r_j \in R_i} p_j \left(\sum_{t=1}^{k(j)} F_t'^j(1) \right)$$

$$= 1 + \sum_{r_j \in R_i} p_j (n(j,1)F_1'^j(1) + \ldots + n(j,k)F_k'^j(1))$$

$$= 1 + a_{i1} F_1'(1) + a_{i2}F_2'(1) + \ldots + a_{ik}F_k'(1)$$

$$= 1 + a_{i1}D_1 + a_{i2}D_2 + \ldots + a_{ik}D_k$$

$$= 1 + \sum_{\ell=1}^{k} a_{i\ell} D_\ell$$

Let $\underline{D} = [D_i]$ a column vector, i=1,2,...,k, and

$\underline{1} = [1]$ a column vector of size k

Then $\underline{D} = \underline{1} + E\underline{D}$

or $(I-E)\underline{D} = \underline{1}$

or $\underline{D} = (I-E)^{-1}\underline{1}$

Therefore $ADL(V_1) = D_1$

= sum of the elements of the first row of the expectation matrix.

For instance, from the expectation matrix mentioned above

$$ADL(A) = \begin{bmatrix} 2.5 & 1.25 \\ 0.5 & 1.25 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$= 2.5 * 1 + 1.25 * 1$$

$$= 3.75$$

7.5 AVERAGE NUMBER OF STATES (ANS)

Any string of symbols from a language generated by a context-free grammar has to be recognized (parsed) before storing its code. The recognition can be represented by a unique sequence of states. Each state is selected, either as a consequence of a shift action or a reduce action, from the previous state. The number of shift actions is equal to the number of terminal symbols in the string (including the end marker). In the right-most derivations, a reduce action is equivalent to a derivation step in the left-most derivations; because in the reduction step, the consequence of a particular production will be replaced by its premise; whereas in the derivation step, the premise of a particular production will be substituted by its consequence in the sentential form (see Section 3.2). So the number of reduce actions is equal to the number of derivations of the string. Therefore, the set of states which represents a string of symbols can be divided into two subsets (see the example in Section 7.9):

1. The states which cause shift actions; and
2. The states which cause reduce actions.

In general, assume that

$$NS = s_1, s_2, \ldots, s_n$$

be a set of states represents a string of symbols where

$$I = s_1, s_2, \ldots, s_i \text{ cause shift actions; and}$$

$$J = s_1, s_2, \ldots, s_j \text{ cause reduce actions;}$$

such that $I + J = n.$

Then $NS = I + J$

The average of (I) is the average word length (AWL) which was explained in Section (7.3); and the average of (J) would be the average derivation

length (ADL) which was explained in Section (7.4). So

the average of (NS) = AWL + ADL

Assume that [A] = the expectation matrix;

$[t_i]$ = the column vector as defined in Section (7.3)

Then

$$ANS = [1000 \ldots 0][A][t_i] + [1000 \ldots 0][A]\begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$$

$$= [100 \ldots 0][A]\left([t_i] + \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}\right)$$

From the definition of the addition of two matrices mentioned in Section (7.1).

$$ANS = [100 \ldots 0][A][t_i+1]$$

or

let e_i be the expected occurrence of the non-terminal symbol v_i

$p(v_i)$ be the probability of v_i

#T be the number of terminal symbols in a production $j \in R_i$

Then

$$ANS = AWL + ADL$$

$$= [e_1 \ e_2 \ e_3 \ \ldots \ e_k][t_i] + ADL$$

Since $ADL = e_1+e_2+\ldots+e_k$

$$p(v_i) = \frac{e_i}{ADL}$$

then $ANS = ADL[p(v_1) \ p(v_2) \ \ldots \ p(v_k)][t_i] + ADL$

$$= ADL \left(\sum_{i=1}^{k} p(v_i) \sum_{j \in R_i} p_j \ \# \ T \right) + ADL$$

$$= ADL \left[\left(\sum_{i=1}^{k} p(v_i) \sum_{j \in R_i} p_j \ \# \ T \right) + 1 \right]$$

An example of obtaining the average number of states is mentioned in Section (7.9).

7.6 THE PROBABILITY DISTRIBUTION OF THE STATES

It has been mentioned that for a context-free grammar, a finite set of states could be generated (using YACC program) to encode (or decode) any string of symbols from the language generated by the grammar. The transition between the states is not independent; each state has certain connections in which it can be entered, and it can exit to other particular states. So the probability of being in a state depends on the previous state (conditional probability), and is noted as p(a|b) which means the probability of seeing the state (a) given that (|) the state (b) has just been seen.

For a set of n states, a convenient way of describing the relation between the states is to arrange the probabilities in matrix form (transition matrix) where each row represents the current state, and each column represents the next state. The matrix entry is the conditional probability (Fig. 7.1). The sum of the elements in each row is equal to

	a	b	c	...
a	p(a|a)	p(b|a)	p(c|a)	...
b	p(a|b)	p(b|b)	p(c|b)	...
c				
⋮				

FIGURE 7.1: Transition matrix

one. For example, suppose that a,b and c are three states with the following probabilities

$$p(a|a) = \frac{1}{3} \quad , \quad p(b|a) = \frac{1}{3} \quad , \quad p(c|a) = \frac{1}{3}$$

$$p(a|b) = \frac{1}{4} \quad , \quad p(b|b) = \frac{1}{2} \quad , \quad p(c|b) = \frac{1}{4}$$
$$p(a|c) = \frac{1}{4} \quad , \quad p(b|c) = \frac{1}{4} \quad , \quad p(c|c) = \frac{1}{2}$$

Then the transition matrix would be:

$$\begin{array}{c} \\ a \\ b \\ c \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \end{bmatrix} \end{array}$$

Assume that $p_1, p_2, \ldots, p_n$ be the probability distributions of the states $s_1, s_2, \ldots, s_n$ respectively. The relations between the probabilities can be easily derived. For example, there are maximum n states in which a new state can be seen. If state 1 is the current state, then state 1 will be the next state with probability p(1|1); if state 2 is the current state, state 1 will be the next state with probability p(1|2),..., and so on. In general, the probability of each state would be:

$$p(1)p(1|1) + p(2)p(1|2) + \ldots + p(n)p(1|n) = p(1)$$
$$p(1)p(2|1) + p(2)p(2|2) + \ldots + p(n)p(2|n) = p(2)$$
$$\vdots$$
$$p(1)p(n|1) + p(2)p(n|2) + \ldots + p(n)p(n|n) = p(n)$$

The above set of equations is arranged as follows (Hamming, 80):

$$(p(1)\ p(2)\ \ldots\ p(n)) \begin{bmatrix} \text{transition} \\ \text{matrix} \end{bmatrix} = (p(1)\ p(2)\ \ldots\ p(n)) \qquad (1)$$

where p(1)+p(2)+...+p(n) = 1. This indicates that the probabilities of the states remain unchanged through shifts in time. From the above example:

$$(p(a)\ p(b)\ p(c)) \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \end{bmatrix} = (p(a)\ p(b)\ p(c))$$

This is equivalent to the equations

$$\frac{1}{3}\,p(a) + \frac{1}{4}\,p(b) + \frac{1}{4}\,p(c) = p(a)$$

$$\frac{1}{3}\,p(a) + \frac{1}{2}\,p(b) + \frac{1}{4}\,p(c) = p(b)$$

$$\frac{1}{3}\,p(a) + \frac{1}{4}\,p(b) + \frac{1}{2}\,p(c) = p(c)$$

It is known that

$$p(a) + p(b) + p(c) = 1$$

By solving the above equations

$$p(a) = \frac{3}{11}, \quad p(b) = p(c) = \frac{4}{11}$$

Equation (1) can be used for limited transition matrix. Nevertheless, it can be reorganized in a way suitable for obtaining the probability distributions for large numbers of states with the help of computer routines which are already available for users. Assume that (A) is the transition matrix and (A^T) is the transpose of (A). Equation (1) can be written as

$$\underline{P}^T A = \underline{P}^T$$

where $(\underline{P}^T)$ is the row vector $(p_1\ p_2 \ldots p_n)$, then

$$A^T \underline{P} = \underline{P} \tag{2}$$

It can be shown that equation (2) has a non-trivial solution $\underline{P} \neq 0$ and one of the eigenvalues of A^T is equal to unity. In this case $\underline{P}$ will be equal to the normalized eigenvector corresponding to the eigenvalue of 1. From the properties of eigenvalues (Jennings, 77), a matrix has the same

eigenvalues as its transpose. So to prove that one of the eigenvalues is one, it is necessary to show that

$$\det(A-I) = 0 \tag{3}$$

where I is the identity matrix. Suppose that a_{ij} is an element of the i^{th} row and j^{th} column in (A). Then

$$\det(A-I)=\det\left(\begin{bmatrix} a_{11}-1 & a_{12} & a_{13} \cdots\cdots & a_{1n} \\ a_{21} & a_{22}-1 & a_{23} \cdots\cdots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & a_{n3} \cdots\cdots & a_{nn}-1 \end{bmatrix}\right)$$

Any row or column can be added to any other row or column of the matrix without affecting the value of the determinant (Campbell, 65). So add all the columns j=2,3,...,n to the first column, then

$$\det(A-I)=\det\left(\begin{bmatrix} \sum_{j=1}^{n} a_{1j}-1 & a_{12} \cdots\cdots & a_{1n} \\ \sum_{j=1}^{n} a_{2j}-1 & a_{22}-1 \cdots\cdots & a_{2n} \\ \vdots & & \\ \sum_{j=1}^{n} a_{nj}-1 & a_{n2} \cdots\cdots & a_{nn}-1 \end{bmatrix}\right)$$

Since the summation of the elements of any row is equal to 1, i.e.

$$\sum_{i=1}^{n} a_{ij} = 1$$

Then

$$\det(A-I)=\det\left(\begin{bmatrix} 0 & a_{12} \cdots\cdots & a_{1n} \\ 0 & a_{22}-1 \cdots\cdots & a_{2n} \\ \vdots & & \\ 0 & a_{n2} \cdots\cdots & a_{nn}-1 \end{bmatrix}\right)$$

