Compiler writing system for attribute grammars*

G. V. Bochmann† and P. Ward

Département d'Informatique, Université de Montréal, Case Postale 6128, Montreal 101, Canada

The paper presents a compiler writing system which is believed to be portable and easily usable. Similar in philosophy to a bottom-up compiler writing system built previously, this system generates compilers for top-down syntax analysis. The system allows the use of regular expressions for the specification of the syntax of the language to be compiled, and the use of inherited and synthesised attributes for the specification of the semantics. The generated compilers are written in PASCAL. The second part of the paper discusses the system in view of certain aspects that are important for the user of a compiler writing system. Among these aspects are discussed the coverage of different problem areas, such as lexical and syntactic analysis, specification of semantics, error treatment, etc. the simplicity and flexibility of the system's use, and the conciseness and readability of the compiler specification language. The portability of the system is obtained by using PASCAL as the implementation language, and as language for the generated compilers.

(Received September 1975; Revised December 1976)

1. Introduction

Compiler writing systems have been built for more than ten years. Unfortunately, many of these systems are merely aids for compiler writing or toys which are not practical for the development of useful compilers for medium size languages. They place on the user's shoulder most of the burden of actually programming the compiler he wants. We think that a compiler writing system should be designed such that it is most valuable to compiler writers, who are the users of the system. We therefore propose the following points as aspects of evaluation, and as development objectives for any future compiler writing system:

1. *The usability of the system*
1.1. *Broad coverage of problem areas*
Different problem areas, such as lexical analysis, syntax analysis, specification of semantics, optimization, code generation, symbol table management, error treatment, etc. are important for the compiler construction. The system should give tools for as many of these areas as possible.

1.2. *Simple usage, and flexibility*
It should be easy for a potential user to learn how to use the system. It should be easy, by means of options provided, to adapt the system to particular user requirements.

1.3. *Efficiency*
The generation of a compiler should be efficient and the generated compiler should be efficient.

2. *The compiler specification language*
The input language to the compiler writing system is in fact a language for the specification of the compiler to be generated, and therefore of the language to be compiled.

2.1. *Formal definition*
The input language should allow for a formal definition of the language and its compiler. We note that, as far as the syntax is concerned, BNF is a well accepted specification language. For the semantics, however, several specification methods have been proposed (see for example Marcotty *et al*, 1976) and it is not clear which one is best suited as an input language for a compiler writing system.

2.2. *Readability*
The input language should have a simple structure, use a readable representation and be powerful enough, so that the specification of a compiler to be generated is easy to write and understand.

3. *The system's installation*
3.1 *Portability*
It should be easy to install the system on different computer systems.

3.2. *Comprehensible system structure*
It should be easy to understand the functioning of the system, and to modify it if necessary.

A '(truly) usable and portable compiler writing system' was built at the University of Montreal, and has been described in respect to the above points by Lecarme and Bochmann (1974). They also give a comparison of the system with some other compiler writing systems. In this paper we present another compiler writing system which was developed subsequently. Its development has been strongly influenced by the former system, but it uses a top-down LL(1) syntax analysis in contrast to the former system, which uses a bottom-up analysis based on weak precedence.

Compiler writing systems have been built which give the user the choice between different parsing methods to be incorporated in the generated compiler (see for example Wilhelm *et al*, 1976). We have not taken this approach because our systems generate one-pass compilers, and in this case the semantic processing by the compiler is closely coupled with the syntax analysis. The use of a top-down syntax analysis facilitates the use of inherited semantic attributes, and the use of several semantic actions per production rule. In the case of bottom-up parsing the evaluation of inherited attributes is more complicated (Crowe, 1972 and Watt, 1974). Also the error treatment is closely related to the parsing method. Therefore, the new compiler writing system for top-down syntax analysis, although similar in philosophy, is practically independent from the system for bottom-up analysis (only the lexical scanner generator is the same in both systems).

