SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 16(3), 269–276 (MARCH 1986)

Syntax-directed Compression of Program Files

JYRKI KATAJAINEN, MARTTI PENTTONEN AND JUKKA TEUHOLA
University of Turku, Department of Computer Science, SF–20500 Turku, Finland

SUMMARY

Parsing can be applied to compress source programs. A suitably encoded parse tree, together with the symbol table, constitutes a very compact representation of the program. The paper reports a Prolog implementation of the method, including automatic, syntax-directed, encoder and decoder generators. The test results show compression gains of 50–60 per cent.

KEY WORDS Text compression Syntax-directed parsing Logic programming

INTRODUCTION

The effectiveness of text compression depends on how much information is available about the structure of the text. The encoding can be based on
(1) single characters (e.g. Huffman code)
(2) pairs, triples or longer blocks of characters (e.g. Rubin's method[1])
(3) overall structure of the text.
As information about the structure of the text increases from (1) to (3), the compression gain, too, is expected to grow. Hence, for example, if the *syntax* of a program file is taken into account, a better compression gain should be achievable than when using compression based on mere characters. Also natural language has a certain structure, but it is so complicated that compression based on that is hardly feasible.

Here we assume that the text to be compressed conforms with a *context-free grammar*. Thus our method is suitable only for a specific subset of text files; source files of programs constitute the main application area. The principle of the method is very simple: given the grammar and a source program, the *parse tree* of the latter is generated. The nodes of the tree are then stored in preorder, encoded as compactly as possible.

Each context-free grammar has a corresponding *Szilard* (or derivation) language.[2] The leftmost Szilard word of a program is the sequence of productions (or production labels) in its leftmost derivation. This sequence is equivalent to the preorder representation of the parse tree. It was observed by Penttonen[3] that every context-free language L can be expressed in the form $h(L')$ where L' is the leftmost Szilard language of a grammar in Greibach normal form generating L and h is a letter-to-letter homomorphism. So, if the grammar is in Greibach normal form, reproduction of the original program simplifies to a homomorphism. In the general case, the program is reconstructed by applying, from left-to-right, the productions in its Szilard word. In our application this means that the reproduction of the original text is a very simple task, basically just replacement of production labels.

0038–0644/86/030269–08$05.00

Received 13 March 1984

GENERAL DESCRIPTION

The steps of the compression method are partly parallel with the phases commonly applied in compilers.[4] Our task is however simpler because we do not have to worry about the semantics of the program, just its syntax. Here the encoding of the parse tree is a counterpart of code generation in compiling.

Figure 1 shows the proposed steps for compression (b) and decompression (c). In order to achieve a general, language-independent system, we have added a preliminary phase (a), which generates programs for scanning, parsing and deparsing. The generators take as input the lexical and syntactical specifications of the language, and are executed only once. Another, but slower, possibility would be to read these specifications each time before the actual (de)compression.

The *lexical analyser* (*scanner*) discards the useless characters, divides the remaining text into *tokens* and classifies them into *syntactic terminals* (keywords, operators, punctuation symbols) and *user terminals* (constants and identifiers). The tokens are defined as regular expressions,[4] and the scanning can be performed by a program operating like a finite automaton. The user terminals are gathered into the *symbol table*.

The parser generates the parse tree, the nodes of which are *indices* of two kinds: labels of grammar productions and pointers to symbol table entries. The tree can be simplified because syntactic terminals are not needed and internal nodes yielding nothing but one non-terminal or one user terminal may be omitted. Linearization of the parse tree means that we list its nodes in preorder. The final encoding of the list elements can be performed using the conventional compression techniques, e.g. Huffman encoding. The result of the compression procedure consists of the encoded list, together with the symbol table, which is also stored as compactly as possible.

