The algebraic anatomy of programs

L. S. Levy and R. Melville*

Department of Statistics and Computer Science, University of Delaware, Newark, Delaware 19711, USA

Game playing and backtracking programs among others can be understood in terms of algebraic operations with zeros. The special properties of the algebraic operations can be used to explain the equivalence of otherwise dissimilar algorithms.

(Received April 1976)

1. Prelude

Programming is undergoing an extensive rethinking due primarily to the leadership of Dijkstra (1972). The effects of this change in our concept of programming will be to set it into perspective with other intellectual and problem-solving activities. In the course of such a revision of programming, we should keep a clear idea of the purpose of redefining programming:

> Programs must be understood by people. Only if we understand the programs will we know if they are correct, and where they are applicable. And only if we know that the programs are correct can we be concerned about efficiency, because the most efficient incorrect program is 'STOP' which takes no time and no physical resources.

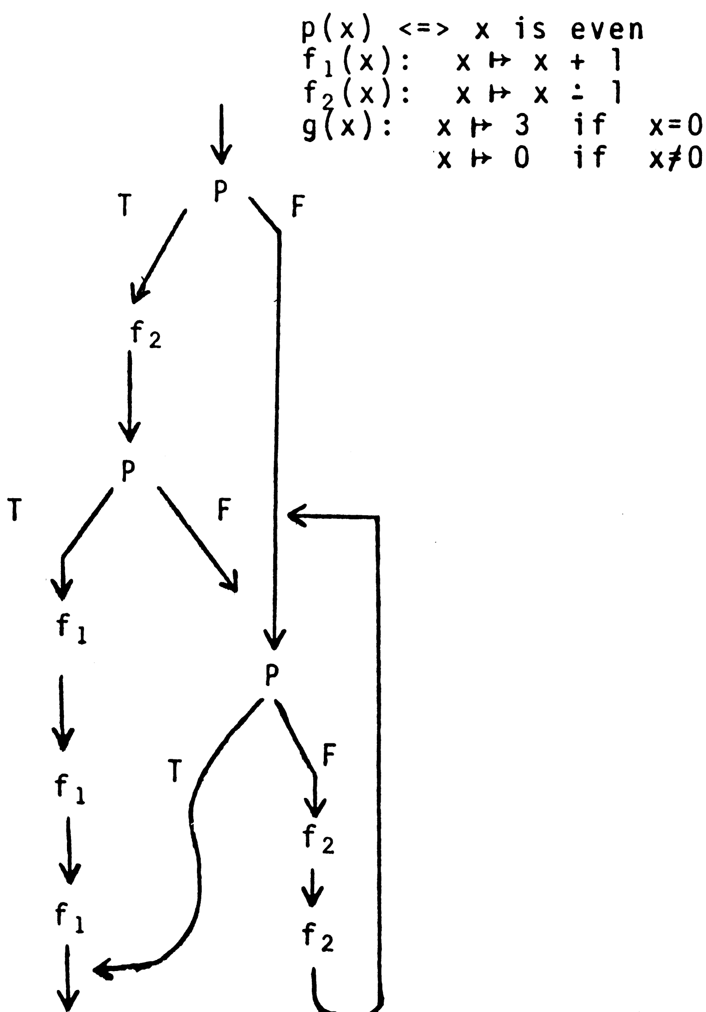


Fig. 1 Program to compute $g(x)$

We define the *deep structure* of a program as the description of the underlying concepts in terms of which the program is understood. The deep structure and the methodology for developing the program text are the aspects of *structured programming* which are considered most important. The syntax of the program text considered by itself is often an inaccurate indicator of program quality.

We shall give only the following example:

