Efficient implementation of a class of recursively defined functions

R. Haskell

Department of Computing and Control, Imperial College of Science and Technology, 48 Prince's Gardens, London SW7 1LU

A method of evaluating recursively defined functions which uses two (or more) stacks is described in this paper. This method is designed to reduce the stack storage needed and requires simpler 'linkage' information than the single stack method. In addition, a study of 'go-to' (or iterative) type recursive definitions is made and it is shown that these require bounded stack storage for their evaluation.

(Received August 1973)

1. Introduction

Recursion is the technique of defining procedures or functions in terms of themselves. Conventional implementations of recursion use a single stack for holding temporary results, return addresses, etc. This stacking mechanism incurs serious storage overheads. To reduce storage overheads work has been done to characterise recursive definitions which could be flowcharted and to devise suitable algorithms to perform the transformation—see McCarthy (1962) and Strong (1971).

The approach we take is different from previous approaches. Instead of trying to reduce storage overheads by transforming recursive definitions into a flowchart we devise a new technique of stacking and erasing arguments to solve this problem. It is shown that with this method, go-to (or iterative) type recursive definitions require only bounded stack storage for their evaluation.

This study of recursion is made within the framework of a functional language (Section 2) and its associated stack language (Section 3). In Section 4 go-to (or iterative) type functions are defined. In Section 5 a stack minimisation method for a single stack is described and in addition the bounded storage theorem is proved. In Section 6 an efficient method of evaluating recursively defined functions which requires at least two stacks is described. The bounded storage theorem also applies to this method.

2. The functional language

The objective in this section is to define a suitable language for studying recursively defined functions and the stacking mechanism. The language we define uses the postfix (reverse polish) notation for writing functions and is thus convenient to describe the evaluation and stacking processes (*See* Barron (1968)).

2.1. *BNF syntax of the functional language*

The usual BNF notation is used with the additional convention that '*abc*,,,' is used to represent any repetitive sequence of zero or more occurrences of the syntactic type '*abc*' *suitably delimited by commas*, e.g. sequences such as '*abc*' or '*abc, abc, abc, abc*'. Furthermore the tacit use of blanks as a separator is taken for granted and is not explicitly specified in the syntax.

1(*a*) function-specification ::= LET (argument , , ,) function-name BE BASIC
1(*b*) ::= LET (argument , , ,) function-name BE expression END
2(*a*) expression ::= constant (which depends on the data domain)
2(*b*) ::= argument
2(*c*) ::= (expression , , ,) function-name
2(*d*) ::= IF expression, expression ARE EQUAL THEN expression ELSE expression FI

The three syntactic types 'constant', 'argument' and 'function-name' have not been specified in detail. All we shall assume about them is that they are mutually distinct. We shall also assume that a function has only one specification.

3. The stack language

The stack mechanism for handling recursion is based on that of Barron (1968). We assume that the reader is conversant with this and only present the concepts in outline.

When a function G is to be applied to arguments a, b, c say, the calling sequence is as follows.

1. a, b, c are loaded onto the stack.
2. The link information is loaded on the stack—namely the return address and backpointer to the previous link.
3. The evaluation of G is then initiated.

There are two pointers T and L associated with the stack. The pointer T points to top of the stack. L points to the link information of the most recently activated function.

The operation of returning from the function G consists of reinstating the stack to its previous status from the link information; copying the result back over the arguments of G, and returning control to the return address.

A test operation is required which compares the top two items of the stack for equality, erases them, and then evaluates different expressions depending on whether or not the items are equal. This operation is used in evaluating IF . . . THEN . . . ELSE . . . FI type expressions in the functional language. We summarise these ideas in the syntactic definition of the stack language.

3.1. *BNF syntax of the stack language*

Program ::= Statement-List RETURN
Statement-List ::= Statement
::= Statement Statement-list
1. Statement ::= LOAD constant
::= LOAD argument
2. ::= ENTER function-name
3. ::= IF EQUAL THEN Statement-list ELSE Statement-list FI

The translation process from function-specifications to stack language programs is trivial. The reader requiring further details is referred to the appendix.

Example 1:
Here the data domain is the set of character strings of the form 'ABC', etc. including the null string. The functions HEAD and TAIL are basic and behave as their name suggests.

e.g. ('ABC') HEAD = 'A'
('ABC') TAIL = 'BC'

We also assume that HEAD and TAIL are defined for the null string by HEAD ('') = TAIL ('') = ''.