The decompression procedure starts with the scanning and decoding the results of the final encoding step of compression, producing the symbol table and the list of indices, in the same form as above. The deparsing of the list re-establishes the parse tree (now with terminal symbols included). The productions are applied from left-to-right, expanding the non-terminals repeatedly. In fact, we can do this in the same manner as *macros* are developed. All we need to know is the number of non-terminals ('arity') and their positions in the right sides of productions. The identifiers are fetched from the symbol table when references to them are encountered. The token list is obtained as the yield of the tree, i.e. the leaves in preorder. Finally, the list must be formatted into the standard 'layout' of the language, using some prettyprinting program.

The currently existing automatic program formatters are often rather restricted, in that their primary purpose is to perform the indentation of lines. What we need is a more sophisticated tool, which also decides about spacing and dividing the text into lines. It should be realized that there are more than one 'correct' way to do the formatting. Different programmers prefer different styles of layout, hence an ideal system would provide several options to choose from. If the formatting requires parsing information, it would be natural to embed the formatter in the decompression procedure (in the 'yield' step), where the parse tree is still available.

Figure 2 presents a simple grammar and some of the data occurring during the compression of a sample program.

PROLOG PROTOTYPE

Warren[5] has demonstrated that the *'logic programming'* language Prolog[6] is suitable for writing compilers. Hence it is not surprising that we found it convenient for our task, too.

