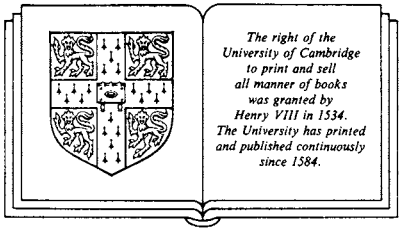
19 Cambridge Computer Science Texts

# Recursion via Pascal

J. S. Rohl

*University of Western Australia*




Cambridge University Press

*Cambridge*

*New York Port Chester*

*Melbourne Sydney*

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia



First published 1984
Reprinted 1986, 1990

Library of Congress catalogue card number: 83-26335

*British Library cataloguing in publication data*

Rohl, J. S.
Recursion via Pascal. – (Cambridge computer science texts; 19)

1. Electronic digital computers – programming
2. Recursive functions
I. Title
001.64'2 QA76.6

ISBN 0 521 26329 8 hardback
ISBN 0 521 26934 2 paperback

Transferred to digital printing 2003

DJ

# Contents

# 1

# Introduction to recursion

What is recursion? It is simply a technique of describing something partly in terms of itself. This notion has wide applicability. We are all used to the idea that an adjectival clause, for example, may contain another adjectival clause. Who has not at sometime or another recited *This is the house that Jack built*?

This is the cock that crowed in the morn
That woke the priest all shaven and shorn
That married the man all tattered and torn
That kissed the maiden all forlorn
That milked the cow with the crumpled horn
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.

On a more prosaic level, if you were asked for the differential with respect to $x$ of $x^2 + 5x$ you would instantly, and correctly, reply $2x + 5$. If you were pressed to explain your answer you would probably reply, firstly, that the differential of $x^2 + 5x$ is equal to the differential of $x^2$ plus the differential of $5x$, and, secondly, that the differential of $x^2$ is $2x$ and of $5x$ is 5. This then is the essence of recursion which consists of two parts:

(i) the *recursive rule*: $\dfrac{\mathrm{d}}{\mathrm{d}x}(x^2 + 5x) = \dfrac{\mathrm{d}}{\mathrm{d}x}(x^2) + \dfrac{\mathrm{d}}{\mathrm{d}x}(5x)$

in which the differential of a sum is defined in terms of the differential of the two terms;

(ii) the *explicitly defined cases*: $\dfrac{\mathrm{d}}{\mathrm{d}x}(x^2) = 2x, \quad \dfrac{\mathrm{d}}{\mathrm{d}x}(5x) = 5$

which terminate the recursion.

For more general expressions, of course, we need further recursive rules, such as those for products and quotients, and more explicitly defined cases, such as that for the differential of a constant. We will return to this example in Chapter 3.

What are the advantages of recursion as a programming technique? From the point of view of this monograph there are four.

(i) For many problems the recursive solution is more natural than the alternative non-recursive solution. Of course naturalness is in the eye of the beholder and for some readers an unfamiliarity with recursion may indeed make the early examples appear unnatural. However the relationship between recursively defined data structures and recursive procedures is very close and by the time trees are introduced in Chapter 3 the appropriateness of recursion will be clear enough.

(ii) It is often relatively easy to prove the correctness of recursive procedures. Inasmuch as recursive procedures are direct transliterations of the mathematical formulations involved, the proofs are often trivial. Even where they are not, the proofs are based on the very familiar process of induction.

(iii) Recursive procedures are relatively easy to analyse to determine their performance. The analysis produces recurrence relations, many of which can easily be solved.

(iv) Recursive procedures are flexible. This is a very subjective statement but, as we demonstrate in Chapter 7 and elsewhere, it is quite easy to convert a general procedure into a more specific one. Indeed this is often a useful design technique: first write a program for a problem which is a generalisation of the given problem, and then adapt it to the problem in hand.

What are the costs to be incurred in using recursion? There are two:

(i) Recursive procedures may run more slowly than the equivalent non-recursive ones. There are two causes for this. Firstly, a compiler may implement recursive calls badly. Most, if not all, Pascal compilers handle recursion quite well and so the cost is small, perhaps 5% to 10%, perhaps nothing. At worst, as we shall shortly show, a recursive procedure may run at half-speed though this applies only to the most trivial procedures. Secondly, the recursive procedures we write may simply be inefficient. It is easy to write such procedures as we shall see, and we must always be on our guard to avoid doing so.

(ii) Recursive procedures require more store than their non-recursive counterparts. Each recursive call involves the creation of an activation record, and if the depth of recursion is large this space

penalty may be significant. This only arises with simple procedures, however: with more complex procedures the depth is small, and, what is more, the non-recursive versions themselves require space which is proportional to the recursive depth. Furthermore, there are some situations where the cost of the recursion, in both time and space, can be eliminated quite simply by a compiler.