Let

$$F = \{f_1 : x \to x + 1, f_2 : x \to x \dot{-} 1\}^*$$

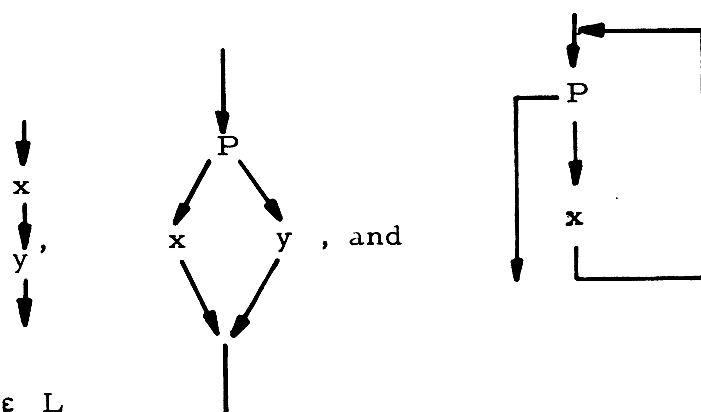
and $P = \{p(x) <=> x \text{ is even}\}$. F is a set of two functions, and P is a set of one predicate, all defined on $N = \{0, 1, 2, \ldots\}$, to be included in our mini language. The program to be written is to compute the function:

$$g(x) = \begin{matrix} 3 \text{ if } x = 0 \\ 0 \text{ if } x \neq 0 \end{matrix}$$

Clearly a program to compute g is shown in **Fig. 1.** On the other hand, suppose that our programming language, L, were restricted in the following way:

(a) $x \in F => x \in L$

(b) $x, y \in F, p \in P =>$



(c) L is the smallest set of programs defined by 1 and 2.

Then L is composed of D-charts, and although g is programmable in L nothing is gained by so doing. (Kosaraju (1974) has the most complete discussion of this issue, but see also Knuth (1974b), Ashcroft and Manna (1972), and the special SIGPLAN issue on control structures, Leavenworth (1972)).

The point of the example is that a conceptual issue in con-

*Current address: Department of Computer Science, Cornell University, Ithaca, New York

$$^*x \dot{-} y = \begin{matrix} x - y & \text{if } x \geqq y \\ 0 & \text{if } x \leqq y \end{matrix}$$

Downloaded from https://academic.oup.com/comjnl/article/20/4/340/393728 by guest on 19 April 2024

structing g from F and P is to understand how a test for evenness can be used to test for zero (when appropriate functions are available). **Fig. 2** shows such a zero test in non-L form.

In this paper we develop the deep structures of some programs in terms of the algebraic properties of the groupoids. Examples are given in which this point of view is then useful to programmers who share this algebraic perspective in developing improved programs.

We acknowledge the following who have influenced our approach to programming: Dijkstra (1972), Burstall (1969), Burstall and Landin (1969), Mills (1972), and Yelowitz (1973).

2. Groupoids

2.1. *Algebraic prelude*

Dijkstra (1972) has posed the question, 'In what sense can the following programs, (i) and (ii) be considered equivalent?'

Let A_i $1 \leqslant i \leqslant n$ be an array of truth-values and we wish to compute the conjunction of the A_i's:

$$p = \bigwedge_{i=1}^{n} A_i \ .$$

```
(i)  condition ← TRUE
     do i = 1 to N
     condition ← condition ∧ A_i
     end
(ii) condition ← TRUE
     while condition do i = 1 to N
     condition ← condition ∧ A_i
     end
```

Program (i) runs through the whole linear array, A, to compute p. Program (ii), however, makes use of an additional fact about the algebraic operation, $\wedge$, namely that it has a zero. The important point is that Program (ii) can be understood as a variation of Program (i) which is applicable whenever the algebra has a zero. Let us review the relevant algebraic notions.†

An algebraic system, $a = \langle A, \Omega \rangle$ is composed of a set of elements A called the carrier of a and a set of operations, Ω. A ranking function ρ assigns to each operation w in Ω a natural number, in $\{0, 1, 2, \ldots\}$, called the rank of w. denoted $\rho(w)$. Each operation w in Ω corresponds to a function w: $A^{\rho(w)} \rightarrow A$, mapping the n-fold Cartesian product of the carrier to the carrier. Operations of rank 0 are constants, operations of rank 1 are called unary, those of rank 2 are binary, those of rank 3 are ternary, etc.

A groupoid, **G**, is an algebraic system, $\mathbf{G} = \langle G, \{\theta\} \rangle$ where θ is a binary operation. If G is finite, then θ can be specified by a groupoid operation table, where $\theta(g_i, g_j)$ is shown in the row of g_i and column of g_j. (In infix notation $\theta(g_i, g_j)$ is denoted $g_i \theta g_j$).

