On the Choice of Grammar and Parser for the Compact Analytical Encoding of Programs

R. G. STONE
Department of Computer Studies, University of Technology, Loughborough, Leicestershire, LE11 3TU

***If a computer program is analysed according to the language in which it is written then it is possible to represent the program by describing its derivation. It is shown that it is also possible to represent the program by recording the decisions made by a parser while parsing it. Thus the term* analytical encoding *embraces* derivation encoding *and* parsing encoding.**

Some general principles are given for choosing an analytical encoding method which yields good compression.

Received November 1983

INTRODUCTION

Conceptually, a program is usually stored and transmitted in token form, i.e. as a string of terminal symbols from a particular grammar. However, it is possible to represent a program by the choices made while deriving it from the starting symbol of the grammar. A leftmost (rightmost) derivation gives rise to a leftmost (rightmost) encoding. It is also possible to represent a program by tracing out the path taken through a parser or recogniser for the corresponding grammar. The parser, often a computer program simulating a finite-state machine, will have decisions at various points, and a program is represented by recording the sequence of decisions taken. This type of representation will be called parsing encoding. The term 'analytical encoding' is intended to embrace both derivation and parsing encoding.

Derivation encodings and parsing encodings are related but not identical. There are usually many parsers for a particular grammar and so many parsing encodings are possible. Given a particular language, e.g. Pascal, many 'equivalent' grammars may be written down, leading to different derivations and parsings of the same program.

If the choices made during derivation or parsing encoding are encoded carefully, taking into account the relative probabilities of each possible decision at a given point, the encoding process makes a useful program-compression scheme.

However, in order to establish the probabilities of the various choices at each decision point, a statistical analysis of a large number of programs must be undertaken. This is a fairly costly process and so it is not attractive to experiment with collecting different statistics for varous grammars and parsers.

The problem then is to decide in advance which grammar and which encoding scheme will produce on average the shortest encodings of programs in the language.

Derivation encoding is discussed in Refs 6 and 8. In Ref. 6 it is recommended both as a method of file compression and as an encryption method. Parsing encoding using an LR parser is discussed in Ref. 1, where practical results of compressing Pascal programs are reported.

1. DERIVATION ENCODING – AN EXAMPLE

Consider the grammar (later referred to as grammar G6)

Production	Code
L:E	L0
L:E , L	L1
E:a	E0
E:b	E1

The 'program' a,a,b has the following leftmost derivation (in a leftmost derivation the leftmost non-terminal in a sentential form is expanded first).

```
Sentential form    Production
L                  L:E , L
| \
E,L                E:a
|
a,L                L:E , L
  | \
a,E,L              E:a
  |
a,a,L              L:E
    |
a,a,E              E:b
    |
a,a,b
```

The derivation encoding is

L1 E0 L1 E0 L0 E1

There is only one possible binary code for two alternatives. One alternative is given the codeword '0' and the other the codeword '1'. So in this case the encoding would actually be the binary string

101001

Given only that this originated from a leftmost derivation the program can be reconstructed. In practice there will often be more than two choices and these would be coded with a (variable-length binary) prefix code depending on the relative probabilities of the derivations (see e.g. Ref. 4 for variable-length prefix codes).

In the case that only one derivation from a non-terminal occurs in the grammar no code at all need be sent. If ever the sentential form contains that non-terminal in the leftmost position the only possible derivation can be assumed.

Downloaded from https://academic.oup.com/comjnl/article/29/4/307/459999 by guest on 09 April 2024

The rightmost derivation (rightmost non-terminal expanded first) is

```
Sentential form    Production
L                  L:E,L
| \
E,L                L:E , L
  | \
E,E,L              L:E
    |
E,E,E              E:b
    |
E,E,b              E:a
  |
E,a,b              E:a
|
a,a,b
```

with derivation encoding

L1 L1 L0 E1 E0 E0

Notice that the rightmost derivation is a permutation of the leftmost derivation and not a reversal.

2. CHOOSING A DERIVATION

If the grammar in use is unambiguous then only one parse tree exists for each program. If the grammar is ambiguous then there will be rules (precedence, dangling else, etc.) which are used to choose a unique parse tree for any program. Thus in practice only one parse tree is associated with any program.

