Partial non-underflow and non-overflow of an arithmetic stack

D. T. Goodwin

Department of Computer Science, University of Keele, Newcastle, Staffs., ST5 5BG

This paper continues analysis of the model of arithmetic stack behaviour set up in Goodwin (1977). There conditions computable at high level language compile time were given under which run time stack overflow or underflow could not occur. Because the conditions are strict, it is worth considering the question 'Is it possible for no overflow nor underflow to occur, even if the conditions do not apply?' A number of results are presented here, including an algorithm for finding the smallest stack size, if any, which permits an infinity of different program runs to take place.

(Received August 1976; revised July 1978)

This paper is a sequel to Goodwin (1977) in which stack non-overflow and non-underflow conditions are given when strings of a CF language are loaded symbol by symbol on to a stack. A particular symbol exists whose instances each cause the deletion of the top cell of the stack, in some undefined way. This model can easily be applied to the operation of a run time arithmetic stack, where the particular source program acts as the grammar of the CF language, and transfer-to-store instructions or arithmetic operations correspond to cell removal. A disadvantage of the conditions is that they are strict and only answer the question 'Is the program sure to run?' The author suspects that many useful programs would not satisfy this test. This paper considers the question 'Might the program run?' in the sense of whether it can be decided for any CF language whether some finite or infinite number of its strings cause no overflow nor underflow. The answer to this question could conceivably justify a compile time decision either to reject the program outright, or to run it with code inserted for over- and underflow checks at execution time, if no suitable interrupt were available. The former article discussed what high level language programs could be analysed in this way. Certainly all ALGOL 60 and FORTRAN programs are included.

It must be stressed that the case where only a finite number of strings cause no overflow nor underflow is not unduly restrictive and would not by itself justify rejection of the program. Each language string is a stream of variable references, constants and operators which is generated at run time and corresponds to the complete execution of the program. However the language string disappears symbol by symbol as it is generated. Each variable reference at once signals either a fetch to the stack or a transfer-to-store from the stack. Similarly a number input function would place its result on the stack. The operators immediately stimulate operations on the top cells of the stack. It may now be seen that a single particular language string could be used in an 'almost infinite' number of program runs, depending on the particular values of data supplied. Here the word 'almost' is only necessary because in practice a numeric variable can only take one of a finite, though very large, number of possible values, depending on its internal representation.

The question 'Might the program run?' is carefully phrased, and cannot be equated with 'Does the program have a good chance of running?' or 'Is it likely to run?'. These are matters of probability which can only be settled by considering the meanings, and hence the patterns of usage, of strings in the language. This difficult problem is not dealt with at all.

The algorithms of the sections below are often laborious and are given just to demonstrate computability. No claims for efficiency are made.

Notation and assumptions

This paper relies so heavily on the concepts, terminology and notation of the previous article that no further introduction is made here. For ease of reference the numbering of the theorems and lemmas is continued from that paper. Unless stated to the contrary no assumptions are made in what follows as to the existence of \( l^+(S) \), \( l^-(S) \), \( m^+(S) \) and \( m^-(S) \).

Shortest stack length

First considered is the question 'What is the shortest stack length necessary to accommodate some string of \( G \) throughout the whole of its deposition on the stack?' This problem turns out to be computable and hence answers the simpler question 'Can any string of \( G \) be accommodated using a stack of given length \( L \) ?' At the moment only the upper bound is considered — thus strings of \( G \) which use only a few cells of the stack are deemed acceptable even though they may make the stack length negative.

Definitions

For an arbitrary string \( s \) of terminals, let \( m\text{u}(s) \) be the maximum number of stack cells used at any time during the deposition of \( s \). Then \( m\text{u}(s) \geq -1 \) for all \( s \).

For an arbitrary non-terminal \( N \), let \( N \Rightarrow s_N \) where \( s_N \) is a terminal string. Then let

\[
m\text{u}(N) = \min_{\text{all } s_N} (m\text{u}(s_N)).
\]

\( m\text{u}(N) \) clearly exists whatever the nature of \( G \), and whether there are a finite or infinite number of \( s_N \).

Similarly for any string \( u \), where \( u \Rightarrow s_u \), let

\[
m\text{u}(u) = \min_{\text{all } s_u} (m\text{u}(s_u)).
\]

Now let \( m\text{u}(N) \) be the least maximum stack length used by all strings \( s_N \) where the derivation \( N \Rightarrow s_N \) begins with the \( i \)'th rule of \( N \). (Let this rule be \( N \Rightarrow e_1 \ldots e_j \ldots e_n \). Then

\[
m\text{u}(N) = \min_{\text{all } s_u} (m\text{u}(s_u)).
\]

Consider how this least maximum can arise in the deposition of terminal strings derived from \( e_1 \ldots e_j \ldots e_n \). From an argument similar to that used to derive the formula for the \( m^+(N) \),

\[
m\text{u}(N) = \min_{\text{all } s_u} \left\{ \begin{array}{ll}
m\text{u}(e_1), \\
m\text{u}(e_1) + m\text{u}(e_2), \\
\vdots \\
m\text{u}(e_1) + \ldots + m\text{u}(e_j), \\
\vdots \\
m\text{u}(e_1) + \ldots + m\text{u}(e_j) + m\text{u}(e_{j+1}), \\
\vdots \\
m\text{u}(e_1) + \ldots + m\text{u}(e_j) + m\text{u}(e_{j+1}) + \ldots + m\text{u}(e_n)
\end{array} \right\}
\]
This minimum is harder to compute than \( m^*(N) \). In that case the \( 1^* \) are maxima which are known beforehand, while here the minimum may arise for any length \( 1(c_j) \) of a terminal string of \( c_j \). It is desired to vary all the lengths involved so that the maximum stack length arising from any line of the formula is minimised.

Consider the \( n \)th line \( 1(c_1) + \ldots + 1(c_{n-1}) + mU(c_n) \). No other line of the formula involves \( c_n \), so the minimum \( mU(c_n) \) can be used, and regarded as a constant when considering other \( c_j \). Now the only lines to involve \( c_{n-1} \) are the \((n-1)\)th and \( n \)th, viz.

\[
\sum_{i=1}^{n-2} (c_i) + mU(c_{n-1}) + \sum_{i=1}^{n-2} (c_i) + 1(c_{n-1}) + mU(c_n).
\]

So it is necessary to choose \( s_{n-1} \), \( c_{n-1} \Rightarrow s_{n-1} \), so that \( mU(s_{n-1}) + 1(c_{n-1}) + mU(c_n) \) is minimised, where \( mU(c_n) \) is a constant. For such \( s_{n-1} \) it is not always true that \( mU(s_{n-1}) = mU(c_{n-1}) \). This suggests a new function defined as follows:

Let \( mU(u, k) = \min \left\{ \max \left\{ mU(s_u), 1(s_u) + k \right\} \right\} \).