Since the first column of the above matrix is all zeros, then (Campbell, 65):

$$\det(A-I) = 0$$

Therefore one of the eigenvalues of the matrix (A^T) is 1. Hence equation (2) has a non-trivial solution $\underline{P} \neq 0$. Notice that equation (2) has multiple solution vectors. That is if $\underline{P}$ is one solution then $\alpha\underline{P}$ is a solution as well, where α is a constant (see Section 7.1). The correct solution can be found by using the fact that

$$\sum_{i=1}^{n} p_i = 1$$

Therefore, once the vector ($\underline{x}$) has been evaluated, then,

$$\underline{P} = \alpha\underline{x}$$

where

$$\alpha = \frac{P}{\underline{x}} = \frac{1}{\sum_{i=1}^{n} x_i}$$

As an example, consider the above transition matrix

$$A^T = \begin{bmatrix} \frac{1}{3} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{2} \end{bmatrix}$$

$$\det(A^T-I) = \det\left(\begin{bmatrix} -\frac{2}{3} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{3} & -\frac{1}{2} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & -\frac{1}{2} \end{bmatrix}\right)$$

$$= -\frac{2}{3}\left(\frac{1}{4} - \frac{1}{16}\right) - \frac{1}{4}\left(-\frac{1}{6} - \frac{1}{12}\right) + \frac{1}{4}\left(\frac{1}{12} + \frac{1}{6}\right)$$

$$= -\frac{2}{3} * \frac{3}{16} + \frac{1}{4} * \frac{3}{12} + \frac{1}{4} * \frac{3}{12}$$

$$= 0$$

For the eigenvalue of 1

$$(A^T - I)\begin{bmatrix} x_a \\ x_b \\ x_c \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} -\frac{2}{3} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{3} & -\frac{1}{2} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} x_a \\ x_b \\ x_c \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$-\frac{2}{3}x_a + \frac{1}{4}x_b + \frac{1}{4}x_c = 0$$

$$\frac{1}{3}x_a - \frac{1}{2}x_b + \frac{1}{4}x_c = 0$$

$$\frac{1}{3}x_a + \frac{1}{4}x_b - \frac{1}{2}x_c = 0$$

By simplifying the above equations

$$x_b = x_c$$

$$x_a = \frac{3}{4}x_b$$

For $x_b = 1$

$$x_c = 1$$

and $$x_a = \frac{3}{4}$$

since $p_a + p_b + p_c = 1$

Then $$\alpha = \frac{1}{\frac{3}{4} + 1 + 1} = \frac{4}{11}$$

Therefore $\underline{p} = \alpha\underline{x}$

$$= \frac{4}{11}\begin{bmatrix} \frac{3}{4} \\ 1 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{3}{11} \\ \frac{4}{11} \\ \frac{4}{11} \end{bmatrix}$$

i.e. $p_a = \frac{3}{11}$, $p_b = p_c = \frac{4}{11}$

7.7 MINIMUM CODE LENGTH (MCL)

Let α be any string from the language (L) with probability $p(\alpha)$, then the minimum code length would be (Hutchins, 72a),

$$H = -\sum_{\alpha \in L} p(\alpha) \log_2 p(\alpha)$$

This can be achieved from the derivation steps and the productions which generate the language. From the definition of the grammar (Section 3.1), let p_j (j=1,2,...,m) be the probability of each production in R_i (i=1,2,...,n) in which v_i is the premise with probability $p(v_i)$. Then

$$h_i = -\sum_{j \in R_i} p_j \log_2(p_j)$$

is the minimum code length of a particular state v_i. Now the minimum code length per non-terminal symbol would be

$$H = \sum_{i=1}^{k} p(v_i) h_i$$

In the derivation process (left-most derivation), each non-terminal symbol is substituted by a particular production which is considered as one step of the process. From Section (7.4); D1 would be the average number of derivation steps required for deriving a string. Hence,

$$\begin{aligned} MCL &= D1\ H \\ &= D1 \sum_{i=1}^{k} p(v_i) h_i \end{aligned}$$

In the right-most derivation, assume that $s_1, s_2, \ldots, s_n$ is a set of states which is used to encode any string from the language (L), and each state has got m choices. Let the probability of each choice be p_j, j=1,2,...,m. Then

$$h_i = -\sum_{j=1}^{m} p_j \log_2(p_j)$$

be the minimum code length for a particular state (s_i). The minimum code length per state would be $\sum_{i=1}^{n} p(s_i)h_i$ where $p(s_i)$ is the probability of the state s_i. Since each string can be represented by a set of states. From Section (7.5), let N be the average number of states required to encode a string, then the minimum code length per string would be

$$MCL = N \sum_{i=1}^{n} p(s_i)h_i$$

An example of finding the MCL of a string is mentioned in Section (7.9).

7.8 AVERAGE CODE LENGTH (ACL)

To determine the average length of the coded string generated by the encoder, assume that:

$c(r_j)$ is the code of each rule r_j;

$\ell(c(r_j))$ is the length of the code.

Then for each non-terminal symbol v_i, i=1,2,...,k

$$\ell_i = \sum_{j \in R_i} p_j \, \ell(c(r_j))$$

would be the average code length of the code for R_i. Since there are k non-terminal symbols, each has a probability $p(v_i)$, i=1,2,...,k, then the average code length for each non-terminal is

$$\sum_{i=1}^{k} p(v_i) \, \ell_i$$

In a left-most derivation, the replacement of a non-terminal symbol by the consequence of a particular production is considered as one step in the derivation process. Hence the average number of non-terminal symbols is equal to the average derivation length of the string. From Section (7.4) the average derivation length D1 can be obtained. Therefore the average code length per string would be (Hutchins, 72a):

$$ACL = D1 \sum_{i=1}^{k} p(v_i) \, \ell_i$$

For the right-most derivation, assume that $s_1, s_2, \ldots, s_n$ be a set of states in which a string can be encoded. Let

$p(s_i)$ be the probability of s_i;

c_j be the code of each choice in the state j=1,2,...,m;

$\ell(c_j)$ be the length of the code;

p_j be the probability of each choice in the state

Then the average code length for a particular state is

$$\ell_i = \sum_{j=1}^{m} p_j \ell(c_j)$$

Now, for the set of state, the average code length per state is

$$\sum_{i=1}^{n} p(s_i)\ell_i$$

From Section (7.5), let N be the average number of states required to encode a string. Then the average code length per string would be:

$$ACL = N \sum_{i=1}^{n} p(s_i)\ell_i$$

An example of the average code length per string is mentioned in Section (7.9).

7.9 COMPARISON

The main objective of designing a compression method is to encode a sequence of source letters into a form such that the encoded data can occupy as little storage space as possible. Obviously, all compression methods have the ability to compress data but it has to be decided which method is the optimal. Therefore, a comparison between different compressing methods is necessary to decide which method does use less storage for the encoded file.

Two practical comparisons exist. The first is to find the number of binary digits that are required to encode a given source letter by these methods. The second is to find the ratio of the size of the encoded data to the size of the data in its original form (Section 3.9).

The encoding method (explained in Chapter 5) has been implemented on Pascal language, and some Pascal programs have been used as sample data (Section 5.11). However, the language itself has not been used for comparing the encoding method with an already existing encoding method because of the difficulties of obtaining the probability of each state (there are more than 300 states) which require the construction of the transition matrix, and also the construction of the expectation matrix. So, it is not possible to calculate the average length of a program (AWL), and also the average length of the encoded file (ACL). Instead, a simple language is used for comparing the encoding method with another method using a parsing encoding technique. The set of productions (rules) of that language is listed in Fig. (7.2) together with the frequency, probability, and the code of each production. The frequencies are obtained from six simple programs (Fig. 7.3). The full listing of these programs is in

Appendix E. Note that the programs are syntactically correct but have no semantic meanings.

		Freq.	Prob.	Length	Code
1.	prog:=series				
2.	series:=series;stmnt	33	0.846	1	0
3.	:=stmnt	6	0.154	1	1
4.	stmnt:=var=exp				
5.	exp:=exp+factor	24	0.157	3	001
6.	:=exp-factor	21	0.137	4	0000
7.	:=exp*factor	36	0.235	2	01
8.	:=exp/factor	12	0.079	4	0001
9.	:=factor	60	0.392	1	1
10.	factor:=var	96	0.628	1	0
11.	:=const	36	0.235	2	10
12.	:=(exp)	21	0.137	2	11

var:=a
:=b
:=c
:
:=z

const:=0
:=1
:=2
:
:=9

FIGURE 7.2: Grammar rules

	prog. size
program 1	176
" 2	62
" 3	112
" 4	53
" 5	71
" 6	166

FIGURE 7.3: Sample programs

The proposed compression method tries to encode a source file by building up a syntax tree of that file starting from the leaves upwards to the root. Hence, a bottom-up parsing method is required to do the encoding. So, an LR(K) parsing technique is used. But, the compression method which already exists, tries to encode a source file by building up a syntax tree starting from the root downwards to the leaves. So, a top-down parsing method is required to do the encoding. An example of such a method is a Recursive-Descent (R-D) technique.

The above grammar rules are not suitable for R-D parsing technique, because of the problem of left recursion. Hence, a slight modification is required to some productions which does not affect the overall outcome. The modified rules are:

series:= stmnt {;stmnt}

exp:= factor{(+|-|*|/) factor}

An encoded file size depends on the number of bits generated for the letters, digits, editing characters (spaces and new lines), and for the parsing. If its assumed that the ways used for encoding those letters, digits, and editing characters are identical, then the comparison will

depend only on the number of bits generated by the encoder for the parsing. That is, the less number of bits generated by one method, the better that method is.

By implementing both methods on a number of sample programs, the following two tables (7.1 and 7.2) have been produced.

Prog.size in bytes	Bits for chars.	Bits for edit.chars.	Total	Bits for parsing	Size in bytes
176	170	408	578	160	96
62	61	138	199	58	40
112	121	248	369	97	64
53	60	108	168	53	32
71	80	156	236	68	40
166	165	374	539	149	88
640	657	1432	2089	585	360

TABLE 7.1: Encoding programs using R-D parsing

Prog.size in bytes	Bits for chars.	Bits for edit.chars.	Total	Bits for parsing	Size in bytes
176	170	408	578	149	96
62	61	138	199	55	32
112	121	248	369	96	64
53	60	108	168	53	32
71	80	156	236	70	40
166	165	374	539	140	88
640	657	1432	2089	563	352

TABLE 7.2: Encoding programs using LR parsing

It can be seen that the number of bits generated for characters and editing characters by both methods for each sample program is identical.

Whereas, the average number of bits generated for the parsing by the proposed method (Table 7.2) is less than that of the already existing method (Table 7.1). This means that the proposed method can build a syntax tree with less number of bits required to be encoded. Hence, the average encoded file size generated by the proposed method is smaller than that generated by the existing method.†

In general, one or both of the following two reasons cause a reduction in the number of bits generated by the proposed method:

1. In the existing method, the frequency of each grammar rule is fixed (i.e. the code is fixed), that is, whenever a rule is recognized during a parsing process, the same code is generated. For instance, whenever the rule

   exp:=exp*factor

   is recognized, always 4 bits are generated. But in the proposed method, the frequency of each input symbol is fixed within each state (Appendix D), not for all states. That is, the same input might have different frequencies in different states. Therefore, the code length is varying from one state to another. For instance, in state 8 the symbol * requires 2 bits, whereas in state 17 requires 3 bits. So, the code of the expression a*b is different from the code of the same expression inside paranthesis.

2. It is considered that the average code length (i.e. average number of bits) generated for any tree could be minimized by

---

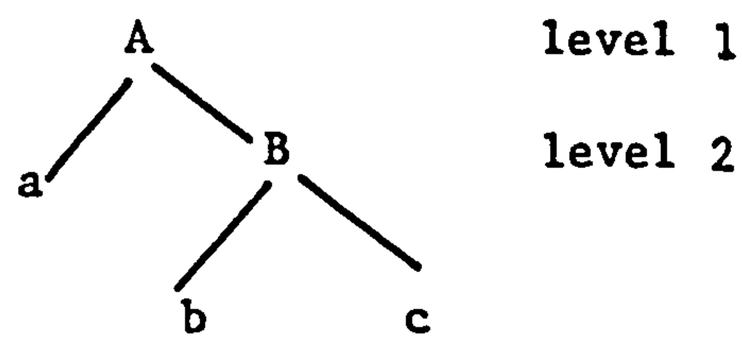
†*In the existing method, no code was generated for productions which were certain to occur.*

reducing the number of levels that the tree has.

In the existing method, the parsing starts at the root and goes down level by level until the process is completed. If at certain levels more than one option exists in a node, then a code should be generated by the encoder to indicate which grammar rule is applied. For instance, consider the following simple grammar rules with their frequencies and code lengths:

	Freq.	No. of bits
A:=B	20	1 bit
:=a	7	1 bit
B:=b	17	1 bit
:=c	3	1 bit

the parsing tree would be:



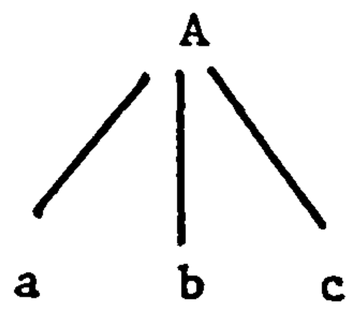
which has 3 levels.