With this in mind we now consider some simple examples all of which exhibit *linear recursion.* In these procedures there is only one recursive call. Others, such as the differentiation procedure referred to above, have two recursive calls, and we refer to this as *binary recursion.* Yet others have an indefinite number (the one written call is within a loop), and we refer to this as *n-ary recursion.*

It would be unreasonable to expect that the advantages listed above should appear manifest in simple examples, since that is where recursion is at its weakest. Consequently we will concentrate in this chapter on explaining recursion and how it works and illustrating some of its characteristics.

## 1.1 Some simple examples

The simplest example is the factorial function, which is defined by:

$$p! = 1, \qquad p = 0,$$
$$= 1 \times 2 \times 3 \times \ldots p, \quad p > 0$$

From this definition the function of Fig. 1.1 follows immediately.

Fig. 1.1. A non-recursive function *Fact.*

```
function Fact(p:natural):natural;
  var i,f:natural;
  begin
  f := 1;
  for i := 1 to p do
    f := f*i;
  Fact := f
  end { of function "Fact" };
```

where *natural* is defined as:

type *natural* = 0 . . *maxint*

In a study of the factorial function one of the first theorems proved is:

$$p! = 1, \qquad p = 0,$$
$$= p \times (p-1)!, \quad p > 0$$

from which the function of Fig. 1.2 immediately follows.

Fig. 1.2. A recursive function *Fact*.

```
function Fact(p:natural):natural;
  begin
  if p = 0 then Fact := 1
  else Fact := p*Fact(p-1)
  end { of function "Fact" };
```

Indeed, for many people, the theorem just mentioned *is* the definition. In either case, it must be said that it is hard to argue that either function is more natural than the other.

As a second example, we consider the highest common factor (*HCF*) of two positive integers $p$ and $q$. A description of Euclid's algorithm for finding the HCF usually goes something like this: 'Divide $p$ by $q$ to give a remainder $r$. If $r = 0$ then the HCF is $q$. Otherwise repeat with $q$ and $r$ taking the place of $p$ and $q$'. From this description the non-recursive version of Fig. 1.3 is usually derived.†

Fig. 1.3. A non-recursive version of *Hcf*.

```
function Hcf(p,q:natural):natural;
  var r:natural;
  begin
  r := p mod q;
  while r <> 0 do
    begin
    p := q;
    q := r;
    r := p mod q
    end;
  Hcf := q
  end { of function "Hcf" };
```

From the same description, the recursive version of Fig. 1.4 follows.

Fig. 1.4. A recursive version of *Hcf*.

```
function Hcf(p,q:natural):natural;
  var r:natural;
  begin
  r := p mod q;
  if r = 0 then Hcf := q
  else Hcf := Hcf(q,r)
  end { of function "Hcf" };
```

This is more natural in the sense that $p$ **mod** $q$ is evaluated in only one place, as in the description, whereas in Fig. 1.3 it is evaluated twice.

† As $q$ must not be 0 we should introduce a type *positive* = 1 . . *maxint* for it. Since we will give a version later in which $q$ may be 0, we do not do so.

Mathematically we can formulate this as:

$$hcf(p, q) = p, \qquad p \bmod q = 0,$$
$$= hcf(q, p \bmod q), \quad p \bmod q \neq 0$$

These two examples are fairly well known. As a third example Fig. 1.5 gives a procedure, rather than a function, which prints out an unsigned integer left-justified, that is, with no spaces preceding the most significant digit.

Fig. 1.5. A procedure for writing an unsigned integer left-justified.

```
procedure WriteNatural(i:natural);
  begin
  if i < 10 then
    write(chr(i + ord('0')))
  else
    begin
    WriteNatural(i div 10);
    write(chr(i mod 10 + ord('0')))
    end
  end { of procedure "WriteNatural" };
```

Its action is fairly clear. If $i$ is less than 10, it has only one digit which is printed. If it is greater than 10 (say 375), the procedure is called recursively to print $i$ **div** 10 (here 37) after which the final digit (5) is printed.

## 1.2 How does recursion work?

The standard run-time storage organisation used in Pascal to ensure the optimal use of store is the *stack*; and this organisation automatically encompasses recursion. We will illustrate this with respect to a program, *Test*, which simply reads $x$ and calls *WriteNatural* to print it. We assume that the *activation record* for a procedure contains, as well as the parameters and local variables, two links. The first, the *return address link* (*ral*), holds the address to which control is to be returned on exit from the procedure. The second is called the *stack link* (*sl*), because it is used to ensure that the stack returns to the same configuration on exit from a procedure as it had on entry. We assume that the stack is accessed by a set of registers, called the *display*, one register being associated with each textual level. In what follows we call them $D1, D2, \ldots$. On entry to a procedure one of the display registers has to be altered to refer to the variables of this procedure. If the procedure is at level $n$, then $Dn$ is changed. It is the original value of this register that is the stack link.

