

Working Paper Series
ISSN 1170-487X

**Compressing computer
programs**

by Rod M. Davies & Ian H. Witten

Working Paper 93/7

October, 1993

© 1993 by Rod M. Davies & Ian H. Witten
Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Compressing Computer Programs

Rod M. Davies

Computer Science, University of Waikato, Hamilton, New Zealand
email rdavies@waikato.ac.nz

Ian H. Witten

Computer Science, University of Waikato, Hamilton, New Zealand
email ihw@waikato.ac.nz

1. Introduction

Computer programs offer great potential for compression. The programming languages in which they are written are formally defined with a specified lexical and syntactic structure. Many items they contain are “reserved words,” that is, pre-defined terms of the language; and other words have particular rules for their introduction and re-use. Taken together, these suggest that high levels of compression should be achievable on this type of data—certainly higher than for normal text.

We know of no previous work on lossless compression of computer programs (except for Barry, 1988, who describes the design of a lossless compression system but includes no results). The small literature that exists on program compression (e.g. Katajainen *et al.*, 1986; Stone, 1986; Cameron, 1988) invariably describes compression methods that are *lossy* in that all white space is discarded—indeed, the first two of these papers omit comments too. It is tempting when considering syntax-directed compression to ignore non-syntactical items like white space and comments and concentrate on coding the parse tree alone (Katajainen and Mäkinen, 1990). The functionality of programs is certainly preserved—and while one is at it one might as well standardize variable names too, and other identifiers, and omit them from the compressed representation. However, to regard source programs as merely encoding functionality is to relegate them to the status of object programs. In actuality they do far more: they are vehicles for the expression of algorithms for people to study and modify, as well as for computers to compile and execute.

From a user’s viewpoint it is often necessary to have programs reproduced exactly. While one might argue that in principle pretty-printers eliminate the need to reproduce white space, we all know that individuals become attached to their own layout styles and conventions and staunchly resist the imposition of someone else’s uniformity on their programs. Comments form an essential part of all non-trivial computer programs, as do user-defined identifiers.

This paper describes a scheme for compressing programs written in a particular programming language—which can be any language that has a formal lexical and syntactic description—in such a way that they can be reproduced exactly. Only syntactically correct programs can be compressed. The scheme is illustrated on the PASCAL language, and compression results are given for a corpus of PASCAL programs; but it is by no means restricted to PASCAL. In fact, we discuss how a “compressor-generator” program can be constructed that creates a compressor automatically from a formal specification of a programming language, in much the same way as a parser generator creates a syntactic parser from a formal language description.

2. Method of compression

The compression scheme works as follows. Given a syntactically correct program, it is parsed into a parse tree. The parser interacts with a lexical analyzer that produces a stream of language tokens. Nodes of the tree are labeled with token types—such as “identifier”—rather than actual instances of tokens. Lossless program compression

involves the compression of lexical elements that compilers generally discard. The lexical analyzer must provide enough extra information to allow the original tokens to be reproduced, and to permit white space and comments to be preserved.

Some language tokens—for example operators, keywords and punctuation—have a single lexical representation, while others—numbers, strings and identifiers—have a large number of different representations. In the first case, knowledge of the token type is sufficient to reproduce it, whereas in the second the actual instance of the token must be preserved separately. For each multiply-represented token, a stream of instances is generated during parsing.

Once parsing is complete, the program is represented by a parse tree, a whitespace/comment stream, and several token streams. Taken together, these provide enough information to allow the original program to be reconstructed exactly. Implementation details of the individual steps follow.

PARSING AND PARSE TREE COMPRESSION

We developed the parser with standard compiler building tools, LEX (Lesk, 1975) and YACC (Johnson, 1975). It was constructed in the usual way, with a lexical analysis phase, a syntax analysis phase, and a main program coordinating the two. Since the compressor is intended to be invoked only on programs that have already been compiled and so are guaranteed to be syntactically correct, no error-recovery facilities were incorporated: this greatly simplifies the task of constructing a syntax analyzer. The output of this module is a list of productions that represent the parse tree for the target program.