We now define a function LAST recursively which will have as

Downloaded from https://academic.oup.com/comjnl/article/18/1/23/454888 by guest on 19 April 2024

'ABC'@	On entry to ('ABC') LAST
'ABC'@'ABC'@	On entry to TAIL
'ABC'@'BC',''	Equals test fails
'ABC'@	Execute ELSE part
'ABC'@'ABC'@	On entry to TAIL
'ABC'@'BC'@	Re-enter ('BC') LAST
'ABC'@'BC'@'BC'@	Enter TAIL
'ABC'@'BC'@'C', ''	Equals test fails
'ABC'@'BC'@	Execute ELSE part
'ABC'@'BC'@'BC'@	Enter TAIL
'ABC'@'BC'@'C'@	Re-enter ('C') LAST
'ABC'@'BC'@'C'@'C'@	Enter TAIL
'ABC'@'BC'@'C'@'', ''	Equals test succeeds
'ABC'@'BC'@'C'@	Execute THEN part
'ABC'@'BC'@'C'@'C'	Return from ('C') LAST
'ABC'@'BC'@'C'	Return from ('BC') LAST
'ABC'@'C'	Return from ('ABC') LAST
'C'	Value found

Fig. 1 Stack behaviour in evaluating ('ABC') LAST

value the last character of the string.

i.e. ('ABC') LAST = 'C'; ('') LAST = ''.

```
    LET (X) HEAD BE BASIC
    LET (X) TAIL BE BASIC
    LET (X) LAST BE
    IF (X) TAIL, '' ARE EQUAL THEN X
                    ELSE ((X) TAIL) LAST
    FI
    END
```

Program to compute (X) LAST

```
        LOAD X
        ENTER TAIL
        LOAD ''
        IF EQUAL THEN LOAD X
                ELSE  LOAD X
                      ENTER TAIL
                      ENTER LAST
        FI
        RETURN
```

The behaviour of the stack in computing ('ABC') LAST is shown in **Fig. 1.** The link information is denoted by @ and the rightmost symbol represents the top of the stack.

Example 2:

We assume the data domain to be the unsigned integers including zero and the define using recursion the highest common factor HCF of two integers. The basic functions are (M, N) DIFF the difference of M and N, i.e. $|M - N|$ and (M, N) MIN minimum of M and N.

```
   LET (M, N) MIN BE BASIC
   LET (M, N) DIFF BE BASIC
   LET (M, N) HCF BE
      IF M, O ARE EQUAL THEN N
              ELSE ((M, N) DIFF, (M, N) MIN) HCF
      FI
   END
```

Program to compute (M, N) HCF

```
        LOAD M
        LOAD O
        IF EQUAL THEN LOAD N
                ELSE  LOAD M
                      LOAD N
                      ENTER DIFF
                      LOAD M
                      LOAD N
                      ENTER MIN
                      ENTER HCF
        FI
        RETURN
```

The behaviour of the stack in evaluating (9, 6) HCF is shown in **Fig. 2.** The link information is denoted by @ and the rightmost symbol represents the top of the stack.

9, 6@	On Entry to (9, 6) HCF
9, 6@9, 0	Equals test fails
9, 6@	Execute ELSE part
9, 6@9, 6@	Enter DIFF
9, 6@3, 9, 6@	Enter MIN
9, 6@3, 6@	Re enter (3, 6) HCF
9, 6@3, 6@3, 0	Equals test fails
9, 6@3, 6@	Execute ELSE part
9, 6@3, 6@3, 6@	Enter DIFF
9, 6@3, 6@3, 3, 6@	Enter MIN
9, 6@3, 6@3, 3@	Re enter (3, 3) HCF
9, 6@3, 6@3, 3@3, 0	Equals test fails
9, 6@3, 6@3, 3@	Execute ELSE part
9, 6@3, 6@3, 3@3, 3@	Enter DIFF
9, 6@3, 6@3, 3@0, 3, 3@	Enter MIN
9, 6@3, 6@3, 3@0, 3@	Re-enter (0, 3) HCF
9, 6@3, 6@3, 3@0, 3@0, 0	Equals test succeeds
9, 6@3, 6@3, 3@0, 3@	Execute THEN part
9, 6@3, 6@3, 3@0, 3@3	Return from (0, 3) HCF
9, 6@3, 6@3, 3@3	Return from (3, 3) HCF
9, 6@3, 6@3	Return from (3, 6) HCF
9, 6@3	Return from (9, 6) HCF
3	Value found

Fig. 2 Stack behaviour in evaluating (9, 6) HCF

4. Go-to (or iterative) type definitions

Examples 1 and 2 are known as go-to (or iterative) type functions. This concept was introduced by McCarthy (1962) to describe the class of recursive functions which are in one to one correspondence with iterative type programs, i.e. programs consisting of assignment, go-to's and conditional statements. Indeed it has been shown that every go-to (or iterative) type function definition may be transformed into an equivalent iterative program and vice-versa (see McCarthy (1962), Barron (1968) and Strong (1971)).

Informally the go-to (or iterative) type recursive definitions have the property that defined functions must never occur 'inside' other functions in the expression which defines a function value. Neither must they occur in the test for equality in any conditional expression.

More formally we specify this concept as follows.

1. A complete set of function definitions is said to be of go-to type if the value of every non-basic function is defined by a go-to type expression.
2. A go-to type expression (g-expr) has the following syntax:

```
g-expr ::= constant
       ::= argument
       ::= (basic-expression, , ,) function-name
       ::= IF basic-expression, basic-expression
                                    ARE EQUAL
              THEN g-expr
              ELSE g-expr FI.
```

3. A basic-expression is any expression constructed from constants, arguments, basic functions, and IF-THEN-ELSE-FIs only.