A procedure call then must:

(i) stack the return address link,
(ii) stack the stack link,
(iii) adjust the display,
(iv) allocate space for the local variables,
(v) branch to the code of the called procedure.

The corresponding procedure exit then:

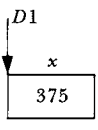
(i) recovers the space of the local variables,
(ii) adjusts the display using the stack link,
(iii) returns to the statement after the call using the return address link.

We illustrate this with respect to the *Test* program mentioned earlier which we give as Fig. 1.6. Note that two points are marked $\alpha$ and $\beta$ by means of comments.

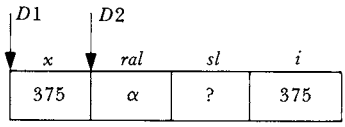
Fig. 1.6. A program to test *WriteNatural.*

```
program Test(input,output);
type natural = 0..maxint;
var x:natural;

procedure WriteNatural(i:natural);
  begin
  if i < 10 then
    write(chr(i + ord('0')))
  else
    begin
    WriteNatural(i div 10);
    { point β }
    write(chr(i mod 10 + ord('0')))
    end
  end { of procedure "WriteNatural" };

begin
read(x);
write(' The value of ',x:1,' is ');
WriteNatural(x)
{ point α }
end.
```

Suppose we run this program with 375 as data. Within the main program there is only one activation record addressed via $D1$. It contains only the variable $x$ since the concept of links is irrelevant for the main program. After *read*($x$) we have:

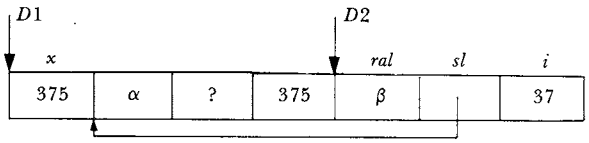


On entry to *WriteNatural* after the call *WriteNatural*($x$), an activation record is created for *WriteNatural* containing the links (*ral* and *sl*) and the parameter $i$. It is addressed via $D2$.

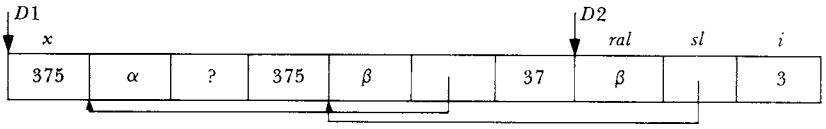



Note that the stack link is irrelevant, since within the main program $D2$ is unused.

On the second entry to *WriteNatural*, as a result of the recursive call *WriteNatural*($i$ **div** 10), a further activation record is created for *WriteNatural*. It is accessed via $D2$, while the previous activation record becomes temporarily inaccessible.

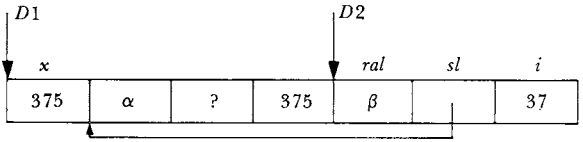



On the third entry to *WriteNatural* we have:



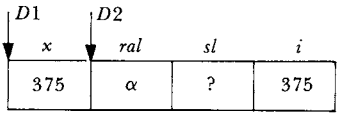

and *chr*($i$ + *ord*('0')), that is the character 3, is then printed.

On exit from this activation of *WriteNatural*, the stack is returned to its previous state so that the second activation record becomes accessible again, and control returns to point $\beta$.



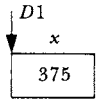

Then *chr*($i$ **mod** 10 + *ord*('0')), that is, the character 7, is printed.

On exit from this activation we have:




and, as control returns again to point $\beta$, 5 is printed.

On the exit from the first activation to *WriteNatural* the stack returns to:




and control returns to $\alpha$, at which point the program stops.

### 1.3 The storage cost of recursion

From the description of the implementation, the cost in terms of storage associated with recursive procedures is clear. If $n$ is the maximum recursive depth, then the store required is $n \times (p + l + 2)$ where $p$ represents the space required by the parameters and $l$ that required by the local variables. Where the alternative non-recursive solution requires only a small number of local variables for its operation, this cost might be significant. (In the two relevant examples given so far, *Fact* and *Hcf*, $n$ is likely to be small but in Chapter 2 we consider situations where $n$ may be large.)

There are, however, some situations where the non-recursive procedure requires an amount of store which is proportional to $n$, in which case the comparison between recursive and non-recursive versions may be less clear-cut. In these situations the extra store is used as a stack† and we will assume that some appropriate facilities have been added to Pascal. This is simply a matter of abstraction: the implementation of the facilities in pure Pascal is trivial.

