A language for algorithms

Richard H. Stark

New Mexico State University, Las Cruces, New Mexico 88001, USA

A syntax for a language to express algorithms in a way which facilitates proofs of equivalence between algorithms as well as translation into computer programs is given and briefly justified.

(Received November 1969)

The cases in which it has been possible to prove the correctness of a computer program or the equivalence of distinct computer programs written to solve the same problem are few. A stumbling block is the potential complexity of program structure when all the capabilities of an algorithmic language like ALGOL, FORTRAN or PL/1 are employed. Yet, in fact, these complexities may come about because of the flexibility of the language and not be inherent in the problem. These observations suggest that it might be fruitful to design a limited language for algorithms. Such a language would need to be mechanically translatable into efficient machine programs. It should lead to algorithms which are structurally simple enough that proofs of equivalence are facilitated. Finally, it should have the necessary richness of expression to be well adapted to some broad class of problems. In this paper, we propose a language for the expression of algorithms to solve mathematical problems to meet these objectives.

The material contained in this paper is a portion of an unpublished report by the author (1968). Shortly thereafter, a letter of Edsger Dijkstra (1968) presented quite similar ideas in the form of a criticism of features of existing languages with some suggestions for improvement. One may look upon the language presented herein as a specific proposal which conforms to his suggestions.

General characteristics of the language

1. There is an alphabet of characters including decimal digits, letters and special symbols from which the statements which make up an algorithm are composed.

2. An algorithm is expressed as a sequence of statements for which the normal execution path is from one statement to its successor in this sequence. A feature of the language is the absence of jump statements, their place being taken by a looping statement and a choice between alternate actions as elaborated in 8.

3. Every operand is itself a numeric value or is the name by which such a value is identified and can be obtained. We assume that values can be recorded and retrieved precisely so that no round-off error occurs.

4. Statements which constitute an algorithm are assignment, delimiter, and control statements. Each type is described just below. Syntactical descriptions are given in Fig. 1.

5. An assignment statement is of the form

$$v := \xi$$

where v represents a variable (identifier followed by a possibly empty argument list) and ξ represents an expression which may be arithmetic or Boolean. The value obtained by substituting current values assigned to variables in the expression represented by ξ is assigned to the variable represented by v. Should that variable already have an assigned value, it is replaced.

6. We distinguish segments of the algorithm which we call actions. Intuitively, an action is a sequence of statements which has exactly one entrance which is its first statement and has only one exit which is from its last statement to the next in sequence. An algorithm is written as an action.

A labelled action has delimiter statements for its first and last statements. The first provides the label followed by a (possibly empty) list of arguments which are names for which parameters or parameter values are to be substituted prior to execution. The last statement serves only as a marker for the end of the action.

For convenience, we provide a standard empty action with label NIL. It is equivalent to the action:

```
BEGIN NIL;
END   NIL;.
```

7. Any action can be invoked by having its first statement be the next to be executed. Only labelled actions can be invoked by other statements. These serve much as the FUNCTION and SUBROUTINE subprograms which would appear in a FORTRAN program and are invoked in much the same way. The combination of action identifier and arguments used in such invocation is called an action initiator. An action initiator may be only an identifier (the label of the action) or it may be an identifier followed by a parenthesised list of parameters. A parameter may be an expression, in which case its value is assigned to the corresponding argument prior to execution. Otherwise it is a variable preceded by # in which case the variable is to be substituted for every occurrence in the action of its corresponding argument prior to execution This substitution of value or name is done for each position common to both lists.

If the action initiator appears as an operand, then its role is just that of a FORTRAN function and—as for FORTRAN functions—its value will be that assigned in the definition of the action to the identifier portion of the action initiator.