The production list representing the parse tree is compressed as follows. Initially, and at each point during parsing, a certain production must be encoded out of a specific set of candidates. The set of candidates is given by the grammar: it comprises all rules that have that the current non-terminal as head. For example, if the current non-terminal is A and there are rules

$$A ::= B \mid C \mid D,$$

then information must be encoded that specifies which of B, C, or D actually occurs. This is done using adaptive arithmetic coding based on the frequencies with which B, C and D have occurred in this context so far. Frequency counts are maintained for each possibility, initialized to 1 to prevent problems with zero coding probabilities, and updated each time one of the three rules is encountered.

While the parse tree compressor that we built and tested is for the PASCAL language, it can readily be modified to deal with any language whose syntax is formally defined. The use of compiler-generator tools means that the lexical and syntactic properties of the language are specified independently from the actual parsing engine. All that must be added to a standard language parser is the ability to output the parse tree in the form of a list of productions.

TOKEN STREAMS AND THEIR COMPRESSION

As noted above, some token types have a single well-defined lexical representation, while others have different lexical representations which must be coded. In the PASCAL language, the ones in the second class are:

- identifiers;
- strings;
- numbers, both integers and reals.

The corresponding token streams are extracted from the target program by adding instructions to the lexical analyzer to output all instances of the tokens to an appropriate file. On completion of parsing, three streams have been generated: an identifier stream, a string stream, and a number stream.

These streams are coded and transmitted in different ways. For identifiers, a static model of occurrence counts is constructed and transmitted, and then the actual identifier sequence is encoded relative to this static model. Figure 1 shows an example. The identifier sequence is reproduced in part (a), part (b) gives the corresponding static model, part (c) shows the string of symbol table indexes corresponding to the identifier sequence, and part (d) indicates what is actually transmitted.

First the model is encoded. We use the standard PPMC method for this (Cleary & Witten, 1984; Moffat, 1990; Bell *et al.*, 1990), encoding the sequence of names and then the sequence of counts. A character that cannot occur in identifiers is used to separate them—comma in this example. For the sequence of counts, PPMC has its alphabet restricted to eleven possible symbols: the digits 0–9 and comma. The number of counts is the same as the number of identifiers and so there is no need for another symbol to terminate the sequence. There are more efficient ways than this of encoding the counts. For example, the total could be encoded using a static model and then all possible ways enumerated in which it might be broken down into the required number of counts. This was not deemed worthwhile because of the relatively small amount of space that the counts consume.

Once the model has been encoded, the actual identifier sequence is coded with respect to it. Figure 1(d) shows the encoding probabilities in the example. The first symbol has to be number 1 and so the first encoding probability is 1 (10/10). Now the “1” count in the model is decremented, and so the probability of the next symbol being number 1 is 1/9. The encoder must merely encode the fact that this is not the case (probability 8/9) and the decoder can infer that the second symbol is number 2. Coding proceeds along these lines, one model count being decremented for each symbol processed.

Each identifier normally occurs at least three times: once when it is declared, once when it is assigned, and again when it is used; this repetition is what makes the coding method worthwhile. However, strings and numbers are not encoded in this way, because they are expected to involve far less repetition. The stream of strings is simply compressed using PPMC. It may be worth priming the model with the list of identifier names, because identifiers are often mentioned in strings (particularly error messages); however this is likely to give only marginal improvement and was not done because it is rather specific to a particular programmer’s writing style.

Integers and real numbers are compressed quite well by the above-mentioned “numeric” variant of PPMC. For tokens of type “integer” only one numeric sequence is encoded, while two are encoded for tokens of type “real”—one for the part preceding