When the parser is at level 1, it only derives either a or B, and from B, it derives either b or c. So, in order to parse c, the parser should pass through B, i.e. $A \Rightarrow B \Rightarrow c$.

The symbol (a) requires 1 bit to be recognized. But (c) and (b) need an extra bit (whatever their frequencies are). This extra bit is generated when the parser has to choose B out of two choices.

However, at any state in the proposed method, the parser can recognize all expected input symbols (i.e. it allows a widespread of options). This sometimes allows the parser to reduce the input according to a specific rule (i.e. goes up one level higher) without a need to generate

any extra code. In other words, the parser can by-pass this level to a higher one. For instance, state 0 (Fig. 7.4) which examines the first input derived from A is expecting a,b or c.



Any input could be reduced to A without generating any extra code.

```
state 0:   prog:=.A$
             a  s3
             b  s4
             c  s5
           others error

state 1:   prog:=A.$
             $   accept
           others error

state 2:   A:=B.
           r1

state 3:   A:=a.
           r2

state 4:   B:=b.
           r3

state 5:   B:=c.
           r4
```

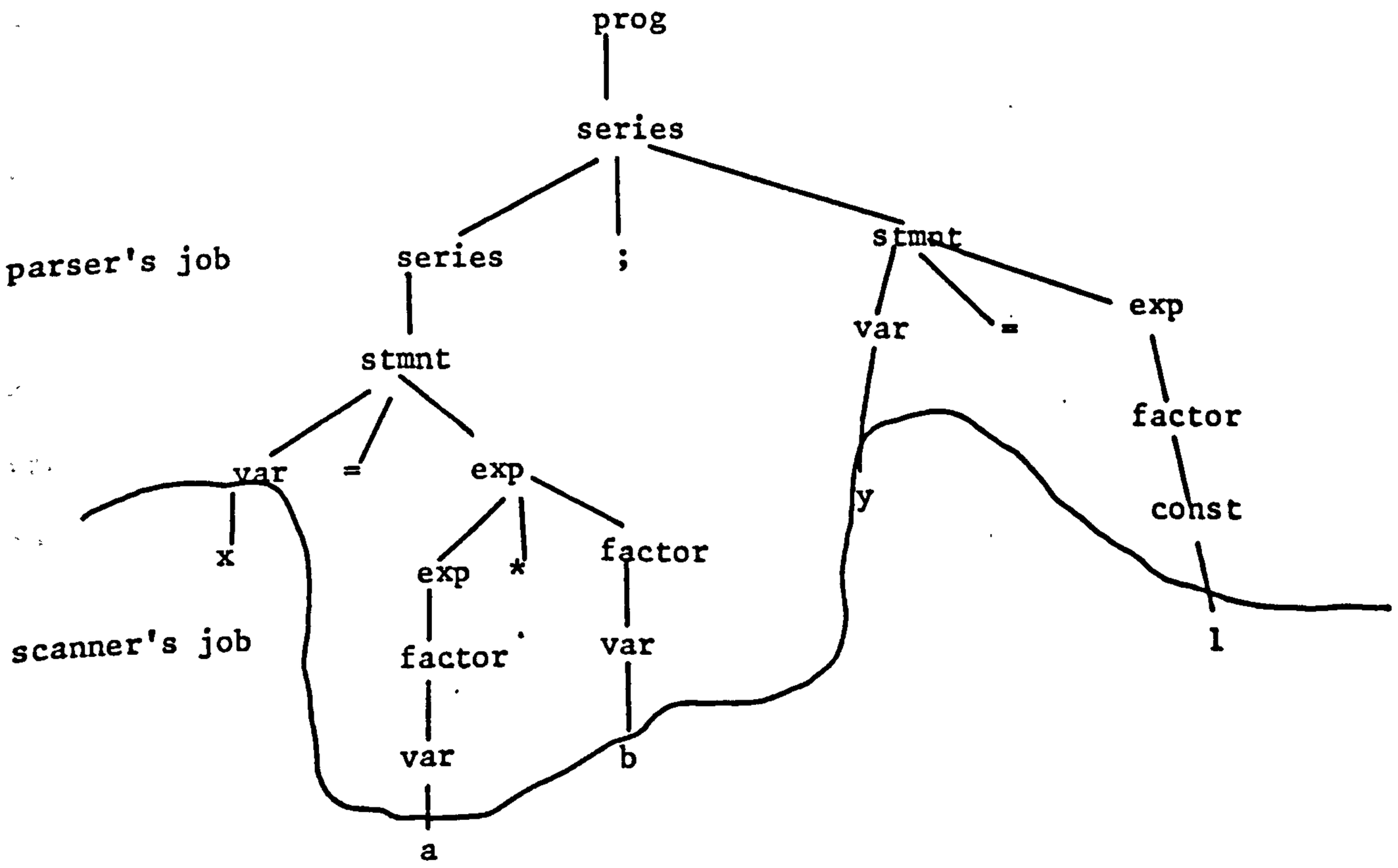
FIGURE 7.4: A set of states for the above example

To compare the number of derivations in R-D parsing method with the number of states used by the LR parser for the same string of symbols, consider the R-D parsing for the following string x=a*b; y=1. From the

productions mentioned eariler in Fig.(7.2), the set of derivations is

prog ⇒ series
⇒ series; stmnt
⇒ stmnt; stmnt
⇒ var=exp; stmnt
⇒ x=exp; stmnt
⇒ x=exp*factor; stmnt
⇒ x=factor*factor; stmnt
⇒ x=var*factor; stmnt
⇒ x=a*factor; stmnt
⇒ x=a*var; stmnt
⇒ x=a*b; stmnt
⇒ x=a*b; var=exp
⇒ x=a*b; y=exp
⇒ x=a*b; y=factor
⇒ x=a*b; y=const
⇒ x=a*b; y=1

The derivation length is 16. Obviously, 5 derivations are part of the scanner's job (see Fig. 7.5). Therefore, the parser needs only 11 derivation steps in order to complete the recognition of the above string.



To parse the same string using LR parsing method (the set of states generated by the YACC program is listed in Appendix D). Assume that ($s_{\_}$) means shift the symbol =; and (R_5) means reduce by the production number 5. Then the sequence of states would be:

$$s_0 \xrightarrow{s_x} s_4 \xrightarrow{s_=} s_6 \xrightarrow{s_a} s_{10} \xrightarrow{R_{10}} s_9 \xrightarrow{R_9} s_8 \xrightarrow{s_*} s_{15}$$

$$\xrightarrow{s_b} s_{10} \xrightarrow{R_{10}} s_{20} \xrightarrow{R_7} s_8 \xrightarrow{R_4} s_3 \xrightarrow{R_3} s_2 \xrightarrow{s_;} s_5$$

$$\xrightarrow{s_y} s_4 \xrightarrow{s_=} s_6 \xrightarrow{s_1} s_{11} \xrightarrow{R_{11}} s_9 \xrightarrow{R_9} s_8 \xrightarrow{R_4} s_3$$

$$\xrightarrow{R_2} s_2 \xrightarrow{R_1} s_1 \xrightarrow{s_{\$end}} accept.$$

The total number of states is 21, 11 states occur due to reduce actions, and 10 states occur due to shift actions. Assume that each derivation in R-D method represents one state in the parsing process, then the total number of derivation steps in R-D is the same as the number of reduce actions in LR.

7.9.1 Using Left-Most Derivations

To find the average word length of a string generated from the grammar mentioned in Fig. (7.2); the average code length; and the minimum code length of the encoded file. By implementing the left-most derivations, the first step is to construct the expectation matrix from the productions, then find the non-terminal expectation matrix.

	prog	series	stmnt	exp	factor
1	0	1	0	0	0
2	0	0.846	1	0	0
3	0	0	0	1	0
4	0	0	0	0.608	1
5	0	0	0	0.137	0

E =

$$(I-E) = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 0.154 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0.392 & -1 \\ 0 & 0 & 0 & -0.137 & 1 \end{bmatrix}$$

$$(I-E)^{-1} = \begin{bmatrix} 1 & 6.494 & 6.494 & 25.465 & 25.465 \\ 0 & 6.494 & 6.494 & 25.465 & 25.465 \\ 0 & 0 & 1 & 3.922 & 3.922 \\ 0 & 0 & 0 & 3.922 & 3.922 \\ 0 & 0 & 0 & 0.537 & 1.537 \end{bmatrix}$$

The average derivation length is

$$ADL = 1 + 6.494 + 6.494 + 25.465 + 25.465$$

$$= 64.918 \text{ steps}$$

There are 5 non-terminal symbols, the probability of each of these symbols would be

$$p(prog) = \frac{1}{64.918} = 0.015$$

$$p(series) = 0.1$$

$$p(stmnt) = 0.1$$

$$p(\text{exp}) = 0.392$$

$$\dot{p}(\text{factor}) = 0.392$$

The average word length would be

$$AWL = [10000]\begin{bmatrix} 1 & 6.494 & 6.494 & 25.465 & 25.465 \\ 0 & 6.494 & 6.494 & 25.465 & 25.465 \\ 0 & 0 & 1 & 3.922 & 3.922 \\ 0 & 0 & 0 & 3.922 & 3.922 \\ 0 & 0 & 0 & 0.537 & 1.537 \end{bmatrix}\begin{bmatrix} 0 \\ 0.846 \\ 1 \\ 0.608 \\ 1.137 \end{bmatrix}$$

$$= 6.494 * 0.846 + 6.494 + 25.465 * 1.745$$

$$= 56.424$$

Now to find the minimum code length per string; first try to evaluate the minimal average code length obtained from each state:

$$h_i = -\sum_{j \in R_i} p_j \log_2(p_j)$$

$$h_1 = 0$$

$$h_2 = -(0.846 \log_2 0.846 + 0.154 \log_2 0.154)$$

$$= 0.619$$

$$h_3 = 0$$

$$h_4 = -(0.392 \log_2 0.392 + 0.157 \log_2 0.157 + 0.137 \log_2 0.137 + 0.235 \log_2 0.235 + 0.079 \log_2 0.079)$$

$$= 2.121$$

$$h_5 = -(0.628 \log_2 0.628 + 0.235 \log_2 0.235 + 0.137 \log_2 0.137)$$

$$= 1.306$$

$$H = \sum_{i=1}^{5} p(v_i)h_i$$

$$= 0.1 * 0.619 + 0.392 * 2.121 + 0.392 * 1.306$$

$$= 1.405$$

So the minimum code per string is

$$MCL = ADL * H$$

$$= 64.918 * 1.405$$

$$= 91.210 \text{ bits}$$

The average code length per a state is

$$\ell = \sum_{i=1}^{5} p(v_i) \quad \sum_{j} p_j \ell_j$$

$$= 0.1(0.846+0.154) + 0.392(0.392+0.157*3+0.137*4$$

$$+0.235*2+0.079*4)+0.392(0.628+0.235*2+0.137*2)$$

$$= 1.499 \text{ bits/state}$$

Now the average code length per string would be

$$ACL = ADL * \ell$$

$$= 64.918 * 1.499$$

$$= 97.312 \text{ bits.}$$

7.9.2 Using Right-Most Derivations

For the right-most derivations, the average number of states would be:

$$ANS = ADL + AWL$$

$$= 64.918 + 56.424$$

$$= 121.342 \text{ states}$$

or

$$ANS = [10000] \begin{bmatrix} 1 & 6.494 & 6.494 & 25.465 & 25.465 \\ 0 & 6.494 & 6.494 & 25.465 & 25.465 \\ 0 & 0 & 1 & 3.922 & 3.922 \\ 0 & 0 & 0 & 3.922 & 3.922 \\ 0 & 0 & 0 & 0.537 & 1.537 \end{bmatrix} \begin{bmatrix} 1 \\ 1.846 \\ 2 \\ 1.608 \\ 2.137 \end{bmatrix}$$

$$= 1 + 6.494 * 3.846 + 25.465 * 3.745$$

$$= 121.342 \text{ states}$$

From the state probability (Table 7.4), and the frequency of choices in each state (Appendix D), it is possible to find the minimum code length of a state:

$$H = -\sum_{i=0}^{22} p(s_i) \sum_{j \in s_i} p_j \log_2(p_j)$$

$$= 0.0499 * 0.619 + 0.0499 * 1.423 + 0.1339 * 2.065 + 0.027 * 0.703$$
$$+ 0.0309 * 1.417 + 0.027 * 0.863 + 0.047 * 1.123 + 0.0158 * 1$$
$$+ 0.063 * 1.749$$
$$= 0.643 \text{ bits/state}$$