The behaviour of the stack in Examples 1 and 2 have one important point in common, i.e. the RETURNS from the defined functions were performed consecutively. Thus we see that when for example ('ABC')LAST calls ('BC')LAST the arguments of ('ABC')LAST are never used again and so need-

Downloaded from https://academic.oup.com/comjnl/article/18/1/23/454888 by guest on 19 April 2024

lessly occupy the stack. This is repeated when ('BC')LAST calls ('C')LAST, i.e. the arguments of ('BC')LAST are not accessed again. So by the time the RETURN operations are executed, the stack contains mainly *useless* information. The same remarks apply to the function HCF. In fact it may be shown that the above remarks apply to all go-to type definitions.

This is wasteful in stack storage space and in the following sections we shall show how to reduce this wastage.

5. Storage reduction for a single stack

In Examples 1 and 2 we have highlighted the problem of needlessly preserving arguments on the stack. This problem is particularly acute with go-to type definitions but is also present in other recursive definitions. What we shall do in this section is find (certain function calls) where arguments may be erased (5.1) and describe a crude method of erasing arguments for a single stack implementation (5.2). In addition the bounded storage theorem (5.3) and a corollary (5.4) will be proved. The problem of erasing arguments efficiently will be tackled in Section 6.

5.1. *Identifying function calls*

Consider a function (A, B, . . .)F whose value is defined by the expression *e*. Consider those functions in *e* which do not occur 'inside' other functions and which are not involved either directly or indirectly in a test for equality. Let $(e_1, e_2, \ldots)G$ be such a function. Clearly if *G* were to be evaluated it would be immediately followed by a return from *F*. Thus as the arguments of *F* would not be accessed after entering *G* there is no harm in erasing the arguments of *F* at this stage. This describes those function calls in *e* where the arguments of *F* may be erased. To help the reader to identify these calls and to prevent any possible misinterpretation of what we have said we also describe these calls using BNF syntax. Square brackets are added to the functional language and the syntax is designed so that any function name whose arguments are in square brackets is of the type required.

```
1(a) function specification ::= LET(argument, , ,) function-
                                name
                                    BE BASIC
1(b)                        ::= LET(argument, , ,)function-
                                name
                                    BE expr END
2(a) expr ::= constant
          ::= argument
          ::= [expression, , ,] function-name
          ::= IF expression,  expression ARE EQUAL
                              THEN expr
                              ELSE expr FI
3(a) expression ::= exactly as before in 2.1 and so contains no
                    square brackets.
```

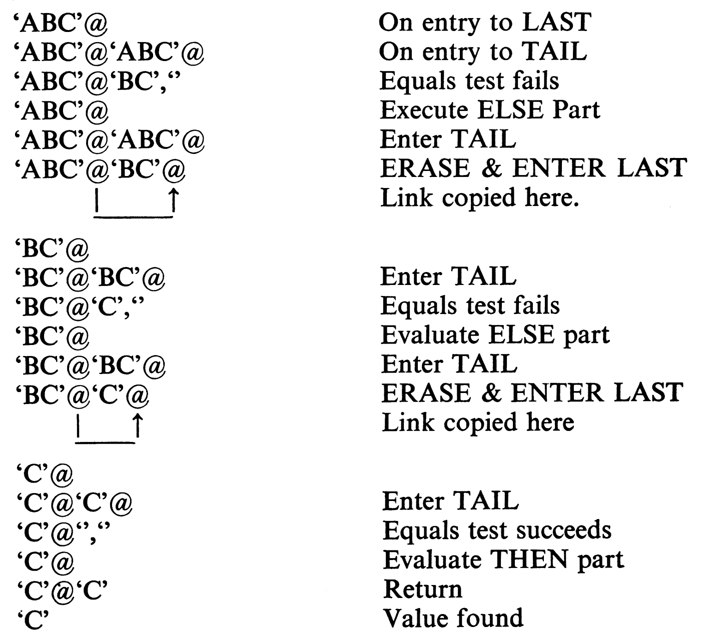
It is easy to transform the old form of function definition (2.1) to this new form. The reader requiring further details is referred to the Appendix.

5.2. *Extending the stack language*

Let *F*, *e*, *G* be as in 5.1 and let us say *F* is being evaluated. Suppose it is necessary to erase the arguments of *F* and enter *G*. We achieve this by:

1. copying the link of *F* to the top of the stack,
2. erasing the arguments and original link of *F* by copying the link and arguments of *G* over them and adjusting pointers as appropriate, (there is no harm in doing this as the arguments of *F* will never be accessed again.)
3. initiating the evaluation of *G*.