(a)	Sequence	a, b, c, b, b, c, a, c, b, b																																																																
(b)	Model	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px;">Index</th> <th style="padding: 2px;">Name</th> <th style="padding: 2px;">Count</th> <th colspan="8"></th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">a</td> <td style="padding: 2px;">2</td> <td colspan="8"></td> </tr> <tr> <td style="padding: 2px;">2</td> <td style="padding: 2px;">b</td> <td style="padding: 2px;">5</td> <td colspan="8"></td> </tr> <tr> <td style="padding: 2px;">3</td> <td style="padding: 2px;">c</td> <td style="padding: 2px;">3</td> <td colspan="8"></td> </tr> <tr> <td colspan="2"></td> <td style="padding: 2px;">10</td> <td colspan="8"></td> </tr> </tbody> </table>										Index	Name	Count									1	a	2									2	b	5									3	c	3											10								
Index	Name	Count																																																																
1	a	2																																																																
2	b	5																																																																
3	c	3																																																																
		10																																																																
(c)	Indexes	1, 2, 3, 4, 4, 3, 1, 3, 2, 2																																																																
(d)	Encoding	<table border="0" style="width: 100%;"> <tr> <td style="padding-right: 20px;">“a, b, c”</td> <td style="text-align: right;">PPMC-encoded</td> </tr> <tr> <td style="padding-right: 20px;">2,5,3</td> <td style="text-align: right;">PPMC-encoded</td> </tr> <tr> <td style="padding-right: 20px;"> $\frac{10}{10}$ $\frac{8}{9}$ $\frac{3}{8}$ $\frac{4}{7}$ $\frac{3}{6}$ $\frac{2}{5}$ $\frac{1}{4}$ $\frac{1}{3}$ $\frac{2}{2}$ $\frac{1}{1}$ </td> <td style="text-align: right;">arithmetically encoded</td> </tr> </table>										“a, b, c”	PPMC-encoded	2,5,3	PPMC-encoded	$\frac{10}{10}$ $\frac{8}{9}$ $\frac{3}{8}$ $\frac{4}{7}$ $\frac{3}{6}$ $\frac{2}{5}$ $\frac{1}{4}$ $\frac{1}{3}$ $\frac{2}{2}$ $\frac{1}{1}$	arithmetically encoded																																																	
“a, b, c”	PPMC-encoded																																																																	
2,5,3	PPMC-encoded																																																																	
$\frac{10}{10}$ $\frac{8}{9}$ $\frac{3}{8}$ $\frac{4}{7}$ $\frac{3}{6}$ $\frac{2}{5}$ $\frac{1}{4}$ $\frac{1}{3}$ $\frac{2}{2}$ $\frac{1}{1}$	arithmetically encoded																																																																	

Figure 1 Encoding an identifier stream

the decimal point and the other for the part following it. Numeric values in scientific notation are not handled by our grammar, but could certainly be accommodated if necessary. An alternative to this character-based method of handling numbers is to encode the length of the number in bytes and then its value in binary as a word of that number of bytes. However, numeric constants account for such a small proportion of our test files that this was not deemed worthwhile.

COMMENTS, AND SPACES

Comments were removed by a preprocessor that stripped them out of the program and placed them in a "comment" file with a pointer to their position in the original program. Comments in PASCAL programs are generally long enough to make the overhead of a pointer for each comment worthwhile. The resulting file was compressed using the PPMC method.

In PASCAL, any amount of white space can appear between any pair of adjacent tokens, and so each token produced by the lexical analyzer was viewed as having leading white space—even if this was of zero length. White space tends to occur in short strings of only a few characters, and so the use of pointers is not justified. Moreover, advantage can be taken of the fact that programs are usually indented systematically to reduce the bandwidth needed to represent space. This can be done by representing each string in terms of two-dimensional spatial offsets. Each occurrence of whitespace is viewed as a horizontal and vertical offset. If the vertical offset is 0, that is, this white space string does not include a "newline," the horizontal offset is measured from the end of the previous token—0 for no space, 1 for a single space, etc.). If the vertical offset is 1 or more, the horizontal offset is measured from the last indent.

Figure 2 shows the computation of offsets for a sample program. Part (b) shows the horizontal and vertical offset that follow each token (using a tab size of 5). These are arithmetically encoded in the context of the preceding token—begin, if, identifier, and so on. Modeling is adaptive, which means that the system will tend to "learn" a particular style of indenting.