In Section 2 of this paper we present the new compiler writing system, explain how the regular expressions of the input language are handled by the LL(1) syntax analysis, and discuss

*This work was supported by an operating grant and a scholarship grant from the National Research Council of Canada.
†Now at: Département de Mathématiques, Ecole Polytechnique Fédérale, Lausanne, Switzerland.

how the concept of attribute grammars is incorporated in the compiler writing system. More details about the system, or some aspects of it, can be found elsewhere (Ward, 1975; Lecarme, 1973; Stasyna, 1977; and Bochmann, 1975 and 1976). In Section 3, we discuss some aspects of the system in the light of the usability criteria listed above. The reader is also referred to a similar discussion of the system for bottom-up syntax analysis by Lecarme and Bochmann (1974). Most parts of the discussion apply for the top-down system as well.

2. The compiler writing system

We present in this section the compiler writing system built at the University of Montreal and discuss the formalism underlying its operation. The system accepts as input the integrated description of the language to be compiled, together with some options which determine the system's actions. The integrated description consists of a set of production rules specifying the syntax and semantics of the language, and a set of supplementary type, variable, procedure and function declarations which are used by the semantic actions. Each production specifies the syntax of a non-terminal in terms of a regular expression (an extension of BNF) and the semantics in terms of semantic attributes and evaluation rules, also called semantic actions, which are written in the programming language PASCAL (Jensen and Wirth, 1974). The system produces as output a complete compiler in the form of a PASCAL program. The user also receives full diagnostics and informative data according to the options chosen.

The system is composed of several program modules, all written in PASCAL which are executed sequentially and which pass the necessary information from one to the next by means of temporary files. The most important module is the first one. It reads the integrated description of the language, extracts the list of terminal symbols for a scanner generator module, verifies the syntax for the possibility of top-down LL(1) analysis and generates a set of recursive procedures which represent the main part of the generated compiler. The system uses the same scanner generator as the compiler writing system described by Lecarme and Bochmann (1974).

A generated compiler performs a deterministic, one-symbol-lookahead, top-down syntax analysis of the program text, based on the LL(1) conditions the grammar has to satisfy. The possible productions for each nonterminal of the grammar are specified by a regular expression. A regular expression α is built from terminal and nonterminal symbols using the formation rules (1) *concatenation:* $\alpha_1 \alpha_2 \ldots \alpha_n$, the only formation rule which applies in the case of grammars in BNF, (2) *alternative choice:* $\alpha_1 + \alpha_2 + \ldots + \alpha_m$, and (3) *repetition:* $(\alpha)^*$ which stands for any number of repetitions of the regular expression α, including none. The system also accepts the formation rules (4) *sequence:* $(\alpha)^+$, a repetition of at least one, and (5) *list:* (α *separator terminal*) which is a sequence except that the occurrences

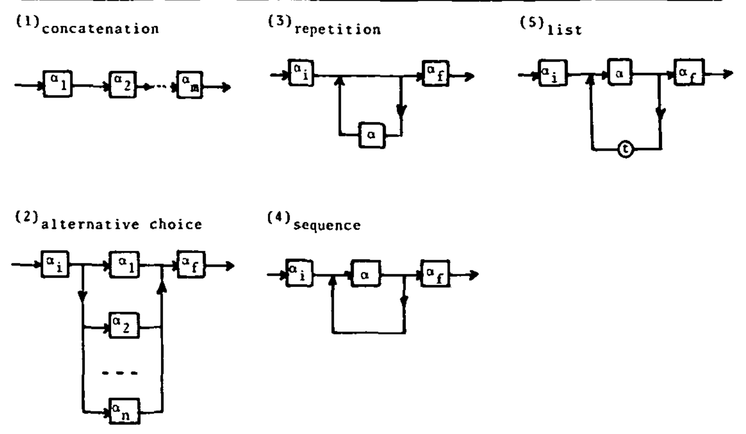

Fig. 1 The corresponding transition diagrams

of α are separated by a terminal symbol.