8. An action initiator may appear as an operand in an expression as indicated above; its only other place is in a control statement. There are two types of control statement, each of them conditional, called branching and looping respectively. A branching statement invokes one of two alternate actions, the choice based on the truth or falsity of its Boolean expression. A looping statement invokes an action for a succession of values of its looping index,

*This work was performed at Washington State University and was sponsored by the National Science Foundation under Project GP-7476.

```
<Boolean value>          ::= TRUE | FALSE
<digit>                  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter>                 ::= A | B | C | ... | Z | < | ≦ | = | ≠ | ≧ | >
<relational op>          ::= < | ≦ | = | ≠ | ≧ | >
<empty>                  ::=
<identifier>             ::= <letter> | <identifier>
                             {<letter> | <digit>}
<unsigned integer>       ::= <digit> | <unsigned integer>
                             <digit>
<integer>                ::= <unsigned integer> |
                             {+ | −}<unsigned integer>
<decimal fraction>       ::= .<unsigned integer>
<decimal number>         ::= <unsigned integer> |
                             <decimal fraction> |
                             <unsigned integer>
                             <decimal fraction>
<exponent part>          ::= ₁₀<integer>
<unsigned number>        ::= <decimal number> |
                             <exponent part> |
                             <decimal number>
                             <exponent part>
<primary>                ::= <unsigned number> | <variable> |
                             <initiator> | (<arithex>)
<factor>                 ::= <primary> | <factor> ↑ <primary>
<term>                   ::= <factor> | <term> {* | / | ÷}
                             <factor>
<arithex>                ::= <term> | {+ | −}<term> |
                             <arithex> {+ | −}<term>
<index list>             ::= <arithex> |
                             <index list>,<arithex>
<indexed variable>       ::= <identifier> [<index list>]
<variable>               ::= <identifier> |
                             <indexed variable>
<relation>                   <arithex> <relational op>
                             <arithex>
<Boolean primary>        ::= <Boolean value> | <variable> |
                             <initiator> |
                             <relation> | (<Boolex>)
<Boolean secondary>      ::= <Boolean primary> |
                             ¬<Boolean primary>
<Boolean factor>         ::= <Boolean secondary> |
                             <Boolean factor> ∧
                             <Boolean secondary>
<Boolean term>           ::= <Boolean factor> |
                             <Boolean term> ∨
                             <Boolean factor>
<implication>            ::= <Boolean term> | <implication> ⊃
                             <Boolean term>
<Boolex>                 ::= <implication> | <Boolex> ≡
                             <implication>
<expression>             ::= <arithex> | <Boolex>
<left part>              ::= <variable> :=
<assign>                 ::= <left part> <expression> ;
<assign sequence>        ::= <assign> | <assign sequence>
                             <assign>
<parameter>              ::= <expression> | <variable>
<param list>             ::= <parameter> | <param list>,
                             <parameter>
<param part>             ::= <empty> | (<param list>)
<argument>               ::= <identifier>
<arg list>               ::= <argument> | <arg list>
                             <argument>
<label>                  ::= <identifier>
<begin label>            ::= BEGIN <label> {<empty>
                             (<arg list>)};
<end label>              ::= END <label>;
<unlabelled action>      ::= <assign sequence> |
                             <action sequence> | <branch>
                             <labelled action>
                             <labelled action> |
                             <loop control> <labelled action>
<labelled action>        ::= <begin label>
                             <unlabelled action> <end label>
<action>                 ::= <unlabelled action> |
                             <labelled action>
<action sequence>        ::= <unlabelled action> |
                             <labelled action>
                             <unlabelled action> |
                             <labelled action>
                             <labelled action> |
                             <action sequence> <action>
<initiator>              ::= <label> <param part>
<branch>                 ::= DO <initiator> IF <Boolex>
                             {<empty> | ELSE DO <initiator>}; |
                             DO <initiator>;
<loop control>           ::= DO <initiator> {<empty> |
                             WHILE <Boolex> }FROM <identi-
                                                  fier> :=
                             {<integer> | <variable>}BY
                             {<integer> | <variable>}THRU
                             <variable>; | DO <initiator> WHILE
                             <Boolex> FROM <identifier> :=
                             {<integer> | <variable>}BY
                             {<integer> | <variable>};
<delimit statement>      ::= <begin label> | <end label>
```

Fig. 1 Syntax for the language

terminating when its Boolean expression is false. Each of these control statements names the action(s) it controls. Consequently, these actions could and would in practice be written apart from the control statements which invoke them. To avoid such complexities as actions which invoke themselves, we insist, however, that the action(s) named by a control statement be written below it and separate it from the next statement to be executed after the action(s) invoked by the control statement have been executed.

