changing all subsequent occurrences of p to a new variable r, thereby saving the value of p; and (ii) saving the value of q in a suitable manner. The second task is accomplished very neatly by having one extra node *link* to hold the value of q in its *left* field. By initialising r to *link* and setting $r\uparrow\cdot left$ to q we not only manage to completely duplicate the body of the while loop, thus reducing it to a repeat loop, but also preserve the initial value of q in $link\uparrow\cdot left$. The final program is therefore as follows:

```
procedure buildright(n: integer, t: tree);
  var p, q, r, link: tree;
    x, m, nl, nr: integer;
    S: stack;
  begin new(link); |S| := 0; S ⇐ (n, t);
    repeat (m, p) ⇐ S;
      if m = 0 then p↑·right := nil else
      begin r := link;
        repeat read(x); new(q); q↑·key := x;
          nl := m div 2; nr := m − nl − 1;
          S ⇐ (nr, q);
          m := nl; r↑·left := q; r := q
        until m = 0;
        r↑·left := nil; p↑·right := link↑·left
      end
    until |S| = 0
  end
```

This procedure is essentially the one given by Wirth in (1976).

4. Results and conclusions

One may justifiably ask whether or not the energy spent on recursion elimination leads to significant gains in efficiency. To answer this question, the three versions of the balanced tree procedure were coded in ALGOL 68-R and run on the University of Reading's 1904S computer together with a timing program. The following table shows the time in seconds required to build a balanced tree of 250 nodes. In the table, *build* refers to the recursive version, *build*1 to the iterative version which uses the ref device and *build*2 to the iterative version given in Section 3. Both *build*1 and *build*2 were coded in two ways; in the first the stack S was represented by a structured linked list, while in the second two linear arrays were used. The results were:

build	*build*1	*build*2
0·072	0·081	0·088
	0·066	0·072

Clearly, the table tells a somewhat disappointing story; only the array version of *build*1 managed to beat the recursive procedure and then by only about 10%. The reason is that, although the balanced tree algorithm possesses a running time which is linear in the number of recursive calls, this is completely dominated by the time spent on manipulating the ALGOL 68 heap. Certainly it does not seem a good idea to involve the heap again by representing the stack as a linked list.

Nevertheless, the problem of recursion elimination is a useful vehicle in which to study and broaden our knowledge of program transformations and a practically useful one for the machine code and FORTRAN programmer who remains obliged to manufacture his or her own implementation of recursion.

Acknowledgements

The author would like to thank Dr J. Ogden for many stimulating discussions on the pros and cons of ALGOL 68 as well as making available his timing program. Thanks are also due to Dr J. Roberts for similar discussions on PASCAL.

References

BIRD, R. S. (1977). Notes on recursion elimination, *CACM*, Vol. 20 No. 6, pp. 434-439.
KNUTH, D. E. (1974). Structured programming with goto statements, *ACM Computing Surveys*, Vol. 6, pp. 261-302.
PARTSCH, H. and PEPPER, P. (1976). A family of rules for recursion removal, *Information Processing Letters*, Vol. 5 No. 6, pp. 174-177.
WIRTH, N. (1976). *Algorithms + Data Structures = Programs*, Prentice-Hall.

Book review

Computer-Aided Design of Digital Systems, by D. Lewin, 1977; 313 pages. (*Edward Arnold*, £15·00)

This book will be of interest mainly to students of computer science at a postgraduate level, and to those practising engineers who are already familiar with 'formal' logic design methods. The book surveys the current status of computer aids in the fields of logic network synthesis, logic simulation and logic testing. The subject of system specification, both by means of register transfer languages and by graph theoretic models is also discussed.

The main barrier to the more widespread acceptance of computer aided logic design is the lack of sufficiently powerful algorithms, especially methods applicable to circuits using MSI and LSI components. Most of the algorithms described in this book are orientated toward design using flipflops and discrete NAND and NOR gates; this limits its usefulness to designers of CAD systems in industry, who have to work with the current technology.

The longest chapter in this book (117 pages) covers the topic of logic network synthesis. A large number of algorithms (over 20) are described, for state reduction, state assignment and for implementation of the resulting switching functions in a particular 'logic family'. The algorithms described first appeared in a variety of journals, Ph.D. theses, etc; this book serves the useful purpose of collecting and comparing such a diversity of methods. The algorithms are described in Professor Lewin's usual lucid style and a large number of helpful worked examples are provided. Many of these algorithms suffer severe limitations on the size of problem which they can handle. Unfortunately, little numeric information is provided in this book to indicate to the reader the limitations of each technique.

The chapter on system specification contains an up-to-date survey of hardware description languages and also describes more recent developments, e.g. the use of Petrinets. The subjects of logic simulation for design verification and for test-program validation are treated in rather less detail; for example, the techniques of deductive fault simulation and the use of worst case timing are mentioned but not described in detail. The book concludes with a review of the subjects of logic circuit testing and testable logic design. The book provides a large number of references (nearly 300), and a useful subject and authors index.

D. BUMSTEAD (Poole)