The LL(1) conditions for grammars in BNF are discussed in the literature (see for example Knuth, 1971; Aho and Ullman, 1972).

In the case of a grammar which contains regular expressions as right sides, additional conditions must hold for each regular subexpression within the right side of a production rule. These conditions can be determined by considering a transformation of the grammar into an equivalent grammar in BNF. This transformation, specified in **Table 1**, introduces for each regular subexpression a new non-terminal, called Z, and the LL(1) condition for the subexpression is simply the LL(1) condition of this new non-terminal in the new BNF grammar.

The main part of the generated compilers are the recursive procedures which perform the syntax analysis and execute the necessary semantic actions. The system generates one such procedure for each non-terminal of the given grammar. The flow of control within a procedure reflects the structure of the regular expression which is the right side of the corresponding production rule. **Fig. 1** shows the transition diagrams that correspond to the different types of subexpressions. More details are given by Ward (1975). As an example, we consider the production rule

$$\langle\text{expr}\rangle \rightarrow \langle\text{term}\rangle\ ((\oplus + \ominus)\ \langle\text{term}\rangle)^*$$

which specifies an expression to be a sequence of terms separated by *plus* or *minus* operators. The system would generate a recursive procedure for the non-terminal ⟨expr⟩ which corresponds to the following transition diagram:

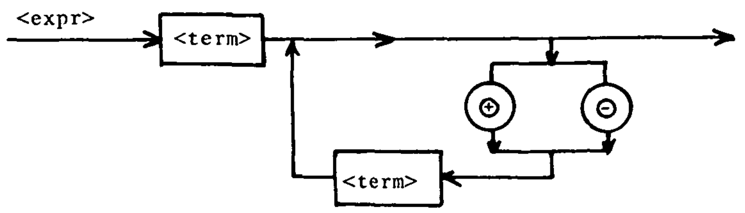

Table 1 Transformation of regular subexpressions into equivalent BNF rules

type of subexpression	*original right side*	*new right side*	*additional production rules*	*corresponding transition diagram of Fig. 1*
concatenation	$\alpha_1\ \alpha_2 \ldots \alpha_m$	no change		(1)
alternative choice	$\alpha_i\ (\alpha_1 + \alpha_2 + \ldots + \alpha_n)\ \alpha_f$	$\alpha_i\ Z\ \alpha_f$	$Z \rightarrow \alpha_1$	(2)
			$Z \rightarrow \alpha_2$	
			...	
			$Z \rightarrow \alpha_n$	
repetition	$\alpha_i(\alpha)^*\ \alpha_f$	$\alpha_i\ Z\ \alpha_f$	$Z \rightarrow e$	(3)
			$Z \rightarrow \alpha\ Z$	
sequence	$\alpha_i(\alpha)^+\ \alpha_f$	$\alpha_i\ Z\ \alpha_f$	$Z \rightarrow \alpha(\alpha)^*$	(4)
list	α_i (α *separator* t) α_f	$\alpha_i\ Z\ \alpha_f$	$Z \rightarrow \alpha(t\ \alpha)^*$	(5)

The semantics of the language to be compiled is specified by attributes, associated with the syntactic symbols of the grammars, and semantic actions, written in PASCAL, which are included in the production rules of the grammar and make use of the supplementary declarations provided by the user. The concept of attribute grammars is a well known method for the specification of semantics (Knuth, 1968; Koster, 1971) and has been adapted by Bochmann (1975) for the case of grammars with regular expressions. The attribute evaluation mechanism which is realised by a compiler generated by the system implements these ideas in terms of a single pass from left to right over the derivation tree. The generated compiler is a one-pass compiler and performs the syntax analysis, with an implicit construction of the derivation tree, together with the attribute evaluation, as specified by the semantic actions, in the same pass over the program text. In the generated compiler, attributes of a non-terminal are represented as parameters of the corresponding recursive procedure. Inherited attributes, which specify the context in which a non-terminal is found, are represented by value parameters (in the sense of PASCAL), whereas synthesised attributes, which specify semantic information derived from the subtree of the non-terminal, are represented by variable parameters (in the sense of PASCAL). Local attributes are represented as local variables and the semantic actions of the production are simply incorporated in the procedure body at the appropriate places.