9. The most general form of the branching statement is

$$\text{DO } a_1 \text{ IF } \mathscr{B} \text{ ELSE DO } a_2;$$

Here a_1 and a_2 are action initiators and $\mathscr{B}$ represents a Boolean expression which is to be evaluated when the branching statement is encountered. If the value of the Boolean expression is TRUE, initiator a_1 is applied; if its value is FALSE, initiator a_2 is applied. In case the statement terminates just after the Boolean expression, the effect is the same as if a_2 were NIL; if it terminates after a_1 the effect is that resulting from $\mathscr{B}$ = TRUE.

10. The most general form of the looping statement is

$$\text{DO } a \text{ WHILE } \mathscr{B} \text{ FROM } \mathscr{L} := \eta_1 \text{ BY } \eta_2 \text{ THRU } \eta_3;$$

where $\mathscr{L}$ represents an identifier for the looping index, η_1 and η_2 represent variables which furnish initial value and increment for the looping index, and η_3 represents a variable to which the final value of the looping index is assigned.

To define the precise sequence of steps initiated by the looping statement, we introduce an auxiliary index j which is zero when the looping statement is encountered. Then the steps are:

1. For the current j, assign the value represented by $\eta_1 + j \cdot \eta_2$ to the identifier represented by $\mathscr{L}$ and then evaluate the Boolean expression.
2. If the Boolean expression has value TRUE, proceed to 3. If it has value FALSE, skip to step 4.
3. Apply initiator a, replace j by $j + 1$ and return to step 1.
4. Assign the value represented by $\eta_1 + (j - 1) \cdot \eta_2$ to the variable represented by η_3. This completes the sequence.

Omitting WHILE $\mathscr{B}$ is equivalent to using $(\eta_3 - \mathscr{L}) \cdot \eta_2 \geqq 0$ for $\mathscr{B}$ and deleting the assignment in 4. Omitting THRU η_3 deletes the assignment in 4 and requires use of the Boolean expression for termination.

A Backus-Naur Syntax Specification modelled after that of ALGOL (Naur, 1963) appears in Fig. 1. Some of its restrictions could well be relaxed. We have striven for simplicity wherever the penalty was not great.

Functions computable in the language

It is to be expected that the class of μ-recursive functions is computable with algorithms in this language. The following actions demonstrate this explicitly. Note that the branching instruction is not even used.

The Zero Function:

```
BEGIN Z ;
Z:=0;
END Z;
```

The Projection Functions:

```
BEGIN PKN(I1, I2, . . . , IN);
PKN: = IK;
END PKN;
```

The Successor Function:

```
BEGIN S(X);
S: = X + 1;
END S;
```

Primitive Recursion with initial function $G(X1, \ldots, XM)$ *and inductive step function* $H(X1, \ldots, XN)$ *where* $N = M + 2$:

```
BEGIN PR(X1, . . . , XM, Y);
Q(0): = G(X1, . . . , XM);
DO INDUCT WHILE N ≦ Y FROM N = 1 BY 1
THRU NZ;
BEGIN INDUCT;
Q(N): = H(X1, . . . , XM, N − 1, Q(N − 1));
END INDUCT;
PR: = Q(NZ);
END PR;
```

Composition: $C(X1, X2, \ldots, XM) = H(G1(X1, \ldots, XM), \ldots, GN(X1, \ldots, XM))$

```
BEGIN C(X1, X2, . . . , XM);
Y1: = G1(X1, . . . , XM);
Y2: = G2(X1, . . . , XM);
YN: = GN(X1, . . . , XM);
C: = H(Y1, Y2, . . . , YN);
END C;
```