(a)	Program	begin if a=b then writeln(a); writeln(b); end;																																										
(b)	Offsets	<table border="0"> <thead> <tr> <th style="text-align: left;">vertical, horizontal</th> <th style="text-align: left;">name</th> <th style="text-align: left;">preceding token representation</th> </tr> </thead> <tbody> <tr><td>0, 0</td><td><start></td><td></td></tr> <tr><td>1, 5</td><td>begin</td><td></td></tr> <tr><td>0, 1</td><td>if</td><td></td></tr> <tr><td>0, 0</td><td>identifier</td><td>a</td></tr> <tr><td>0, 0</td><td>=</td><td></td></tr> <tr><td>0, 1</td><td>identifier</td><td>b</td></tr> <tr><td>1, 5</td><td>then</td><td></td></tr> <tr><td>0, 0</td><td>writeln</td><td></td></tr> <tr><td>0, 0</td><td>(</td><td></td></tr> <tr><td>0, 0</td><td>identifier</td><td>a</td></tr> <tr><td>0, 0</td><td>)</td><td></td></tr> <tr><td>1, -5</td><td>;</td><td></td></tr> <tr><td>...</td><td>...</td><td>...</td></tr> </tbody> </table>	vertical, horizontal	name	preceding token representation	0, 0	<start>		1, 5	begin		0, 1	if		0, 0	identifier	a	0, 0	=		0, 1	identifier	b	1, 5	then		0, 0	writeln		0, 0	(0, 0	identifier	a	0, 0)		1, -5	;	
vertical, horizontal	name	preceding token representation																																										
0, 0	<start>																																											
1, 5	begin																																											
0, 1	if																																											
0, 0	identifier	a																																										
0, 0	=																																											
0, 1	identifier	b																																										
1, 5	then																																											
0, 0	writeln																																											
0, 0	(
0, 0	identifier	a																																										
0, 0)																																											
1, -5	;																																											
...																																										

Figure 2 Encoding white space

Each vertical/horizontal offset pair is encoded as a single unit. First, if it has occurred before in the context of the current token it is encoded with its occurrence frequency in that context. If not, an escape symbol is coded and an amalgamated model is used. If it has occurred before in some other context it will have a non-zero count in the amalgamated model, and will be coded using that count. Otherwise a second escape is coded and both offsets in the pair are encoded individually using a fixed model. The probabilities in the model are chosen to approximate Elias's gamma code (Elias, 1975); they are

$$\Pr[n] = \frac{6 / \pi^2}{(n+1)^2} \approx \frac{0.608}{(n+1)^2}$$

to code the number n . The cumulative values of this distribution cannot be obtained analytically, but are easily calculated in advance and stored in a table.

4. Experimental Results

The new compression method was tested on a corpus of twenty PASCAL programs collected from the Internet. All of the programs are quite short: this seems to be typical of files containing PASCAL programs. Table 1 shows the result. The original file size is given, followed by the space occupied by the eight components of the compressed representation: the parse tree, the symbols in the symbol table, the counts in the symbol table, the identifier list, strings, numbers, white space, and comments. These are totaled in the penultimate column, and finally the compression obtained by the PPMC method is given for reference. The average of each column is shown at the bottom, along with the percentage of the total that is accounted for by each component of the compressed representation.

The compression rate of the new method averaged about 1.8 bits per character over the whole corpus. Comparing it with PPMC, very similar results are achieved, with an

Program	Original size	Parse tree	Symbol table symbols	Symbol table counts	Identifier list	Strings	Numbers	White space	Comments	Total	PPMC
1	4137	115	165	37	72	226	22	111	469	1217	1153
2	4235	154	357	41	68	96	54	138	3	911	1112
3	4474	136	178	38	69	318	16	126	489	1370	1329
4	4822	13	271	8	1	3	168	98	190	752	813
5	5237	52	94	24	36	4	8	75	1021	1314	1239
6	5799	94	127	28	40	3	132	105	1059	1588	1600
7	6523	315	123	41	167	218	73	195	181	1313	1137
8	8031	220	267	62	218	3	31	213	736	1750	1862
9	9235	133	483	64	232	3	3	209	1051	2178	2088
10	9392	385	431	74	268	278	202	232	649	2519	2472
11	9402	319	402	75	273	550	40	252	473	2384	2480
12	9543	192	498	61	340	4	313	285	166	1859	2094
13	10314	328	406	69	260	720	106	193	15	2097	2438
14	13596	254	128	53	231	282	85	258	1918	3209	3352
15	15334	707	567	116	623	257	123	388	443	3224	3442
16	17299	805	637	123	726	648	219	412	438	4008	4133
17	18112	478	569	94	377	1841	33	380	1205	4977	5153
18	22652	963	333	93	961	1534	39	491	448	4862	509
19	24957	873	405	106	845	1413	62	433	897	5034	5269
20	28188	898	812	177	1434	719	398	604	646	5688	6134
Mean	11564	372	363	69	362	456	106	260	625	2613	2715
% of total		14.2%	13.9%	2.7%	13.9%	17.5%	4.1%	9.9%	23.9%	100%	

Table 1 Compression results (in bytes)

average improvement of only about 4%. The new method was about 14% better than PPMC on program 12 but about 13.5% worse on program 7.