Given the unique parse tree for a program, all possible derivations including the leftmost and rightmost can be deduced. All these derivations will contain the same list of productions, though possibly permuted into different orders. This means that the encoded lengths of all derivation encodings must be the same. Therefore, if derivation encoding is chosen as a means of compression the most convenient derivation may be chosen freely. This will probably be dictated by the choice of a favourite parser to produce the derivation. Notice however that while a top-down parser produces a leftmost derivation, a bottom-up parser produces a rightmost derivation 'in reverse'. In this sense the popular LR parser is at a disadvantage.

3. CHOOSING A GRAMMAR

In what follows, distinctive production sets are analysed to discover their contribution to a derivation encoding. In particular, simple instances of the following four cases are discussed:

(i) Sequence	P:A B C . . .
(ii) Selection	P:A
	P:B
	. . .
(iii) Repetition	P:empty
	P:A P
(iv) Lists	P:A
	P:A , P

3.1 Sequence

Consider the grammar rule

P:A B C . . .

Once this production has been chosen for expansion in a leftmost derivation, non-terminal A will be expanded with probability 1.0. At some time later B, then C, etc. will necessarily be expanded. Thus there is no choice associated with the sequence rule, and no code is needed during a derivation encoding. Of course productions with a left-hand side A will be used, and these may give rise to choices and hence codes, but the decision to parse an A is taken with certainty. Exactly the same argument applies to rightmost derivations except that the sequence is expanded in reverse order.

Similar remarks apply to 'unit productions'. They cause no code to be generated and thus cannot affect the length of the encoding.

In the sense described above these rules are not critical in obtaining a best encoding.

3.2 Selection

Consider the following common productions in human-oriented syntax rules for Pascal.

Stmt: Structured-Stmt
Stmt: Simple-Stmt
Structured-Stmt: begin . . . .
Structured-Stmt: while . . . .
Simple-Stmt: Variable := Expression
Simple-Stmt: Proc-Ident Actual-Param-List-Option

The essential structure of these rules can be abstracted, giving

$$\left.\begin{array}{l} S:C \\ S:U \\ C:b \\ C:w \\ U:v \\ U:p \end{array}\right\} \text{Grammar G1}$$

Another common version of the same rules would abstract to

$$\left.\begin{array}{l} S:b \\ S:w \\ S:v \\ S:p \end{array}\right\} \text{Grammar G2}$$

In order to compare the encoding capabilities of these grammars let the probabilities of the four 'statements' b, w, v, p be p_b, p_w, p_v, p_p respectively. (In practice this corresponds to counting the relative frequencies of compound statements, while-loops, assignment statements and procedure calls in Pascal programs.)

In this simple case the program probabilities can easily be transformed into production probabilities.

For grammar G1

Production	Probability	Codeword
S:C	p_b+p_w	S0
S:U	p_v+p_p	S1
C:b	$p_b/(p_b+p_w)$	C0
C:w	$p_w/(p_b+p_w)$	C1
U:v	$p_v/(p_v+p_p)$	U0
U:p	$p_p/(p_v+p_p)$	U1

Downloaded from https://academic.oup.com/comjnl/article/29/4/307/459999 by guest on 09 April 2024

For grammar G2

Production	Probability	Codeword
S:b	p_b	Sb
S:w	p_w	Sw
S:v	p_v	Sv
S:p	p_p	Sp

The codes for G1, G2 are represented by symbols S0, S1, etc. which stand for strings of binary digits. (*Note.* The actual codes for G1 could have been inserted, since for only two alternatives the code is independent of the probabilities.)

The four 'programs' have the following leftmost encodings

Program	G1-encoding	G2-encoding
b	S0 C0	Sb
w	S0 C1	Sw
v	S1 U0	Sv
p	S1 U1	Sp

For a given set of messages and probabilities the Huffman code[5] is known to be optimal. So if the 4 codes (3 for G1 and 1 for G2) are Huffman codes they will be the best possible. The average encoded length of the four programs will be

G1

$(p_b+p_w)*L(\mathrm{S0})$
$+$
$(pv+p_p)*L(\mathrm{S1})$
$+$
$(p_b+p_w)*(p_b/(p_b+p_w))*L(\mathrm{C0})$
$+$
$(p_b+p_w)*(p_w/(p_b+p_w))*L(\mathrm{C1})$
$+$
$(p_v+p_p)*(p_v/(p_v+p_p))*L(\mathrm{U0})$
$+$
$(p_v+p_p)*(p_p/(p_v+p_p))*L(\mathrm{U1})$

G2

$p_b*L(\mathrm{Sb})$
$+$
$p_w*L(\mathrm{Sw})$
$+$
$p_v*L(\mathrm{Sv})$
$+$
$p_p*L(\mathrm{Sp})$

where $L(s)$ is the length of the string s.