It follows that \( mU(u, 0) = mU(u) \).

Now if \( u \Rightarrow s_u \) and \( v \Rightarrow s_v \), then

\[
mU(uw, k) = \min \left\{ \max \left\{ mU(s_u, 1(s_u) + k) \right\} \right\},
\]

\[
= \min \left\{ \max \left\{ mU(s_u), 1(s_u) + mU(s_u), 1(s_u) + 1(s_u) + k \right\} \right\},
\]

\[
= \min \left\{ \max \left\{ mU(s_u), 1(s_u) + \max(mU(s_u), 1(s_u) + k) \right\} \right\},
\]

\[
= \min \left\{ \min \left\{ \max \left\{ mU(s_u), 1(s_u) + \max(mU(s_u), 1(s_u) + k) \right\} \right\} \right\}.
\]

From the definition of \( mU(u, k) \) it is clear that it increases monotonically with \( k \), so that the above minimum is attained by minimising the second parameter, i.e., \( mU(uw, k) = mU(u, \min \left\{ \max mU(s_u), 1(s_u) + k \right\} \) all \( s_u, s_v \).

A third parameter \( r \) for the \( mU \) function is now introduced. This restricts the terminal strings of \( u \) to those whose generation trees are no more than \( r \) rule-applications deep. Then if \( N \rightarrow c_1 \ldots c_j \ldots c_n \) is the \( i \)th rule of \( N \),

\[
mU(N, k, r + 1) = mU(c_1 \ldots c_j \ldots c_n, k, r)
\]

\[
= mU(c_1 \ldots c_{n-1}, mU(c_{n-1}, k, r), r)
\]

\[
= mU(c_1 \ldots c_{n-2}, mU(c_{n-1}, k, r), r, mU(c_{n-1}, k, r), r, r)
\]

\[
= mU(c_1, mU(c_2, mU(c_3, \ldots mU(c_{n}, k, r), r) \ldots r), r)
\]

and

\[
mU(N, k, r + 1) = \min \left\{ \max \left\{ mU(N, k, r), mU(N, k, r + 1) \right\} \right\}, \quad r \geq 0.
\]

For low \( r \) some of the \( mU \) function values may not exist since they may eventually call for the evaluation of some \( mU(N, k, 0) \) —impossible since \( N \) is a non-terminal.)

Now since the \( c_j \) for all \( j, i \) and \( N \) are either terminals or non-terminals of \( G \), the above relations define a simple recursive procedure for finding the \( mU(N, 0, r + 1) \), and as many of the \( mU(N, k, r + 1) \) as are required for computing the \( mU(N, k, r + 2) \). The use of the \( r \) parameter guarantees termination of the procedure, since it specifies that strings of a certain finite measure of complexity are analysed in terms of simpler ones, until eventually terminal strings are reached. No doubt a much faster iterative method could be found which preserves all intermediate \( mU(c_j) \) values found so that no duplicated evaluations were made.

It still needs to be shown how to terminate the algorithm, having arrived at values for the \( mU(N) \). Certainly

\[
mU(N) = \lim_{r \rightarrow \infty} [mU(N, 0, r)]
\]

because by taking sufficiently large \( r \) every desired terminal string derived from \( N \) will be considered. Also as \( r \) increases, \( mU(N, k, r) \) decreases monotonically and \( mU(N, k, r) \geq mU(N, 0, r) \geq mU(N) \). Termination takes place at a certain \( r \) value when for each \( N, k \)

\[
mU(N, k, r + 1) = mU(N, k, r)
\]

because then the algorithm would give the same values for the \( mU(N, k, r + 2) \) as it had for the \( mU(N, k, r + 1) \). However for this condition to be useful the \( k \) values that can arise must be shown to be bounded for all \( N, i \) and \( r \). This is true for each \( N, i \) because as \( r \) increases \( mU(c_i) \) is bounded, \( mU(c_{i-1}) \), \( mU(c_{i-2}) \), \( mU(c_{i-3}) \), \( r \), \( r \), \( r \), and so on. To avoid unnecessary evaluations just to test for termination the condition could be expressed as

\[
mU(N, 0, r + 1) = mU(N, 0, r)
\]

for all \( N \), and

\[
mU(N, k, r) = mU(N, k, r - 1)
\]

for all \( N \) and all \( k \) already evaluated.

It is also easy to see that by following the progress of the above algorithm one may construct a string of \( G \) which uses only the minimum stack length \( mU(S) \). This is by taking note of:

1. the \( i \) value used when evaluating each \( mU(N, k, r + 1) \);
2. the lowest \( r \) for which each of the \( mU(N, k, r) \) attains its minimum value.

**Definition**

Let \( mL(N) \) for each \( N \) be the maximal lower bounds analogous to the \( mU(N) \). The algorithm to compute them would clearly be similar to the one above. Of course it would be unusual for a single string \( s \) of \( G \) to have the property that \( mU(s) = mU(S) \) and \( mL(s) = mL(S) \).

These results are summarised in the following theorem.

**Theorem 7**

1. It is possible to compute the shortest stack length necessary to accommodate some string of \( G \) through the whole of its deposition on the stack. A string which uses just this stack length can be found.
2. Suppose the strings of \( G \) are to be deposited on a stack where some cells are occupied already. Then it is possible to compute the least inroad that a string of \( G \) can make upon the cells already occupied. A string of \( G \) which does this can be found.

**Corollary**

Given a positive integer \( L \) it is decidable whether there is any string of \( G \) which can be deposited on a stack of length \( L \). If there is, then such a string can be found.

A consequence of this theorem is that for many high level language programs which at run time use an arithmetic stack, whether known or accessible to the programmer or not, it is decidable at compile time what the minimum stack length must be for any program to run.

The above discussion is particularly relevant to FORTRAN and ALGOL programs since for these languages \( m^*(S) = 0 \) so that only upper bounds need be investigated.
Upper and lower bounds together

Definition
A string $s$ of $G$ is ‘stack-bounded in $[L_1, L_2]$’ if the length of the stack during the deposition of $s$ remains in the finite interval $[L_1, L_2]$. It is only necessary to consider intervals which include the integer zero, since it is normally assumed that the stack length is zero before deposition of a string of $G$.