We assume a new structured mode, **stack of**, so that, for example, the declaration:

**var** *s*:**stack of** *natural*

declares *s* to be a stack of natural numbers. This stack is initialised, to an empty stack by:

**clear** *s*

Only two accessing statements are available. The first:

**push** *i* **onto** *s*

pushes the value of the expression *i* onto the top of *s*, while:

**pop** *i* **from** *s*

pops the top value from *s* and assigns it to *i*. Finally:

*s* **empty**
*s* **not empty**

are predicates which test the state of the stack.

† The term *stack* thus refers to two concepts which are alike in their *first-in, last-out* characteristics but have different rules of access.

Fig. 1.7 gives a non-recursive version of *WriteNatural* using these facilities.

Fig. 1.7. A non-recursive version of *WriteNatural*.

```
procedure WriteNatural(i:natural);
  var s:stack of natural;
  begin
  clear s;
  while i >= 10 do
    begin
    push i onto s;
    i := i div 10
    end;
  write(chr(i + ord('0')));
  while s not empty do
    begin
    pop i from s;
    write(chr(i mod 10 + ord('0')))
    end
  end { procedure "WriteNatural" };
```

Clearly in *WriteNatural* the size of the stack will be small†, perhaps 5 or 6, but the general principle is clear: the amount of store required is proportional to the recursive depth, though as there will be fewer links required (here there are none) the constant of proportionality will be smaller than that for the recursive version.

Fig. 1.7 illustrates another point: that the procedures themselves occupy space and the differences in procedure size must be considered. These are generally of a lower order, since there is only one copy of a procedure code, whereas there may be many activation records.

## 1.4 The time cost of recursion

We indicated in the opening paragraphs of this chapter that even where they have been well written, recursive procedures may run more slowly than their non-recursive counterparts. We illustrate this here by using what is perhaps the most extreme example, the factorial functions given earlier. In Fig. 1.8 we give counts of those of the so-called *structured operations* that are involved: arithmetic, assignment, loop traverse, procedure call and so on.

We also count the number of *elementary operations* by assigning appropriate weights to the structured operations: arithmetic, simple tests and assignments at 1, for-loop entry at 2 (for the assignment

† Indeed for this particular example we could avoid the use of a stack by trading space for time, and using quite a different technique.

Fig. 1.8. Analysis of the *Fact* functions.

| Number of operations of the type | Non-recursive (Fig. 1.1) | Recursive (Fig. 1.2) |
|---|---|---|
| Arithmetic | $p$ | $2p$ |
| Assignment | $p+2$ | $p+1$ |
| Test | | $p+1$ |
| Parameter evaluation | 1 | $p+1$ |
| Procedure call and exit | 1 | $p+1$ |
| For-loop entry | 1 | |
| For-loop traverse | $p$ | |
| Elementary operations | $5p+10$ | $10p+8$ |
| Elementary operations ($p$=10) | 60 | 108 |
| Time on Cyber 73 ($p$=10) | 210 $\mu$s | 380 $\mu$s |

and test involved), for-loop traverse at 3 (for the test, increment and assignment involved), parameter evaluation at 1 (for the implied assignment) and procedure call and exit at 5 (for assigning two links and setting the display register on entry, resetting two links on exit). From Fig. 1.8 we see that the recursive procedure is perhaps twice as slow.† This is probably an upper limit on the differences between a linear recursive procedure and the equivalent non-recursive version because the body of *Fact* is quite trivial.

Fig. 1.8 gives as well some timings for the procedures run on a Cyber 73, as do subsequent tables. The figures indicate that the model is a fair approximation to the Cyber Pascal system. The discrepancies arise from the simplicity of the model and from the relative inaccuracy of the timer used.

## 1.5 Recurrence relations

The analysis of most of the procedures considered in this chapter and the next (those exhibiting linear recursion) is very simple and really needs no formalism. However this is not so with binary and $n$-ary recursion, and so we will consider an analysis based on the use of a *recurrence relation.* It is convenient to have the notion of the *size* of a problem, so that if $T_k$ represents the cost, however defined, of evaluating a procedure of size $k$ then the recurrence relation defines $T_k$ in terms of the cost of evaluating the smaller problem(s) into which it is broken down. For linear recursion the size is closely related to the recursive depth and $T_k$ is defined in

† This set of weights is very arbitrary and may not be appropriate to some machines and some compilers, particularly where procedures are handled by a subroutine call.

terms of $T_{k-1}$. A typical recurrence relation, which applies to *Fact*, is:

$$\begin{aligned} T_k &= b + T_{k-1}, \quad k > 0 \\ &= a, \qquad\quad k = 0 \end{aligned}$$

where $a$ and $b$ are appropriate constants. $T_n$ can be determined quite simply by a process of substitution.

$$\begin{aligned} T_n &= b + T_{n-1} \\ &= b + b + T_{n-2} \\ &= b \times 2 + T_{n-2} \\ &\quad\vdots \\ &= b \times n + T_0 \\ &= bn + a \end{aligned}$$

This is linear in $n$ which coincides nicely with our use of the phrase linear recursion. It is not the only form of recurrence relation that arises in linear recursion as we shall see. However, in all recurrence relations that do arise, the coefficient of $T$ on the right-hand side is always 1.