Example:
Groupoid of logical values and $\wedge$:

$$\mathbf{G} = \langle \{\text{TRUE, FALSE}\}, \{\wedge\} \rangle \ .$$

The operation table is:

$\wedge$	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

An element x of a groupoid with operation θ is called a *right identity* if $\theta(y, x) = y$ for every y in the carrier; x is called a *left identity* if $\theta(x, y) = y$ for every y in the carrier. In the groupoid of logical values and $\wedge$, TRUE is both a left identity and a right identity.

†A readable introduction for computer scientists is contained in Preparata and Yeh (1973). More advanced treatment is given in texts on Universal Algebra; for example, Grätzer (1968).

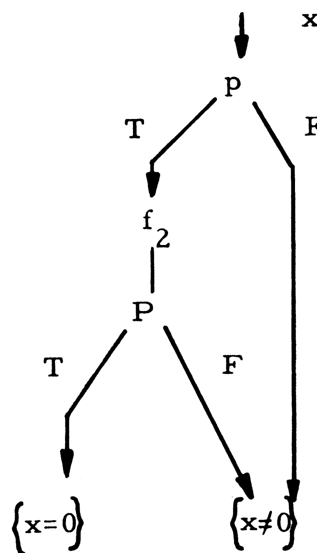


Fig. 2 Zero test in terms of evenness test

An element x of a groupoid with operation θ is called a right zero if $\theta(y, x) = x$ for every y in the carrier. x is called a left zero if $\theta(x, y) = x$ for every y in the carrier. In the groupoid of logical values and $\wedge$, FALSE is both a left zero and a right zero.

The following facts about groupoids are well-known: A groupoid may have many left zeros or many right zeros. But if the groupoid has both a left zero and a right zero, then the left zero and the right zero are the same, and there are no other left zeros or right zeros.

Similarly a groupoid may have many left identities or many right identities. But if the groupoid has both a left identity and a right identity then the left identity and the right identity are the same, and there are no other left identities or right identities.

2.2. *Evaluating a sequence over a groupoid*

Let $(S_1, S_2, \ldots, S_n)$ be a sequence of elements of the groupoid, $=(G, \theta)$. The evaluation of the sequence over the groupoid is the computation of $(\ldots(S_1 \theta S_2) \ldots \theta S_n)$. The following programs are applicable:

Program 1—most general

```
v ← S_1
i ← 2
while i ≤ n do begin
    v ← v θ S_i
    i ← i + 1
    end
```

Program 2—the groupoid has a left identity

```
v ← 1
i ← 1
while i ≤ n do begin
    v ← v θ S_i
    i ← i + 1
    end
```

Program 3—the groupoid has a left zero

```
v ← S_1
i ← 2
while i ≤ n and v ≠ 0 do begin
    v ← v θ S_i
    i ← i + 1
    end
```


Downloaded from https://academic.oup.com/comjnl/article/20/4/340/393728 by guest on 19 April 2024

Program 4—the groupoid has a left zero and a left identity

```
v ← 1
i ← 1
while i ⩽ n and v ≠ 0 do begin
    v ← v θ S_i
    i ← i + 1
    end
```

Program 5—the groupoid has a left zero and a left identity and no zero divisors.

```
v ← 1
i ← 1
while i ⩽ n do begin
    if S_i = 0 then do begin
        v ← 0
        exit
        end
    v ← v θ S_i
    i ← i + 1
end
```

Programs (i) and (ii) of Section 2.1 can now be described as the evaluation of a sequence over the groupoid

$$G = (\{\text{TRUE, FALSE}\}, \{\wedge\}) \ .$$

If S_i $1 \leqslant i \leqslant n$ is the array of truth values, the most general program is:

```
(i) a)  v ← S_1
        i ← 2
        while i ⩽ n do begin
            v ← v ∧ S_i
            i ← i + 1
            end
```

while recognition of both the presence of a left zero and left identity and no zero divisors allows the program to be written as:

```
(ii) a) v ← 1
        i ← 1
        while i ⩽ n do begin
            if S_i = 0 then do begin
            v ← 0
            exit
            end
        v ← v ∧ S_i
        i ← i + 1
        end
```

Note
If the groupoid may have zero divisors, then the test must be done on v, as in program 4.