μ-Recursion: $MUY(X1, \ldots, XN) = \mu Y(F(Y, X1, \ldots, XN) = 0)$

```
BEGIN MUY(X1, . . . , XN);
DO NIL WHILE F(I − 1, X1, . . . , XN) ≠ 0 FROM I
: = 1BY 1 THRU Y;
MUY: = Y;
END MUY;
```

Justification for the language

It should be clear that any algorithm written in the language we have defined is easily translated into FORTRAN or ALGOL and presumably into any reasonable algorithmic language including machine code. Hence, the justification needs only demonstrate that (1) algorithms written in the language have some special useful properties, (2) writing algorithms in the language is convenient enough and produces short enough execution times to be practical. The analysis of algorithm structure which follows is directed toward (1). The example then gives some feeling for the appearance of an algorithm in the language and perhaps even to some extent of its capability for producing economical computation. Thus, the example is directed toward (2) but is far short of demonstrating it.

Analysis of program structure

In proving the correctness of an algorithm, the author has found it essential to demonstrate an ordering of potential execution (defined later in this section) for pairs of actions. One reason is to assure that if action A relies, for a particular setting of looping indices, on action B for an operand, then if A is executed for that setting, B must have already made the operand available. Similarly, if action A relies, for a particular setting of looping indices, on an operand which action C destroys, then if A is executed for that setting, it must precede such destruction. Such properties were used by Rechard and Stark (1969) in a proof of the equivalence of two algorithms. We now show that proofs of the ordering of potential execution are facilitated when algorithms are written in the proposed language.

To each action in an algorithm, we assign a level within the algorithm, (denote the level of action A by $l(A)$) as follows:

1. If action A is the algorithm, then $l(A) = 1$.
2. If action A is formed from an action sequence and action B is an element of that sequence, then $l(B) = l(A) + 1$.
3. If action A is a control action and its control statement contains an initiator for action B, then $l(B) = l(A) + 1$.

Consider an action in the text of an algorithm. Either it is the complete algorithm or it is a component of another action. Whenever action A is formed with action B as one of its components, we say B is nested in A. Starting from A, we can form a sequence of action identifiers $A_1, A_2, \ldots, A_m$ in which $A_m = A$ and such that A_{i+1} is nested in $A_i (i = 1, \ldots, m - 1)$. We call this the nesting sequence for A and define A_m to be nested in each action preceding it in the sequence. In the example of Fig. 2, this sequence for the action beginning at statement 9 is EUCLID, LOOP, REDUCE.

The nesting sequence will point to a place in the text of the algorithm, but it is not sufficient to distinguish among executions of the same action for distinct settings of control indices. For that, and for the more complex case of distinct actions, we need an execution pointer.

Consider the possible executions of an action A. Let its nesting sequence be $A_1, A_2, \ldots, A_m$. For such a sequence we may define an index sequence $i_1, i_2, \ldots, i_m$ where i_j is defined as follows:

1. If A_j is not a looping action, then $i_j = 0$.
2. If A_j is a looping action and $j \neq m$, then i_j is a value from the domain of the looping index of the control statement of A_j.
3. $i_m = 0$ (if A is a looping action, then its execution consists of all executions of the action it controls, not just one).

Any execution of A can be selected by giving its nesting sequence and an index sequence with current values for looping indices.

A pair $(AS; IS)$ consisting of a nesting sequence and an index sequence is called a potential execution. Referring again to Fig. 2, (EUCLID, LOOP, REDUCE; 0, 7, 0) would represent a potential action. An execution determines a nesting sequence and an index sequence and therefore a potential execution. But not every potential execution determines an execution, since Boolean conditions which depend on input data influence the execution path. Nevertheless, much that needs to be proved can be based on these potential executions.

Consider two potential executions

$$(AS; IS) \text{ and } (BS; JS)$$

where

$$AS = A_1, \ldots, A_m$$
$$BS = B_1, \ldots, B_n$$
$$IS = i_1, \ldots, i_m$$
$$JS = j_1, \ldots, j_n$$

Let k be the largest integer such that $A_l = B_l (l = 1, \ldots, k)$ and $i_l = j_l (l = 1, \ldots, k)$. If $m = k = n$, then these are the same potential action. If $m = k$ and $n > k$, then $(BS; JS)$ is nested in $(AS; IS)$. Similarly, if $m > k$ and $n = k$, then $(AS; IS)$ is nested in $(BS; JS)$.

