Subrecursive Programming

Languages for $\mathcal{Q}^n$

By

Robert L. Constable

Technical Report

70-53

April, 1970

Department of Computer Science
Cornell University
Ithaca, New York 14850

ACKNOWLEDGEMENTS

This work was supported by NSF contract GJ-579, and has profited greatly from the author's discussions with Professor Larry Travis of Wisconsin and John Hopcroft of Cornell. Its appearance is enhanced by the fine typing of Camille Taylor.

Page

CONTENTS

Subrecursive Programming Languages for $\mathcal{R}^n$

Robert L. Constable

Cornell University

§1 Introduction

In the last 10 years, computer scientists investigating computational complexity have turned their attention to classes of recursive functions such as the primitive recursive and elementary functions, which were well known in recursive function theory. They have found elegant new definitions for these previously difficult to motivate classes [3],[13]. They have discovered new and significant results about them, [3], [12], [13], and they have applied these classes in developing a theory of computation which in some instances may be useful to the applied computer scientist, [5], [13].

This paper will present new computer oriented definitions of certain well-known classes of recursive functions, namely the multiply recursive $\mathcal{R}^n$, functions of Péter. Hopefully this will make accessible to the various theoreticians in computer science more of the results of "ancient" (pre-1963) logic. At least it will facilitate the discussion of ultra complex functions when the need arises. Surprisingly the need to discuss them does arise frequently at least for $\mathcal{R}^2$ functions when teaching results about $\mathcal{R}^1$ functions and the various languages for computing them. Students want examples of non-$\mathcal{R}^1$ functions and computations. Researchers also want simple examples of functions which have very complex structural as well as computational complexity. $\mathcal{R}^2$ functions and the language given here serve these purposes.

§2 Preliminaries

<u>A model for program language semantics</u>

In the next subsection certain theoretical "programming languages" will be defined. It is well-known that an abstract machine model is convenient for describing the semantics of such languages. The following machine model is presented for this purpose. The model is an instance of Elgot & Robinson's [8] RASP (<u>Random</u> <u>Access</u> <u>Stored</u> <u>Program</u> <u>machine</u>), which in the way used here, is a very natural model whose characteristics will be familiar even to the reader unacquainted with the model.

Let the machine have as <u>contents</u> $B = \mathbb{N}$, and as <u>addresses</u> $A = \mathbb{N}$ for $\mathbb{N} = \{0,1,2,\ldots\}$.

The addresses identify computer <u>words</u> (or registers) which are (memory) locations holding elements of B. The contents of word a is denoted by k(a) for $k(\ ):A\to B$. A <u>memory configuration</u> of the RASP is simply a content function k( ). The set of all possible memory configurations is the set $B^A=K$ of all functions from A into B. The subset of functions of finite support is denoted K_o.

The operation of a RASP is determined by defining the effect of an instruction on the memory configuration, $k(\ )\in K$, and on the location of the next instruction which will control the computation. For the reader familiar with digital computers this definition is quite natural. A specific RASP is determined below by presenting the instruction schemata which generate its change of memory and change of control mappings

Def 2.1:

<u>instruction schemata</u>		<u>intuitive meaning</u>
mnemonic form	machine form	meaning
TRN(v,n)	$2^o\cdot 3^v\cdot 5^n$	<u>tran</u>sfer the <u>n</u>umber n to the location having address v

TRV(v,w)	$2^1 \cdot 3^v \cdot 5^w$	transfer the value in location w to the location having address v.
AN(v,w)	$2^2 \cdot 3^v \cdot 5^w$	add one to the value in location w and place the result in location v. (leaving w unchanged)
SN(v,w)	$2^3 \cdot 3^v \cdot 5^w$	if the contents of w are positive then subtract one and place the result in location v, otherwise place o in v; in either case leave the value of w unchanged
TRI(v,w)	$2^4 \cdot 3^v \cdot 5^w$	transfer indirectly by using the value of w, k(w) as the address, i.e. execute TRV(v,k(w))

The above table describes the effect of the instructions on memory. A complete description of the instruction includes specification of where to find the next instruction. This depends on how the instructions are stored in memory. For simplicity it is assumed here that all instructions are stored sequentially in <u>even</u> addressed locations. The <u>odd</u> numbered locations hold data. Intuitively this means that memory is divided into two banks - program bank and data bank.

For each of the above instructions, if it is located at address a (even), then the next instruction is found at location a+2. The conditional branch allows interruption of this fixed flow of control.

c(v.b)	$2^5 \cdot 3^w \cdot 5^b$	if the value of w is non-zero (positive) then the next instruction is taken from address b otherwise from address a+2.