It can now be established that the average length for G1 is never shorter than the average length for G2.

Let S0C0 stand for S0 and C0 concatenated, etc. Then S0C0, S0C1, S1U0, S1U1 are four codewords that could be used to code the four programs in the style of G2, giving an average length of

$$p_b*L(\mathrm{S0C0})+p_w*L(\mathrm{S0C1})+p_v*L(\mathrm{S1U0})+p_p*L(\mathrm{S1U1})$$

Since $L(\mathrm{S0C0}) = L(\mathrm{S0})+L(\mathrm{C0})$ this is the same as the first expression for the average length for G1. If this were less than that for G2 it would mean that G2 did not have the best set of codeword lengths. This would contradict the fact that the code in use for G2 is already optimum.

A simple generalisation of the above argument shows that no 'layering' of potential choices can improve the encoding. In fact the coding is liable to be worse. This is the first occurrence of the general principle of keeping the number of choices as large as possible.

For the most compact encoding it would seem that grammars should be written so that the right-hand side of every production begins with a terminal symbol (or is empty), i.e. in GNF or Regular form.

3.3 Repetition

Consider the grammar

$$\left.\begin{array}{l}\mathrm{P:empty}\\ \mathrm{P:A\ P}\\ \mathrm{A:a}\\ \mathrm{A:b}\end{array}\right\}\ \text{Grammar G3}$$

With this grammar a derivation encoding would encode each 'a' with two bits (one for P:A P, and one for A:a), each 'b' with two bits and the end of the sequence (P:empty) with one bit. Using N_a, N_b to represent the number of 'a's and 'b's in a program the coded length can be written

$$\text{length} = 2*N_a+2*N_b+1$$

From the lesson learned in section 3.2, the grammar is better written as

$$\left.\begin{array}{l}\mathrm{P:empty}\\ \mathrm{P:a\ P}\\ \mathrm{P:b\ P}\end{array}\right\}\ \text{Grammar}$$

Each program is now derived by $L+1$ ternary choices, where L is the length of the program ($L = N_a+N_b$). If the choices of 'a', 'b', 'empty' are encoded by sequences of length L_a, L_b, L_e respectively then the coded length of a program can be expressed as

$$\text{length} = L_a*N_a+L_b*N_b+L_e$$

If the choices are encoded using a Huffman code then the values of L_a, L_b, L_e are bound to be 1, 2, 2 in some permutation. If empty is the most probable choice then the lengths would be $L_a = 2$, $L_b = 2$, $L_e = 1$, giving the same result as before. If 'a' or 'b' were the most probable choice then this second version will have $L_a = 1$ or $L_b = 1$ and will produce a shorter average length.

Having come this far it is clear that a further 'improvement' can be gained by writing

$$\left.\begin{array}{l}\mathrm{P:empty}\\ \mathrm{P:a}\\ \mathrm{P:b}\\ \mathrm{P:a\ a\ P}\\ \mathrm{P:a\ b\ P}\\ \mathrm{P:b\ a\ P}\\ \mathrm{P:b\ b\ P}\end{array}\right\}\ \text{Grammar G5}$$

There is no theoretical limit to the potential 'improvement' in this style, but there is a serious practical problem with pursuing this expansion of the set of productions. This problem is taken up again in section 7.