Many more variations are possible if the groupoid is associative or commutative, or other special properties arise, such as:

```
i ← 1
while (i < n and S_i = TRUE)
    i ← i + 1
    end
v ← S_i
```

which can be used for the iterative evaluation over the groupoid of truth values and $\wedge$.

The style of presentation adopted to facilitate the discussion emphasises the iterative control structure. No generality is lost in so doing. When the algebra is associative and commutative, as in the case of the minimax algorithm, set operations may be used rather than sequential iteration.

Another style of presentation which 'hides' the counting from the programmer is a pseudo-LISP style. Any of the given programs can readily be recast in that form (See e.g. Burge, 1975). For example, program 4 of this section becomes:

Program 4a

```
ez a α = if null(α)
            then a
            else if head(α) = 0
                then 0
                else ez a θ head(α) tail(α)
```

3. Game trees

3.1. *Minimax prelude*

In two-person game playing situations (see Nilsson, 1971), it is common to represent the strategies available to the players as a game tree as in **Fig. 3.**

The numbers on the leaves of the tree indicate the gain to A if the game is played in such a way as to arrive at that leaf. For example, at his initial move player A may choose a, b or c. If A chooses b, then player B may choose to play g or h. If player B chooses g, then the gain to A will be four units. If player B chooses h then the gain to A will be two units.

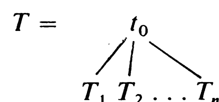
A strategy for playing this game which is widely used is the so-called *minimax strategy*. The assumption underlying this strategy is that at each move player A will play to maximise his gain, and player B will play to minimise A's gain, and each player is aware of the other's rule of play.

A pair of recursive programs are readily written to compute the gain to A at a node.

Let T be the rest of a game tree seen from any node in the tree. If the node is not a leaf then we may write

$$T = t_0(T_1, T_2, \ldots, T_n)$$

where T_i, $1 \leqslant i \leqslant n$, are the subtrees of T from left to right:



The recursive programs are:

max (T) $\stackrel{=}{\Delta}$ **if** T is-a-leaf **then** T
else maximum $\{\min(T_1), \ldots, \min(T_n)\}$

min (T) $\stackrel{=}{\Delta}$ **if** T is-a-leaf **then** T
else minimum $\{\max(T_1), \ldots, \max(T_n)\}$

In writing these programs we have used a style of pseudo-PL/1 and allowed operations *maximum* which selects the largest element of a set, and *minimum* which selects the smallest element of a set.

The fact that these programs could be written as a pair of mutually recursive programs is because the evaluation of a game tree corresponds to a bottom-up automaton evaluating the tree. It is known that this form of tree evaluation is a

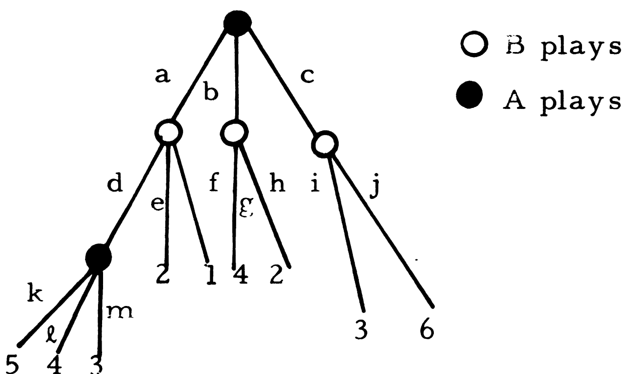


Fig. 3 A typical (?) game tree

Downloaded from https://academic.oup.com/comjnl/article/20/4/340/393728 by guest on 19 April 2024

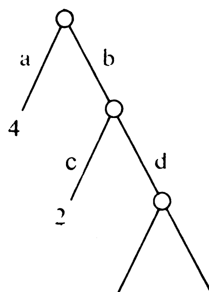


Fig. 4 Part of a game tree illustrating $\alpha\beta$

homomorphism from the tree algebra to the value algebra, as discussed in Levy (1976). Hence, this style of programming is itself an illustration of the principle of incorporating the algebraic structure of the problem into the program.