## 1.6 The choice of the explicitly defined case

We want now to consider in the next two sections two aspects which are important in the design of recursive procedures. Firstly the choice of the explicitly defined case. There is often some flexibility in this choice. For example, we have chosen 0! = 1 as the explicitly defined case in the factorial function. We might have chosen 1! = 1 as in Fig. 1.9; and provided we always called *Fact* with a parameter >0 it would have operated successfully.

Fig. 1.9. The function *Fact* modified to use 1! = 1.

```
function Fact(p:natural):natural;
  begin
  if p = 1 then Fact := 1
  else Fact := p*Fact(p-1)
  end { of function "Fact" };
```

But note the implication that two functions for the same problem with different explicitly defined cases are different in that one function might fail in cases where the other does not. For example the evaluation of *Fact*(0) using Fig. 1.9 would fail as $p$ went out of range!†

† As we noted in §1.1 with respect to the parameter $q$ of *Hcf*, it would be better to define $p$ to be of the type *positive*.

Considering the example *Hcf*, if we stop the recursion one step later, that is when $q = 0$ rather than when $p \bmod q = 0$, we produce the elegant function of Fig. 1.10.

Fig. 1.10. A function *Hcf* stopping one step later.

```
function Hcf(p,q:natural):natural;
  begin
  if q = 0 then Hcf := p
  else Hcf := Hcf(q,p mod q)
  end { of function "Hcf" };
```

Note that the local variable $r$ has disappeared. Note, too, that this function gives an interpretation to $Hcf(7, 0)$ where the previous one did not.

The recurrence relation enables us to determine the effect of the change. For the new version of *Fact* we have:

$$T_k = b' + T_{k-1}, \quad k > 1$$
$$= a', \quad k = 1$$

Note we have used constants $a'$ and $b'$ since they will in general be different from $a$ and $b$, even though this is not true for the factorial functions. The solution is simply:

$$T_n = b'n + (a' - b')$$

Which is the faster depends on the values of $a$, $b$, $a'$ and $b'$. In any event the different will be small. Thus the choice of explicit case is usually made on the grounds of elegance or simplicity or generality. When we consider binary recursion, the difference, however, may turn out to be significant.

## 1.7 Two-level procedures

The second aspect is the use of *two-level procedures*, in which the main procedure contains within itself a procedure which is recursive and which it calls initially. This technique has a number of advantages which we now consider.

It is clear from the discussion of costs that the number of parameters is significant in that it affects both space and time requirements. Consider a function for evaluating the polynomial:

$$a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n$$

This is usually evaluated by Horner's method of nested multiplication:

$$(\ldots(((a_0)x + a_1)x + a_2) \ldots + a_{n-1})x + a_n$$

Fig. 1.11 gives a function in which the coefficients are assumed to be in an array $a$.†

† Very often, as here, we will leave some types unspecified, where it is clear what an appropriate definition might be.

Fig. 1.11. A non-recursive version of *Poly*.

```
function Poly(var a:coeff; x:real; n:natural):real;
  var y:real;
      i:natural;
  begin
  y := 0;
  for i := 0 to n do
    y := y*x + a[i];
  Poly := y
  end { of function "Poly" };
```

Note that we have called *a* as a variable even though it serves only to transmit a value to *Poly*. The reason is simply one of efficiency. Since each element of *a* is accessed only once, the cost of copying the whole array (which calling it by value would involve) is more than the cost of the indirect access (which calling as a variable implies). Further we require less space, since here it requires a single location (for the indirect address) whereas it would require space for a copy if it were called by value. We will use this criterion for the choice between call-by-value and call-as-a-variable extensively in this book.

The standard recursive version also follows directly from Horner's re-arrangement as Fig. 1.12 shows.

Fig. 1.12. A recursive version of *Poly*.

```
function Poly(var a:coeff; x:real; n:natural):real;
  begin
  if n = 0 then Poly := a[0]
  else Poly := Poly(a,x,n-1)*x + a[n]
  end { of function "Poly" };
```

Here *a* and *x* are unaltered between calls, and we consume both time and space for them on each recursive call.

To avoid repeatedly assigning these redundant parameters we can use a two-level approach as shown in Fig. 1.13.