Now, the minimum code length per a string of symbols is

$$MCL = ANS * H$$
$$= 121.342 * 0.643$$
$$= 78.024 \text{ bits}$$

From Table (7.4), the average code length of a state is

$$\ell = \sum_{i=0}^{22} p(s_i) \sum_{j \in s_i} p_j \ell_j$$

$$= 0.0499 + 0.0499 * 1.436 + 0.1339 * 2.154 + 0.027 * 1.190$$
$$+0.0309 * 1.542 + 0.027 * 1.286 + 0.047 * 1.278 + 0.0158 * 1.5$$
$$+0.063 * 1.857$$
$$= 0.725 \text{ bits/state}$$

So, the average code length per string is

$$ACL = ANS * \ell$$
$$= 121.342 * 0.725$$
$$= 87.973 \text{ bits}$$

If a fixed code length for each character (8 bits) is used, then the average code length would be $ACL = 56.414 * 8$

$$= 451.312 \text{ bits}$$

character	frequency	code length	character	frequency	code length
=	39	3	g	4	7
*	36	3	q	4	7
;	33	4	k	3	7
+	24	4	m	3	7
-	21	4	q	3	7
(	21	4	4	3	7
)	21	4	5	3	7
1	16	5	p	2	8
a	14	5	z	2	8
c	12	5	O	2	8
r	12	5	f	1	8
w	12	5	j	1	8
/	12	5	θ	1	8
b	11	5	s	1	8
h	10	5	t	1	8
e	8	6	y	1	9
x	8	6	3	1	9
d	7	6	8	1	9
ℓ	7	6	u	0	11
i	6	6	v	0	11
2	6	6	6	0	11
n	5	7	7	0	11

TABLE 7.3: Character code lengths

Fig. (7.6) shows diagrammatically the above values. Note that the editing characters are excluded.

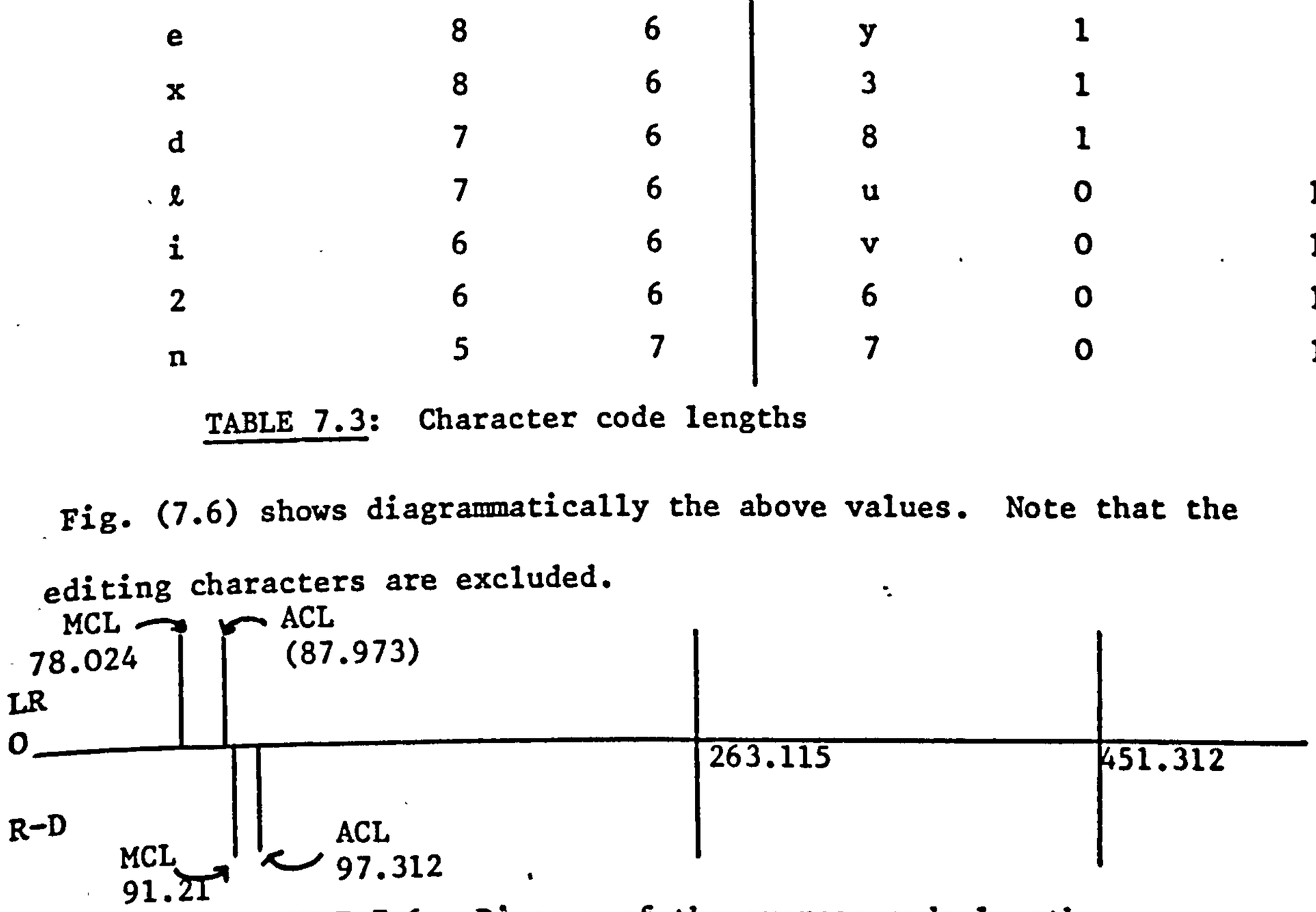


FIGURE 7.6: Diagram of the average code length

State no.	Freq.	Prob.	$\sum p_j \ell_j$	$-\sum p_j \log_2(p_j)$
0	6	0.008	0	0
1	6	0.008	0	0
2	39	0.0499	1	0.619
3	6	0.008	0	0
4	39	0.0499	0	0
5	33	0.0429	0	0
6	39	0.0499	1.436	1.423
7	33	0.0429	0	0
8	104	0.1339	2.154	2.065
9	60	0.0778	0	0
10	96	0.1238	0	0
11	36	0.0469	0	0
12	21	0.027	1.190	0.703
13	24	0.0309	1.542	1.417
14	21	0.027	1.286	0.863
15	36	0.047	1.278	1.123
16	12	0.0158	1.5	1
17	49	0.063	1.857	1.749
18	24	0.0309	0	0
19	21	0.027	0	0
20	36	0.047	0	0
21	12	0.0158	0	0
22	21	0.027	0	0

TABLE 7.4: Probability distributions, average code lengths and minimum code lengths of the states

Suppose that the average length of a string is given, then from Table (7.3) the average code length per character is

$$\ell = \sum p_i \ell_i$$

$$= 4.664 \text{ bits}$$

since the average word length of a string is 56.414, then the average code length would be:

$$ACL = 56.414 * 4.664$$

$$= 263.115 \text{ bits}$$

The efficiency of the first method (left-most derivation) would be

$$\eta = \frac{97.312}{451.312} = 0.216$$

whereas the efficiency of the second method (right-most derivation) would be:

$$\eta = \frac{87.973}{451.312} = 0.195$$

Samples of programs (Table 7.1) have been chosen to find the average code length of a string:

Total number of characters excluding the editing characters = 378

Total number of bits generated = 1242

$$AWL = \frac{378}{6} = 63 \text{ characters}$$

$$ACL = \frac{1242}{6} = 207 \text{ bits}$$

From Table (7.2)

Total number of bits generated = 1220

$$ACL = \frac{1220}{6} = 203.333 \text{ bits}$$

For a fixed code length per character (8 bits)

$$ACL = 63 * 8$$

$$= 504 \text{ bits}$$

Fig.(7.7) shows diagrammatically how far apart the average code length obtained from the samples, and the minimum and average code lengths obtained from the grammar, were in this case.

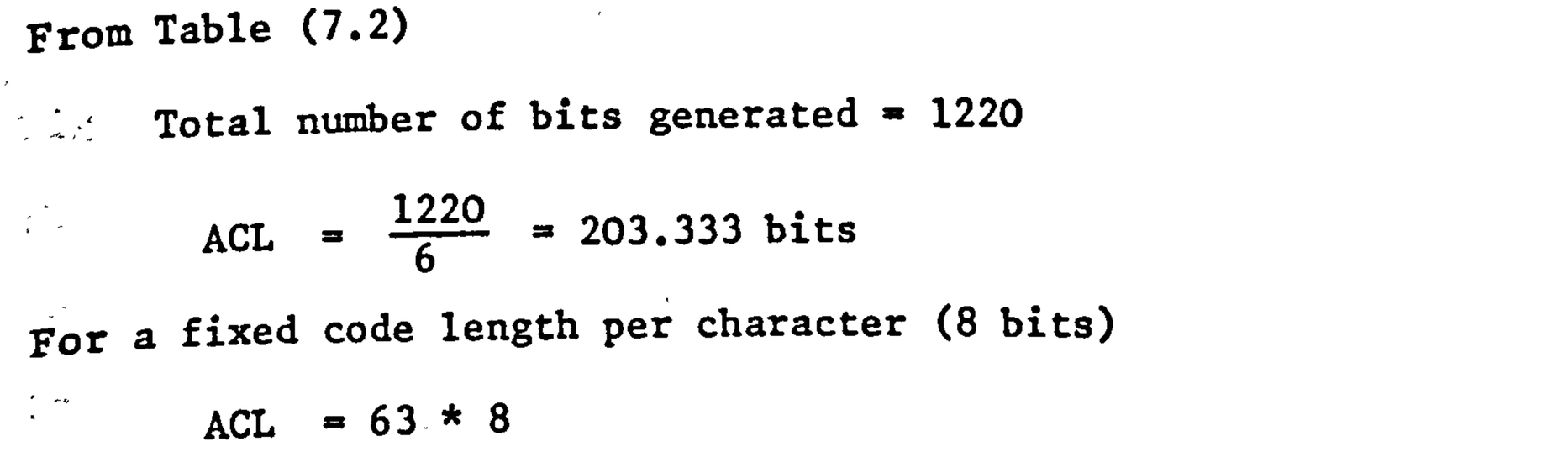


FIGURE 7.7: Diagram of the average code length

The same samples of programs (Table 7.1) have been chosen to find the average code length of a string. The difference here is that the editing characters are included.