Since the syntactic analysis and the semantic evaluation are performed during the same pass over the program text, one can use semantic information for solving an ambiguity of syntax analysis in cases where the LL(1) conditions are not satisfied. Although we believe that this practice should be avoided, the system provides means for handling such cases.

The automatic provision by a compiler writing system of the handling of syntactic errors (i.e. proper detection, significant message and safe recovery) is a complicated subject. However, in the case of LL(1) parsing, some relatively simple schemes for error recovery are compared by Stasyna (1977) and seem to give satisfactory results. The system provides for error handling according to such a scheme.

3. From the user's viewpoint . . .

The system we have described has been used for research projects and a compiler writing course, but it is relatively new, and little experience has been gained in using it. Rather than trying to summarise this existing experience, we discuss in this section some aspects of the system which we believe particularly important for the potential user. In the following, the different evaluation aspects listed in the Introduction are discussed one after the other.

Broad coverage of problem areas

The system tries to offer aids to the user in all aspects of compiler writing, and not only in those which are easiest to formalise.

For the lexical analysis the system uses the scanner generator already used in the bottom-up system (Lecarme, 1973). Adopting a pragmatic approach, the system produces scanners similar to handmade ones, and provides for scanning of most programming languages, except languages like FORTRAN or PL/1, in which the absence of reserved or delimited keywords makes it impossible to separate scanning from parsing and semantic analysis.

For syntax analysis the system uses the LL(1) parsing method extended to regular expressions, as explained in Section 2. Although more restrictive than certain bottom-up parsing methods this is, we believe, a reasonable approach because of the simplicity of the method. In fact, Hoare (1973) suggests to use only precedence or top-down grammars for the design of new languages, since a programmer can easily understand programming languages with only a simple type of syntax. We note that the system automatically includes a mechanism in the generated compilers for handling syntactic program errors.

For the semantic processing the use of semantic attributes is very valuable. They simplify the specification of the semantic actions, while being extremely simple to understand and to use. The use of PASCAL for the specification of semantic actions, because of the intrinsic quality of the language, provides for easily attained readability and efficiency of the semantic actions. In addition a set of utility procedures gives powerful yet minimal tools for handling common situations, and an intermediate language may be used for code generation.

Similarity with the bottom-up system

We do not discuss here those aspects of the system that are the same as for the bottom-up system (Lecarme and Bochmann, 1974). In particular, we believe that the system is simple to use and flexible. The fact that it is written in standard PASCAL and uses no system dependent features makes the system easily portable to any computer where PASCAL is available. The use of PASCAL and structured programming methods increases the comprehensibility of the system's structure and makes it easy to read. The same applies for the generated compilers.

Readability of the input language

An integrated description of the syntax and semantics of the language to be compiled serves as input to the compiler writing system. Grouping all parts of the language specification into one integrated description gives much more readability than the customary approach, which separates scanning, parsing, semantic analysis, code generation and error analysis into different description parts, with very error-prone interfaces.

The integrated description of a language is structured according to its syntax; the production rules for each non-terminal of the language form a certain unit of description. Similarly as the non-terminal symbols define the syntactic interface between these different units of description, the semantic attributes of the symbols define the interface between these units as far as the semantic actions are concerned. The use of regular expressions and several semantic actions, in the right sides of production rules, makes it possible to combine into a single production rule parts of the language description that, in the case of a description in BNF, would have necessitated several production rules. Therefore, the units of description can be constructed larger than in the case of BNF. This facilitates a natural structuring of the language specification. Local attributes, as described by Bochmann (1975), can be used like variables for handling the semantics of repeated subexpressions, or for making some semantic information available to or from several alternative subexpressions.

Another advantage of regular expressions, in the case of top-down syntax analysis, is that they can be used to specify production rules which would otherwise be specified with left recursion, which is not allowed in top-down parsing.