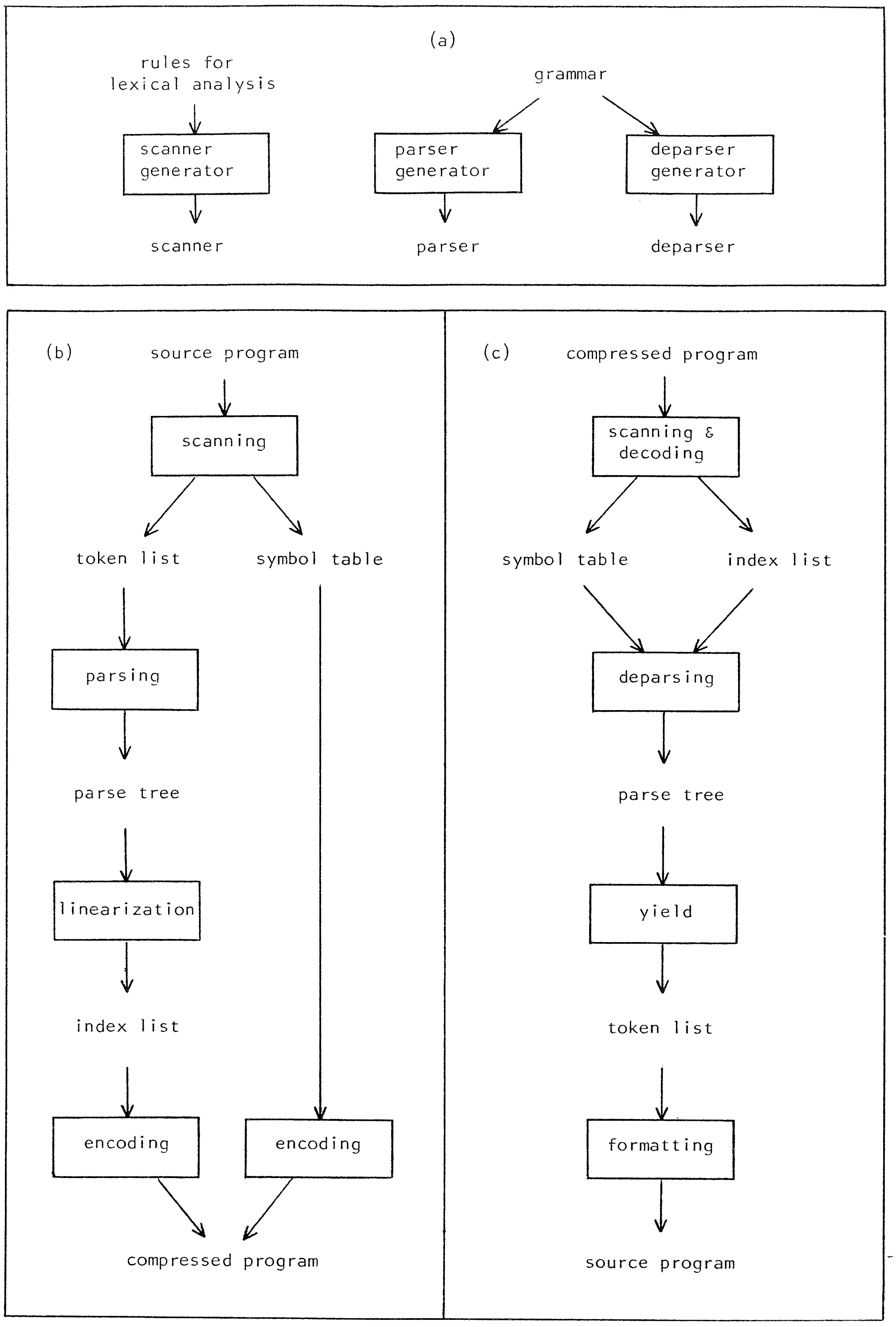
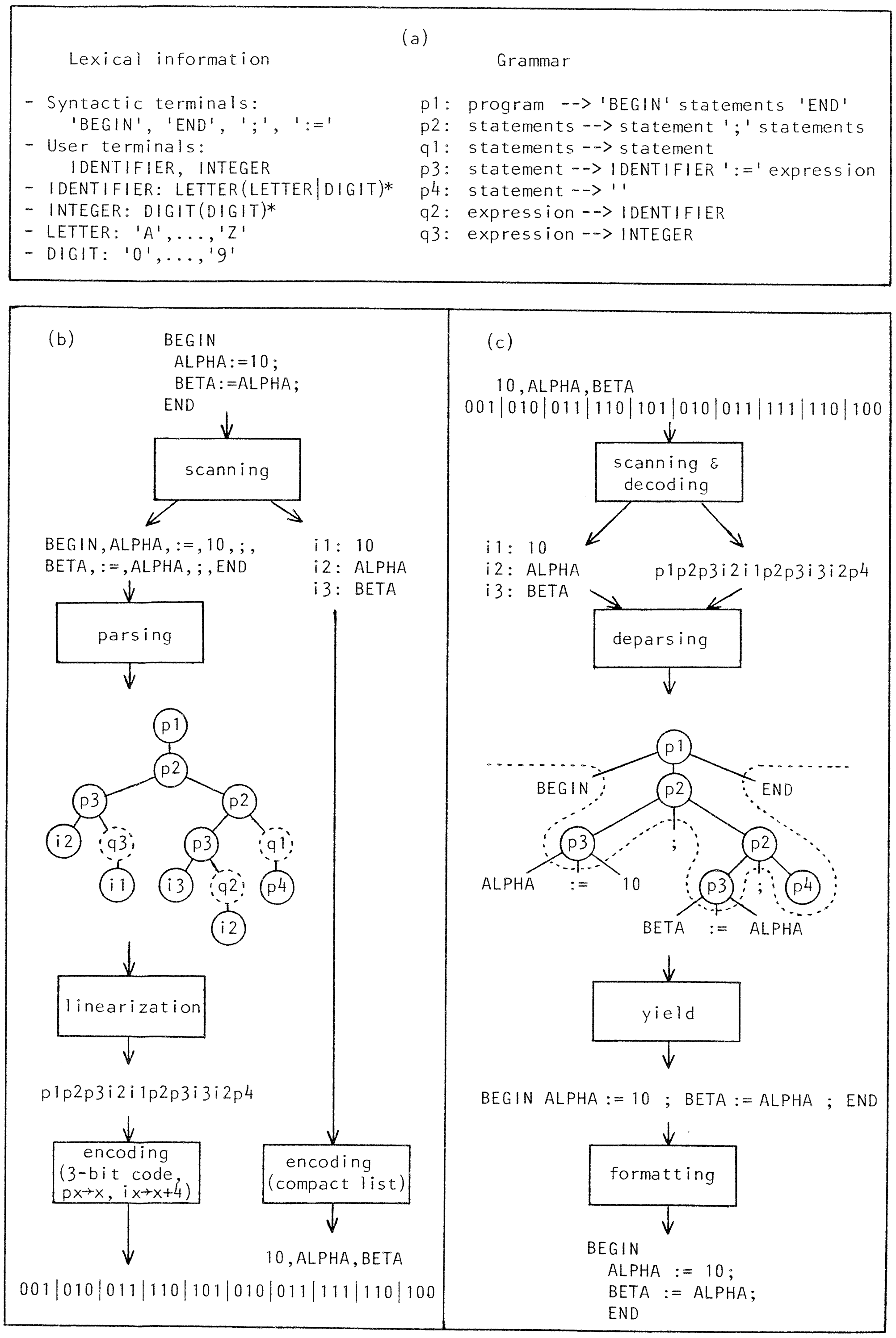