Total number of characters = 640

Total number of bits generated = 2674

$$AWL = \frac{640}{6} = 106.667 \text{ characters}$$

$$ACL = \frac{2674}{6} = 445.667 \text{ bits}$$

From Table (7.2) the average code length of a string would be:

Total number of bits generated = 2652

$$ACL = \frac{2652}{6} = 442 \text{ bits}$$

If a fixed code length per character is used (8 bits), then

ACL = 106.667 * 8

= 853.336 bits

Fig. (7.8) shows that the inclusion of the editing characters as part of the program to be encoded, increases the code length and consequently, the average code length becomes far from the average code length generated from the grammar.

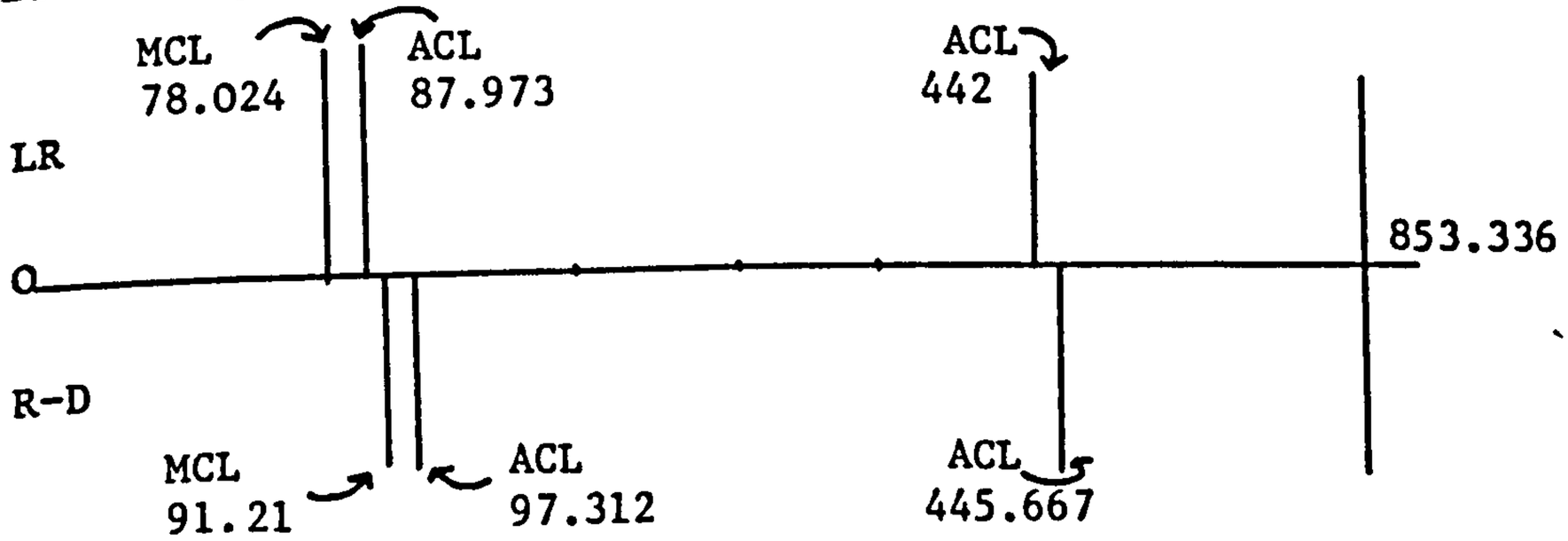


FIGURE 7.8: Diagram of the average code length

It seems that for the small grammar tested, the minimal theoretical average code length is

$$\frac{91.21-78.024}{91.21} * 100 = 14.45\ \%$$

smaller for the LR method than the R-D method. However, in practice over the sample programs used, the average code length was:

$$\frac{97.312-87.973}{97.312} * 100 = 9.59\ \%$$

smaller.

CHAPTER 8

SUMMARY AND CONCLUSIONS

General definitions and illustrations have been given of the concepts of probabilistic context-free languages, and the probabilistic context-free grammars which generate such languages. The construction of the parsing method LR(K) has been explained. The definition and the properties of a code were illustrated. The method of construction of a variable-length code known as Huffman code has been given.

The objectives of this study were:

1. To design a model for the compression of a text the content of which is partially recognized by a context-free grammar. The model was to be based on the parsing encoding technique.
2. To design a model for the decompression of the encoded string which must retain the exact original text.
3. To investigate the properties of the probabilistic context-free language, and the probabilistic context-free grammar.

The encoder has been implemented on the Pascal language. The results were encouraging since a considerable saving in storage space has been achieved. The encoder program consists of the parsing part and the encoding part. The program verifies the grammatical structure of the input by parsing it, and then generates the required codes. The parsing method used was LR(K) which is the most practical method for a right-most derivation technique. However, two main obstacles have been found. The first was that the manual construction of the set of states was tedious and time consuming. This has been overcome with the help of the program generator YACC. The second obstacle was the large size of the parsing tables. This was optimized by splitting the ACTION table into a number

of subtables, and also by compressing the GOTO table. Thus the size of the tables was dramatically reduced. For the encoding part, it generates three different Huffman codes. That is, codes for the grammatical actions; codes for user names, constants, and comments; codes for editing characters.

The decoder has also been implemented. The output was exactly as the original file. This has proved that the encoded data was indeed representing the original data. The program consists of two parts; the parsing part and the decoding part. The parsing part is similar to the parsing part of the encoder program. For the decoding part, each Huffman code has to be recognized before outputting any symbol. Since Huffman codes are variable-length codes, then a fair amount of computer time is spent on code manipulations because this involved a tree search. This is acceptable because computer space, not computer time, is the main concern in this study. However, the amount of computer time spent was small compared to the time spent on transmitting information to and from the storage devices. Since the volume of the information transmitted was reduced, thus the transmission time would be reduced.

Different statistical properties have been obtained from the probabilistic language and the probabilistic grammar. These include the average size of the input string, the average size of the encoded data, and the average number of states required to parse a string. A comparison has been done with an already existing parsing encoding method using a simple example of a probabilistic context-free grammar. The encoding method discussed earlier has produced a smaller average size of the encoded data than the average size of the encoded data produced by the

other method. Nevertheless, the method can not always guarantee the production of a smaller size for the encoded data for any string; the reason for this is that in a left-most derivation process, each non-terminal symbol can be represented by one state, and the first terminal symbols derived from the non-terminal symbol represent all the possible choices of that state. Thus the probability of selecting a particular choice would always be the same every time the corresponding terminal symbol is expected. Hence the same code would be generated. But in a right-most derivation process, the same choices could occur in more than one state depending on the grammar rules. Thus a particular choice might have different probabilities, and hence different codes, depending on which state it belongs to.

Thus the LR encoder method is feasible. It seems to be better than the corresponding top-down encoder on average but not for every program. It has also been demonstrated to be practical by using it to encode Pascal programs. The encoder and decoder can be used to compress programs in other languages (context-free languages) by changing the encoding and decoding tables.

REFERENCES

ABRAMSON, N. (1963), *"Information theory and coding"*, Mc-Graw Hill Inc.

AHO, A.V., & JOHNSON, S.C., (1974), *"LR parsing"*, Computing Surveys 6, pp.99-124.

AHO, A.V. & ULLMAN, J.D. (1972), *"Optimization of LR(K) parsers"*, J. Computer & Systems Sciences, 6:6, pp.573-602.

AHO, A.V. & ULLMAN, J.D. (1973), *"A technique for speeding up LR(K) parsers"*, SIAM J. Computing, 2:2, pp.106-127.

AHO, A.V. & ULLMAN, J.D. (1977), *"Principles of Compiler Design"*, Addison-Wesley.

ANDERSON, T., EVE, J. & HORNING, J.J. (1973), *"Efficient LR(1) parsers"*, ACTA Informatica, 2:1, pp.12-39.

BOOTH, T.L. & THOMPSON, R.A. (1973), *"Applying probability measures to abstract languages"*, IEEE Trans. on Computers, C-22:5, pp.442-450.

BORNAT, R. (1979), *"Understanding and writing compilers"*, The Macmillan Press Ltd.

CAMPBELL, H.G. (1965), *"An introduction to matrices, vectors and linear programming"*, Appleton-Century-Crofts.

DEMERS, A.J. (1975), *"Elimination of single productions and merging of non-terminal symbols in LR(K) grammars"*, J.Computer Languages, 1:2, pp.105-119.

DeREMER, F.L. (1971), *"Simple LR(K) grammars"*, CACM, 14, pp.453-460.

FELLER, W., (1957), *"An introduction to probability theory and its applications"*, Volume 1: John Wiley and Sons, Inc.

GALLAGER, R.G. (1968), *"Information theory and reliable communication"*, John Wiley.

GRIES, D. (1971), *"Compiler construction for digital computers"*, John Wiley & Sons.

HAHN, B. (1974), *"A new technique for compression and storage of data"*, CACM, 17:8, pp.434-436.

HAMMING, R.W. (1980), *"Coding and information theory"*, Prentice-Hall Inc.

HOLBOROW, C., McNEMAR, J. & STONEBURNER, P. (1976), *"A review of data compression algorithms"*, DCA 100- 73-C-0015, CCTG-TM-122-76, ADA 035786.

HOPCROFT, J.E. & ULLMAN, J.D. (1969), *"Formal languages and their relation to automata"*, Addison-Wesley.

HUANG, T. & FU, K.S. (1971), *"On stochastic context-free languages"*, Information Sciences 3, pp.201-224.

HUFFMAN, D.A. (1952), *"A method for the construction of minimum-redundancy codes"*, Proceedings of the I.R.E. 40, pp.1098-1101.

HUTCHINS, S.E. (1972a), *"Data compression in context-free languages"*, Information Processing, 71, North-Holland Publishing Co., pp.104-109.

HUTCHINS, S.E. (1972b), *"Moments of string and derivation lengths of stochastic context-free grammars"*, Information Sciences 4, pp.179-191.

JENNINGS, A. (1977), *"Matrix computation for engineers and scientists"*, John Wiley.

JONES, D.S. (1979), *"Elementary information theory"*, Oxford Press.

JOHNSON, S.C. (1978), *"YACC - Yet Another Compiler-Compiler"*, Bell Laboratories, Murray Hill, New Jersey, 07974.

JOLIAT, M.L. (1976), *"A simple technique for partial elimination of Unit productions from LR(K) parsers"*, IEEE Trans. on Computers, 25:7, pp.763-764.

KNUTH, D.E. (1965), *"On the translation of languages from left to right"*, Information & Control, 8:6, pp.607-639.

LEWIS II P.M., ROSENKRANTZ D.J. & STEARNS, R.E. (1976), *"Compiler Design Theory"*, Addison-Wesley.

MARTIN, W.C. Jr. (1976), *"An optimizing variable power data compression method"*, Ph.D. Thesis, Texas A & M University.

MAURER, W.D. (1969), *"File compression using Huffman coding"*, Conference: Computing Methods in Optimization Problems - 2, Academic Press, pp.247-256.

McGETTRICK, A.D. (1980), *"The definition of programming languages"*, Cambridge University Press.

PAGE, E.S. & WILSON, L.B. (1973), *"Information Representation and Manipulation in a Computer"*, Cambridge University Press.