Alternatively, we might write the programs using the iterative control construct, **do** $i = 2, \ldots, n$ with binary operations ⌈ and ⌊, to compute the greater and smaller respectively of their operands. (Thus 5 ⌈ 3 is 5 and 5 ⌊ 3 is 3). The iterative programs are:

```
max (T) ≜ if T is-a-leaf then return T
          else begin value ← min (T1)
            do i = 2, . . ., n
              value ← value ⌈ min (Ti)
              end
            return value end

min (T) ≜ if T is-a-leaf then return T
          else begin value ← min (T1)
            do i = 2, . . ., n
              value ← value ⌈ max (Ti)
              end
            return value end
```

The groupoid structure in the evaluation of a node which is made explicit in the sequential version of the program using the iterative control construct is very important in the transformed version of this program discussed in the next section.

3.2. *Alpha-Beta* ($\alpha\beta$)

Consider the situation shown in **Fig. 4.**

Once player B has evaluated the path labelled c, he need not consider further the path labelled d, since the value of the node at b cannot be more than 2—if it were player B he would evaluate the node as at most 2, since he always chooses the minimum—and since the root of the tree cannot have a value less than 4, since player A chooses the maximum, the subtree reached via d cannot have any effect on the value of the tree to A. Therefore, the subtree reached via d need not be evaluated.

This is reminiscent of the situation we had encountered in Section 2, in evaluating a sequence over a groupoid with a zero. The value at the node b is the minimum of the values of its subtrees, evaluated from left to right, with the added condition that any value less than or equal to 4 is a zero of the groupoid. Summarising, the evaluation of the sequence of subtrees at a node is performed in a groupoid with a zero, the value of the zero being determined at the node.

A special case occurs when evaluating the left-most subtree of a node, since the left-most subtree does not have a zero. It is usual in presentations of the alpha-beta algorithm to introduce fictitious zeros—$+\infty$ and $-\infty$—for this case. We feel that little is gained by this artifice, and that since such an artifice tends to conceal the algebra at the node it should be avoided.

With this much introduction we can now present the alpha-beta algorithm:

```
max (T) ≜ if T is-a-leaf then return T
          else begin top ← min (T1)
            do i = 2, . . ., n
              top ← top ⌈ min (Ti, top)
              end
            return top end

min (T) ≜ if T is-a-leaf then return T
          else begin bottom ← max (T1)
            do i = 2, . . ., n
              bottom ← bottom ⌊ max (Ti, bottom)
              end
            return bottom end

max (T, bottom) ≜ if T is-a-leaf then return T
                  else begin top ← min (T1)
                    do i = 2, . . ., n while (top < bottom)
                      top ← top ⌈ min (Ti, top)
                      end
                    return top end

min (T, top) ≜ if T is-a-leaf then return T
               else begin bottom ← max (T1)
                 do i = 2, . . ., n while (bottom > top)
                   bottom ← bottom ⌊ max (Ti, bottom)
                   end
                 return bottom end
```

Note

max (T, bottom) is the larger of max (T) and bottom while min (T, top) is the smaller of min (T) and top. In fact, the situation is slightly more complicated since in max (T, bottom), for example, we are only interested in increasing the computed value of the subtree if it can effect the computed value of the whole tree. (See Knuth, 1974b).

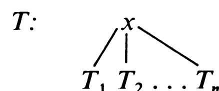
4. Problems

4.1. *Backtracking*

The technique of enumeration known as backtracking is discussed in Dijkstra (1972) and Nilsson (1971). However, the best introduction and discussion of backtracking is still Golomb and Baumert (1965). We can summarise the algorithm as follows.

Arrange the set of alternatives as a decision tree. The set of leaves is called the frontier. Each path in the tree from root to frontier corresponds to a sequence of choices. The set of solutions in the tree is the set of paths which do not fail.

An algorithm which does not use the zeros of the algebrae to simplify the computation would calculate the set of paths from a node x to the frontier, S_T, in the tree



as $S_T = \bigcup_{i=1}^{n} x * S_{T_i}$, where x is the label of the root. The set of solutions is then the subset of the set of paths which are successful, i.e. whose value is non-zero.

However, the backtrack program uses the fact that if the value of an initial segment of the path from root to frontier is zero, then the value of the complete path must be zero. In the following programs we again identify the node of a tree with its label. The complete programs are:

```
b ≜ for a in A1 do
      if not zero (a) then ba (a, a)

ba(α, x) ≜ if x is-a-leaf then print (α)
```

Downloaded from https://academic.oup.com/comjnl/article/20/4/340/393728 by guest on 19 April 2024

```
else do i = 1, . . ., n
  if not zero (α*x_i) then
    ba(α*x_i, x_i)
  end
```

The procedure *ba* has parameters α and x, where x is a node label and α is the sequence of node labels encountered on a path from root node to x.