These schemata will generate the transition functions H_1, H_2 of the RASP. Thus by the terminology of [ ] the abstract machine is $\langle \mathbb{N}, \mathbb{N}, 0, K_0, H_1, H_2 \rangle$ where K_o is the set of functions of finite support in $\mathbb{N}^{\mathbb{N}}$.

The language based on these instruction schemata is intuitively clear enough for the purposes of this paper that no difficulties should result from following this intuition. However, the formal RASP model in [ ] or [ ] is available to the reader if difficulties arise.

A higher level universal language G

The RASP machine language provides a formal semantic basis for all further programming languages considered here.

But in order to use the language for that purpose it is convenient to introduce a simple high level language as an intermediate step. Such a language is presented below using BNF supplemented by a restriction on program formation.

Def 2.2, Syntax for G:

<Letter> :: = A|B|...|X|Y|Z

<+digit>:: = 1|2|...|9

<digit> :: = 0| <+digit>

<label> :: = <+digit>|<label><digit>

<numeral> :: = <digit>|<label>

<variable> :: = <Letter>|<Letter><label>

<term> :: = <numeral>|<variable>|<variable>+1|<variable>∸1

<assignment> :: = <variable>←<term>|<label>,<assignment>

<conditional> :: = if <variable>≠0 then <label>|<label>,<condi

<go to+> :: = go to+<label>|<label> , <go to>

<go to-> :: = go to-<label> <label> , <go to>

<statement>:: = <assignment>;<conditional>:<go to+>;<go to->

<program> :: = <statement>|<program>;<statement>

A program is assumed to be deterministic, i.e. every stateme has unique lebels.

In simpler terms, variables are $A, A_1, A_2, \ldots, B, B_1, B_2, \ldots$ labels are 1,2,3,... assignment statements have the form

$$v \leftarrow w$$
$$v \leftarrow n$$
$$v \leftarrow w+1$$
$$v \leftarrow w-1$$

for v,w any variables and n a number. The conditional is of the form "if v≠o then L" for L a label. The language is quite elementary yet is capable of expressing algorithms for all computable functions over $\mathbb{N}$. For a proof see [8] or [19]. The following are examples of G programs

```
1   X←X-1                S←X
    if X≠o then 1        if X≠o then 1
    X←X+1                S←S+1
    X←X+1                X←X-1
                         go to -3
                     1   X←X+1
```

This high level type language is interpreted on a RASP by assigning variables to data locations, statements to machine language instructions, and machine instructions to locations. The details are as follows. Let

$\pi = s_1;s_2;\ldots;s_n$ be any G program and let $Var(\pi) = \langle V_1,\ldots,V_p\rangle$ be the list of distinct variables (in order of occurrence). A 1-1 function addr: $Var(\pi) \to$ Odd addr: $\{s_1,\ldots,s_n\} \to$ Even where $addr(s_i) = 2(i-1)$ is called a <u>loader table</u> for π. The table determines an even address corresponding to each label L of π. Let this address be denoted L^1. With respect to this table the statements of π are interpreted on the RASP as follows.

Def 2.3 Semantics for G:

<u>Location in π</u>	<u>G statement</u>	<u>machine language statement</u>
s_i	$v \leftarrow n$	TRN(addr(v),n)
s_i	$v \leftarrow w$	TRV(addr(v),addr(w))
s_i	$v \leftarrow w+1$	AN(addr(v),addr(w))
s_i	$v \leftarrow w \dot{-} 1$	SN(addr(v),addr(w))
s_i	go to + n	$C(p_1 : a_i+2n)$ where p_1 is the address of constant > o a is the address of s_i, i.e. $addr(s_i)$
s_i	go to − n	$C(p_1, a_i-2n)$
s_i	if $v \neq o$ then b	C(addr(v),b') where b' is the address of the statement of π labeled

The Loop language

A very elegant language for expressing primitive recursive functions has been defined independently by M. Minsky [ ] and D. Ritchie [ ]. Following Ritchie, the language is called the "Loop language". The syntax for a version of Loop is presented below using a context free grammar.

Def 2.4 Loop Syntax:

<Letter> :: = A|B|...|X|Y|Z

<label> :: = 1|2|3|...