Let us now add to the stack language the statement 'ERASE AND ENTER function-name' and let us define it to behave as



'ABC'@	On entry to LAST
'ABC'@'ABC'@	On entry to TAIL
'ABC'@'BC','	Equals test fails
'ABC'@	Execute ELSE Part
'ABC'@'ABC'@	Enter TAIL
'ABC'@'BC'@	ERASE & ENTER LAST
|____↑	Link copied here.
'BC'@	
'BC'@'BC'@	Enter TAIL
'BC'@'C','	Equals test fails
'BC'@	Evaluate ELSE part
'BC'@'BC'@	Enter TAIL
'BC'@'C'@	ERASE & ENTER LAST
|____↑	Link copied here
'C'@	
'C'@'C'@	Enter TAIL
'C'@','	Equals test succeeds
'C'@	Evaluate THEN part
'C'@'C'	Return
'C'	Value found

Fig. 3 Stack behaviour in evaluating ('ABC')LAST—Two methods of entry

above. We now use it to enter all functions whose arguments are enclosed in square brackets. (See 5.1)

Let us look at Examples 1 and 2 again.

Example 1 (Continued)
Functional definition of LAST using square brackets

```
LET (X) LAST BE
IF   (X) TAIL, '' ARE EQUAL
        THEN X
        ELSE [(X) TAIL] LAST
FI
END
```

New stack language program for (X) LAST

```
LOAD X
ENTER TAIL
LOAD ''
IF EQUAL THEN LOAD X
         ELSE LOAD X
              ENTER TAIL
              ERASE & ENTER LAST
FI
RETURN
```

The behaviour of the stack in evaluating ('ABC')LAST when both entry methods are used for entering a function is shown in **Fig. 3.**

Example 2 (Continued)
Functional definition of HCF using square brackets.

```
LET (M, N)HCF BE
     IF M, O ARE EQUAL THEN N
        ELSE [(M, N) DIFF, (M, N) MIN] HCF
     FI
END
```

Program for HCF

```
LOAD M
LOAD O
IF EQUAL THEN LOAD N
         ELSE LOAD M
              LOAD N
              ENTER DIFF
              LOAD M
              LOAD N
```

Downloaded from https://academic.oup.com/comjnl/article/18/1/23/454888 by guest on 18 April 2024

ENTER MIN
ERASE & ENTER HCF
FI
RETURN

The behaviour of the stack in evaluating (9, 6) HCF when both methods of entry are used is shown in **Fig. 4.**

It is clear that if recursion is implemented by using two methods of entering a function then a great saving of stack storage can be achieved. To make estimates of storage requirements let us make the simplifying assumption that each data or link item can be represented on the stack using a single cell. While this is not strictly true it is a convenient measure of stack storage required which *is independent* of the values of the arguments under consideration. Another possible measure of stack storage required is the number of links on the stack. This measure is also independent of the values of the arguments under consideration.

The function LAST (see Example 1) requires a maximum stack of four cells if both methods are used as opposed to $2 \times (\text{length}(X) + 1)$ cells, if only one method is used. Thus if X is a long character string the storage saved is very great. Similarly in Example 2 the storage saved is considerable if the depth of recursion is high.

Examples 1 and 2 are functions which require a bounded stack if both methods of function call are used. This result holds for all functions of go-to type and not just for the examples illustrated. We shall now proceed to prove the general theorem. We first note, however, that when Examples 1 and 2 were executed using both methods of function entry, there were at *most* two links on the stack. This property will play a key part in proving the result.

5.3. *The bounded storage theorem:*
Let S be a complete set of function definitions of go-to type and assume that every link or data item can be represented by using one stack cell. Then any function in S requires a *bounded* stack for its evaluation providing both methods of function entry are used as indicated. This bound depends on S but is independent of the values of the arguments of the function in question.

9, 6@	On Entry to HCF
9, 6@9, 0	Equals test fails
9, 6@	Execute ELSE part
9, 6@9, 6@	Enter DIFF
9, 6@3, 9, 6@	Enter MIN
9, 6@3, 6@	ERASE & ENTER HCF
|___↑	Link copied here
3, 6@	
3, 6@3, 0	Equals test fails
3, 6@	Execute ELSE part
3, 6@3, 6@	Enter DIFF
3, 6@3, 3, 6@	Enter MIN
3, 6@3, 3@	ERASE & ENTER HCF
|___↑	Link copied here.
3, 3@	
3, 3,@3, 0	Equals test fails
3, 3@	Execute ELSE part
3, 3@3, 3@	Enter DIFF
3, 3@0, 3, 3@	Enter MIN
3, 3@0, 3@	ERASE & ENTER HCF
|___↑	Link copied here.
0, 3@	
0, 3@0, 0	Equals test succeeds
0, 3@	Execute THEN part
0, 3@3	Return
3	Value found

Fig. 4 Stack behaviour in evaluating (9, 6)HCF—Two methods of entry

Proof:
In a go-to type function definition non-basic functions do not occur 'inside' other functions in the expression defining a function value. Neither are they involved in a test for equality. (See 3.1). Hence all non-basic functions will be entered by the ERASE & ENTER statement (See 5.1, 5.2).