Fig. 1.13. The two-level function *Poly*.

```
function Poly(var a:coeff; x:real; n:natural):real;

  function P(k:natural):real;
    begin
    if k = 0 then P := a[0]
    else P := P(k-1)*x + a[k]
    end { of function "P" };

  begin
  Poly := P(n)
  end { of function "Poly" };
```

Here the body of the outer procedure *Poly* contains simply a call to the inner procedure *P* with just the one parameter *k* which is initialised to *n*. Within *P* the values of *a* and *x* are accessed non-locally. We will use these two-level functions (and procedures) quite extensively in this book and, by convention, we will generally give the inner function (or procedure) a name which is the first letter of the name of the outer one, unless that happens to have a name which starts with a prefix which is common to a group of procedures.

This function certainly uses less space since the inner recursive function has only one parameter. The stack space we require is 5 locations for the outer function plus $3(n+1)$ for *P*, as against $5(n+1)$ for the single-level recursive function. (Of course, the non-recursive function requires only 7 locations for the parameters and the local variables.)

An analysis of all three functions is given in Fig. 1.14. It shows that the two-level recursion requires fewer operations than the one-level recursive function, but more than the non-recursive one. However some of the operations involve non-local accesses which the model assumes to be no more costly than local ones. This is a fairly simplistic assumption, and Fig. 1.14 shows that it is not appropriate for the Cyber.

Fig. 1.14. An analysis of the *Poly* functions.

| | Wt | Non-recursive (Fig. 1.11) | Recursive (Fig. 1.12) | Two-level (Fig. 1.13) |
|---|---|---|---|---|
| Arithmetic | 1 | $2n+2$ | $3n$ | $3n$ |
| Assignment | 1 | $n+3$ | $n+1$ | $n+2$ |
| Subscripting | 1 | $n+1$ | $n+1$ | $n+1$ |
| Test | 1 | | $n+1$ | $n+1$ |
| Parameter evaluation | 1 | 3 | $3n+3$ | $n+4$ |
| Procedure call and exit | 5 | 1 | $n+1$ | $n+2$ |
| For-loop entry | 2 | 1 | | |
| For-loop traverse | 3 | $n+1$ | | |
| Elementary operations | | $7n+19$ | $14n+11$ | $12n+18$ |
| Elementary operations ($n$=10) | | 89 | 151 | 138 |
| Time on Cyber 73 ($n$=10) | | 350 $\mu$s | 540 $\mu$s | 540 $\mu$s |

However the two-level solution has other advantages which are indisputable. Firstly, it enables us to maintain an acceptable interface to the user. For example suppose we wished to write a procedure to evaluate the polynomial:

$$a_0 + a_1x + a_2x^2 + \ldots + a_n^n$$

(The one used earlier was $a_0x^n + a_1x^{n-1} + \ldots a_n$, so we will call this

*PolyUp*, reflecting that the coefficients are increasing along the polynomial.) Using Horner's method we evaluate:

$$(\ldots(((a_n)x + a_{n-1})x + a_{n-2})\ldots + a_1)x + a_0$$

A one-level procedure requires an extra parameter as Fig. 1.15 shows.

Fig. 1.15. A one-level function *PolyUp*.

```
function PolyUp(var a:coeff; x:real; i,n:natural):real;
  begin
  if i = n then PolyUp := a[n]
  else PolyUp := PolyUp(a,x,i+1,n)*x + a[i]
  end { of function "PolyUp" };
```

This means that the user requires an extra (to him, useless) parameter in each call such as *PolyUp*$(a, x, 0, n)$.

The two-level function enables us to retain the usual function heading as shown in Fig. 1.16.

Fig. 1.16. A two-level function *PolyUp*.

```
function PolyUp(var a:coeff; x:real; n:natural):real;

  function P(i:natural):real;
    begin
    if i = n then P := a[n]
    else P := P(i+1)*x + a[i]
    end { of function "P" };

  begin
  PolyUp := P(0)
  end { of function "PolyUp" };
```

Secondly, the use of a two-level solution enables us to accommodate special cases quite simply. Consider a function *Power* whose value is the $n$th power of $x$ with the added constraint that $0^n$ is 0. Fig. 1.17 gives a single-level function.

Fig. 1.17. A poor function for *Power*.

```
function Power(x:real; n:integer):real;
  begin
  if x = 0 then Power := 0
  else if n < 0 then Power := 1/Power(x,-n)
  else if n = 0 then Power := 1
  else Power := x*Power(x,n-1)
  end { of function "Power" };
```

Note that on each call $x$ is compared with 0, even though, if it is different from 0 on the first call, it will remain different from 0 for all calls. Similarly $n$ is tested to ensure it is not less than 0 at each

call, when, if it were negative initially, its value would have been immediately negated. The two-level solution of Fig. 1.18 avoids this by dealing with these cases in the outer procedure.

Fig. 1.18. A two-level function for *Power*.

```
function Power(x:real; n:integer):real;

  function P(k:natural):real;
    begin
    if k = 0 then P := 1
    else P := x*P(k-1)
    end { of function "P" };

  begin
  if x = 0 then Power := 0
  else if n < 0 then Power := 1/P(-n)
  else Power := P(n)
  end { of function "Power" };
```

Note that this is the recursive equivalent of moving constants outside loops.