3.4 Lists

Consider the grammar

$$\left.\begin{array}{l}\mathrm{L:E}\\ \mathrm{L:E\ ,\ L}\\ \mathrm{E:a}\\ \mathrm{E:b}\end{array}\right\}\ \text{Grammar G6}$$

which can be rewritten

$$\left.\begin{array}{l}\mathrm{L:a}\\ \mathrm{L:b}\\ \mathrm{L:a\ ,\ L}\\ \mathrm{L:b\ ,\ L}\end{array}\right\}\ \text{Grammar G7}$$

Downloaded from https://academic.oup.com/comjnl/article/29/4/307/459999 by guest on 09 April 2024

It is clear that this type of rule can continue to be 'improved' in the same way as the repetition above, e.g.

$$\left.\begin{array}{l} L:a \\ L:b \\ L:a\ ,\ a \\ L:a\ ,\ b \\ L:b\ ,\ a \\ L:b\ ,\ b \\ L:a\ ,\ a\ ,\ L \\ L:a\ ,\ b\ ,\ L \\ L:b\ ,\ a\ ,\ L \\ L:b\ ,\ b\ ,\ L \end{array}\right\} \text{Grammar G8}$$

Using the right recursive grammar G7 the probabilities of finishing a list with 'a' or 'b' are singled out. By using the corresponding left recursive grammar G9 the probabilities of starting a list with 'a' or 'b' become important.

$$\left.\begin{array}{l} L:a \\ L:b \\ L:L\ ,\ a \\ L:L\ ,\ b \end{array}\right\} \text{Grammar G9}$$

4. PARSING ENCODING OR DERIVATION ENCODING

Given a particular grammar there are usually several parsers that could recognise it. This section tries to establish whether a parsing encoding could be expected to improve on a derivation encoding.

Throughout this section the LR parser is used in examples as it is a well-understood, popular parser which can parse a broad class of grammars. A comparison of parsing encodings obtained by different parsing methods is left to section 5.

4.1 Sequence

As in section 3.1, it is argued that sequencing is never coded in parsing encoding and can be ignored.

4.2 Selection

It might be supposed that for optimum results, derivation encoding of grammars in the style of G2 is to be preferred. However, an LR parser can 'see through' rules of the form in G1 and makes decisions in the style of G2 automatically.

The set of states used by an LR parser for G1 would be

State	Set of items
0:	[A:_S] [s:_C] [S:_U] [C:_b] [C:_w] [U:_v] [U:_p]
1:	[A:S_]
2:	[S:C_]
3:	[S:U_]
4:	[C:b_]
5:	[C:w_]
6:	[U:v_]
7:	[U:p_]

The notation used follows that of Ref. 2. The corresponding parsing program is

State	Parsing alternatives
0:	case b: shift 4
	case w: shift 5
	case v: shift 6
	case p: shift 7
1:	default: accept
2:	default: reduce using S:C
3:	default: reduce using S:U
4:	default: reduce using C:b
5:	default: reduce using C:w
6:	default: reduce using U:v
7:	default: reduce using U:p

(*Note.* In this and subsequent parsing programs the possibility of errors is ignored. It is not possible to encode analytically a syntactically incorrect program. Errors should be detected and cause the encoding process to stop.)

In parsing any of the four programs a four-way choice is made away from state 1 and then two more unconditional reduce actions are made. Since no code will be needed for any of states 1 to 7 the LR encoding of G1 will be as good as the previous coding of G2.

4.3 Repetition

Consider grammar G3 again

P: empty
P: A P
A: a
A: b

The sets of states and parsing program for the corresponding LR parser are

State	Set of items
0:	[A:_P] [P:_] [P:_A P] [A:_a] [A:_b]
1:	[A:P_]
2:	[P:A_P] [P:_] [P:_A P] [A:_a] [A:_b]
3:	[A:a_]
4:	[A:b_]
5:	[P:A P_]

State	Parsing alternatives	Codeword
0:	case a: shift 3	S00
	case b: shift 4	S01
	default: reduce using P: empty	S02
1:	default: accept	
2:	case a: shift 3	S20
	case b: shift 4	S21
	default: reduce using P: empty	S22
3:	default: reduce using A:a	
4:	default: reduce using A:b	
5:	default: reduce using P:A P	

The empty program causes the parser to trace through the states 0, 1 giving a parsing encoding of S02.