As an example we consider again the production rule for the non-terminal ⟨expr⟩ given in Section 2. **Table 2** (*a*) contains the integrated description of the production rule for this non-terminal specifying, in addition to the syntax, a particular semantics. The example contains the synthesised attributes *type* and *location* which indicate the type of the expression or a term, i.e. the value *integer* or *real*, and the location where the value of the expression or a term is stored during the execution phase of the program. The local attributes *interm-type* and *interm-location* take on corresponding intermediate values. The local attribute *operation* of the subexpression ($\oplus + \ominus$) indicates the arithmetic operation to be performed, i.e. the value *plus* or *minus*. The function *generate* has the side effect of generating the appropriate code for executing the operation and furnishes the location where the result of this operation will be stored. The definition of this function must be given by the compiler writer in the supplementary declarations.

The integrated description (*a*) of Table 2 conforms with the concept of attributes for regular expressions as described by Bochmann (1975). The description (*b*) is an equivalent but optimised version of it. This second version makes use of the fact that the synthesised attributes are represented, in the generated procedure of the compiler, by variable parameters which can be assigned successively different values during the parsing from left to right over the program text.

We hope that the input language of the system is flexible enough so that very few changes have to be done by the compiler writer to the 'description grammar', which was first written by the language designer to obtain the 'compilation grammar' which is accepted by the system. The fewer the necessary changes the less are the chances of human errors.

An advantage of LL(1) parsing is its relative simplicity. This is reflected in the fact that the structure of the generated compiler is closely related to the syntax of the language to be compiled (see also Section 2). For example, the production rule (*b*) of Table 2 gives rise to the PASCAL procedure (*c*) of the table, which is part of the generated compiler. The fact that this procedure is similar to the production rule given as input to the system facilitates the understanding and debugging of the semantic parts of the generated compiler.

Efficiency

The cost of using a compiler writing system is on the one hand related to the effort necessary for installing the system and understanding its use, and on the other hand to the computer time and space necessary for generating a compiler. The time and space requirements of the system on the CDC Cyber 74 computer are as follows. The system needs less than 13,000 words, plus about 5,000 for treating a large grammar, for a language such as ALGOL 60. The complete generation of a medium size compiler takes about 10 seconds central processor time. These values are similar to those found for the bottom-up system. The generated compilers are comparable, in size and performance, to one which would be produced by hand, and may easily be improved and modified, since they are written in PASCAL. We found a space and time performance better than for the compilers generated by the bottom-up system.

Availability

Anyone interested in obtaining a copy of the system should contact the authors.

4. Conclusions

We have presented a compiler writing system for top-down syntax analysis which was designed following the principles outlined in the Introduction. These principles are the same as those used for the design of the bottom-up compiler writing system of Lecarme and Bochmann (1974). The different parsing methods and the introduction of regular expressions, in the case of the top-down system, for the integrated description of the language to be compiled imply certain differences between the two systems which are discussed in this paper. In general it was found that the choice of the LL(1) top-down parsing method simplified many aspects of the compiler construction, such as semantic processing, syntactic error handling, obtaining a readable integrated description of the language to be compiled, etc. For a comparison of the parsing aspects of top-down and bottom-up syntax analysis methods we refer to the literature (see for example Aho and Ullman, 1972; Griffiths and Petrick, 1969).