Figure 1. The compression system: (a) preliminary phases for the language; (b) compression; (c) decompression



```
                                   (a)
   Lexical information                        Grammar

- Syntactic terminals:               p1: program --> 'BEGIN' statements 'END'
    'BEGIN', 'END', ';', ':='        p2: statements --> statement ';' statements
- User terminals:                    q1: statements --> statement
    IDENTIFIER, INTEGER              p3: statement --> IDENTIFIER ':=' expression
- IDENTIFIER: LETTER(LETTER|DIGIT)*  p4: statement --> ''
- INTEGER: DIGIT(DIGIT)*             q2: expression --> IDENTIFIER
- LETTER: 'A',...,'Z'                q3: expression --> INTEGER
- DIGIT: '0',...,'9'
```

Figure 2. An example of compression: (a) input for the preliminary phase; (b) steps in a sample compression; (c) corresponding decompression

Our prototype implementation is in accordance with the main lines of the previous section. However, some simplifications were made, partly due to Prolog. The scanner generator was not implemented, instead, we wrote language specific scanners for our test cases. The scanning was performed in two commonly applied steps:[4] prescanning and actual scanning. In the latter step the tokens were recognized by Prolog predicates, which were specifically ordered to ensure correct interpretation.

As the data structure of the symbol table we used the *AVL tree*, the implementation of which was proposed in Reference 7 (see also Reference 8). Here we use the typical trick of Prolog that a single procedure can be used for both searching and inserting, as described by Warren.[5]

The parser generator transforms the grammar productions one-by-one into corresponding Prolog productions, making up the parsing grammar. For the simple language of Figure 2 the parsing productions would become as follows:

```
program(p1(X)) --> ['BEGIN'], statements(X), ['END'].
statements(p2(X,Y)) --> statement(X), [';'], statements(Y).
statements(X) --> statement(X).
statement(p3(X,Y)) --> [identifier(X)], [':='], expression(Y).
statement(p4) --> [].
expression(X) --> [identifier(X)].
expression(X) --> [integer(X)].
```

This grammar, appended to the common *driver* program, constitutes the final language-specific compression program. The deparsing rules are generated similarly, but now they need not be written in the form of productions, simple predicates are sufficient:

```
p1(['BEGIN', Statements, 'END']).
p2([Statement, ';', Statements]).
p3([Identifier, ':=', Expression]).
P4([]).
```

The terms in Prolog productions are equipped with arguments which carry along the intermediate results of parsing, so that the parse tree is generated as a by-product of syntax checking. The use of arguments in Prolog non-terminals bears a close resemblance with so called *attribute grammars*[9], a device developed for the description of the semantics of a programming language. Attributes, when attached to the non-terminals of a context-free grammar, give it a facility of information exchange between different parts of a program, producing a non-context-free effect.