Note that the backtrack program differs from the usual versions in that the backtracking, or reduction of the index k, is not explicitly shown. Thus the algebraically motivated backtrack program manages to 'hide' the stack control from the programmer—a feature that is desirable in recursive programming.

4.2. *The problem of the eight queens*

The eight queens problem is often used in the literature (Dijkstra, 1972) as a standard example of the application of backtracking. The problem is to find ways to place eight queens on a chessboard so that no queen is attacking any other queen. A brute force approach might examine the $\binom{64}{8}$ possible positions in which eight queens can be placed. With the observation that at most one queen can be placed in a row, the search space can be narrowed to $(8)^8$ positions. We examine the $(8)^8$ positions by placing queens in rows $1, \ldots, i$ and then attempting to place a queen in row $i + 1$, in the eight possible column positions of that row. When placement of a queen in row $i + 1$, column j is not allowed—because that square is under attack—the attempted partial solution, including $(i + 1, j)$ will never lead to a valid solution. In the algebra of backtracking, that partial solution is a *zero*.

We present first the abstract version of the eight queens problem followed by the program text of the B6700 ALGOL program. In these programs the following notations have been used:

qa(*k*):	queen has been placed in row k; if $k = 7$ then write out the solution, otherwise attempt to place a queen in row $k + 1$.
col [*c*] ⟨=⟩	column c is free
up [*u*] ⟨=⟩	upward diagonal u is free
down [*d*] ⟨=⟩	downward diagonal d is free
Q[0 . . . *k*]:	placement of queens $0, \ldots, k$ representing a partial solution

The abstract program is:

```
qa(k) ≜ if k = 7 then print Q
          else for i to 8 do
            if not zero(i) then
              Q[k + 1] := 1;
              qa(k + 1)
            fi
      fi;
for i to n do (Q[0] := i; qa(0))
```

Note that the abstract program follows the general backtracking pattern with $k = 7$ corresponding to a leaf of the search tree. In the complete program following, the zero test uses the three arrays, *col*, *up*, *down*, to check if the square being evaluated is under attack.

The complete ALGOL 60 program is:

```
begin
 file out (kind = printer);
 integer array Q[0:7];
 boolean array col [0:7], up [−7:+7], down [0:14];
 procedure qa(k); value k; integer k;
  if k = 7 then write(out, ⟨8(x2, i2)⟩, Q)
   else begin integer c, u, d;
    c := 0;
    u := (k + 1) − c;
    d := (k + 1) + c;
    while c <= 7 do begin
     if col[c] and up[u] and down[d]
     then begin
      Q[k + 1] := c;
      col[c] := up[u] := down[d] := FALSE;
      qa[k + 1];
      col[c] := up[u] := down[d] := TRUE;
    end
  end qa;
 integer c, j;
  j := 0;
  while j <= 7 do begin
         col[j] := TRUE;
         j := * + 1
         end;
  j := 0;
  while j <= 14 do begin
         up[j − 7] := down[j] := TRUE;
         j := * + 1
         end;
  c := 0;
  while c <= 7 do begin
         col[c] := up[−c] := down[c] := FALSE;
         Q[0] := c;
         qa[0];
         col[c] := up[−c] := down[c] := TRUE;
         c := * + 1
         end
end.
```

5. Conclusion

We have shown several examples in which the algebra of an algorithm can be used to clarify a program text and its accompanying documentation. The programmer who knows the relevant algebra will be better able to understand the programs. More, an understanding of the deep structure can lead to improved versions of programs in which properties of the underlying algebra can be exploited in the control structure of the program.

Downloaded from https://academic.oup.com/comjnl/article/20/4/340/393728 by guest on 19 April 2024

References

ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'go to' programs to 'while' programs, *Proc. IFIP Congress 1971*, Vol. 1, North Holland Publ. Co., Amsterdam, The Netherlands, p. 250-255

BURGE, W. (1975). *Recursive Programming Techniques*, Addison-Wesley, Reading, Mass.