Table 2

(*a*) *The integrated description of one production rule*

```
⟨expr⟩ ↑location0 : typloc ↑type0 : typtype
        = local intermlocation : typloc ; intermtype : typtype
        ⟨term⟩ ↑location1 ↑type1
        : intermlocation := location1 ; intermtype := type1 $
        *[ [local operation : typop
                "+" : operation := plus $
              ∨ "−" : operation := minus $]
           ⟨term⟩ ↑location2 ↑type2
           : intermlocation := generate (operation, type1,
                                         type2, location1,
                                         location2) ;
             if type2 = real then intermtype := real $
        ]*
        : location0 := intermlocation ; type0 := intermtype $
$
```

(*b*) *An optimised version of the integrated description of* (*a*)

```
⟨expr⟩ ↑location : typloc ↑typ : typtype
     = ⟨term⟩ ↑location ↑typ
       *[      [local operation : typop
                 "+" : operation := plus $
                 "−" : operation := minus $]
               ⟨term⟩ ↑location2 ↑typ2
               : location := generate (operation, typ,
                                       typ2, location, location2) ;
                 if typ2 = real then typ := real $
       ]*
$
```

(*c*) *The corresponding generated procedure*

```
procedure proc3 (var location : typloc ; var typ : typtype) ;
          (*⟨expr⟩*)
var operation : typop ; location2 : typloc ; typ2 : typtype ;
begin
proc4 (location, typ) ;
if not fenêtredans (follow-of-⟨expr⟩)
then repeat
  if fenêtredans ([1])
  then begin if fenêtre = 1 then lexical else pastrouvé (1, first-
         of-⟨term⟩);
         operation := plus
        end
  else if fenêtredans ([2])
       then begin if fenêtre = 2 then lexical else pastrouvé
            (2, first-of-⟨term⟩);
            operation := minus
            end
       else nondans ([1, 2], first-of-⟨term⟩) ;

  proc4 (location2, typ2) ;
  location := generate (operation, typ, typ2, location,
              location2);
  if typ2 = real then typ := real ;
  until fenêtredans (follow-of-⟨expr⟩)
end ;
```

The procedure uses the following internal representation of the syntactic symbols in terms of integers :

terminal symbols :	"+"	1
	"−"	2
nonterminal symbols :	⟨expr⟩	3
	⟨term⟩	4

The procedure uses the following standard compiler procedures :

fenêtredans (S : set-of-terminal) : boolean

verifies if the value of the variable *fenêtre*, i.e. the window, is contained in the set given as parameter.

lexical

is the scanner which places into the window the internal representation of the next syntactic symbol of the program text.

pastrouvé (Sym : terminal ; recovery : set-of-terminal),

i.e. 'terminal symbol not found', or

nondans (Sym, recovery : set-of-terminal),

i.e. 'window not contained in . . .', are called in the case of syntax errors. They print a message indicating that the window is not equal to, or respectively contained in the first parameter. The second parameter is used for the error recovery, which consists of possibly skipping text until the syntactic symbol of the window is an element of the recovery set.

References

AHO, A. V., and ULLMAN, J. D. (1972). *The theory of parsing, translation, and compiling*, Prentice-Hall.

BOCHMANN, G. V. (1975). Semantic attributes for grammars with regular expressions, Publication # 195, Département d'Informatique, Université de Montréal.

BOCHMANN, G. V. (1976). Semantic evaluation from left to right, *CACM*, Vol. 19, pp. 55-62.

CROWE, D. (1972). Generating parsers for affix grammars, *CACM*, Vol. 15, pp. 728-734.

GRIFFITHS, T. V., and PETRICK, S. R. (1969). Top-down versus bottom-up analysis, *Inform. Processing 68*, North Holland Publ. Co.

HOARE, C. A. R. (1973). Hints on programming language design, *ACM Symposium on principles of programming languages*, Boston.

JENSEN, K., and WIRTH, N. (1974). *PASCAL User Manual and Report*, Springer-Verlag, Berlin.

KNUTH, D. E. (1968). Semantics of context-free languages, *Math. Systems Th.*, Vol. 2, p. 127, and Vol. 5, p. 95 (1971).

KNUTH, D. E. (1971). Top-down syntax analysis, *Acta Informatica*, Vol. 1, pp. 79-110.

KOSTER, C. H. A. (1971). Affix grammars, in *Algol 68 Implementation*, North Holland Publishing Co., Amsterdam.

LECARME, O. (1973). Un générateur d'analyseurs lexicaux, Document de travail # 40, Département d'Informatique, Université de Montréal (in French).