We first show that during execution there are at most two links on the stack. Let S consist of the functions, $F_1, \ldots, F_p$ of $n_1, n_2, \ldots, n_p$ arguments respectively.

Initially say F_i is just entered. Then the stack configuration will be:

$$\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_{n_i} \ @$$

Now if a constant or an argument is loaded onto the stack this will increase the stack length but will not produce a new link. If a basic function is entered (by either method) this will produce a second link which will be erased after the basic function returns a value. The only other possibility is that a defined function say F_j will be entered. If this happens then it must be entered by the ERASE & ENTER statement. Hence a second link will appear at the top of the stack, then the arguments and original link of F_i will be erased and execution of F_j will be commenced with the stack as below.

$$\text{arg}_1^*, \ldots \text{arg}_{n_j}^* \ @$$

This leaves us in a position to apply the same analysis to F_j and establishes that there will be at most two links on the stack.

We now show that the stack is bounded. Suppose F_i is being evaluated. When can a new value be added to the stack? Clearly this can only occur when a constant or argument is loaded on the stack or when a basic function returns a value on the stack. We know that initially there are n_i arguments on the stack. Also, there can be at most two links on the stack. Thus if k_i is the number of constants, arguments and basic functions occurring in the expression defining F_i, the stack can grow to at most $(n_i + k_i + 2)$ cells. Of course after F_j is entered by the ERASE & ENTER statement there will be n_j arguments and one link on the stack and the same analysis can be applied to F_j. Hence since any function in S may be entered during the evaluation process, the storage required is at most

$$M = \max_{F_i \in S} (n_i + k_i + 2)$$ (Note this is just a bound, not necessarily the best bound.)

This proves the theorem.

In the discussion leading to 5.3—The Bounded Storage Theorem—we mentioned that we could get a measure of the storage space required which was independent of the values of the arguments in question by either:

1. assuming that each data or link item can be represented in one stack cell, or
2. taking the number of links on the stack as a measure of the stack storage required.

The bounded storage theorem is phrased in terms of the first measure. The next corollary is phrased in terms of the second measure.

5.4. *Corollary:*
Let S be as in 5.3. Then any function in S when executed will cause at most two links to appear on the stack.

Proof: See proof of 5.3.
We can calculate bounds (not the best bound) for Examples 1 and 2 using the methods described in 5.3.

Downloaded from https://academic.oup.com/comjnl/article/18/1/23/454888 by guest on 19 April 2024

A bound for Example 1 is 6 which is consistent with the behaviour of LAST in Fig. 3.

A bound for Example 2 is 9 which is consistent with the behaviour of HCF in Fig. 4.

The reader may ask how does the method described in Section 5 compare with that of Section 3 as far as storage requirements are concerned. It may be shown that if the method of Section 3 is used, go-to type definitions would:

1. require storage roughly proportional to 'n', the number of defined functions called during the evaluation,
2. require $n + 1$ links on the stack, n as above.

This demonstrates that if the two methods of entering a function are used, then a great deal of space on the stack can be saved, particularly with go-to type definitions. The author's ideas have been implemented by T. J. Pierre—his student, and these findings have been confirmed. Pierre (1972) used a single stack and implemented the ERASE & ENTER command by actually copying the link and arguments on the top of the stack *over* the previous arguments and adjusting the stack pointers accordingly.

Using this simple approach is not really satisfactory as it is rather inefficient in execution. The efficiency problem has been solved and we now present a technique of implementing the ERASE & ENTER command which eliminates the need for copying information down the stack. Two (or more) stacks are used and the bounded storage theorem applies to the method which will be described in the next section.

6. The two stack (or multi stack) method

Instead of implementing recursion using a single stack which is the conventional approach we use two (or more) stacks. Roughly speaking, at any stage only one stack is active and wherever a function is to be entered (by *either* method) the active stack is changed. By doing this we ensure that the link and arguments of the function being currently evaluated must always be on the top of the inactive stack. So the ERASE & ENTER command can now erase them by merely resetting a pointer. There is no copying involved any longer.

An important consequence of handling function entry in this way is that the back pointer to the link of the previous function is no longer required. This is because at all stages in the computation the link of the function being currently evaluated must be on the top of the inactive stack. There is thus no need to chain all the links on the stack together. Therefore by using this method of switching stacks on function entry we are able to simplify the link information and thus speed up the stacking mechanism. Furthermore, as we are merely describing a new way of executing the ERASE & ENTER command, it follows that the bounded storage theorem also applies to the two stack method.

Our description is based on using a single block of sequential storage for the implementation and letting one stack grow from the left hand end and the other stack grow from the right hand end. Of course there are other methods that may equally well be used for implementing two (or more) stacks but this one is particularly convenient for our purpose.

This completes the outline description of our method. (A two stack method has been used by Hassitt *et al* (1971) for handling storage allocation in APL, However they did not use it for handling function calls.)

The actual mechanisms required and the behaviour of the various operations are as follows.