If $m > k$ and $n > k$, then either $A_{k+1} \neq B_{k+1}$ or $i_{k+1} \neq j_{k+1}$. If $A_{k+1} = B_{k+1}$, then A_{k+1} is a looping action and the potential action whose index occurs first as looping index will not be executed after the other. If $A_{k+1} \neq B_{k+1}$, then these are two distinct actions at the same level. That which occurs earlier in the text of the algorithm will not follow the other.

Analysis of an algorithm in the language

We have chosen, because of its familiarity, to write the Extended Euclid's Algorithm in our language and then to prove the validity of this formulation. There is one advan-

tage to choosing such a commonly used example, namely that it makes comparison easy. Knuth (1968) has a proof on p. 15 according to a method formulated by Floyd (1967). Our algorithm for finding the greatest common divisor, C, of integers M and N (neither negative and one positive) is given in Fig. 2. At the beginning and after each execution of the REDUCE algorithm, gcd(M, N) = gcd(C, D) and C = ASTAR*M + BSTAR*N, D = A*M + B*N.

We would have chosen to retain indices in the statements of our algorithm rather than recreating them as below, but that would have made comparison with Knuth's proof less direct.

For purpose of analysis, we need to be able to label, in a unique way, every quantity assigned to a variable during computation. The notion is to use the statement number in which a value is assigned and the current values of indices in loops controlling that statement for the purpose. Rather than propose any general labelling scheme, we apply a straightforward labelling scheme for this example.

We may designate the value assigned to any identifier at the completion of a statement which is not in the loop by applying the statement number as a superscript to the identifier. Thus, $D^6 = N$. When the statement is within the action controlled by the DO statement, we supply the current value of the looping index as a subscript as well as statement number as a superscript. For example,

$$R_1^{11} = M - IQ(M, N)*N$$

where IQ(M, N) is the integer quotient of M by N.

With this symbolism, we can abstract from the instructions of Fig. 2 the set of relations given in Fig. 3. Again, we have chosen not to formalise the abstractions from the algorithm. They should be intuitively clear.

Our proof of the validity of the algorithm in Fig. 2 is now made completely from the relations in Fig. 3. As a first step we relate values at statement 20 to those at statement 9 by single substitution sequences:

$$\text{(A)}\quad \left.\begin{aligned} ASTAR_K^{20} &= ASTAR_K^{14} = T_K^{12} = A_K^9 \\ BSTAR_K^{20} &= BSTAR_K^{17} = T_K^{15} = B_K^9 \\ A_K^{20} &= A_K^{13} = ASTAR_K^9 - Q_K^{10}*A_K^9 \\ B_K^{20} &= B_K^{16} = BSTAR_K^9 - Q_K^{10}*B_K^9 \\ C_K^{20} &= C_K^{18} = D_K^9 \\ D_K^{20} &= D_K^{19} = R_K^{11} = C_K^9 - Q_K^{10}*D_K^9 \end{aligned}\right\} (K = 1, \ldots, \$1)$$

It is a simplification to define

$$X_0^{20} = X^6 \text{ where } X\varepsilon\{ASTAR, BSTAR, A, B, C, D\}$$

so that

$$\text{(B)}\qquad X_K^9 = X_{K-1}^{20} (1 \leqq K \leqq \$1)$$

```
 0   BEGIN EUCLID;
 1   ASTAR:=1;
 2   BSTAR:=0;
 3   A:=0;
 4   B:=1;
 5   C:=M;
 6   D:=N;
 7   BEGIN LOOP;
 8   DO REDUCE WHILE D ≠ 0 FROM I:=1 BY 1 THRU $I;
 9   BEGIN REDUCE;
10   Q:=IQ(C,D);
11   R:=C-Q*D;
12   T:=A;
13   A:=ASTAR-Q*A;
14   ASTAR:=T;
15   T:=B;
16   B:=BSTAR-Q*B;
17   BSTAR:=T;
18   C:=D;
19   D:=R;
20   END REDUCE;
21   END LOOP;
22   END EUCLID;
```