BURSTALL, R. M. (1969). Proving Properties of Programs by Structural Induction, *The Computer Journal*, Vol. 12, p. 41-48

BURSTALL, R. M. and LANDIN, P. J. (1969). Programs and their Proofs: an Algebraic Approach, *Machine Intelligence*, 4. B. Meltzer and D. Michie, (eds.) American Elsevier, p. 17-43

DIJKSTRA, E. W. (1972). Notes on Structured Programming in *Structured Programming* by Dahl, Dijkstra, and Hoare, Academic Press

GOLOMB, S. and BAUMERT, L. (1965). Backtrack Programming, *JACM*, Vol. 12, No. 3, p. 516-524

GRÄTZER, G. (1968). *Universal Algebra*, Van Nostrand, Princeton, NJ

KNUTH, D. E. (1974a). Structured Programming with go to Statements, *ACM Computing Surveys*, Special Issue: Programming, Vol. 6, No. 4, p. 261-302

KNUTH, D. E. (1974b). An Analysis of Alpha-Beta Pruning, Stanford University Report STAN-CS-74-441

KNUTH, D. E. and FLOYD, R. W. (1974). Notes on avoiding 'go to' statements, *Information Processing Letters*, Vol. 1, No. 1, p. 23-31

KOSARAJU, S. R. (1974). Analysis of Structured Programs, *Journal of Computer and System Sciences*, Vol. 9, p. 232-255
LEAVENWORTH, B. M. ed. (1972). Special Issue on Control Structures in Programming Languages, *ACM Sigplan notices*, Vol. 7, No. 11
LEVY, L. S. (1976). Automata on Trees: A Tutorial Survey, *Egyptian Computer Journal* (forthcoming).
MILLS, H. D. (1972). Mathematical Foundations for Structured Programming, FSC 72-6012, Federal Systems Division, IBM Corporation, Gaithersburg, Md.
NILSSON, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, NY
PREPARATA, F. P. and YEH, R. T. (1973). *Introduction to Discrete Structures*, Addison-Wesley, Reading, Mass.
YELOWITZ, L. (1973). A Symmetric, Top-Down Structured Approach to Computer Program/Proof Development, IBM Report FSC 73-5001, IBM Federal Systems Division, Gaithersburg, Md.

Book reviews

Informal Introduction to Algol 68, by C. H. Lindsey and S. G. van der Meulen, 1977; 361 pages. (*North Holland*, for IFIP, US$14.50)

ALGOL 60 appeared in 1960; the report was quite difficult reading to a generation not brought up on Backus Naur Form, but it became the users' bible. A few errors were detected and a revised report appeared in 1962. ALGOL 68 appeared in 1968, by which time the whole scale of everything had increased. The difficulty of mastering the extended grammatical techniques used in defining it meant that few but specialists gave the report more than a single shuddering glance, and so in 1971 an official informal introduction appeared. As a result of the 'field trials' stage, a revised report appeared in 1974. This revision went further than the ALGOL 60 revision did; it made significant improvements in the language (as well as extensive additions to the descriptive techniques). The 1971 introduction was no longer an accurate one, and the work here under review is its consequential revision.

Every serious computer science undergraduate who uses ALGOL 60 has his own (often Xeroxed) copy of the ALGOL 60 report. Because it carries the guarantee not only of the authors but also of the whole of IFIP WG 2.1, this work is nearly as authoritative as the Report itself, and might fulfil the same function for ALGOL 68 were it not far too big to do so. It starts with a sixty page 'Very informal introduction'; one wonders whether authors and publishers have considered making this available by itself to penurious students. As a substitute for the report in this context its only defect, which it shares with the main work, is that it dispenses entirely with any formal grammar. I found it very clearly written. Apart from an unintended and distracting mnemonic in using **fun** rather than **fn** or **func** for **function**, and an unidiomatic usage of **constantly** (using **constantly yielded** for **yielded . . . in the form of a constant**), my only complaint is that I still, after considerable investigation of **scope**, **range** and **reach**, cannot see how a local generator in a for-loop can produce storage that outlives the control variable. (That the work goes too fast for anyone to absorb properly on a single reading is inevitable.) The main work is directed to the specialist student, whether advanced programmer or implementer; it covers the whole language by a profusion of program snippets illustrating each point as it arises. Like the 1971 version, it is organised as a two-dimensional array of sections so that it may be read by rows (each introducing a new development of phrase structure) or by columns (each dealing with a new concept in data structure and its associated notation). The final row consists of more extended examples appropriate to each column. Appendices include a description of the sublanguage ALGOL 68S and of the recently approved stropping conventions.