1. Two stacks called the left stack and right stack and two top of stack pointers.
2. An indicator giving the current active stack, i.e. stack on which arguments, constants, links will be loaded and test operations performed.
3. The ENTER statement firstly creates a new link on the top of the active stack consisting of:
   (*a*) The return address
   (*b*) The value of the active stack indicator.
   Secondly the active stack is changed.
   Thirdly the function evaluation is initiated.
4. The ERASE & ENTER statement firstly creates a new link on top of the active stack by copying the link of the current function from the top of the inactive stack onto the top of the active stack.
   Secondly the link and arguments of the current function are erased from the top of the inactive stack.
   Thirdly the active stack is changed.
   Fourthly the function evaluation is initiated.
5. The RETURN statement in addition to its normal actions (see Section 4) will now also use the link on top of the inactive stack to reinstate the correct active stack.
   The value to be returned is put on top of the activated stack.
6. The LOAD and IF EQUAL-THEN-ELSE-FI statements work exactly as before. They use the *active* stack only. (See 2).

We now illustrate the behaviour of the two stacks for Examples 1 and 2. The symbols L or R at the start of each line indicate which stack is active. The links are now written as L or R so that we can tell which stack was active when the appropriate function was entered. The link and arguments of the function being currently evaluated are on top of the inactive stack.

Example 1 (Continued)
The two stacks' behaviour in evaluating ('ABC') LAST is shown in **Fig. 5.** We assume that initially the left stack is active and that the function is initially entered by executing LOAD 'ABC', ENTER LAST.

Example 2 (Continued)
The two stacks' behaviour in evaluating (9, 6) HCF is shown in **Fig. 6.** We assume that initially the left stack is active and that the function is initially entered by executing LOAD 9, LOAD 6, ENTER HCF.

L	'ABC'		Load 'ABC'
R	'ABC' L		Enter LAST
L	'ABC' L	R 'ABC'	Enter TAIL
R	'ABC' L	'', 'BC'	Equals test fails
R	'ABC' L		Execute ELSE part
L	'ABC' L	R 'ABC'	Enter TAIL
R	'ABC' L	'BC'	
L	'ABC' L link copied here →	L 'BC'	ERASE & ENTER LAST
L		L 'BC'	
R	'BC' L	L 'BC'	Enter TAIL
L	'C', ''	L 'BC'	Equals test fails
L		L 'BC'	Execute ELSE part
R	'BC' L	L 'BC'	Enter TAIL
L	'C'	L 'BC'	
R	'C' L ← link copied here	L 'BC'	ERASE & ENTER LAST
R	'C' L		
L	'C' L	R 'C'	Enter TAIL
R	'C' L	'',' '	Equals test Succeeds
R	'C' L		Execute THEN part
R	'C' L	'C'	Return
L	'C'		Value found

Fig. 5 Two stacks' behaviour in evaluating ('ABC')LAST

Downloaded from https://academic.oup.com/comjnl/article/18/1/23/454888 by guest on 19 April 2024

	Left stack	Right stack	
L	9		Load 9
L	9, 6		Load 6
R	9, 6 L		Enter HCF
R	9, 6 L	0, 9	Equals test fails
R	9, 6 L		Execute ELSE part
L	9, 6 L	R 6, 9	Enter DIFF
R	9, 6 L	3	
L	9, 6 L	R 6, 9, 3	Enter MIN
R	9, 6 L	6, 3	
L	9, 6 L → Link copied here	L 6 3	ERASE & ENTER HCF
L		L 6, 3	
L	3, 0	L 6, 3	Equals test fails
L		L 6, 3	Execute ELSE part
R	3, 6 L	L 6, 3	Enter DIFF
L	3	L 6, 3	
R	3, 3, 6 L	L 6, 3	Enter MIN
L	3, 3	L 6, 3	
R	3, 3 L ← Link copied here	L 6, 3	ERASE & ENTER HCF
R	3, 3 L		
R	3, 3 L	0, 3	Equals test fails
R	3, 3 L		Execute ELSE part
L	3, 3 L	R 3, 3	Enter DIFF
R	3, 3 L	0	
L	3, 3 L	R 3, 3, 0	Enter MIN
R	3, 3 L	3, 0	
L	3, 3 L → Link copied here	L 3, 0	ERASE & ENTER HCF
L		L 3, 0	
L	0, 0	L 3, 0	Equals test succeeds
L		L 3, 0	Execute THEN part
L	3	L 3, 0	Return
L	3		Value found

Fig. 6 Two stacks' behaviour in evaluating (9, 6) HCF

Example 3

Examples 1 and 2 have been go-to type function definitions. To illustrate the generality of the two stack method this example is of a function which is highly recursive and which always has value one. The data domain is the same as Example 2. The function (N) SUB—subtract one from N—is now basic.

```
LET (N) SUB BE BASIC
LET (M, N) H BE
        IF M, 1 ARE EQUAL THEN 1
                          ELSE [(N, (M)SUB)H,
                               (M)SUB] H
        FI
END
```

Two stack program for (M, N)H

```
LOAD N
LOAD 1
IF EQUAL THEN LOAD 1
        ELSE  LOAD N
              LOAD M
              ENTER SUB
              ENTER H
              LOAD M
              ENTER SUB
              ERASE & ENTER H
FI
RETURN
```

The Stacks' behaviour in executing (2, 2)H is shown in **Fig. 7.** We assume the left stack is initially active and that the function is entered by executing LOAD 2, LOAD 2, ENTER H.