PAGER, D. (1977), *"A practical general method for constructing LR(K) parsers"*, ACTA Informatica, 7:3 pp.249-268.

SCHUEGRAF, E.J. (1976), *"A survey of data compression methods for non-numeric records"*, Can. J. Inf. Science, Canada, 2:1 pp.93-105.

SCHWARTZ, E.S. (1964); *"An optimum encoding with minimum longest code and total number of digits"*, Information & Control, 7, pp.37-44.

SCHWARTZ, E.S. & KALLICK, B. (1964), *"Generating a canonical prefix encoding"*, CACM, 7:3, pp.166-169.

SMITH, A.J. (1976), *"A queueing network method for the effect of data compression on system efficiency"*, National Computer Conference, pp.457-465.

THOMPSON, R.A. (1971), *"Compact encoding of probabilistic languages"*, Ph.D. Thesis, The University of Connecticut.

THOMPSON, R.A. (1974), *"Determination of probabilistic grammars for functionally specified probability measure languages"*, IEEE Trans. on Computers C-23:6, pp.603-614.

THOMPSON, R.A. & BOOTH T.L. (1971), *"Encoding probabilistic context-free languages"*, Conference: Theory of Machines and Computations, Academic Press, pp.169-186.

WELLS, M. (1972), *"File compression using variable length encodings"*, The Computer Journal, 15:4, pp.308-313.

WETHERELL, C.S. (1980), *"Probabilistic languages: a review and some open questions"*, Computing surveys, 12:4, pp.361-379.

WILKINSON, J.H. (1965), *"The algebraic eigenvalue problem"*, Oxford University Press.

YOUNG, J.F. (1971), *"Information theory"*, Butterworth & Co.

APPENDIX A

PASCAL PRODUCTIONS

```
%token IDR DIGIT UNSREAL
%token STRING NOTEQ LESEQ GRTEQ ASSIGN DOTS
%token ARRAY PACKED CONST DO FILE SET FOR
%token TO DOWNTO IF THEN LABEL FUNCTION GOTO
%token OF PROGRAM ELSE RECORD TYPE UNTIL
%token VAR WHILE BEGIN END REPEAT WITH CASE
%token IN NIL MOD NOT PROCEDURE OR AND DIV
%token FORWARD EXTERN
%%
prog : PROGRAM IDR '(' idrlist ')' ';' block '.' ;
idrlist : IDR |
          idrlist ',' IDR ;
block : lpart cpart tpart vpart ppart spart;
lpart : LABEL llist ';' |
          ;
llist : DIGIT |
          llist ',' DIGIT ;
cpart : CONST clist ';' |
          ;
clist : IDR '=' const |
          clist ';' IDR '=' const ;
const : int |
          real |
          STRING |
          IDR |
          '+' IDR |
          '-' IDR ;
int : DIGIT |
      '+' DIGIT |
      '-' DIGIT ;
real : UNSREAL |
       '+' UNSREAL |
      '-' UNSREAL ;
tpart : TYPE tlist ';' |
          ;
tlist : IDR '=' type |
          tlist ';' IDR '=' type ;
type : '(' idrlist ')' |
       const DOTS const |
       IDR |
       PACKED unptype |
       unptype |
       '^' IDR ;
unptype : ARRAY '[' indlist ']' OF type |
          RECORD flist END |
          SET OF type |
          FILE OF type ;
indlist : type |
          indlist ',' type ;
flist : fpart |
        fpart ';' varpart |
        varpart ;
fpart : idrlist ':' type |
        fpart ';' idrlist ':' type |
         ;
```

```
varpart : CASE IDR ':' type OF varnlist|
          CASE type OF varnlist ;
varnlist : varnt |
           varnlist ';' varnt ;
varnt : caselab ':' '(' flist ')' |
        ;
caselab : const |
          caselab ',' const ;
vpart : VAR vlistd ';' |
        ;
vlistd : idrlist ':' type |
         vlistd ';' idrlist ':' type ;
ppart : fdec  ';' |
        ;
fdec : pdec |
       fdec ';' pdec ;
pdec : PROCEDURE IDR ';' block1 |
       PROCEDURE IDR '(' formlist ')' ';' block1 |
       FUNCTION IDR ':' IDR ';' block1 |
       FUNCTION IDR '(' formlist ')' ':' IDR ';' block1 |
       IDR ';' block1 ;
formlist : fparam |
           formlist ';' fparam ;
fparam : idrlist ':' IDR |
         VAR idrlist ':' IDR |
         PROCEDURE idrlist |
         FUNCTION idrlist ':' IDR ;
spart : BEGIN series END ;
series : stmt |
         series ';' stmt ;
stmt : DIGIT ':' st |
       st ;
st : var ASSIGN expr |
     IDR |
     IDR '(' outlist ')' |
     GOTO DIGIT |
     BEGIN series END |
     WHILE expr DO stmt |
     REPEAT series UNTIL expr |
     FOR IDR ASSIGN expr TO expr DO stmt |
     FOR IDR ASSIGN expr DOWNTO expr DO stmt |
     IF expr THEN stmt |
     IF expr THEN stmt ELSE stmt |
     CASE expr OF caselimbs END |
     WITH varlist DO stmt |
     ;
expr : sexpr |
       sexpr relop sexpr ;
sexpr : '+' term |
        '-' term |
        term |
        sexpr addop term ;
term : factor |
       term multop factor ;
```

```
factor : var |
         unsconst |
         IDR '(' aparlist ')' |
         '[' elmlist ']' |
         '(' expr ')' |
         NOT factor ;
var : IDR |
      var '[' aparlist']' |
      var '.' IDR |
      var '^' ;
unsconst : DIGIT |
           UNSREAL |
           STRING |
           NIL ;
relop : '=' |
        '<' |
        '>' |
        IN |
        NOTEQ |
        LESEQ |
        GRTEQ ;
addop : '+' |
        '-' |
        OR ;
multop : '*' |
         '/' |
         DIV |
         MOD |
        AND ;
elmlist : elmt |
          elmlist ',' elmt |
           ;
elmt : expr |
       expr DOTS expr ;
aparlist : expr |
           aparlist ',' expr ;
caselimbs : caselimb |
            caselimbs ';' caselimb ;
caselimb : caselab ':' stmt |
            ;
varlist : var |
          varlist ',' var ;
outlist : outval |
          outlist ',' outval ;
outval : expr |
         expr ':' expr |
         expr ':' expr ':' expr ;
pdec : FUNCTION IDR ';' block1 ;
block1 : block |
         FORWARD |
         EXTERN ;
```