The largest component of the compressed representation is comments, followed by strings. The new syntax-directed method offers no advantage for these components: they are merely isolated and passed to PPMC. Between them they account for over 40% of the compressed file size. In the uncompressed files, comments and strings account for only about 27% of the file size.

If the compression were permitted to be lossy, preserving only the functionality of the program, it would not be necessary to transmit comments, white space, or the symbol table. Then the compressed size would be halved compared with PPMC, representing a compression rate of 0.94 bits per character. Of course, such programs would be quite unreadable on decompression.

Compression can be improved by priming the models first rather than starting them out from scratch each time. To investigate this, the test corpus was split into two, odd-numbered programs being used for priming and even-numbered ones being used to evaluate the compression achieved. It was found worthwhile to prime all models except those for the parse tree and the identifier list: it seems reasonable that knowledge of these structures for one program does not really help in compressing another.

Table 2 shows the results obtained. Priming improves the overall compression figure by about 10%. However, it naturally improves the compression obtained by PPMC as well, and the net result is that the new method offers about 9% improvement over PPMC.

5. Automatic compressor generation

The method described above and illustrated with a PASCAL program compressor can, with some fairly routine work, be applied to any formally-defined programming language. The question arises as to whether such a compressor could be created automatically from the language description. This would require generating:

- a parser which during parsing creates a parse tree and the relevant token streams;
- a deparser which reconstitutes the source code from the parse tree and token streams;
- a compressor and decompressor for the parse tree and the token streams.

This section sketches the design of such a compressor generator.

Figure 3 shows the definition of a simple “arithmetic expression” language that will

Program	Parse tree	Symbol table <i>symbols</i> <i>counts</i>	Identifier Strings list	Numbers	White space	Comments	Total	PPMC primed	
2	154	276 28	68	69	38	132	1	766	1014
4	13	345 7	1	1	171	223	153	914	1000
6	94	91 14	40	1	133	89	971	1433	1219
8	220	205 38	218	1	22	193	644	1541	1466
10	385	276 54	268	212	189	208	352	1944	2040
12	192	471 50	340	1	311	290	89	1744	2079
14	254	101 34	231	306	83	221	1774	2449	3325
16	805	384 95	726	474	194	419	152	3249	3312
18	963	155 73	961	1231	27	548	338	4296	4510
20	898	610 141	1434	617	388	667	377	5132	5774
Mean	398	291 53	429	291	156	299	485	2346	2574
% of total	17.0%	12.4% 2.3%	18.2%	12.4%	6.6%	12.7%	20.7%	100%	

Table 2 Compression results (in bytes) after priming

be used to illustrate the working of the generator. The language definition is assumed to be available to both compressor and decompressor. It comprises a list of terminal symbols, regular expressions that define the form of each non-constant token and of a comment string, and a grammar. The regular expressions use the notation “[x]” to represent sets, “a+” and “a*” to represent sequences of zero or more and one or more a’s respectively, “.” for any character, and parentheses for grouping. Thus an identifier is a letter followed by any number of letters, digits or underscores, while a number is one or more digits. A comment has the form “(* ... *)”; also any characters following a “%” on a line are comments. It is assumed that all tokens can be separated by arbitrary amounts of white space, with or without comments. The productions in the grammar are numbered for reference.

We first examine the analysis and parsing process, then look at how the information so produced is used to reconstruct the original program, and finally consider how each component is compressed. As a working example, Figure 4 shows a small program and the information that the program compressor generates.