Example 4

Examples 1, 2 and 3 have illustrated our method for direct recursion. This example is of an indirect recursive definition which is of go-to type. The data domain and basic function are as in Example 3. The functions to be defined by mutual recursion are:

(N) MOD0 = N modulo 3
(N) MOD1 = N + 1 modulo 3
(N) MOD2 = N + 2 modulo 3

The definitions are:

```
LET (N) SUB BE BASIC        —See Example 3.
LET (N) MOD0 BE
        IF N, O ARE EQUAL THEN 0
                          ELSE [(N) SUB] MOD1
        FI
END
LET (N) MOD1 BE
        IF N, O ARE EQUAL THEN 1
                          ELSE [(N) SUB] MOD2
        FI
END
LET (N) MOD2 BE
        IF N, O ARE EQUAL THEN 2
                          ELSE [(N) SUB] MOD0
        FI
END
```

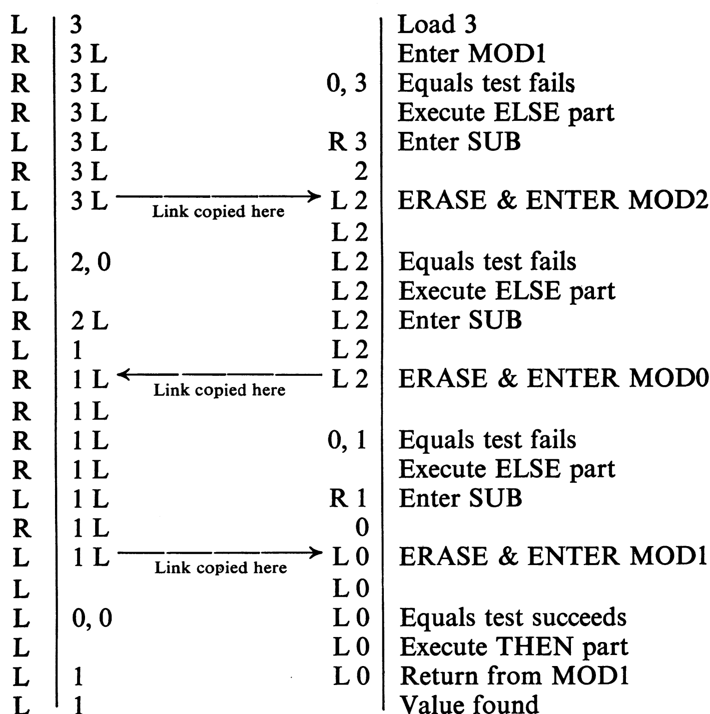
The two stack program for (N) MOD0 is given below. The programs for (N) MOD1 and (N) MOD2 are similar.

```
    LOAD N
    LOAD O
    IF EQUAL THEN LOAD O
            ELSE  LOAD N
                  ENTER SUB
```

	Left stack	Right stack	
L	2		Load 2
L	2, 2		Load 2
R	2, 2 L		Enter (2, 2) H
R	2, 2 L	1, 2	Equals test fails
R	2, 2 L		Execute ELSE part
L	2, 2 L	R 2, 2	Enter SUB
R	2, 2 L	1, 2	
L	2, 2 L	R 1, 2	Enter (2, 1) H
L	2, 2 L 2, 1	R 1, 2	Equals test fails
L	2, 2 L	R 1, 2	Execute ELSE part
R	2, 2 L 1, 2 L	R 1, 2	Enter SUB
L	2, 2 L 1, 1	R 1, 2	
R	2, 2 L 1, 1 L	R 1, 2	Enter (1, 1) H
R	2, 2 L 1, 1 L	1, 1 R 1, 2	Equals test succeeds
R	2, 2 L 1, 1 L	R 1, 2	Execute THEN part
R	2, 2 L 1, 1 L	1 R 1, 2	Return from (1, 1) H
L	2, 2 L 1	R 1, 2	Resume (2, 1) H
R	2, 2 L 1, 2 L	R 1, 2	Enter SUB
L	2, 2 L 1, 1	R 1, 2	
R	2, 2 L 1, 1 R ← Link c. h.	R 1, 2	ERASE & ENTER (1, 1) H
R	2, 2 L 1, 1 R		
R	2, 2 L 1, 1 R	1, 1	Equals test succeeds
R	2, 2 L 1, 1 R		Execute THEN part
R	2, 2 L 1, 1 R	1	Return from (1, 1) H
R	2, 2 L	1	Resume (2, 2) H
L	2, 2 L	R 2, 1	Enter SUB
R	2, 2 L	1, 1	
L	2, 2 L → Link copied here	L 1, 1	ERASE & ENTER (1, 1) H
L		L 1, 1	
L	1, 1	L 1, 1	Equals test succeeds
L		L 1, 1	Execute THEN part
L	1	L 1, 1	Return from (1, 1) H
L	1		Value found