The program aba is parsed as follows

State-stack	Input	Parsing action
0	aba	shift 3
0 3	ba	reduce using A:a
0 2	ba	shift 4
0 2 4	a	reduce using A:b
0 2 2	a	shift 3
0 2 2 3		reduce using A:a
0 2 2 2		reduce using P: empty
0 2 2 2 5		reduce using P:A P
0 2 2 5		reduce using P:A P
0 2 5		reduce using P:A P
0 1		accept

Downloaded from https://academic.oup.com/comjnl/article/29/4/307/459999 by guest on 09 April 2024

The sequence of states in the parse is

0 3 2 4 2 3 2 5 5 5 1

From the point of view of parsing encoding, states 1, 3, 4, 5 can be ignored, since they generate no code. Thus the parsing encoding of program aba is

S00 S21 S20 S22

Once again it is seen that the LR parser is making a wider choice than the grammar seems to suggest. In fact it is making choices in the style of grammar G4. In addition there are two separate states each with a three-way choice. State 0 is used at the start of the sequence of 'a's and 'b's and state 2 is used thereafter. If the opportunity is taken to collect separately the frequencies with which the actions of states 0 and 2 occur, separate codes could be used for each state. This should lead to a further improvement if (say) the probability of starting a list with 'a' was very different from the probability of an 'a' later on in the sequence.

4.4 Lists

Consider grammar G6 again

Production	Codeword
L:E	L0
L:E , L	L1
E:a	E0
E:b	E1

The leftmost encoding of some short programs is given below.

Program	Encoding
a	L0E0
b	L0E1
a,a	L1E0L0E0
a,b	L1E0L0E1
. . .	

Because all the parsing alternatives are binary choices the encodings will be $L+1$ bits long, where L = length of the original program.

The set of states and parsing program for an LR parser is given below.

State	Set of items
0:	[A:_L] [L:_E , L] [E:_a] [E:_b]
1:	[A:L_]
2:	[L:E_] [L:E_, L]
3:	[E:a_]
4:	[E:b_]
5:	[L:E ,_L] [L:_E] [L:_E , L] [E:_a] [E:_b]
6:	[L:E , L_]

State	Parsing alternatives
0:	case a: shift 3
	case b: shift 4
1:	default: accept
2:	case ,: shift 5
	default: reduce using L:E
3:	default: reduce using E:a
4:	default: reduce using E:b
5:	case a: shift 3
	case b: shift 4

The program 'a,b' is recognised by tracing through the state sequence 0, 3, 2, 5, 4, 2, 6, 1. States 1, 3, 4 and 6 can be ignored since they contain no alternative. States 0, 2 and 5 each contain binary choices and thus have codes with length 1. State 0 is entered once by every program. State 2 is entered once for each comma and once more. State 5 is entered once for each 'a' or 'b' except the first. So the coded length of programs by this method will be the same as for the leftmost derivation encoding.

In this case the LR scheme seems to have no advantage. Certainly it is bound to agree with the leftmost encoding in the coding of commas. However, if there were three or more choices of elements

E:a
E:b
E:c

the LR scheme might regain the advantage. The point to notice is that the code used to show every derivation from E in the leftmost derivation will be established by the 'total' probabilities of 'a', 'b' and 'c'. The LR parser will use the probabilities of a list *beginning* with 'a', 'b' or 'c' in state 0 and the probabilities of subsequent occurrences of 'a', 'b' or 'c' in state 5. This will be to the advantage of the LR scheme if (say) the probability of starting a list with 'c' was quite high, but the probability of further 'c's was quite low.

Returning to the encoding of the commas, it can be seen that both systems code the length of the list essentially as follows

Length	Encoding
1	0
2	10
3	110
4	1110
. . .	

This method of counting has been called 'stone age binary', and will only be a satisfactory coding scheme if the probability of a list of length i is approximately twice the probability of a list of length $i+1$, for all i.

5. CHOOSING A PARSER

First compare an LL parsing encoding against an LR parsing encoding. Consider the grammar G3 again

P:empty
P:A P
P:a
P:b

The LL parser is essentially described by a table with rows indexed by non-terminals and columns indexed by tokens.[3] In this case the table is

	a	b	$end
P	P:A P	P:A P	P:empty
A	A:a	A:b	

The LL parser operates using a stack of grammar symbols and makes parsing decisions when the top-of-stack symbol is a non-terminal. It uses this non-terminal and the next token in the input to index the table. The entry in the table is the grammar rule which is to be expanded next.