## 1.8 Developing the power example: a cautionary tale

The powering procedures implemented the definition:

$$\begin{aligned} x^n &= 0, && x = 0 \\ &= 1/x^{-n} && x \neq 0, n < 0 \\ &= 1, && x \neq 0, n = 0 \\ &= x \times x^{n-1}, && x \neq 0, n > 0 \end{aligned}$$

As many readers will have noticed, this procedure is not very efficient for large $n$. It is $O(n)$ whereas the method often called 'halving and squaring' is $O(\log n)$. This technique calculates $x^{14}$, for example, by squaring $x^7$ whereas the original multiplies $x$ by itself 13 times.

Formally we can specify the function:

$$\begin{aligned} x^n &= 0, && x = 0 \\ &= 1/x^{-n}, && x \neq 0, n < 0 \\ &= 1, && x \neq 0, n = 0 \\ &= x^{n/2} \times x^{n/2} \times x, && x \neq 0, n \text{ odd} \\ &= x^{n/2} \times x^{n/2}, && x \neq 0, n \text{ even and} > 0 \end{aligned}$$

From this the function of Fig. 1.19 is easily produced.

Fig. 1.19. A faster version of *Power*.

```
function Power(x:real; n:integer):real;

  function P(k:natural):real;
```

```
    begin
    if k = 0 then P := 1
    else if odd(k) then P := sqr(P(k div 2))*x
    else P := sqr(P(k div 2))
    end { of function "P" };

begin
if x = 0 then Power := 0
else if n < 0 then Power := 1/P(-n)
else Power := P(n)
end { of function "Power" };
```

The analysis of Fig. 1.19 is a little more difficult than those considered previously because of the different actions taken depending on whether $k$ is even or odd. However, the difference is small and we can, as an approximation, assume that $k$ is equally likely to be even or odd. The recurrence relation is:

$$\begin{aligned} T_k &= b + T_{\lfloor k/2 \rfloor}, & k > 0 \\ &= a, & k = 0 \end{aligned}$$

where $\lfloor k/2 \rfloor$, the *floor* of $k/2$, is the largest integer less than $k/2$. We can solve this for $T_n$ again by simple substitution:

$$\begin{aligned} T_n &= b + T_{\lfloor n/2 \rfloor} \\ &= b + (b + T_{\lfloor n/4 \rfloor}) \\ &= 2b + T_{\lfloor n/4 \rfloor} \\ &= 2b + (b + T_{\lfloor n/8 \rfloor}) \\ &= 3b + T_{\lfloor n/8 \rfloor} \end{aligned}$$

We can see that, as $n$ is progressively halved, the coefficient of $b$ is increased by 1. Thus we ultimately arrive at:

$$\begin{aligned} T_n &= b\lfloor \log n \rfloor + T_1 \\ &= b\lfloor \log n \rfloor + b + T_0 \\ &= b\lfloor \log n \rfloor + (b + a) \end{aligned}$$

This is only the cost of the call of $P$, of course. We must also add the small cost of the body of *Power*.

This derivation suggests other alternatives, such as stopping the recursion one step earlier (where $k = 1$) and modifying the body of *Power* appropriately. Note that this illustrates another advantage of a two-level procedure: we can stop the recursion earlier without needing to alter the specification. We leave it to the reader to pursue this solution.

We indicated earlier that it is trivially easy to write inefficient recursive procedures. Here is a case in point. Suppose we unthinkingly

used explicit multiplication instead of squaring as shown in Fig. 1.20.

Fig. 1.20. A bad version of *Power*.

```
function Power(x:real; n:integer):real;
  function P(k:natural):real;
    begin
    if k = 0 then P := 1
    else if odd(k) then P := P(k div 2)*P(k div 2)*x
    else P := P(k div 2)*P(k div 2)
    end { of function "P" };
  begin
  if x = 0 then Power := 0
  else if n < 0 then Power := 1/P(-n)
  else Power := P(n)
  end { of function "Power" };
```

Unless we have a compiler which can recognise that the multiplications can be replaced by squarings, we find that the procedure is actually worse than the original two-level solution of Fig. 1.18. This is because, at each level, $P$ is called twice. The recurrence relation is:

$$\begin{aligned} T_k &= b + 2T_{\lfloor k/2 \rfloor}, & k \neq 0 \\ &= a, & k = 0 \end{aligned}$$

whose solution is $(a+b)\bar{n} - a$, where $\bar{n}$ is $2^{\lfloor \log n \rfloor + 1}$, that is the smallest power of 2 which is greater than $n$.

Fig. 1.21 gives a detailed analysis of these procedures, including the body of *Power*, in which $n$ is the absolute value of the parameter, which is assumed as likely negative as positive.