APPENDIX B

ENCODER PROGRAM LISTING

```
# define NULL 0
# define stelmts 300
# define zero 0
# define one 1
# define newline '\n'
# define symlen 256
# define idrsize 25
# define bufsize 2
# define blocksize 512
int  fd1,  fdw;
char filename[14];
char *name;
int pointer;  /*stack pointer */
int endflag;
int errorflag;
int dotsflg;
int ch;  /* input character */
int class;
int l =1;
int d =2;
int others =3;
char *keyword[] ={
        "array","packed","const","do","file","set",
        "for","to","downto","then","label","function",
        "goto","of","program","else","record",
        "type","until","var","while","begin","end",
        "repeat","with","case","if",
        "in","nil","mod","not","procedure","or",
        "and","div","forward","extern",0 };
int token;
int idr =61;
int digit =62;
int unreal =37;
int noteq =38;
int leseq =39;
int grteq =40;
int assign =41;
int dots =42;
int lpar =43;
int dot =47;
int colon =48;
int les =57;
int grt =58;
int endfile =59;
int string =60;
int tokens[] ={
        44,45,46,49,50,51,52,53,54,55,56};
char *chars  ="),;[]^+-*/=";
int lenth, bits;  /* length and value to be encoded*/
struct symtag {
        int length;
        int code;
};
# include "realdata"  /* parsing tables */
# include "encodedata" /* encoding tables */
int stack[stelmts];
int word[idrsize];
int upper;
```

```
int bufpntr;  /*points to the current input in the buffer*/
char inpbuf[blocksize];
int idrposit[] ={
        2,4,8,16,32,64,128,256};
int symtab[symlen][idrsize];
int sympntr;  /* symbol table pointer */
int idradrs;
int oldidr;
int loc;  /* (editab) pointer*/
char editab[3];  /* store editing chars before encoding */
int stnglnth =6;
int stngcode =44;
int cmntlnth =13;
int cmntcode =1646;
int nlcount;
int counter;  /*no. of bits stored in a current word*/
int buff[bufsize];  /* output buffer*/
int index;  /* points to a current word*/

nextch()   /*get one char at a time*/
{
        int n;
        if(bufpntr >= blocksize) {
                n=read(fd1,inpbuf,blocksize);
                if(n != blocksize) inpbuf[n] = '\0';
                if(n<=0)error(4);
                else bufpntr=0;
        }
        ch=inpbuf[bufpntr++];
        if ((ch >= 'A' &&ch <= 'Z') || (ch >= 'a' &&ch <= 'z'))class=l;
        else if ( ch >= '0' && ch <= '9' ) class = d;
        else class = others;
        return ;
}
chckidr()
{
        register  i, j, r;
        upper = 0;
        while ( class == d || class == l || ch == '_')
        {
                word[upper++] = ch;
                nextch();
        }
        word[upper] = '\0';
        for ( i = 0; keyword[i] != 0; i++)
        {
                for (j=0;(r= keyword[i][j])==word[j] &&r!= '\0';j++);
                if ( r== word[j])
                {
                        token = i;
                        return;
                }
        }
        token = idr;
        lookup();
        return;
}
```

```
crname(s)   /*creats an encoded file name*/
char *s;
{
        name = filename;
        while (*name++ = *s++);
        name -= 2;
        *name = 'e';
}

codeproc(len,bitseq)  /*generate codes */
int len,bitseq;
{
int temp,j, wa;
register i;
temp = counter + len;
if(temp <= 32)
{
        counter = temp;
        buff[index] = buff[index]<< len | bitseq;
}
else {
        bitseq <<= (32 - len);
        for(i=counter +1 ;i<= 32; i++)
        {
                wa = bitseq & 020000000000;
                bitseq <<= 1;
                len--;
                buff[index]=buff[index]<<1 | (wa? one : zero);
        }
if(index == bufsize -1)  /*buffer is full*/
{
        j=write(fdw,buff,8);
        index = counter = 0;
}
else {
        index++;
        counter = 0;
}
for(i=0; i<len; i++)
{
        wa = bitseq & 020000000000;
        bitseq <<= 1;
        counter++;
        buff[index]=buff[index]<<1 | (wa? one : zero);
}
        }
}
```

```
oldproc()  /*encode the position in symbol table of an old idr*/
{
        register i;
        for(i=0;i<8;i++) if(sympntr < idrposit[i]) break;
        i += 1;
        codeproc(i,idradrs);
}

lastwrite()
{
        int i,j;
        for(i=counter + 1; i<= 32; i++) buff[index] <<= 1;
        j = write(fdw,buff,8);
}

storechar()  /* encode each char in a new idr or const*/
{
        register i,j;
        for(i=0; i< upper; i++)
        {
                j = word[i] ;
                codeproc(chtab[j].length,chtab[j].code);
        }
}

check()  /*encode idrs or constants*/
{
        int i;
        if(token==idr || token==digit || token==unreal)
        {
                codeproc(1,oldidr);
                if(oldidr==1) oldproc();  /*an old idr or const*/
                else {
                        i=upper;  /*new idr or const*/
                        while(1)
                        {
                                if(i>=7)
                                {
                                        codeproc(3,7);
                                        i -= 7;
                                }
                                else {
                                        codeproc(3,i);
                                        break;
                                }
                        }
                        storechar();
                }
        }
        else if(token == string) rdstring();
}
```

```
lookup()  /*search the sym. table for idr or const; store it if new*/
{
        int r;
        register i,j;
        for(i=0; i<sympntr; i++)
        {
                for(j=0; (r=symtab[i][j]) == word[j] && r !='\0';j++);
                if(r==word[j])  /*an old idr*/
                {
                        oldidr = 1;
                        idradrs = i;  /*position in sym. table*/
                        return;
                }
        }
        if(sympntr <= symlen)  /*new idr*/
        {
                for(j=0; j<=upper; j++) symtab[i][j] = word[j];
                oldidr = 0;
                idradrs =0;
                sympntr++;
                return;
        }
        else {
                printf("symbol table is full");
                exit();
        }
}
rdigit()
{
        while ( class == d)
        {
                word[upper++] = ch;
                nextch();
        }
        return;
}
more()
{
        if( class==d) rdigit();
        else if (ch == '-' || ch == '+')
        {
                word[upper++] = ch;
                nextch();
                rdigit();
        }
        else error(1);
        return;
}
```

```
chckint()
{
        upper = 0;
        rdigit();
        if(ch=='.')
        {
                nextch();
                if (class != d)
                {
                        if(ch =='.')
                        {
                                token = digit;
                                word[upper] ='\0';
                                lookup();
                                dotsflg = 1;
                        }
                        else error(1);
                        return;
                }
                word[upper++] = '.';
                rdigit();
                if( ch != 'E')
                {
                        token = unreal;
                        word[upper] = '\0';
                        lookup();
                        return;
                }
                word[upper] = 'E';
                nextch();
                more();
                token = unreal;
                word[upper] = '\0';
                lookup();
                return;
        }
        else if(ch == 'E')
        {
                word[upper++] = 'E';
                nextch();
                more();
                token = unreal;
                word[upper] = '\0';
                lookup();
                return;
        }
        else {
                token = digit;
                word[upper] = '\0';
                lookup();
                return;
        }
}
```

```
rdcomnt()  /*encode comments*/
{
        int temp;
        while(ch != '*')
        {
                codeproc(chtab[ch].length,chtab[ch].code);
                nextch();
        }
        nextch();
        if(ch==')') codeproc(cmntlnth,cmntcode);
        else {
                temp = '*';
                codeproc(chtab[temp].length,chtab[temp].code);
                rdcomnt();
        }
        return;
}

editproc()  /*encode editing chars */
{
        char wa;
        register i;
        if (loc)
        {
                codeproc(1,1);
                codeproc(2,loc);
                for(i =0; i< loc; i++)
                {
                        wa = editab[i];
                        if(wa==' ') codeproc(1,0);
                        else if(wa == '\t') codeproc(2,2);
                        else codeproc(2,3);
                }
                loc = 0;
        }
}
rdstring()  /*encode strings*/
{
        while (ch != '\'') {
                codeproc(chtab[ch].length,chtab[ch].code);
                nextch();
        }
        nextch();
        if( ch== '\'') {
                codeproc(chtab[ch].length,chtab[ch].code);
                codeproc(chtab[ch].length,chtab[ch].code);
                nextch();
                rdstring();
        }
        else codeproc(stnglnth,stngcode);
        return;
}
```

```
chckothr()
{
        int  c;
        register i;
        switch(ch) {
        case '<':
                nextch();
                if(ch == '>') token = noteq;
                else if (ch == '=') token = leseq;
                else token = les;
                if (ch == '>' || ch == '=') nextch();
                break;
        case '>':
                nextch();
                if (ch == '='  ){
                        token = grteq;
                        nextch();
                }
                else token = grt;
                break;
        case ':':
                nextch();
                if(ch == '=') {
                        token = assign;
                        nextch();
                }
                else token = colon;
                break;
        case '.':
                if(dotsflg== 1)
                {
                        token = dots;
                        nextch();
                        dotsflg = 0;
                        break;
                }
                nextch();
                if (ch == '.') {
                        token = dots;
                        nextch();
                }
                else token = dot;
                break;
        case '\'':
                nextch();
                token = string;
                break;
        case '\0':
                token = endfile;
                break;
        default:
                for(i =0; (c=chars[i]) != '\0' && c != ch ; i++);
                if (c == ch) {
                        token = tokens[i];
                        nextch();
                }
                else error(2);
                break;
        }
        return;
}
```

```
error(i)
int i;
{
        errorflag = 0;
        endflag = 0;
        switch(i) {
        case 1 :
                printf("lex.error___real number\n");
                break;
        case 2 :
                printf("lex.error___unknown input\n");
                break;
        case 3 :
                printf("syntax error \n");
                break;
        case 4 :
                printf("reached end of file\n");
                break;
        default:
                printf("unknown error\n");
                break;
        }
        printf("line number %d\n",nlcount);
        return;
}
```

```
scan()
{
        while(ch==' ' || ch=='\n' || ch=='{' || ch=='\t' || ch=='(')
        {
                switch (ch) {
                case ' ' :
                case '\t' :
                case '\n' :
                        editab[loc++] = ch;
                        if(ch=='\n') nlcount++;
                        if(loc >= 3) editproc();
                        nextch();
                        break;
        case '(' :
                        nextch();
                        if(ch != '*') {
                                token = lpar;
                                editproc();
                                codeproc(1,0);
                                return;
                        }
                        else{
                                editproc();
                                codeproc(4,9);
                                nextch();
                                rdcomnt();
                        }
                        nextch();
                        break;
                default :
                        editproc();
                        codeproc(4,8);
                        while(ch != '}')
                        {
                                nextch();
                                codeproc(chtab[ch].length,chtab[ch].code);
                        }
                        nextch();
                        break;
                }
        }
        editproc();
        codeproc(1,0);
        if ( class==1) chckidr();
        else if (class == d) chckint();
                else chckothr();
                return;
        }
```

```
action()  /*find the action and a code to be generated with its length*/
{
        register tbno, rwno, tok;
        int val;
        tbno = stab[stack[pointer]][0];  /*table number*/
        rwno = stab[stack[pointer]][1];  /*row number*/
        tok = toktab[token][tbno];  /*column number*/
        switch(tbno) {
        case 0:
                val = tab0[rwno];
                lenth=0;
                bits=0;
                break;
        case 1:
                val = tab1[rwno][tok];
                lenth = entab1[rwno][tok].length;
                bits = entab1[rwno][tok].code;
                break;
        case 2:
                val = tab2[rwno][tok];
                lenth = entab2[rwno][tok].length;
                bits = entab2[rwno][tok].code;
                break;
        case 3:
                val = tab3[rwno][tok];
                lenth = entab3[rwno][tok].length;
                bits = entab3[rwno][tok].code;
                break;
        case 4:
                val = tab4[rwno][tok];
                lenth = entab4[rwno][tok].length;
                bits = entab4[rwno][tok].code;
                break;
        case 5:
                val = tab5[rwno][tok];
                lenth = entab5[rwno][tok].length;
                bits = entab5[rwno][tok].code;
                break;
        case 6:
                val = tab6[rwno][tok];
                lenth = entab6[rwno][tok].length;
                bits = entab6[rwno][tok].code;
                break;
        case 7:
                val = tab7[rwno][tok];
                lenth = entab7[rwno][tok].length;
                bits = entab7[rwno][tok].code;
                break;
        case 8:
                val = tab8[rwno][tok];
                lenth = entab8[rwno][tok].length;
                bits = entab8[rwno][tok].code;
                break;
        case 9:
                val = tab9[rwno][tok];
                lenth = entab9[rwno][tok].length;
                bits = entab9[rwno][tok].code;
                break;
```

```
case 10:
        val = tab10[rwno][tok];
        lenth = entab10[rwno][tok].length;
        bits = entab10[rwno][tok].code;
        break;
case 11:
        val = tab11[rwno][tok];
        lenth = entab11[rwno][tok].length;
        bits = entab11[rwno][tok].code;
        break;
default :
        printf("unknown table number %d\n",tbno);
                exit();
        }
        return(val);
}
parse()  /*parsing routine*/
{
        register temp, tpntr, nonterm;
        int nont;
        while(( temp = action()) <0 && temp != -500) /*reduce action*/
        {
                temp = temp * (-1);
                if (lenth != 0) codeproc(lenth,bits);  /*generate a code*/
                pointer -= gramtab[temp][0];  /* popped off the stack*/
                tpntr = stack[pointer];
                nont = gramtab[temp][1];
                nonterm = nontab[nont];
                stack[++pointer] = gototab[gotopntr[tpntr]][nonterm];
        }
        if ( temp >= 0 && temp < 800 )  /*shift action*/
        {
                stack[++pointer] = temp;
                if (lenth != 0) codeproc(lenth,bits);
                check();
        }
        else if ( temp == -500) {
                endflag = 0;  /*accept the input*/
                lastwrite();  /*store the buffer*/
        }
        else error(3);
        return;
}
```

```
main(argc, argv)
int argc;
char **argv;
{
        int i, num;
        for (i=1; i< argc;i++)
        {
                fd1 = open(argv[i], 0);
                if(fd1 < 0) printf("%s not found\n",argv[i]);
                else {
                        crname(argv[i]);
                        fdw = creat(filename,0644);
                        pointer = 0;
                        stack[pointer] = 0;
                        counter = index = sympntr = nlcount =0;
                        bufpntr = 0;
                        num = read(fd1,inpbuf,blocksize);
                        if(num != blocksize) inpbuf[num] = '\0';
                        nextch();
                        scan();
                        endflag = errorflag = 1;
                        while( endflag)
                        {
                                parse();
                                if(errorflag==0 || token==endfile) endflag=0;
                                else scan();
                        }
                        if(errorflag == 0) {
                                printf("an error is occured On %s\n",argv[i]);
                                close(fdw);
                                unlink(filename);
                        }
                        else {
                                printf("%s is encoded\n",argv[i]);
                                close(fdw);
                        }
                        close(fd1);
                }
        }
        printf("end of execution\n");
        exit();
}
```