Fig. 2. A formulation of Euclid's Algorithm

$ASTAR^6 = 1, BSTAR^6 = 0, A^6 = 0, B^6 = 1, C^6 = M, D^6 = N$

$X_1^9 = X^6$ where $X\varepsilon\{ASTAR, BSTAR, A, B, C, D\}$

$X_K^9 = X_{K-1}^{20}$ $(1 < K \leq \$I)$ where $X\varepsilon\{ASTAR, BSTAR, A, B, C, D\}$

For any K in [1,$I]

$Q_K^{10} = IQ(C_K^9, D_K^9)$,

$R_K^{11} = C_K^9 - Q_K^{10}*D_K^9$,

$T_K^{12} = A_K^9$,

$A_K^{13} = ASTAR_K^9 - Q_K^{10}*A_K^9$,

$ASTAR_K^{14} = T_K^{12}$,

$T_K^{15} = B_K^9$,

$B_K^{16} = BSTAR_K^9 - Q_K^{10}*B_K^9$,

$BSTAR_K^{17} = T_K^{15}$,

$C_K^{18} = D_K^9$,

$D_K^{19} = R_K^{11}$,

$C_K^{20} = C_K^{18}, D_K^{20} = D_K^{19}, A_K^{20} = A_K^{13}, B_K^{20} = B_K^{16}, ASTAR_K^{20} = ASTAR_K^{14}$,

$BSTAR_K^{20} = BSTAR_K^{17}$.

$X^{21} = X_{\$I}^{20}$ where $X\varepsilon\{ASTAR, BSTAR, A, B, C, D\}$

Fig. 3. Abstractions from the algorithm of Fig. 2

where $X \varepsilon \{ASTAR, BSTAR, A, B, C, D\}$.

This permits a restatement of (A) as:

$$(A') \quad \left.\begin{aligned} ASTAR_K^{20} &= A_{K-1}^{20} \\ BSTAR_K^{20} &= B_{K-1}^{20} \\ A_K^{20} &= ASTAR_{K-1}^{20} - Q_K^{10} * A_{K-1}^{20} \\ B_K^{20} &= BSTAR_{K-1}^{20} - Q_K^{10} * B_{K-1}^{20} \\ C_K^{20} &= D_{K-1}^{20} \\ D_K^{20} &= C_{K-1}^{20} - Q_K^{10} * D_{K-1}^{20} \end{aligned}\right\} (K = 1, \ldots, \$1)$$

where $ASTAR_0^{20} = 1$, $BSTAR_0^{20} = 0$, $A_0^{20} = 0$, $B_0^{20} = 1$, $C_0^{20} = M$, and $D_0^{20} = N$. From the last two equations of (A'), we have that if X divides C_{K-1}^{20} and D_{K-1}^{20}, it divides C_K^{20} and D_K^{20} and vice versa. Hence,

$$\gcd(C_K^{20}, D_K^{20}) = \gcd(C_{K-1}^{20}, D_{K-1}^{20}) = \ldots = \gcd(M, N)$$

From the definition of integer quotient, we have that

$$0 \leqq R_K^{11} < D_K^9$$

which may be restated by use of (A) and (B) as

$$0 \leqq D_K^{20} < D_{K-1}^{20} .$$

Hence, there is a K such that $D_K^{20} = 0$. That K is \$I, since $D_K^{20} = 0$ terminates the loop.

We define

$$\Gamma_K = ASTAR_K^{20} * M + BSTAR_K^{20} * N$$

$$\Delta_K = A_K^{20} * M + B_K^{20} * N$$

Then from (A')

$$\Gamma_K = \Delta_{K-1}$$

$$\Delta_K = \Gamma_{K-1} - Q_K^{10} * \Delta_{K-1}$$

where $\Gamma_0 = M$ and $\Delta_0 = N$.