First, notice that the LL parser does not naturally expand the choices available like the LR parser (see 4.3). Secondly, the LL parser treats all 'a's and 'b's in the same

Downloaded from https://academic.oup.com/comjnl/article/29/4/307/459999 by guest on 09 April 2024

way and does not distinguish the first 'a' or 'b' from the rest like the LR parser.

The critical observation here is that it is common in an LR parser for the recognition of a derivation from a particular production to be reported from one of several states depending on context. This never happens in an LL parser, so the advantage shown by this small example is likely to be obtained with any grammar.

This context sensitivity of the LR parser could be expected to be useful in practice. Consider the grammar

```
STMT:begin STMTSEQ end
STMT:while EXP do STMT
STMT:repeat STMT until EXP
STMTSEQ:STMT
STMTSEQ:STMT STMTSEQ
```

The state containing the core item

[STMT:**while** EXP **do** _ STMT]

will be able to record a large probability of the following STMT having 'begin' as its first token. The state containing the core item

[STMT:**begin** _ STMTSEQ **end**]

will be able to record a very small probability of the first STMT in the STMTSEQ opening with 'begin'. So the evidence seems to suggest that the LR parser is more information-conscious than the LL parser.

If the leftmost derivation is being produced by a recursive descent parser then it is committing itself to a parsing decision on seeing the first token derived from a non-terminal. An LR parser on the other hand waits till all the tokens derived fom a non-terminal have been seen before making its reduction. In this sense the LR parser has more information available at the time of a decision. The only type of top-down parser which could be expected to delay its decisions even longer would be one which backtracked. This could (potentially) read all the program before announcing its parse.

6. THEORETICAL MINIMUM AVERAGE CODED LENGTH

If a message source is imagined which emits the programs P_0, P_1, P_2, etc. with the probabilities p_0, p_1, p_2, etc. then the source is said to have entropy

$$-\sum_i p_i * \log(p_i)$$

If the logarithms are taken to base 2 this gives a theoretical minimum average coded length of programs in bits. It would be nice to use this as a base to measure various actual coding schemes, but for real languages this is clearly impractical.

However, if a probabilistic grammar for the language is available it is possible to deduce the entropy. Before describing how this is done it is shown that dividing a source does *not* change its entropy. This means that all probabilistic grammars which describe a given language have the same entropy. This is in contrast to the result of section 3.2 where it is shown that dividing a source can affect the average code length.

Suppose again that a source emits messages with probabilities p_0, p_1, ..., p_n. Without loss of generality assume that these messages are to be split into two groups containing the messages 0 to I and $I+1$ to n. Let

$$PL = p_0 + p_1 + \dots + p_I$$

and

$$PH = p_{I+1} + p_{I+2} + \dots + p_n$$

Then the entropy of the lower group (in isolation) is

$$HL = -\sum_{i=0}^{I} (p_i/PL) * \log(pL/p_i)$$

Similarly for the entropy of the higher group HH. Thus the entropy of the remodelled source is

$$\begin{aligned} &-PL * \log(1/PL) + PL * HL \\ &\qquad -PH * \log(1/PH) + PH * HH \\ &= -PL * \log(1/PL) - \sum_{i=0}^{I} p_i * \log(PL/p_i) \\ &\qquad -PH * \log(1/PH) - \sum_{i=I+1}^{n} p_i * \log(PH/p_i) \\ &= -\sum_{i=0}^{n} p_i * \log(1/p_i) \end{aligned}$$

Hence the entropy of a divided source is the same as the entropy of the original source.

Most of the following steps in the calculation of the entropy are from Wetherell.[9]

First, define a matrix Q with elements Q_{ij} such that if non-terminal i occurs on the left-hand side of production j, Q_{ij} = probability of production j, otherwise $Q_{ij} = 0$.