Consider first a string $s$ of terminals, which may be part of a complete terminal string of $G$. Let the stack length before loading $s$ be $l_1$ and afterwards $l_2$. Let the Boolean function $s_b(s, l_1, l_2)$ be true if $s$ is stack bounded in the interval $[L_1, L_2]$ and false otherwise. Then for any $s_1, s_2$,

$$s_b(s_1, s_2, l_1, l_2) = s_b(s_1, l_1, l) \land s_b(s_2, l, l_2)$$

where $l$ is the stack length after the loading of $s_1$. Now let

$$s_b(s) = (s_b(s, l_1, l_2))$$

the Boolean matrix of $s_b(s, l_1, l_2)$ values, where $l_1, l_2$ may each take all the values in $[L_1, L_2]$. Then

$$s_b(s_{12}) = (\bigcup_{l_1, l_2} s_b(s_{12}, l_1, l_2))$$

Furthermore, let

$$s_b(N) = \bigcup_{s_{12}} s_b(s_{12}).$$

The $(l_1, l_2)$th element of this matrix is true if there is some $s_N$ such that if deposition of loading starts at stack length $l_1$, it finishes at length $l_2$ without exceeding the bounds $L_1, L_2$. Introduce now a second parameter $r$ which restricts the $s_N$ considered to those who generation trees are no more than $r$-rule applications deep. Also denote by $s_b(N, r)$ the matrix arising from the application of the $r$th rule of $N, N \to c_1 \ldots c_j \ldots c_n$. Then

$$s_b(N, r + 1) = s_b(N, r) \cup s_b(N, r + 1),$$

and

$$s_b(N, r + 1) = s_b(c_1, r) \cup s_b(c_2, r) \cup \ldots \cup s_b(c_n, r).$$

Some of the $s_b(N, r)$ may not exist for low $r$, since their evaluation may depend on some $s_b(N, 0)$—which cannot exist once $N$ is a non-terminal. One could set $s_b(N, 0) =$ (‘False’).

The above rules define a simple though laborious procedure for finding the $s_b(N, r)$ for all $N$, and so $s_b(S, r)$ in particular.

The procedure must terminate, i.e. there is an $r_1$ such that for every $N$ $s_b(N, r_1 + 1) = s_b(N, r_1)$. This is because the number of elements in each $s_b(N, r)$ is fixed, and as $r$ increases the number of true elements can only increase. If the number of distinct non-terminals is $m$, then there can be at most $m(L_2 - L_1 + 1)^2$ iterations before termination.

Once termination has been reached the truth-values of the matrix elements $s_b(S, 0, l)$ for any $l$ in $[L_1, L_2]$ show whether any string of $G$ is stack-bounded in $[L_1, L_2]$. Furthermore if one of them is true then by working back through the computation one can construct the generation tree of a terminal string which performed in this way.

Theorem 8
It is decidable whether any string of $G$ is stack-bounded in a given interval $[L_1, L_2]$. Where it is shown to exist, such a string can be found.

Theorem 9
It is possible to find a least interval $[L_1, L_2]$ in which any string of $G$ can be stack-bounded.

Proof
Choose any string $s$ of $G$ and find $m(U(s))$ and $m(L(s))$. Then there are only a finite number of smaller intervals than $[m(L(s)), m(U(s))]$ which include zero. By examining these in turn as in Theorem 8, a least interval can be found.

Definition
$G$ is ‘infinitely stack-bounded in $[L_1, L_2]$’ if there exist an infinity of strings of $G$ which are stack-bounded in $[L_1, L_2]$.

Theorem 10
It is decidable whether $G$ is infinitely stack-bounded in a given interval $[L_1, L_2]$.

Proof
If $G$ is infinitely stack-bounded in $[L_1, L_2]$ then choosing any integer $r$, there exists $r' > r$ such that a string of $G$ is stack-bounded in $[L_1, L_2]$ and has a generation tree which is $r'$-rule applications deep. It might be hoped that examination of the $s_b(S, r)$ for large $r'$ would enable this infinite sequence of $r'$ values to be detected or proved absent, but this is not so. This is because the presence of a true element in $s_b(S, r)$ only implies the existence of a string with a generation tree no more than $r'$-rule applications deep. However, it is possible to modify the $s_b(S, r)$ so as to derive an acceptable test.

Consider the rule

$$s_b(N, r + 1) = \Pi s_b(c_j, r)$$

which arises from the $i$th rule of $N$, namely $N \to c_1 \ldots c_i$. While it is not necessary to restrict $s_b(N, r + 1)$ to $(r + 1)$-deep derivations only, it is now important to include within $s_b(N, r + 1)$ a record of these new derivations, which have no mention in $s_b(N, r)$. Such a derivation only arises if one or more of the $c_j$ components is $r$-deep. Hence it is necessary to record within each of the $s_b(c_j, r)$ whether any derivations of $c_j$ are new, i.e. exactly $r$-deep. Therefore let each element—say the $(l_1, l_2)$th element—of the matrices $s_b(N, r), s_b(N, r)$ have one of the possible values:

1. ‘False’—no terminal derivation of $N$ which is no more than $r$-deep remains within $[L_1, L_2]$ for the starting length $l_1$ and the ending length $l_2$.
2. ‘Old’—such a terminal derivation of $N$ exists but is less than $r$-deep.
3. ‘New’—such an exactly $r$-deep terminal derivation of $N$ exists. Possibly ‘Old’ derivations exist too.