Fig. 1.21. Analysis of the *Power* functions.

| | | One-level | | Two-level | |
|---|---|---|---|---|---|
| | Wt | (Fig. 1.17) | (Fig. 1.18) | (Fig. 1.19) | (Fig. 1.20) |
| Arithmetic | 1 | $2n+1$ | $2n+1$ | $2\frac{1}{2}\lfloor \log n \rfloor + 3\frac{1}{2}$ | $3\bar{n}+n-2$ |
| Assignment | 1 | $n+1\frac{1}{2}$ | $n+2$ | $\lfloor \log n \rfloor + 3$ | $2\bar{n}$ |
| Test | 1 | $3n+4$ | $n+3$ | $2\lfloor \log n \rfloor + 5$ | $3\bar{n}$ |
| Parameter evaluation | 1 | $2n+3$ | $n+3$ | $\lfloor \log n \rfloor + 4$ | $2\bar{n}+1$ |
| Procedure call and exit | 5 | $n+1\frac{1}{2}$ | $n+2$ | $\lfloor \log n \rfloor + 3$ | $2\bar{n}$ |
| Elementary operations | | $13n+17$ | $10n+19$ | $11\frac{1}{2}\lfloor \log n \rfloor + 30\frac{1}{2}$ | $20\bar{n}+n-1$ |
| Elementary operations ($n$=240) | | 3137 | 2419 | 111 | 5359 |
| Time on Cyber 73 ($n$=240) | | 10200 $\mu$s | 8700 $\mu$s | 400 $\mu$s | 20900 $\mu$s |

Note that this small error has changed the order of complexity of the procedure from $O(\log n)$ to $O(n)$. Note, too, that the possibility of such a drastic effect for such a trivial change does not usually occur with iterative procedures.

## 1.9 Searching

One of the fundamental operations of computer science is searching for an item of a given key in a collection of such items. We assume that the items are of a type *itemtype* defined:

type *itemtype* = **record**
*key* : *keytype*;
*info* : *infotype*
**end**

where both *keytype* and *infotype* are left unspecified.

Let us assume that the items are held in an array whose type is defined by:

type *sizetype* = 1 . . *max*;
*arraytype* = **array** [*sizetype*] **of** *itemtype*

where *max* is an appropriate constant.

Let us assume that the items are not ordered on their keys. In Fig. 1.22 we give an obvious function which proceeds through the array until either the key is found, or all items have been compared.

Fig. 1.22. Searching an array.

```
function InArray(var a:arraytype; n:sizetype; k:keytype):Boolean;

  function I(j:sizetype):Boolean;
    begin
    if k = a[j].key then I := true
    else if j = n then I := false
    else I := I(j+1)
    end { of function "I" };

  begin
  InArray := I(1)
  end { of function "InArray" };
```

On average half the elements will be compared so that the function is $O(n)$.

If the items are held in ascending order of their keys we can do much better by using the method known as *binary-chopping*, which operates as follows. We compare the key of the item being sought with the key of the item in the middle of the array. If it is the smaller, then the item, if it is present, must be in the lower half of

the array; otherwise it must be in the upper half. Fig. 1.23 gives an appropriate function.

Fig. 1.23. Binary-chopping.

```
function InArray(var a:arraytype; n:sizetype; k:keytype):Boolean;

  function I(l,u:sizetype):Boolean;
    var mid:sizetype;
    begin
    if l = u then I := k = a[l].key
    else
      begin
      mid := (l+u) div 2;
      if k <= a[mid].key then I := I(l,mid)
      else I := I(mid+1,u)
      end
    end { of function "I" };

  begin
  InArray := I(1,n)
  end { of function "InArray" };
```

Clearly this procedure is $O(\log n)$ since at each stage the size of the array is halved.

## 1.10 Recursion and reversal

The procedure *WriteNatural* prints out the natural number which is its parameter in the usual way: the procedure *WriteReversedNatural* of Fig. 1.24 prints it out in reverse. That is, if $i = 375$, it prints 573.

Fig. 1.24. A procedure for writing natural numbers reversed.

```
procedure WriteReversedNatural(i:natural);
  begin
  if i < 10 then
     write(chr(i + ord('0')))
  else
    begin
    write(chr(i mod 10 + ord('0')));
    WriteReversedNatural(i div 10)
    end
  end { of procedure "WriteReversedNatural" };
```

The only difference between the procedures is the position of the recursive call: in *WriteNatural* it occurs before the writing of a character, in *WriteReversedNatural* it occurs after. Thus it is often trivial to modify a recursive procedure to produce a reversed form of output - and to accept a reversed form of input. We shall see a useful example in Chapter 2.

With non-recursive procedures the changes are less trivial. In Fig. 1.25 we give an iterative procedure for *WriteReversedNatural.*