Working with grammar G3 and symbolic probabilities

Production	Probability
P:empty	P0
P:A P	P1
A:a	A0
A:b	A1

Q is found to be

$$Q = \begin{vmatrix} P0 & P1 & 0 & 0 \\ 0 & 0 & A0 & A1 \end{vmatrix}$$

Define a matrix C with elements C_{ij} such that C_{ij} is the number of times that non-terminal j occurs on the right-hand side of production i.

$$C = \begin{vmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 0 \\ 0 & 0 \end{vmatrix}$$

Define a matrix A to be the matrix product $Q * C$.

$$A = \begin{vmatrix} p1 & p1 \\ 0 & 0 \end{vmatrix}$$

Finally, after checking that the grammar is consistent (by showing that the spectral radius is less than 1) define a matrix A' to be $1/(1-A)$

$$A' = \begin{vmatrix} 1/p0 & p1/p0 \\ 0 & 1 \end{vmatrix}$$

The first row of A' (associated with the start symbol) gives the expected number of each of the non-terminals in an arbitrary derivation. Thus for grammar G3, $1/p0$ Ps are expected and $p1/p0$ As. Treating the non-terminals as separate sources their entropies can be found. The entropy of the language is then

$$\sum_{j=1}^{N} A_{1j} * H_j$$

Downloaded from https://academic.oup.com/comjnl/article/29/4/307/459999 by guest on 09 April 2024

where H_j is the entropy of non-terminal j, and N is the total number of non-terminals.

This is a practical method of establishing the entropy of the language and thus the minimum average coded length of programs.

7. SUGGESTED ENCODING SCHEME

The main weakness exposed in the encoding schemes discussed so far has been in dealing with repetition and lists. The following discussion works towards a practical scheme which can be used to improve the *information consciousness* of the encoding of repetition and lists.

Consider the grammar below, which defines the same language as G6

Production	Codeword
L:E	L0
L:Pairs	L1
L:E , Pairs	L2
Pairs:E , E	P0
Pairs:E , E , Pairs	P1
E:a	E0
E:b	E1

The leftmost derivation begins with a ternary choice, and then for longer programs uses binary choices to show how many *pairs* of elements there are in the program. The LR parser for this grammar contains four states which choose between 'a' and 'b', and three states to choose comma-continuation or not. Thus the LR parsing encoding will be a sequence of binary choices as before (section 4.4). Since both methods encode the elements of the lists in the same way, only the encoding of the lengths of the lists is compared below (B indicates a binary decision).

Length	Leftmost	LR
1	L0	B
2	L1P0	BB
3	L2P0	BBB
4	L1P1P0	BBBB
5	L2P1P0	BBBBB
6	L1P1P1P0	BBBBBB
7	L2P1P1P0	BBBBBBB
8	L1P1P1P1P0	BBBBBBBB

The lengths of L0, L1, L2 will be 1, 2, 2 in some permutation and the lengths of P1, P0 will each be 1. So for lists of length 5 or more the leftmost encoding is bound to be the shortest. The leftmost encodings of lists of lengths 3 and 4 cannot be longer then the LR parsing encodings. So the only lists that could be longer in the leftmost encoding are those of lengths 1 and 2. So provided the probabilities of the longer lists are high enough the leftmost derivation is the best, as predicted at the end of section 5. However, the leftmost derivation will have to be produced by a backtracking parser. If a recursive descent parser were to be used then the grammar would have to be left-factored, which would result in grammar G6 again.

The ideal of identifying larger sublists can be extended indefinitely, e.g.

L:E
L:E , E
L:E , E , E
L:Quads
L:E , Quads
L:E , E , Quads
L:E , E , E , Quads
Quads:E , E , E , E
Quads:E , E , E , E , Quads
E:a
E:b

The leftmost derivation now contains a 7-way choice, and stone age binary is used to indicate how many Quads there are in long lists. Since the grammar is beginning to look a little ridiculous the common abbreviation using a star rule (* = as many as you like of) is now considered.

The star rule

L:E (, E) *
E:a
E:b

The encoding of the left most derivation in this case could be

$$(E0 \mid E1)\ (n)\ \underbrace{(E0 \mid E1)\ (E0 \mid E1) \ldots (E0 \mid E1)}_{n \text{ times}}$$

Now the need arises for a prefix code for the ordinal numbers which is optimal for the observed probability of each length of list.

Apart from stone age binary, other infinite prefix codes exist (e.g. Ref. 7) but they have very definite patterns of lengths and could not be expected to provide the optimum code for an arbitrary set of probabilities.