Fig. 7 Two stacks' behaviour in evaluating (2, 2) H

Downloaded from https://academic.oup.com/comjnl/article/18/1/23/454888 by guest on 19 April 2024



L	3			Load 3
R	3 L			Enter MOD1
R	3 L		0, 3	Equals test fails
R	3 L			Execute ELSE part
L	3 L		R 3	Enter SUB
R	3 L		2	
L	3 L	Link copied here →	L 2	ERASE & ENTER MOD2
L			L 2	
L	2, 0		L 2	Equals test fails
L			L 2	Execute ELSE part
R	2 L		L 2	Enter SUB
L	1		L 2	
R	1 L	← Link copied here	L 2	ERASE & ENTER MOD0
R	1 L			
R	1 L		0, 1	Equals test fails
R	1 L			Execute ELSE part
L	1 L		R 1	Enter SUB
R	1 L		0	
L	1 L	Link copied here →	L 0	ERASE & ENTER MOD1
L			L 0	
L	0, 0		L 0	Equals test succeeds
L			L 0	Execute THEN part
L	1		L 0	Return from MOD1
L	1			Value found

Fig. 8 Two stacks' behaviour in evaluating (3) MOD1

```
                    ERASE & ENTER MOD1
    FI
    RETURN
```

The stacks' behaviour in executing (3) MOD1 is shown in **Fig. 8.** Initially we assume the left stack is active and that the function is entered by executing LOAD 3, ENTER MOD1.

Conclusion

The importance of the methods of evaluating recursive definitions we have described is that go-to (or iterative) type definitions require bounded stack storage for their evaluation. This means that go-to type definitions can be used for long iterative calculations *without* excessive storage requirements.

In addition the two stack method has simpler link information than the single stack method. This provides an additional saving in space required.

The question of execution time is one where it is difficult to give a definitive answer. Consider the problem of comparing the execution time of a non-recursive program with its equivalent go-to type recursive definition which say is evaluated by the two stack method. With a software implementation of the stacks the recursive definition would operate slower than its equivalent non-recursive program because of the overheads involved in stacking and unstacking arguments on function entry and exit. On the other hand with a hardware implementation of the two stacks with say several fast top of stack registers for both stacks this speed degradation may not occur in some cases and perhaps an increase in speed could result. This is because if the stacks are short they may both reside in fast registers—consider example 1 for instance. Thus it is difficult to give a definitive answer to this question because of the variety of feasible implementations of the two stack method. Perhaps this is an area where further research would be useful.

Finally, the author believes that further useful work could be done to evaluate the usefulness of purely recursive programming languages, i.e. languages with no assignment statement. There are many arguments for and against such languages and it is recognised that the basic method described here to make such a possibility practical would have to be extended. Nonetheless, the establishment of the bounded storage theorem suggests that this scheme may be more than just an interesting theoretical possibility.

Appendix Transforming function definitions

In this appendix we describe how to transform function definitions with round brackets only (see 2.1) into an equivalent stack language program (see 3.1) and into an equivalent function definition which utilises round and square brackets (see 5.1).

To transform a function definition in form 2.1 into a stack language program in form 3.1 the expression defining the function value is processed as follows.

1. Prefix all constants and variables by the word LOAD.
2. Prefix the word ENTER to all function names.
3. Delete all occurrences of '(), IF'.
4. Replace all occurrences of ARE EQUAL THEN by IF EQUAL THEN.
5. Replace the word END by RETURN.

To transform a function definition in form 2.1 into form 5.1 proceed as follows.

1. Set a counter C to zero.
2. Scan from left to right the expression defining the function value.
3. Whenever (or IF is encountered, increase C by 1.
4. Whenever) or THEN is encountered, decrease C by 1.
5. If just *before* encountering (, C is zero then change (to [.
6. If just *after* encountering), C is zero then change) to].

References

BARRON, D. W. (1968). *Recursive Techniques in Programming*, Macdonald/Elsevier Publishers.

HASSITT, E. A., LAGESHUTTE, J. W., and LYON, L. E. (1971). Implementation of a High Level Language Machine, *ACM 4th Annual Workshop on Microprogramming*.

HAUCK, E. A., and DENT, B. A. (1968). Burroughs B6500/7500 Stack Mechanism, *AFIPS SJCC*, pp. 245-251.

MCCARTHY, J. (1962). Towards a Mathematical Science of Computation, in *Information Processing* 1962, North Holland Publishers.

PIERRE, T. J. (1972). *Recursion in Functional Languages*, M.Sc. Thesis, Imperial College, London.

STRONG, H. R. (1971). Translating Recursion Equations into Flowcharts, *Journal of Computer and System Sciences*, pp. 254-285.

Downloaded from https://academic.oup.com/comjnl/article/18/1/23/454888 by guest on 19 April 2024