ANALYSIS AND PARSING

Standard compiler-construction tools are used to produce the lexical analyzer and parser. The lexical analyzer is generated automatically from the information in Figure 3(a)–(c), and produces the token stream of Figure 4(b) for the example program. The standard lexical analyzer is modified (automatically) to write each occurrence of the non-constant tokens to a file, one file per token type, with a separator that is chosen automatically after examining the language definition. This creates the streams shown in Figures 4(c) and 4(d).

The parser is generated automatically from the information in Figure 3(d), and produces the parse tree shown in Figure 4(e). It is modified (automatically) to dump each production that is derived into a “parse tree” file. The productions are shown in Figure 4(f), and they can be represented as the list of productions of Figure 4(g). This production list is the representation of the parse tree that is used for compression.

Now that the program has been analyzed, it is represented by the information in Figures 4(c), 4(d), and 4(g).

(a)	Terminals	id, num, +, -, *, /, (,)	
(b)	Tokens	id = [a-z]([a-z0-9_]*) num = [0-9]+	
(c)	Comments	comment = “(*” .* “*)” % .* \n	
(d)	Grammar	expr → expr op expr expr → (expr) expr → - expr expr → id expr → num op → + op → - op → * op → /	expr/1 expr/2 expr/3 expr/4 expr/5 op/1 op/2 op/3 op/4

Figure 3 “Arithmetic expression” language

DEPARSING AND SYNTHESIS

The deparsing process reconstitutes the target program from the information that has been generated from it. With knowledge of the language structure as set out in Figure 3, the list of productions and token streams are used to recreate the original program. It is not difficult to envisage how this information is used.

The first production in the list is identified from the language grammar; it is

$$\text{expr} \rightarrow \text{expr op expr.}$$

The right-hand side of this expression is processed from left to right. On encountering a non-terminal symbol, the next production in the list is read and the process continues recursively. On encountering a terminal that is a constant token, it is interpreted literally as an element of the final program. On encountering a non-constant token, the next element is removed from the appropriate token stream and interpolated into the program at that point.

COMPRESSION

The parse tree compressor encodes the list of productions using adaptive arithmetic coding, just as for the PASCAL grammar earlier. Note that the prefixes of each member

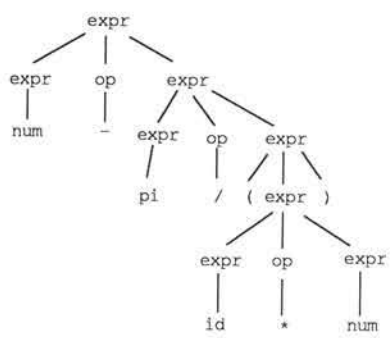
(a)	Example program	15 - pi/(index * 2)	
(b)	Lexical tokens	num - id / (id * num)	
(c)	Id stream	pi, index	
(d)	Num stream	15, 2	
(e)	Parse tree		
(f)	Productions	$\text{expr} \rightarrow \text{expr op expr}$ $\text{expr} \rightarrow \text{num}$ $\text{op} \rightarrow -$ $\text{expr} \rightarrow \text{expr op expr}$ $\text{expr} \rightarrow \text{id}$ $\text{op} \rightarrow /$ $\text{expr} \rightarrow (\text{expr})$ $\text{expr} \rightarrow \text{expr op expr}$ $\text{expr} \rightarrow \text{id}$ $\text{op} \rightarrow *$ $\text{expr} \rightarrow \text{num}$	$\text{expr}/1$ $\text{expr}/5$ $\text{op}/2$ $\text{expr}/1$ $\text{expr}/4$ $\text{op}/4$ $\text{expr}/2$ $\text{expr}/1$ $\text{expr}/4$ $\text{op}/3$ $\text{expr}/5$
(g)	List of productions	$\text{expr}/1, \text{expr}/5, \text{op}/2, \text{expr}/1, \text{expr}/4, \text{op}/4, \text{expr}/2, \text{expr}/1,$ $\text{expr}/4, \text{op}/3, \text{expr}/5$	

Figure 4 Example program

of the list of Figure 4(g) are not encoded. Instead, the string of numbers

1, 5, 2, 1, 4, 4, 2, 1, 4, 3, 5

is sufficient to represent the list of productions, under the assumption that the top-level grammar element (which is "expr" in the example) is the one defined by the first group of productions. Each of these numbers is coded, and decoded, in the context named by the prefix in the list of Figure 4(g).