ALGOL 68 on the defensive is its own worst enemy, and some of this survives in this book. Thus although it is to be commended for restoring to us such familiar terms as **expression** and **statement,** it does its subject no good by suggesting that coercion saves us from writing **widenedtoreal**(i) when everyone knows that all it saves us from is **float**(i). Nevertheless, these are minor criticisms. We have too often learned the hard way that, where the concealments and protections of high level languages are concerned, ignorance is not always bliss. ALGOL 68 has chosen to develop a machine independent way of describing some of what goes on behind the scenes; just enough to restore the bliss without incurring the folly. Consequently this is a work that should be in every computer library, and that will be invaluable to anyone involved with a real understanding of what computing involves.

B. HIGMAN (Lancaster)

Digital Picture Analysis, edited by A. Rosenfeld, 1976; 351 pages. (*Springer-Verlag*, Topics in Applied Physics, Vol. 11, US$29.60)

Visual inspection as a routine task in industry, medicine and commerce represents an enormous labour bill. This fact has fascinated computer scientists and engineers for several decades now and has spawned massive research efforts to automate these tasks with a consequent vast literature. This book is a valuable source of references to that literature except for the singular omission of the substantial body of work on computer interpretation of images of three dimensional scenes. The five substantive chapters are separately authored, and each treats a different topic area. There is an (obligatory) account of character recognition by Professor Ullmann which is poorly illustrated and fails to give a sense of the challenges remaining in that field, e.g. recognition of badly written material like signatures. A chapter by Haralick on the interpretation of images of the land surface, e.g. air photos, does little more than parade the mathematics of decision theory: it gives us no sense of how well that mathematics performs nor of what the state of the art is. Both authors remain remote from the giver of the task. McIlwain's account of work in high energy physics—basically of the automatic analysis of bubble chamber photographs—is at the other extreme. The physics underlying these images is well understood and the recognition techniques are carefully tuned to the explicit recovery of the three dimensional geometry of the particle track. We are presented with large amounts of high energy physics but no general principles of picture analysis emerge. Preston's account of work on the analysis of cell images including work on chromosomes done in the UK by Rutoviztz's MRC group is a massive piece of scholarship, with more than 400 situations.

The best chapter however is that written by Harlow and his colleagues, in which they describe their work in devising computational tools to accomplish automatic diagnosis of a variety of clinical abnormalities—bone malformations, heart disease, brain tumours, etc.—starting from conventional radiographs. The paper is full of interesting problem details, e.g. the ways in which congestion can occur in the heart—an important assessment in diagnosing congenital heart disease. Estimating this from a radiograph involves 'seeing' the heart boundary: a difficult problem made worse by the fact that 'there is some disagreement as to the location of the top of the heart, (so) we chose an arbitrary criterion suggested by radiologists'. The active collaboration between potential users—in this case radiologists—and computer scientist produces something in which we can see how the problem interacts with the hoped for means of its solution, namely the computing system. The collaboration also provides the opportunity to compare results: human v. computer interpretation—an evaluation all too often missing in the engineer's sorties into applications.

Perhaps the most interesting feature of their work however is the clear sense of the variety of processes that have to be programmed into an effect system. In his introduction the book's editor Professor Rosenfeld offers a unifying framework for the subject in terms of four goals that he believes characterise pictorial pattern recognition—matching, classifying, segmenting and recognising. If such a taxonomy is to be useful, then it should inform the designer; while Haralick and Ullmann would, I'm sure, be happy with it Harlow and his colleagues seem to me to have felicitously indulged in what Preston in his chapter disparagingly refers to as 'the adhockery' of shape analysis. But shape and the meanings of shape is what it's all about, isn't it Professor Rosenfeld?

MAX CLOWES (Brighton)

Downloaded from https://academic.oup.com/comjnl/article/20/4/340/393728 by guest on 19 April 2024