It is easy to see how the logical operators $\land$ and $\lor$ can be modified suitably. For consider $s_b(c_j, l_1, l) \land s_b(c_{j+1}, l, l_1)$. As before the result of the operator must be ‘False’ if either operand is ‘False’, and if both are ‘Old’ the result must be ‘Old’. However if either is ‘New’ and the other is ‘Old’ the result must be ‘New’ because the result expresses the fact that this combination of derivations of $c_j, c_{j+1}$ will help to form a new $(r + 1)$-deep derivation of $N$ if at least ‘Old’ derivations can be found for the other $c_1 \ldots c_{j-1}$. Thus $\land$ is given by the following table, and the discussion for the new $\lor$ is similar:

<table>
<thead>
<tr>
<th>$\land$</th>
<th>F</th>
<th>F</th>
<th>N</th>
<th>O</th>
<th>F</th>
<th>O</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td>O</td>
<td>F</td>
<td>N</td>
<td>N</td>
<td>O</td>
<td>O</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>F</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Now let the unary operator $O[ ]$ have the effect of replacing all ‘New’ elements in the matrix by ‘Old’. The formulae for the $s_b$ matrices now become:

$$s_b(N, r + 1) = O[s_b(N, r)] \cup s_b(N, r + 1) \geq 0,$$

for all $N$, where $s_b(N, r + 1) = \Pi s_b(c_j, r)$ as $i$ varies over all rules of $N$, for each $N$, and

$$s_b(N, 0) = \text{('False')} ,$$

and for every terminal $t$, and $r > 0$

$$s_b(t, r) = O[s_b(t, 0)].$$
Example
Consider the simple grammar whose rules are $S \rightarrow aSC$, $S \rightarrow a$, and the interval $[0, 2]$.

$$sb(a, 0) = \begin{pmatrix} F & N & F \\ F & F & N \\ F & F & F \end{pmatrix}, \quad sb(c, 0) = \begin{pmatrix} F & F & F \\ N & F & F \\ F & N & F \end{pmatrix},$$

$$sb(S, 0) = \begin{pmatrix} F & F & F \\ F & F & F \\ F & F & F \end{pmatrix}$$

Then $sb(S, 1) = sb_2(S, 1) = sb(a, 0) = \begin{pmatrix} F & N & F \\ F & F & N \\ F & F & F \end{pmatrix}$

Now $sb_1(S, 2) = sb(a, 1) \cdot sb_1(S, 1) \cdot sb(C, 1) = \begin{pmatrix} F & O & F \\ F & F & O \\ F & F & F \end{pmatrix}, \begin{pmatrix} F & N & F \\ F & F & N \\ F & F & F \end{pmatrix}, \begin{pmatrix} F & F & F \\ F & F & F \\ F & F & F \end{pmatrix}$

$$= \begin{pmatrix} F & O & F \\ F & F & O \\ F & F & F \end{pmatrix}, \begin{pmatrix} F & F & F \\ F & F & F \\ F & F & F \end{pmatrix}, \begin{pmatrix} F & F & F \\ F & F & F \\ F & F & F \end{pmatrix}$$

and $sb_2(S, 2) = sb(a, 1) = O[sb(a, 0)] = \begin{pmatrix} F & O & F \\ F & F & O \\ F & F & F \end{pmatrix}$

Hence $sb_2(S, 2) = O[sb_1(S, 1), sb_1(S, 2)] = \begin{pmatrix} F & N & F \\ F & F & F \\ F & F & F \end{pmatrix}$

Similarly $sb_2(S, 3) = \begin{pmatrix} F & F & O \\ F & F & F \end{pmatrix}$

from which clearly if $r \geq 3$, then $sb(S, r)$ has no 'new' elements and there are no more stack-bounded strings.

Proof of Theorem 10 continued
As with the former $sb$ matrices let evaluation continue until $r = r_1$ (say), after which no more 'false' elements are overwritten with 'old' or 'new'. For the 'old' and 'new' elements in the $sb(N, r_1)$, for all $N$, one of three possibilities remains:

1. The 'new' elements are all replaced by 'old' elements as $r$ increases, so that for some $r_2 \geq r_1$ no 'new' elements remain in any $sb(N, r)$ if $r \geq r_2$.
2. The 'new' and 'old' elements become fixed for $r \geq r_3 \geq 1$

This is a special case of 3.

3. While the number $k$, (which is no more than $m(L_2 - L_1 + 1)^2$) and identity of the 'true' elements are fixed, they oscillate in possessing 'old' or 'new' values, the only restriction being that for any $r$, they cannot all be 'old', for otherwise they would all continue to be 'old' for all higher $r$. Thus there are $2^k - 1$ possibilities for the set of 'old' and 'new' values. Cases 1 and 3 can now be distinguished by examining the $sb(N, r_1 + 2^k)$ matrices for all $N$. Either all elements will be 'false' or 'old', or there will exist distinct $r_4$, $r_5$, in the interval $[r_1, r_1 + 2^k]$ for which $sb(N, r_4) = sb(N, r_5)$ for all $N$. The former condition is case 1, showing that $G$ is not infinitely stack-bounded in $[L_1, L_2]$, and the latter is case 3. Then provided an element $sb(S, 0, l)$, for some $l$, has the 'new' value for some $r'$ in the oscillating sequence, then there will be an infinite number of such $r'$, and $G$ is infinitely stack-bounded in $[L_1, L_2]$. Hence the theorem is proved.

Theorem 11
If $G$ is known to be infinitely stack-bounded in some interval $[L_1, L_2]$ it is possible to find a least interval in which it is infinitely stack-bounded.

Proof
Since zero must be in any such interval, there are only a finite number of intervals shorter than $[L_1, L_2]$ which need to be examined. The procedure of Theorem 10 can be used for each such interval.

The remainder of the article is devoted to proving computable conditions under which $G$ is infinitely stack-bounded in some interval, and determining a least such interval in the general case. Various trivial 'theorems' arise on the way which have some slight value of their own—it is useful to know that the $l(N)$ can be generated parametrically as well as with unwieldy recurrence relations.

Some Diophantine equations
Some Diophantine equations are used a number of times in the succeeding sections. Their relevant properties are therefore summarised here. One is:

$$a_1x_1 + a_2x_2 + \ldots + a_ix_i + \ldots + a_nx_n = a^* \quad (1)$$

where the $a_i$ and $a^*$ are any given integers and solutions in the $x_i$ are sought for integers $x_i > 0$, for all $i$. The results given in Mordell (1969) may be summarised and trivially extended as follows:

1. A solution can only exist if the highest common factor of all the $a_i$ divides into $a^*$.
2. Given 1, then there are always an infinite number of solutions if the $a_i$ do not all have the same sign. The $x_i$ can then be expressed as linear functions of a finite number of parameters.
3. If all the $a_i$ do have the same sign then solutions exist only if $a^*$ also has the same sign. If so then there are either no solutions or a finite number of solutions.
4. Given fixed $a_i$ and integers $A+$ and $A-$ it is computable whether there exists any $a^* < A+$, or alternatively $a^* > A-$, for which there is a solution of (1), and if so what values it may respectively take.