LECARME, O., and BOCHMANN, G. V. (1974). A (truly) usable and portable compiler writing system, *Proceedings IFIP Congress 1974*, pp. 218-221.

MARCOTTY, M., LEDGARD, H. F., and BOCHMANN, G. V. (1976). A Sampler of Formal Definitions, *Computing Surveys*, Vol. 8, pp. 191-276.

STASYNA, J. (1977). Error recovery in LL(1) syntax analysis, Thesis in preparation, McGill University, Montreal.

WARD, P. (1975). Un système d'écriture de compilateurs à analyse syntaxique descendante, Manuel d'utilisation, Document de travail # 55, Département d'Informatique, Université de Montréal (in French).

WATT, D. A. (1974). LR Parsing of affix grammars, Report No. 7, Computing Science Department, University of Glasgow.

WILHELM, R. *et al.* (1976). Design evaluation of the compiler generating system MUG1, in *Second International Conference on Software Engineering*, San Francisco.

Book review

Computational Analysis with the HP-25 Pocket Calculator, by P. Henrici, 1977; 280 pages. (*John Wiley*, £7·75)

First, what is the HP-25? It is a pocket calculator with approximately 25 function keys, eight registers, is programmable to 49 steps, costs around £100 (mid-1977) and comes with a comprehensive manual containing handy dandy programs. Model HP-25C, with a continuous memory, can be switched off and on yet not forget anything.

Second, what about the author? He is Professor of Mathematics at the Eidgenossiche Technische Hochschule, Zurich and the author of several well known books on numerical analysis.

Third, what is the book all about? It contains 35 program descriptions with their purpose, method, flow chart, storage and program, operating instructions, together with examples and timing. Although the programs are machine and language dependent, with thought they can be adapted on any similar or more powerful computer. They are grouped into number theory, iteration, polynomials, power series, numerical integration and special functions (e.g. Gamma, Bessel). The library of programs covers well the gamut of the numerical aspects of a university degree course in mathematics except for statistics, which are covered in the complementary *Scientific Analysis on the Pocket Calculator* by J. M. Smith. There is a curious deficiency in omitting the Monte Carlo technique, because in my opinion it is the most practical method to perform integration; however this method certainly works best on fast and big computers. This last statement is true for everything described, however the tremendous feature of this book is that technology has at last (or very soon will) produced for us a cheap, portable microcomputer so that students can quickly learn for themselves relative accuracy and timing of calculations. Indeed this area of mathematics can be made very exciting with the ideas presented in this book.

I. R. WILLIAMS (London)

Multi-Coordinate Data Presentation, by V. Z. Priel, 1977; 150 pages. (*Business Books*, £12·00)

The object of this book is to provide a technique for converting data into information. Mr Priel defines the difference between the two by stating

'Data can be regarded as building blocks, as part of a jigsaw puzzle or as daubs of paint on a canvas. The various components only make sense—convey a message—if the arrangement is right, i.e. if it forms a complete whole.'

The book is divided into four parts. The first part provides the argument that what managers are receiving is data when it ought to be information. The second part defines the types of and relationships between data and the uses to which it is put. The author states seven postulates which he uses to support the use of the MCDP technique. In part three he defines the coordinates against which data must be viewed to ensure meaning. The fourth and final part introduces MCDP proper and by means of worked examples shows the technique in use. There are three further examples in the appendix which compare the conventional layout with the equivalent MCDP layout.

There are a number of unfortunate typographical errors, which in at least two cases reverse the meaning of the sentences in which they appear. The values in table 4 in part 2 are incorrect, but the correct values can be easily established. The criticism must also be made that the worked examples, which should be the easiest parts of the book to follow, are in fact the most difficult. This should not deter anyone who is interested in providing management with information as opposed to data, it just makes these parts of the book take longer to understand. The technique is designed to reduce the time managers spend studying large volumes of paper, if this can be achieved it is worth attempting.

D. D. BLACK (Crawley)