As the prototype uses an extremely simple parsing strategy, non-deterministic *top-down left-to-right* parsing controlled by *backtracking*, there are some restrictions and recommendations in writing the grammar:

(1) Left recursive productions are prohibited, because they would lead to an infinite loop.
(2) For the sake of compression, empty productions should be avoided, because they will occur explicitly in tne result (cf. Figure 2).
(3) If the definition of a non-terminal involves many productions, then, for efficiency, they should be ordered so that first are those where the right side begins with a terminal.
(4) If there are two productions of the form $x \rightarrow y_1 y_2 \ldots y_i \ldots y_n$ and $x \rightarrow y_1 y_2 \ldots y_i$, the former should be placed before the latter in the grammar.

It may be possible to satisfy requirements (1) and (2) automatically. Recommendation (3) can be fulfilled by simple sorting. The purpose of recommendations (3) and (4) is to speed up the parsing when the above-mentioned parsing strategy is used. As for empty productions, there is a trade-off between compression gain and efficiency. For instance in case (4) the most time efficient form of productions would be: $x \rightarrow y_1 y_2 \ldots y_i z$, $z \rightarrow y_{i+1} \ldots y_n$, $z \rightarrow$ *empty*.

The final compression of production and symbol indices was performed using fixed-length binary coding, where the length = $\lceil \log(\#\text{productions} + \#\text{symbols}) \rceil$. The advantage of the Huffman method in the final coding proved to be marginal in our moderate test cases.

EXPERIMENTS

We tested our system by compressing Pascal programs. Pascal's syntax[10] was rewritten in order to gain the best possible efficiency. The syntax consisted of 159 productions, out of which 126 needed a label, i.e. the rest were of the type *non-terminal* → *non-terminal* or *non-terminal* → *user-terminal*, and their labels were dropped out of the parse tree in the reduction step. Note that the reduction has a considerable effect upon the compression gain for two reasons: first, the number of elements to be encoded decreases, and secondly, the number of bits required to encode one element may decrease.

In our first program version we found out that parsing of complex mathematical expressions was a bottleneck. Normally the parsers use additional information about the precedence and associativity of the operators. To solve the inefficiency, we however chose another solution: in our application there is no need to produce a semantically correct parse tree, as long as the original program can be recovered. Hence we applied a left-to-right parsing scheme for expressions, i.e. the operators were considered as if they were right associative and had no order of precedence.

Table I shows some performance figures about the compression of sample programs. The computer used was DEC-2060, and the Prolog compiler was version 3.3 developed by D. Warren, F. Pereira and L. Byrd (1981). For the sake of comparison, all sizes are expressed using the same unit (7-bit ASCII code).

Table I. Test results for Pascal programs

Original program (characters)	Symbol table (characters)	Index list (characters)	Compressed size (characters)	Compression gain (%)	Compression time (s)	Decompression time (s)
388	100	95	195	49·7	2·5	0·9
790	152	159	311	60·6	3·6	1·6
1797	288	451	739	58·9	38·5	4·1
3435	397	1012	1409	59·0	21·6	10·4
4254	438	1091	1529	64·1	29·2	10·9
6265	1360	1351	2711	56·7	146·0	15·0

It can be noted that neither the compression gain nor the time consumed grow smoothly with the size of the source program. Instead, they heavily depend on the programming style, the length of user symbols and the complexity of expressions. It was unfortunate that the growth of storage requirements as well as execution time prevented us from trying out really big programs.

Although scanning in principle takes only linear time, it proved to be rather slow in our prototype system, as noticed also by Warren.[5] Thus it may be wise to use some procedural language, instead of Prolog, for that phase.

No great effort was taken to parse the programs efficiently; the productions were used almost in the form in which they were input, and the non-deterministic parsing was controlled only by the built-in backtracking facility of Prolog. Hence, if a subderivation fails, all work done thereby is lost, which leads to an exponential worst case complexity.

Some improvement in efficiency could be achieved by carefully adding some *cut symbols*, decreasing the degree of non-determinism. Sato and Tamaki have done some research[11] which aims at detecting automatically the deterministic features of Prolog programs and taking advantage of them. A more conventional approach is to use a better parsing algorithm—one of the simplest, the Cocke–Kasami–Younger algorithm,[12] already has a cubic time complexity.