A practical solution is as follows. Suppose the rule in question is the Pascal statement sequence rule. In any finite group of programs analysed, the largest number of statements in a sequence can be found (say N). Let the lengths from 0 to N occur with probability p_i, $i = 0(1)\ N$. Create a Huffman code for $N+2$ messages, being the numbers 0 to N with probabilities p_i and an extra message with infinitesimal probability. If, in use, a program with more than N statements in a sequence is found the special message can be sent together with the excess length in stone age binary.

This technique allows a good encoding of the expected cases plus the capability to cope with the unexpected. However, a backtracking parser is needed to produce the derivation required unless the following scenario is accepted. Modify an LR parser built on the list rule of grammar G6 so that it produces the derivation expected by the star rule. The counts can be output *after* the derivations of the repeated entries since the parser naturally produces the derivation in reverse. This avoids backtracking during parsing but requires the whole output to be reversed.

8. APPLICATION OF METHOD

The term 'program' as used in this paper should be taken to mean a sentence in the language generated by some context-free grammar. This means that analytical compression can be applied to any data which have formal syntactic structure (e.g. database, perhaps).

Downloaded from https://academic.oup.com/comjnl/article/29/4/307/459999 by guest on 09 April 2024

Since the data are parsed as part of the compaction process they must be syntactically correct. However (semantically) incorrect programs in the process of development can still be compacted. For example, programs with missing procedure declarations or dummy procedure bodies (begin {null} end) can be syntactically correct and successfully compacted.

The compaction process described here does not preserve editing characters (white space) or comments. In the absence of comments a pretty-printer could be used on the decoded form to make it 'human-readable'. If this is not satisfactory then the compaction scheme must be extended to encode and decode the white space and comments as for example in Ref. 1.

9. CONCLUSION

A method of program compaction has been introduced which encodes the decisions made by a parser while parsing the program. It is felt that this method of encoding deserves the name *parsing encoding*, while previous use of the term (e.g. in Ref. 8) should be replaced by *derivation encoding*. The term *analytical encoding* has been used to embrace derivation and parsing encoding.

Four distinctive sets of productions have been discussed to demonstrate their effect on the derivation encoding and parsing encoding of programs.

An optimum encoding of (potentially infinite) repetition and lists seems to require a grammar containing an infinite number of productions. Since this is impractical, some compromise over the encoding of repetition and lists is necessary.

It seems that the LR parser, currently popular for other reasons, is also a good practical choice for parsing encoding. Its encoding of sequence and selection is optimal. Its encoding of repetition and lists can only be improved upon by a backtracking parser, which might be rejected on the grounds of efficiency.

The LR parser displays a small amount of context sensitivity which can be utilised during parsing encoding. An example from the Pascal language showed that this could be useful in practice.

Acknowledgements

The author would like to record his thanks to Professor M. Wells of Leeds University and Dr C. J. Hinde of Loughborough University for their detailed comments on draft copies of this paper.

REFERENCES

1. A. M. M. Al-Hussaini, File compression using probabilistic grammars and LR parsing. *Ph.D. Thesis*, Loughborough University (1983).
2. A. V. Aho and S. C. Johnson, LR Parsing. *Computing Surveys* **6** (2), 99–124 (1974).
3. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Addison Wesley, London (1977).
4. R. W. Hamming, *Coding and Information Theory*. Prentice-Hall, Englewood Cliffs, NJ (1980).
5. D. A. Huffman, A method for the construction of minimum-redundancy codes. *Proc. IRE* **40**, 1098–1101 (1952).
6. S. E. Hutchins, Data compression in context-free languages. *Proc. IFIP* **1**, 104–109 (1971).
7. R. G. Stone, On encoding commas between strings. *CACM* **22** (5), 310–311 (1979).
8. R. A. Thompson and T. L. Booth, *Encoding of Probabilistic Context-Free Languages*, (Conference Report: Theory of Machines and Computations). Academic Press, London (1971).
9. C. S. Wetherell, Probabilistic languages: a review and some open questions. *Computing Surveys* **12** (4), 361–379 (1980).

Downloaded from https://academic.oup.com/comjnl/article/29/4/307/459999 by guest on 09 April 2024