Now let $z_{ij} = b_{i1}b_{i1}x_1 + b_{i2}x_2 + \ldots + b_{i(j-1)}x_{j-1} + \ldots + b_{ijn}x_n$, for $i = 1$ or 2 and $1 \leq j \leq n$, and consider the two simultaneous equations

$$z_{i1}y_1 + z_{i2}y_2 + \ldots + z_{ij}y_j + \ldots + z_{in}y_n = 0,$$
$$z_{i1}y_1 + z_{i2}y_2 + \ldots + z_{ij}y_j + \ldots + z_{in}y_n = 0 \quad (2)$$

to be solved for non-negative $y_j$ and $x_{i(j-1)}$ in terms of the arbitrary integers $b_{i(j-1)}$. The needed result is:

5. It is computable whether there is a solution to the equations (2) for given values of the $b_{i(j-1)}$. The author has a proof of computability, but for brevity it is omitted from this paper.

N-non-recursive subtrees
In subsequent sections it will be shown how to compute whether a particular subset of the basic cycles of $G$ can appear in the derivation of a single string of the language. These subsets will in turn be used to provide formulae for generating all the $l(N)$. As a first step $N$-non-recursive subtrees are defined and their maximum number of possible occurrences is shown to be computable.

Definitions
A generation tree is 'N-non-recursive' if its base-node is the non-terminal $N$ and it contains no other instance of $N$. The base-node of such a tree is termed 'non-recursive' for brevity in what follows. However, note that N-non-recursiveness does not necessarily imply zero-recursiveness.

The marks $N \Rightarrow u$ are defined to mean either that $N = u$, or that $N \Rightarrow u$. For each pair of non-terminals $N'$ and $N$ of $G$, let the maxi-
The number of N-non-recursive subtrees which can exist in the
tree of any expansion of N' be I(N', N). Trivially also
define I(t, N) = 0 for any terminal t.

**Examples**

All zero I(, ) values are omitted.

1. 
   \[
   \begin{align*}
   S &\rightarrow NN \\
   N &\rightarrow d \\
   S &\rightarrow e
   \end{align*}
   \]
   \[
   I(S, N) = 2, \\
   I(S, S) = 1, \\
   I(N, N) = 1.
   \]

2. 
   \[
   \begin{align*}
   S &\rightarrow NS \\
   N &\rightarrow d \\
   S &\rightarrow e
   \end{align*}
   \]
   \[
   I(S, S) = 1, \\
   I(N, N) = \infty, \\
   I(N, N) = 1.
   \]

3. 
   \[
   \begin{align*}
   S &\rightarrow MSS \\
   M &\rightarrow Nf \\
   N &\rightarrow d \\
   S &\rightarrow e
   \end{align*}
   \]
   \[
   I(S, S) = \infty, \\
   I(S, M) = \infty, \\
   I(N, N) = \infty, \\
   I(N, M) = 1.
   \]

4. 
   \[
   \begin{align*}
   S &\rightarrow PS \\
   P &\rightarrow p \\
   S &\rightarrow Nk \\
   N &\rightarrow fN \\
   N &\rightarrow d \\
   S &\rightarrow e
   \end{align*}
   \]
   \[
   I(S, S) = 1, \\
   I(S, N) = 1, \\
   I(N, N) = 1, \\
   I(N, P) = \infty.
   \]

It is easy to see that for any N, either I(N, N) = 1 or I(N, N) = \infty. Also if N_1 \Rightarrow uN_2v for some u, v then I(N, N_1) \leq I(N, N_2). Note that it does not follow that if N' is in cycle then I(N', N) will be infinite. The following lemma gives a precise condition for infiniteness.

**Lemma 5**

I(N', N) is infinite if and only if there exists N_1 such that

\[
N' \Rightarrow uN_1v
\]

and

\[
N_1 \Rightarrow u_1N_1v_1
\]

where

\[
u_1v_1 \Rightarrow u_2N_2v_2
\]

The condition of this lemma is clearly computable, for it is equivalent to determining whether certain paths exist between nodes of a subgraph of G, which is finite.

Let \( R(N') = N' \rightarrow C_{11}C_{12} \cdots C_{ij} \cdots C_{ip} \)

Then if I(N', N) is finite,

\[
I(N', N) = 1 \text{ if } N' = N, \text{ and } \\
I(N', N) = \max_{i,j} \left\{ \sum_{N_{ij}} I(C_{ij}, N) \right\}
\]

from its definition. From Lemma 5 above the formula is never usefully recursive in any quantity I(N_1, N), which makes its computation finite and straightforward.

**Co-existent basic cycle subsets**

**Definition**

A subset of the basic cycles of G is 'co-existent' if there exists a string of the language which uses each of the cycles of the subset at least once, and no others at all.

In this section an algorithm is developed for determining whether a basic cycle subset is co-existent. Hence all co-existent subsets can be found.

There are three possible co-existence relationships between two basic cycles:

1. The use of one cycle may prohibit the use of the other.
   \[
   \begin{align*}
   S &\rightarrow N \\
   S &\rightarrow M
   \end{align*}
   \]
   e.g. \( N \Rightarrow aN \quad M \Rightarrow bM \)
   \[
   N \Rightarrow a \\
   M \Rightarrow b
   \]
   The strings are \( \{a^n \mid n \geq 1\} \quad \{b^m \mid m \geq 1\} \)

2. The cycles may be used independently.
   \[
   \begin{align*}
   S &\rightarrow MN \\
   S &\rightarrow a
   \end{align*}
   \]
   e.g. \( N \Rightarrow aN \quad M \Rightarrow bM \)
   \[
   N \Rightarrow a \\
   M \Rightarrow b
   \]
   The strings are \( \{a^n \mid n, m \geq 1\} \)

3. One cycle may be used only if the other one is also used.
   \[
   \begin{align*}
   S &\rightarrow N \\
   S &\rightarrow a
   \end{align*}
   \]
   e.g. \( N \Rightarrow a \\
   M \Rightarrow g \)
   \[
   N \Rightarrow MNd
   \]
   The strings are \( \{a\} \cup \{b^m a d^n \mid m \geq 0, n \geq 1\} \)

Note that in the simple grammar \( \{S \Rightarrow SS, S \Rightarrow a\} \), the first rule defines two basic cycles. A particular application of this rule, however, may be regarded as the use of either cycle, but not both. Thus these cycles may be used independently, and this grammar formally has three co-existent basic cycle subsets, according to the selection of either or both of these cycles.