A serious question concerns our paradigm of making the compression system independent of the target programming language. Referring to Kowalski,[13] an algorithm consists of a *logic* part and a *control* part. In our case, the grammar corresponds to the logic of parsing, containing almost no control information except those restrictions and recommendations mentioned in the previous section. A general parser generator which does not use control information typical of the grammar in question, can hardly compete in efficiency with a parser made especially for that grammar.

In contrast to compression times, the decompression times in Table I grow almost linearly, which could also be expected. The times do not include program formatting.

DISCUSSION

The compression gains achieved in our experiments were fairly good, but we want to emphasize that it is not justified to compare the power of our method with others, because all information was not preserved about the source programs. The result of decompression equals the original program only if both are written applying the same formatting principles. Slight differences in the outlook of the result are usually unimportant, but, if not allowed, the scanning phase could be changed so that the token separators are extracted and stored in a special table. References to them, attached to tokens, would be carried along during parsing, and would occur explicitly in the result. Comments could be handled in a similar fashion. Hereby, of course, the compression gain somewhat reduces. However, if the amount of commentary is large, compression methods designed for plain text could be applied.

Although our compression method is based on syntactic analysis and in principle reveals syntactic errors, it is not reasonable to use the compression program for that purpose. We require that the programs to be compressed are syntactically correct. This restriction is not serious, because incorrect programs are not usually stored for long periods.

Syntax checking is involved, at least partly, in three different tasks: compression, formatting and compiling. For saving effort it might be worth while to combine these tasks, so that parsing is done only once. The compressed program can be regarded as an intermediate code, from which we can proceed in two directions: either recover the original program (e.g. for making changes) or complete the compilation process. Hence the source and compiled versions might be dispensed with, thus increasing the total space saving.

A problem for possible future work is to determine the minimal size of the parse tree for a given program and a corresponding grammar. Above we used one compaction technique: the internal nodes with only one child were discarded. However, we feel that the resulting tree is by no means minimal. The form of the grammar has a considerable impact on the size of the parse tree. Another problem is thus, what is the optimal form of a given grammar, with respect to minimizing the parse trees, and how an arbitrary context-free grammar can be automatically transformed into the optimal form.

Note added in proof. Some further improvement in compression can be achieved by the following observation. Two productions need different labels only if they have the same left-hand-side non-terminals. During decompression the leftmost non-terminal is known at every moment. Hence, it is sufficient to know which production for that non-terminal will be used. We will investigate this idea in a future work.

REFERENCES

1. F. Rubin, 'Experiments in text file compression', *Comm. ACM,* **19**, (11), 617–623 (1976).
2. A. Salomaa, *Formal Languages*, Academic Press, 1973.
3. M. Penttonen, 'Szilard languages are log n tape recognizable', *Elektr. Inf. und Kyb.,* **13**, (11), 595–602 (1977).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.
5. D. H. D. Warren, 'Logic programming and compiler writing', *Software—Practice and Experience,* **10**, 97–125 (1980).
6. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.
7. M. van Emden, 'AVL-tree insertion: a benchmark program biased towards Prolog', *Logic Programming Newsletter*, 2 (1981).
8. P. Vasey, 'AVL-tree insertion revisited', *Logic Programming Newsletter,* 3 (1982).
9. F. G. Pagan, *Formal Specification of Programming Languages: A Panoramic Primer,* Prentice-Hall, 1981.
10. K. Jensen and N. Wirth, *Pascal User Manual and Report* (second edition), Springer-Verlag, 1975.
11. T. Sato and H. Tamaki, 'Enumeration of success patterns in logic programs', *10th Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 154, Springer-Verlag, 1983, 640–652.
12. M. A. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, 1978.
13. R. A. Kowalski, 'Algorithm = logic + control', *Comm. ACM,* **22**, (7), 424–436 (1979).