APPENDIX C

DECODER PROGRAM LISTING

```
# include <stdio.h>
# define stelmts 300
# define symlen 256
# define idrsize 25
# define bufsize 128
# define blocksize 512
# define reduce -100
# define reduce1 100
# define zero 0
# define one 1
# define binsize 683
# include "realdata" /*parsing tables*/
# include "decodedata" /*decode tables*/
int buff[bufsize];  /*input buffer */
int idrposit[] ={
        2,4,8,16,32,64,128,256};
char symtab[symlen][idrsize];
int sympntr;  /* symbol table pointer*/
int nofchrs;
int index; /*points to current input word*/
int counter; /* no. of bits read from a word*/
char filename[14];
char *name;
FILE *fd2;
int fd1;
int pointer; /* stack pointer*/
int endflag, comntflag;
int errorflag;
char *keyword[] ={
        "array","packed","const","do","file","set",
        "for","to","downto","then","label","function",
        "goto","of","program","else","record",
        "type","until","var","while","begin","end",
        "repeat","with","case","if",
        "in","nil","mod","not","procedure","or",
        "and","div","forward","extern"," ","<>","<=",">=",":=","..",
        "(",")",",",";",".",":","[","]","^","+","-","*",
        "/","=","<",">","\0",0};
int token;
int idr =61;
int digit =62;
int unreal =37;
int endfile =59;
int string =60;
int stack[stelmts];
int nlcount;
```

```
gtbit()  /*get one bit from the buffer*/
{
        int  j;
        register wa;
        if(counter >32)
        {
                if(index== bufsize - 1)  /*read a block */
                {
                        j = read(fd1,buff,blocksize);
                        if(j< 0) error(2);
                        else index =0;
                }
                else index++;
                counter = 1;
        }
        wa = buff[index] & 020000000000;
        buff[index] <<= 1;
        counter++;
        return(wa ? one : zero);
}

binserch(pntr)  /*search a binary tree*/
int pntr;
{
        register i, j, r;
        i = gtbit();
        while ((r = bintree[pntr][i]) <0)
        {
                j = r * (-1);
                pntr += j;
                i = gtbit();
        }
        if(r != reduce1)  token = r;
        else token = 0;
        return;
}
error(i)
int i;
{
        errorflag = 0;
        switch(i) {
        case 1 :
                printf("an error number in decod table\n");
                break;
        case 2 :
                printf("an error in the number of bytes read\n");
                break;
        case 3 :
                printf("syntax error \n");
                break;
        default:
                printf("unknown error\n");
                break;
        }
        printf("line number %d\n",nlcount);
        return;
}
```

```
oldproc()
{
        register i;
        for(i=0; i<8;i++) if(sympntr < idrposit[i])break;
        return(i + 1);
}

countchr()  /*find no. of chars in an idr or const */
{
        register i, bits, num;
        num = 0;
        for( i=1; i<= 3; i++)
        {
                bits = gtbit();
                num = num<< 1 | bits;
        }
        nofchrs += num;
        if(num== 7) countchr();
}

lookup()  /* print idrs or constants */
{
        int bit, count, postion, l, r;
        register i,j,k;
        if((bit = gtbit()) ==1)  /*already in symbol table */
        {
                postion = 0;
                count = oldproc();  /*length of the location */
                for(i=1; i<= count; i++)
                {
                        bit = gtbit();
                        postion = postion<< 1 | bit;
                }
                fprintf(fd2,"%s",symtab[postion]);
        }
        else {
                nofchrs = 0;  /*new idr or const*/
                countchr();
                for(i=0; i<nofchrs; i++)  /* store in symbol table */
                {
                        j= gtbit();
                        k = 0;
                        while ((r=chartab[k][j]) <0)
                        {
                                l = r * (-1);
                                k += l;
                                j = gtbit();
                        }
                        symtab[sympntr][i] = r;
                }
                symtab[sympntr][i] = '\0';
                fprintf(fd2,"%s",symtab[sympntr]);
                sympntr++;
        }
}
```

```
rdchars()  /* print chars of strings or comments */
{
        int k, l, ch;
        register i,j;
        while(1)
        {
                i=0;
                j=gtbit();
                while((k=chartab[i][j])<0)
                {
                        l=k*(-1);
                        i+= l;
                        j= gtbit();
}
ch= chartab[i][j];
if(ch==128){
        fprintf(fd2,"'");
        break;
}
else if(ch==129){
        fprintf(fd2,"*)");
        break;
}
else if(ch=='}' && comntflag ==0){
        fprintf(fd2,"}");
        break;
}
else fprintf(fd2,"%c",ch);
                }
        }
```

```
editproc()  /* output editing chars */
{
        register i,bit;
        while(gtbit())
        {
                nofchrs=0;
                for(i=1;i<=2; i++)  /*find no. of editing chars */
                {
                        bit = gtbit();
                        nofchrs = nofchrs<< 1 | bit;
                }
                if(nofchrs==0)  /*start a comment*/
                {
                        if(gtbit()) {
                                fprintf(fd2,"(*");
                                comntflag = 1;
                        }
                        else {
                                fprintf(fd2,"{");
                                comntflag =0;
                        }
                        rdchars();
                }
                else{
                        for(i=1; i<= nofchrs; i++)
                        {
                                if((bit=gtbit())==0) fprintf(fd2," ");
                                else if((bit=gtbit())==0) fprintf(fd2,"\t");
                                else {
                                fprintf(fd2,"\n");
                                nlcount++;
                                }
                        }
                }
        }
}
```

```
chktokn()
{
        if(token == idr || token == digit || token == unreal)
                lookup();
        else if(token== string) {
                fprintf(fd2,"'");
                rdchars();
        }
        else  fprintf(fd2,"%s",keyword[token]);  /* keyword or symbol*/
        return;
}

decode()   /*decodr routine */
{
        register i, j;
        i = stack[pointer];
        j = dcodtab[i];
        if(j == reduce) return;
        else if (j >= 1000 && j <= 1062) token = j - 1000;
        else if (j >= 0 && j <= binsize)
                binserch(j);  /* search the apprpriate tree */
        else error(1);
        return;
}

crname(s)  /*creat a file name */
char *s;
{
        name = filename;
        while (*name++ = *s++);
        name -= 2;
        *name = 'p';
}
```

```
action()  /* find the action */
{
        register tbno, rwno, tok;
        int val;
        tbno = stab[stack[pointer]][0];  /*table no */
        rwno = stab[stack[pointer]][1];  /*row number*/
        tok = toktab[token][tbno];  /*column no*/
        switch(tbno) {
        case 0:
                val = tab0[rwno];
                break;
        case 1:
                val = tab1[rwno][tok];
                break;
        case 2:
                val = tab2[rwno][tok];
                break;
        case 3:
                val = tab3[rwno][tok];
                break;
        case 4:
                val = tab4[rwno][tok];
                break;
        case 5:
                val = tab5[rwno][tok];
                break;
        case 6:
                val = tab6[rwno][tok];
                break;
        case 7:
                val = tab7[rwno][tok];
                break;
        case 8:
                val = tab8[rwno][tok];
                break;
        case 9:
                val = tab9[rwno][tok];
                break;
        case 10:
                val = tab10[rwno][tok];
                break;
        case 11:
                val = tab11[rwno][tok];
                break;
        default :
                printf("unknown table number %d\n",tbno);
                exit();
        }
        return(val);
}
```

```
parse()  /*parsing routine */
{
        register temp, tpntr, nonterm;
        int nont;
        while(( temp = action()) <0 && temp != -500)  /*reduce action*/
        {
                temp = temp * (-1);
                pointer -= gramtab[temp][0];  /*popped off the stack*/
                tpntr = stack[pointer];
                nont = gramtab[temp][1];
                nonterm = nontab[nont];
                stack[++pointer] = gototab[gotopntr[tpntr]][nonterm];
                decode();
        }
        if ( temp >= 0 && temp < 800 )  /*shift action*/
        {
                stack[++pointer] = temp;
                chktokn();
        }
        else if ( temp == -500)  endflag = 0;  /*accept*/
        else error(3);
        return;
}
```

```
main(argc, argv)
int argc;
char **argv;
{
        int i, j;
        for (i=1; i< argc;i++)
        {
                fd1 = open(argv[i], 0);
                if (fd1 < 0)printf("%s not found\n",argv[i]);
                else {
                        crname(argv[i]);
                        fd2 = fopen(filename,"w");
                        pointer = 0;
                        stack[pointer] = 0;
                        counter = 1;
                        nlcount = 0;
                        index = sympntr = 0;
                        j = read(fd1,buff,blocksize);
                        editproc();
                        decode();
                        endflag = errorflag = 1;
                        while( endflag)
                        {
                                parse();
                                if(errorflag==0 || token==endfile) endflag=0;
                                else {
                                        editproc();
                                        decode();
                                }
                                }
                                if(errorflag == 0)
                                {
                                        printf("an error is occured on %s\n",argv[i]);
                                        fclose(fd2);
                                }
                                else {
                                        printf("%s is decoded\n",argv[i]);
                                        fflush(fd2);
                                        fclose(fd2);
                                }
                                close(fd1);
                        }
                }
                printf("end of execution\n");
                exit();
}
```

APPENDIX D

LIST OF THE SET OF STATES FOR THE EXAMPLE

IN SECTION (7.9)

The following set of states is generated by YACC for the language used for the comparison.

state 0

$ accept:.prog $end

var s_4

. error

state 1

$ accept:prog. $end

$ end a

. error

state 2

prog: series.

series:series.;stmnt

; s_5

. r_1

state 3

series: stmnt.

. r_3

state 4

stmnt: var.=exp

= s_6

. error

state 5

series: series;.stmnt

var s_4

.error

freq.	length
33	1 bit
6	1 bit

state 6

stmnt: var=.exp

		freq.	length
var	s_{10}	22	1 bit
const	s_{11}	9	2 bits
(	s_{12}	8	2 "
. error			

state 7

series:series; stmnt.

. r_2

state 8

stmnt: var=exp.

exp:exp.+factor

exp:exp.-factor

exp:exp.*factor

exp:exp./factor

+	s_{13}	23	2 bits
-	s_{14}	5	3 "
*	s_{15}	26	2 "
/	s_{16}	11	3 "
.	r_4	39	2 "

state 9

exp:factor.

. r_9

state 10

factor:var.

. r_{10}

state 11

factor: const.

. r_{11}

state 12

factor:(.exp)

var	s_{10}	17	1 bit
const	s_{11}	4	2 bits
(	s_{12}	0	2 "

• error

state 13

exp:exp+.factor

var	s_{10}	10	2 bits
const	s_{11}	11	1 bit
(	s_{12}	3	2 bits

• error

state 14

exp:exp-.factor

var	s_{10}	15	1 bit
const	s_{11}	6	2 bits
(	s_{12}	0	2 "

• error

state 15

exp:exp*.factor

var	s_{10}	26	1 bit
const	s_{11}	6	2 bits
(	s_{12}	4	2 bits

• error

state 16

exp:exp/.factor

var	s_{10}	6	1 bit
const	s_{11}	0	2 bits
(	s_{12}	6	2 bits

• error

state 17

exp:exp.+factor

exp:exp.-factor

exp:exp.*factor

exp:exp./factor

factor:(exp.)

+	s_{13}	1	4 bits
-	s_{14}	16	2 bits
*	s_{15}	10	3 bits
/	s_{16}	1	4 bits
)	s_{22}	21	1 bit
.	error		

state 18

exp:exp+factor.

. r_5

state 19

exp:exp-factor.

. r_6

state 20

exp:exp*factor.

. r_7

state 21

exp:exp/factor.

. r_8

state 22

factor:(exp).

. r_{12}

APPENDIX E

LISTS OF SAMPLE PROGRAMS USED FOR THE COMPARISON IN SECTION (7.9)

```
i = i + 1;
w = 1 + l * 2;
r = w * 5 + 4 * w * h + w * h + 9 * (1 - w) * h;
x = w * x +(1-w) * g;
d = d / c;
d = (d - b * e) / c;
r = (r - a * r - b * r) /c;
n = r - d* x -e *x
```

```
w = 9 + k * 2;
n = 8;
a = b *(i - 1) + 1;
x = w * x +(1-w) *h
```

```
b = 0;
a = 0;
m = m - a;
h = 1/ (m-1);
t = 2 * e/ ( c*z);
p = 2;
x = (i-1) *h;
a = 5 + 2 * 1;
f = a + 5 * 1 * b
```

```
a = (c*3) /(b -2);
g = g +1;
h = (g*i) +j;
c = h - c
```

```
e = e * 4 +1;
d = (e - o) /(p * q +1);
a = e + d;
b = a + r +y *(9 /c)
```

```
k = a + b + 1;
s= (a*4) / (z - 9);
q = q + 1;
n = n *(l - 1) + 1;
x = w* i + (l - w) * h;
c = e + a - h;
c = c / b;
r = r / b;
r = (r - a * r) / (b-a * c);
k = n * l
```