Consider the tree of a string of a language in which two cycles are used. Let \( N_1 \) and \( N_2 \) be non-terminal instances, one in each of the two cycle occurrences. Consider the paths from the
base-node $S$ to $N_1$ and to $N_2$. These paths must diverge at some node of the tree, perhaps at the base-node, and when they do they must be such a particular rule:

$$N \rightarrow u'N_1v'N_2w'$$

is applied, for some $u', v', w'$, where

$$S \Rightarrow uN_0, N_1 \Rightarrow u_1N_1v_1, \text{ and } N_2 \Rightarrow u_2N_2v_2$$

for some strings $u, v, u_1, v_1, u_2$, and $v_2$. Conversely, the existence of such a rule of $N$ guarantees that the two cycles can co-exist.

More generally, suppose paths on $G$ from $S$ to $N_1, N_2, \ldots, N_s, \ldots, N_n$ in distinct basic cycles all diverge at a particular $N$, and it is desired to find out whether all these basic cycles can co-exist. The cycles can be grouped depending on which rule of $N$ can be used to connect each $N_j$ to $S$. Then all the cycles can co-exist if sufficient instances of $N$ can occur to allow all the necessary rules of $N$ to be used. Some of these rules may be parts of cycles, and if so can appear as many times as desired (providing of course that the cycles involved are in the subset currently being investigated). Once any such rules have been allowed for, the desired number $D(N)$ of instances of $N$ must all be taken from the non-redundant instances of $N$. Hence the condition for co-existence of the cycles when the paths join at $N$ is

$$D(N) \leq I(S, N)$$

This condition can now be made the basis of the algorithm for determining whether a subset of basic cycles is co-existent. Firstly, for each cycle determine all the non-cyclic paths on $G_R$ from $S$ to each non-terminal in the cycle. Now also allow paths with circuits to be added provided they introduce further distinct rule instances.

Now select one path for each cycle out of the finite set just established and determine as above whether this selection of paths can all appear in any one string of the language. Repeat until a co-existent selection is found, or otherwise until all such selection combinations have been exhausted to prove non-co-existence.

An analogous algorithm can be applied to find the basic cycle subsets which may co-exist in some expansion of a non-terminal $N$ distinct from $S$. Here the computation described would use not $G_R$ and $G$ but the subgraph of $G_R$ containing only nodes accessible from $N$ together with the associated sub-grammar of $G$. Of course, the cycles of this sub-grammar would be a subset of the cycles of $G$.

**Generating functions for the $l(N)$**

Since the recurrence relations

$$l(N) \leftarrow \sum_{j=1}^{r} l(C_j)$$

are not easy to work with, some linear functions $f_N^h$ are now introduced whose arguments can take all the positive integer values and whose domains, taken together, constitute all the $l(N)$.

**Definitions**

For any recursive chained derivation $N \Rightarrow uNv$ call the string $uNv$ the 'result' of the derivation, or of the cycle which is involved. Here the order of the symbols in $uv$ is ignored.

Now let all the $B^h_k$, the co-existent basic cycle subsets of $N$, be denoted by

$$B^1_k, \ldots, B^s_k, \ldots, B^h_k,$$

and let the basic cycles in $B^h_k$ be specially numbered

$$C_1^h, \ldots, C^s_N, \ldots, C^h_N.$$

Now for fixed $i$ and $j$ there may well be more than one zero-cyclic length $l_0(C_j^h)$. Call them

$$l_1^h, \ldots, l_s^h, \ldots, l^h_i,$$

where $i$ varies depending on $C^h_N$.

In the algorithm for determining the co-existence of the subset $B^h_k$, terminal expansions were found which each included a path selection but had no further cycles. Call them

$$e_1^h, \ldots, e_s^h, \ldots, e^h_i,$$

and let the number of instances of

$$C^h_N \in e^h_i$$

be $n^{h_i}$. For any particular $j, 1 \leq j \leq s$.

The generating functions for the $l(N)$ can now be presented.

**Lemma 6**

1. Every terminal expansion of $N$ has a length

$$f^h_N(\bar{x}_{ijk}) = l(e^h_N) + \sum_{j=1}^{r} \sum_{k=1}^{r} x_{ijk}l^h_{jk},$$

for some $i, h$ and a vector $\bar{x}_{ijk}$ of non-negative integers $x_{ijk}$ provided that for fixed $i, h$ and $j$

$$\sum_{k=1}^{r} x_{ijk} = 0 \text{ only if } n^{h_i} > 0.$$

2. Conversely, for every selection of $i, h$ and $\bar{x}_{ijk}$ (according to the rule just given), the above expression gives the length of some expansion of $N$.

Before giving the proof an explanation of the $f^h_N$ is useful: $f$ fixes the cycle subset and $h$ specifies a particular minimal expansion of $N$ which will permit all the cycles of $B^h_k$ to be used. The $x_{ijk}$ fix the number of additional times a particular cycle is to appear, with its result yielding a particular zero-recursive tree structure. Recursiveness in the expansion of the result of the cycle is allowed for by varying some of the $x_{ijk}$. The condition of 1 specifies that each cycle in the subset must appear at least once. Only when the $e^h_N$ concerned contains a particular cycle $C^h_N$ can

$$\sum_{j=1}^{r} x_{ijk} = 0$$

**Proof of Lemma 6**

1. Consider a terminal expansion $e_N$ of $N$. Then either it is 0-recursive or its tree contains a basic cycle whose root is 0-recursive. In the former case the lemma is proved. In the latter case it means that a node $N'$ has a proper subtree with base-node $N'$ such that

$$N' \Rightarrow uN'v \Rightarrow uvw,$$

where $uw$ is 0-recursive. Now form a new tree and a corresponding new expansion $e_{NT}$ by putting the latter instance of $N'$, with its subtree ($N' \Rightarrow w$), in place of the former instance of $N$. This has the effect of removing the basic cycle with the 0-recursive result:

$$l(e_N) = l(e_{NT}) + l(uw) = l(e_{NT}) + l^h_{jk}$$

for some $i, j, k$.

The next step is to reduce $e_{NT}$ in the same way, ensuring when removing cycles that for each non-terminal in the removed cycles some instance is retained in the new reduced subtree. Eventually no further cycle instances can be removed, bearing the above restriction in mind, and the resultant tree's expansion is one of the $e^h_N$. This establishes the formula.