Since the pairs (Γ_K, Δ_K) have the same initial values and the same inductive definition as the pairs (C_K^{20}, D_K^{20}), they are identical; that is

$$C_K = ASTAR_K^{20} * M + BSTAR_K^{20} * N$$

$$D_K = A_K^{20} * M + B_K^{20} * N$$

Conclusion

The statement of Euclid's Algorithm and subsequent proof of its validity demonstrate a high potential for mechanisation in the proof procedure. Clearly, the relations of Fig. 3 can be taken from the algorithm by a simple formal procedure. It is not difficult to picture continuing mechanically through (A'). There seem to be two non-routine steps. One is to recognise the importance of and prove the relation

$$\gcd(C_K^{20}, D_K^{20}) = \gcd(C_{K-1}^{20}, D_{K-1}^{20}) \; (K = 1, \ldots, \$1) .$$

The other is to define the sequence $\{(\Gamma_K, \Delta_K)\}$ and then prove that it is calculated by the same rules as the sequence $\{(C_K, D_K)\}$ and is therefore identical to it.

It is our contention that a language which forces into the algorithm a facility for ordering the execution of statements provides a major simplification for the analyst who proposes to prove its validity. The language proposed in this paper accomplishes this task, and by so doing opens new avenues for development of proofs of validity.

References

DIJKSTRA, W. W. (1968). Go to statement considered harmful, *CACM*, Vol. 11, pp. 147-148.

FLOYD, R. W. (1967). Assigning meanings to programs, *Proc. Symp. Appl. Math., AMS* 19, pp. 19-32.

KNUTH, Donald E. (1968). *Fundamental Algorithms: The Art of Computer Programming*, Vol. 1, Addison-Wesley, pp. 14-17.

NAUR, P. (Ed.). (1963). Revised report on the algorithmic language ALGOL 60, *CACM*, Vol. 6, pp. 1-17.

RECHARD, O., and STARK, R. (1969). Equivalence of two algorithms for Cooper's generalised factorial function, *The Computer Journal*, Vol. 12, pp. 33-37.

STARK, R. H. (1968). On means to record algorithms to facilitate generation of error-free programs, WSU Computing Center Report, 68-1.

Book review

Numerical Approximation to Functions and Data, by J. G. Hayes, 1970; 177 pages. (*The Athlone Press*, 60*s* = £3.00)

This book is the proceedings of the conference with the same title held by the IMA at the University of Kent in September 1967. The editor and publishers have used the intervening time to produce a very polished product but I must say that my own preference would be towards foregoing some of this polish for the sake of publication within, say, a year of the conference.

The main emphasis is on the practical aspects of approximation theory, that is the representation of mathematical functions (which are defined exactly) and the fitting of empirical data (which are subject to errors so that some smoothing is necessary). This makes it a useful companion volume to 'Methods of Numerical Approximation', edited by D. C. Handscomb, which has a more theoretical flavour.

The chapters written by Clenshaw and Hayes provide plenty of practical advice on the use of polynomials and jointed polynomials. Their emphasis is on the user understanding the difficulties associated with his problem and making such decisions as which order is appropriate, where different polynomials should be joined and whether a transformation of the independent variable is appropriate. They rely on the computer to do the actual fitting and to provide information upon which these decisions may be made.

On the other hand, the chapters of Curtis, Powell, and Payne place emphasis on automatic algorithms with almost all decision-making taken out of the hands of the user and given to the computer. Curtis, considering the approximation of mathematical functions, and Powell, considering the fitting of experimental data, both use cubic splines to construct algorithms. Payne also discusses the curve-fitting problem but he uses jointed polynomials whose order is not fixed.

There are two papers which, although they are very interesting in their own right, do not really match the flavour of the remainder. These are by Barrodale and Young presenting their experience in solving a number of linear operator equations using Chebyshev approximation and by Meinguet whose paper is a theoretical discussion of approximation in a general context.

Also included are an excellent introduction by Fox and a concluding survey by Davis which is as pleasing to read now as it was to hear at the time.

J. K. REID (Harwell)