The elements in each token stream satisfy a regular expression which defines that token type. In principle, this expression could be translated automatically into a finite-state machine and used for compression by encoding the transitions that occur during the parsing of any particular token. In practice, however, most programming languages have very similar components—identifiers, strings, and numbers—and these assume nearly the same form in every language. For example, C and PASCAL tokens are identical except for case sensitivity, the quotation marks used to delimit comments and strings, and the additional number formats permitted in C. In our work on compressing PASCAL programs we have found that there is little appreciable advantage to be gained by using specialized compression methods, and have in fact ended up using PPMC, specialized only to expect the correct alphabet for each token type. The same applies to the comment stream too.

The one area where a specialized model is worthwhile is white space. The strategy described for PASCAL of specifying space incrementally both horizontally and vertically and conditioning it as a pair on the preceding token is applicable generally. Common indentation conventions are very similar from language to language, and this model is appropriate even for quite different languages, such as LISP and PROLOG. While it would be more general to allow white space to be defined as a regular expression in the language specification of Figure 3 and compress it using PPMC like the other tokens, compression is greatly improved by drawing the line here, accepting a fixed definition of white space, and treating it as explained previously.

6. Conclusions

We have described a model for the compression of computer programs that outperforms the best methods for text compression by around 9% on our test files, which is an appreciable improvement. If the compression models are unprimed the improvement is only about 4%.

One area where gains could be achieved by more sophisticated analysis of the target program is the identifier list. Upcoming identifiers can be predicted more accurately using type and scope information. Most programming languages have pre-defined data types and allow the user to define further ones; in both cases the type of identifiers that may occur is restricted by operators, procedure parameter declarations, and so on. Moreover, there are different kinds of identifiers: constant identifiers, type identifiers, variables, procedure and function names; and these are restricted as to where they can occur. Most languages permit the scoping of identifiers and this also restricts the places in which a particular identifier can occur. And finally, advantage could be taken of the fact that variables are usually assigned before they are used. All these improvements involve the semantics of the language, however, and so would be difficult to accommodate in a compressor-generator program that works from nothing more than a formal language description.

Gains are possible in a number of other areas. White space encoding could usefully take into account the following token as well as the preceding one. More effort could be made to "learn" a user's indenting style, or to select it from a library of pre-defined possibilities. Lexical elements that are defined by regular expressions could use a finite-state machine to predict upcoming symbols. Models for compressing strings and comments could be primed with identifier names. However, the gains achieved by such means are likely to suffer from the law of diminishing returns.

More profitable from a practical viewpoint is to investigate the use of compression in a source code management scheme, to prevent the wastage of space caused by proliferation of slightly different versions of programs. This seems to be a productive area for future research.

References

- Barry, M. (1988) "Data compression of structured input." Honours Report, Department of Computer Science, University of Melbourne.
- Bell, T.C., Cleary, J.G. and Witten, I.H. (1990) *Text compression*. Prentice Hall, Englewood Cliffs, NJ.
- Cameron, R.D. (1988) "Source encoding using syntactic information source models." *IEEE Trans Information Theory IT-34*(4): 843–850.
- Cleary, J.G. and Witten, I.H. (1984) "Data compression using adaptive coding and partial string matching." *IEEE Trans Communications COM-32*: 396–402.
- Elias, P. (1975) "Universal codeword sets and representations of the integers." *IEEE Trans Information Theory IT-21*(2): 194–203; March.
- Johnson, S.C. (1975) "YACC—yet another compiler compiler." Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J.
- Katajainen, J., Penttonen, K. and Teuhola, J. (1986) "Syntax-directed compression of program files." *Software—Practice and Experience 16*(3): 269–276.
- Katajainen, J. and Mäkinen, E. (1990) "Tree compression and optimization with applications." *Int J Foundations of Computer Science 1*(4): 425–447.
- Lesk, M.E. (1975) "Lex—a lexical analyser generator." Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J.
- Moffat, A. (1990) "Implementing the PPM data compression scheme." *IEEE Trans Communications COM-38*(11): 1917–1921; November.
- Stone, R.G. (1986) "On the choice of grammar and parser for the compact analytical encoding of programs." *Computer Journal 29*(4): 307–314.