2. From the construction of any expansion $e^h_N$, it is possible to add as many instances of cycles $C^h_N$ with 0-recursive results as desired. Hence any particular formula certainly has a corresponding expansion of $N$. 

158
Computability of the $I(N)$

**Theorem 12**

1. It is computable whether any expansion of $N$ has a given length $L$, and also whether there is a finite or an infinite number of such expansions. In the former case they can all be determined. In the latter case formulae can be derived from which as many such expansions can be generated as required.

2. As 1, but with the expansions required to be in a finite interval $[L_1, L_2]$. As 2 but with the expansions in infinite intervals $[-\infty, L_2]$ or $[L_1, +\infty]$.

**Proof**

1. From Lemma 6 the problem is equivalent to solving all the equations

$$f_N^h(x_{ijk}) = L$$

in the $x_{ijk}$, for all $i$ and $h$. These are not simultaneous equations. Each separately may contribute solutions for the $x_{ijk}$ and hence expansions of $N$. Such an equation is an example of the Diophantine equation (1) given above. A solution in the $x_{ijk}$ specifies the number of instances of each basic cycle that each corresponding satisfactory expansion may have. There are only a finite number of terminal expansions for each such solution. They can be found exhaustively by developing generation trees in which all combinations of alternative expansions of the non-terminals are taken.

2. Repeat 1 for all $L$ in $[L_1, L_2]$.

3. Derive all suitable $a^*$ in $[-\infty, L_2]$ or in $[L_1, +\infty]$ using Diophantine equation result 4 above (Mordell, 1969). Then continue as just above.

**Theorem 13**

As Theorem 12, but considering any string $u$ rather than $N$.

**Proof**

1. Permute the symbols of $u$ until $u = vv$, where $v$ consists of all the terminals of $u$, and

$$w = N_1 \ldots N_y \ldots N_b,$$

being all the non-terminals of $u$. Then the problem is equivalent to solving separately all the equations

$$L = l(v) + \sum_{y = 1}^{b} f_N^h(x_{ijk})$$

for all possible combinations of the $i$ and $h$ varying separately for each $N$. These equations have the same Diophantine form as those of Theorem 12 above.

2., 3. As for Theorem 12.

Is there any infinite stack-boundedness?

**Theorem 14**

$G$ is infinitely stack-bounded in some interval $[L_1, L_2]$ if and only if there exists a cycle $C$, not necessarily basic, such that $I(C) = 0$ and $lhI(C) = 0$.

**Proof**

1. **Sufficiency**: take a string of $G$ whose derivation includes a non-terminal of the cycle for which the conditions are true. Then apply the cycle to generate an infinity of stack-bounded strings.

2. **Necessity**: suppose it is known that $L_1, L_2$ exist for which an infinity of strings are stack-bounded in $[L_1, L_2]$. The idea of the proof is to show that there is a suitable cycle with zero lefthand length, and then that there are sufficient of these to guarantee that one of them must have zero righthand length as well.

Consider the infinite set

$$B = \{b_1 \ldots b_i \ldots \}$$

which consists of all the finite strings $b_i$ of $G$ which are stack-bounded in $[L_1, L_2]$. Since there are only a finite number of rules $S \rightarrow w$, then there must exist one of them, say $S \rightarrow w_1$, which is the first rule applied in the derivations of an infinite subset $B_1$ of $B$. Similarly there must exist a rule $N_1 \rightarrow w_2$, where $S \rightarrow w_1 = u_1N_1v_1$, for some $u_1, v_1$, such that an infinite subset $B_2$ of $B_1$ start off their derivations with $S \rightarrow w_1, N_1 \rightarrow w_2$, so that $S \Rightarrow u_1w_1\ldots$. Similarly for any integer $k$, there exists an infinite set $B_{k+1}$ each of whose members has a derivation which begins

$$S \Rightarrow u_1u_2 \ldots u_kw_{k+1}v_{k+1} \ldots v_{2i+1},$$

with a corresponding non-terminal sequence $N_1 \ldots N_f \ldots N_k$, each of which is derived from the last. By choosing $k$ greater than the number of distinct non-terminals of $G$, the later $N_f$ must coincide with earlier ones in the sequence. Hence there exists a non-terminal $N$ such that, for any given integer $f$, $N$ occurs $f + 1$ times in the sequence $N_1 \ldots N_f \ldots N_k$, for sufficiently large $k$. Call these instances of $N$, $N_1, \ldots N_{f+1}$.

Let $u_1u_2 \ldots u_k \equiv u', v_1 \equiv v' \ldots$, and let $u_{j+1} \ldots u_{j+1} \equiv u_j, v_{j+1} \ldots v_{j+1} \equiv v_j$.

Then $N_{f+1} = u_1 \ldots u_{f}N_{j+1}v_{j+1} \ldots v_{j+1}$ and $S \Rightarrow u_{f+1}u_{f+2} \ldots u_{f+k}N_{j+1}v_{j+1} \ldots v_{j+1}$.

Consider the loading of a particular terminal derivation of $y_{f+1}$ on to the stack, and let the stack-length after the deposition of the terminal string derived from $u'$ be $l_{f}$, for $0 \leq f \leq f$. Choose $f > L_2 - L_1 + 1$. Then because the terminal string does not exceed the bounds $L_1$ and $L_2$, there must be distinct integers $r$ and $s$ such that

$$l_{r} = l_{s}, i.e. l(u'_{r+1}u'_{s+1} \ldots u') = 0,$$

and there exists a chained recursive derivation

$$N_{f+1} \overset{\Delta}{=} u'_{r+1} \ldots u'_{f}N_{j+1}v'_{j+1} \ldots v'_{j+1},$$

whose lefthand length is zero. More generally, for sufficiently large $f$ there exists $l, L_1 \leq l \leq L_2$, and distinct integers $r_1, r_2 \ldots r_{L_2 - L_1 + 2}, \leq l$, such that

$$l = l_{r_1} = l_{r_2} = \ldots = l_{r_{L_2 - L_1 + 2}}.$$

Consider now the loading of a terminal derivation of $v_{r_1} \ldots v_{r_{L_2 - L_1 + 2}} \ldots v_{r_{L_2 - L_1 + 2}}$ on to the stack. Let $l_{r_1}$ be the stack length just before $v_{r_1}$ is loaded. Consider the stack lengths $l_{r_1}$ for $1 \leq p \leq L_2 - L_1 + 2$. Then there exist integers $a, b, a > b$, such that

$$l_{r_1} = l_{r_2}, i.e. l(v_{r_{1}}'v_{r_{2}}' \ldots v_{r_{a+1}}') = 0,$$

and from above

$$l(u'_{r_{a+1}} \ldots u'_{r_{b-1}}u'_{r_{b+1}}) = 0.$$

Hence the cycle corresponding to the chained recursive derivation

$$N \overset{\Delta}{=} u'_{r_{a+1}} \ldots u'_{r_{b-1}}N_{r_{a+1}}v'_{r_{b+1}} \ldots v'_{r_{a+1}}$$

has a zero lefthand length and zero righthand length, and hence zero overall length, as required.

**Theorem 15**

It is computable whether there exist integers $L_1, L_2$ such that $G$ is infinitely stack-bounded in $[L_1, L_2]$. If so such an interval can be found.
Proof
From Theorem 14, it is sufficient to show that it is computable whether there exists a cycle $C$ for which $l(C) = 0$ and $lh(C) = 0$. Let $\{C_1, \ldots, C_t\}$ be a composutable subset of the basic cycles of $G$. Then consider a cycle $C$ composed from $y_j$ instances of $C_j$, for each $j$, $1 \leq j \leq t$. Then

$$l(C) = \sum_{j=1}^{t} y_j \cdot l(C_j)$$
$$lh(C) = \sum_{j=1}^{t} y_j \cdot lh(C_j).$$

In general there are a finite number of alternative expressions for $l(C)$ and $lh(C)$, each on the lines of the proof of Theorem 13. For each combination of alternatives the simultaneous equations

$$l(C) = 0$$
$$lh(C) = 0$$

take the form of Diophantine equations (2) and the Diophantine result 5 shows it to be computable whether a solution exists.

If one of these sets of equations proves to be solvable, then the algorithm for deciding that also leads readily to one solution at least. This means that one cycle with the desired properties is known. Suppose that $N$ is a non-terminal in the cycle, say $N \Rightarrow u_i N_i v_i$, where $u_i$ and $v_i$ are terminal strings. Then choose any derivation $S \Rightarrow u \cdot u_i N_i v_i$ for any terminal $u'$ and $v'$, and a non-cyclic derivation $N \Rightarrow s_N$, one can then show that an infinity of terminal strings generated by the cycle is $u \cdot u_i s_N v_i$, $n \geq 1$, whose upper and lower bounds are the same as those for $u \cdot u_i N_i v_i$, where $n = 1$. Thus one suitable interval $[L_1, L_2]$ can be found if any exist at all.

**Theorem 16**
It is possible to find a smallest interval, if any, in which $G$ is infinitely stack-bounded.

**Proof**
This theorem just brings together the results of Theorems 11 and 15.

**References**

**Book reviews**


The text is drawn from an advanced course on software portability given at the University of Kent at Canterbury in Spring 1976 and comprises the copious session material of the individual speakers. The course was sponsored by SRC under the auspices of the Informatics training group of the EEC Scientific and Technical Research Committee. Speakers were drawn from both sides of the Atlantic and appeared very much as a listing in ‘who’s-who’ in Software Portability.

The book is divided into eight ‘parts’ each being subdivided into chapters representing individual speaker’s contributions. Each chapter has appended a list of references and these define the state-of-the-art at the time of writing. Parts 1 and 2 are an introduction which outlines the format of the text and a basic concepts review which is really a justification of what follows. Part 3 concerns itself with mechanisms which affect and aid portability. It includes descriptions of the roles played by verifiers and filters, computer architecture and microprogramming, and microprocessors. The problems associated with portable compilers and the translation between high level languages are discussed. Part 4 considers pragmatics, the practical minutiae which so greatly affect portability as a useful tool. Considerations include engineering, the system interface, performance and optimisation. This ‘part’ is very practical and informative. Part 5 has a novelty as well as great general use. It addresses itself to legal aspects and provides a valuable insight into the legal standing of ‘information’. The session was driven by the portability concept but turns out to be of more general interest. In part 6 a number of case studies are presented, ranging from the portability of ALGOL 60, through SNOBOL 4, BCPL, commercial software, data, FORTRAN and GENESYS, an operating system, and NAG’s approach, to the view of a manufacturer (ICL). The experiences recounted certainly ring true to the reviewer. In Part 7 two of the leading lights in the field (Waite and Griswold) combine to discuss current research in the area and take the possibly dangerous step of predicting the future. Part 8 presents two study group reports on portability, those of CNRS/SRC and EEC.

The text is a wide ranging survey of the nature, problems and mechanisms involved in software portability and succeeds in this area. Since it is formed of a series of papers from different contributors the book suffers to a certain degree because of variations in style, depth and general presentation (including typesetting); however the effect is not as bad as that experienced in many similarly organised texts.

The fragmentation of the book does have an advantage: each chapter is largely self contained and therefore the text easily lends itself as a reference document with respect to the different facets of portability. In general the level of the material is at least final year undergraduate. The contents will be of interest to the software industry at large, since many practical lessons may be learnt from within its covers, without the pain of finding out on a ‘real’ system.

ROY NEWTON (Middlesbrough)

*The Challenge of Microprocessors* by M. G. Hartley and Anne Buckley, 1979; 200 pages. (*Manchester University Press*, £7-95)

Hartley and Buckley are the editors of the *International Journal of Electrical Engineering Education*, and this book consists mainly of papers from a special issue of this journal. It is divided into four main sections: Overall Perspective; General Laboratory Activity; Laboratory Experiments; and Courses and Services. Unfortunately, the editors seemed unable to find sufficient material about microprocessors and have padded out the book with chapters on the use of programmable calculators and a description of a large computerised education system, which are totally unrelated to the rest of the book.

An attempt is made to provide a coverage of a wide range of microprocessor topics, which succeeds only in generating confusion. There is little coherence between the chapters, with the result that some are aimed at educators who are totally unfamiliar with microprocessors, while there are three chapters on microprogramming and bit-slice processors, which seem to be aimed at budding minicomputer designers. The typical educational institution approach of costing equipment merely by the value of the electronic components is noticeable, and highlights the lack of information about the commercial side of microprocessors, such as the development costs of hardware and software, project management and so on.

There is little in this book to commend it to the general reader, and since it does not aim at an identifiable part of the microprocessor arena, or of the educational spectrum, it can also not be recommended to any specific audience.

HARLEY QUILLIAM (Guildford)