<variable> :: = <Letter>|<Letter><label>

<term> :: = 0|<variable>|<variable>+1

<assignment> = <variable> ← <term>

<iterative> :: Do <variable> : <program>; END

<program> :: = <assignment>|<iterative>|<program>;<progr

Def 2.5 Loop semantics:

The Loop language is easily interpreted in terms of G, which in turn has been interpreted in the RASP machine language. Let v and w be any variables.

<u>Loop statement</u> or <u>program</u>	<u>interpretation</u> in G
$v \leftarrow o$	$v \leftarrow o$
$v \leftarrow v+1$	$v \leftarrow v+1$
if π is a program,	is interpreted as the program
DO v	$\bar{v} \leftarrow v$ TRV(add
π	1 if $\bar{v}=0$ then 2
END	$\bar{\pi}$
	$\bar{v} \leftarrow \bar{v}-1$
	go to 1
	where 2 labels the statement following END and where $\bar{v}$ does not occur in $\bar{\pi}$, the translation of π.

Note: the variables $\bar{v}$ is called the <u>loop</u> <u>control</u> variable of the DO-loop.

The Loop language as originally defined used the more restricted syntax of "$v \leftarrow v+1$", but the languages have the same computational power. In fact one can easily define in Loop programs which will carry out the following G instructions, for v, w_i any variables, $n \in \mathbb{N}$.

(1) $v \leftarrow n$

(2) $v \leftarrow w$

(3) $v \leftarrow w \dot{-} 1$

(4) $v \leftarrow w_1 + w_2$

(5) $v \leftarrow w_1 \cdot w_2$

(6) $v \leftarrow w_1 \div w_2$ †

for example. $x \leftarrow n$ is accomplished by the n+1 statements

$X \leftarrow 0$

$X \leftarrow X+1$

$\vdots$

$X \leftarrow X+1$

The instruction $X \leftarrow X \dot{-} 1$ is defined by the program.

```
S←0
DO X
X←S
S←S+1
END.
```

† assign to v the integer part of w_1/w_2.

and X←Y is defined by

```
X←0
DO Y
X←X+1
END
```

It is shown in Meyer & Ritchie [13] that the Loop language as defined by Ritchie, hence by the above remarks this version of the Loop language, expresses algorithms for all and only primitive recursive functions. That is, the class $\mathcal{R}^1$ of primitive recursive functions characterizes Loop.

The class $\mathcal{R}^1$ is the initial class of the earliest subrecursive hierarchy, $\mathcal{R}^1 \subset \mathcal{R}^2 \subset \ldots \subset \mathcal{R}^n \subset \ldots$, called the Péter hierarchy [15]. The class $\mathcal{R}^n$ is called the class of (nested) <u>n-fold</u> <u>recursive</u> <u>functions</u>. A language called Loopstack will be defined. It is characterized by $\mathcal{R}^2$. The language is then generalized to define $\mathcal{R}^n$ for all n.

The critical idea in defining $\mathcal{R}^2$ is to use a push down <u>stack</u> along with the DO-statement to define a deeper type of recursion than can be defined without the stack. A push down stack is a "first-on-last-off" memory device. The formal semantics for it can be defined on the RASP model. The operations wanted are:

Def 2.6: (a) $STACK_i \leftarrow X$ and (b) $X \leftarrow STACK_i$
where $STACK_i$ is the i-th stack which has been defined and (a) means put the value of X on top of the stack (i.e. push it down) and (b) means load the top element of the stack into X ("pop" the stack, thus releesing the top element).

In the RASP model the stack is defined by allocating an infinite subset of $\mathbb{N}$ as the possible memory region of the stack and using "pointers" to indicate its top element. The pointer is a memory location, named $STACK_i$, which contains the address of the top of the stack. Each computer word in the region allocated to $STACK_i$ has the form <data,address>. This form is achieved by using pairing functions to store and retrieve data.

The instruction (a) $STACK_i \leftarrow X$
has the effect of

(1) pair the contents of X and contents of the pointer, $STACK_i$

(2) place the pair in the next (free) location in the region allocated to $STACK_i$ after first "zeroing-out" the new location (this is accomplished by applying a function "$next_i(\ )$" to $STACK_i$).

(3) replace the contents of $STACK_i$ by "$next_i(STACK_i)$".

The instruction (b) $X \leftarrow STACK_i$ has the effect of

The instruction (b) $X \leftarrow STACK_i$ has the effect of

(1) "unpair" the contents of the word whose address is in $STACK_i$, and put the first component of the pair in X.

(2) replace the contents of $STACK_i$ with second component of the above pair.

These informal instructions can be directly translated into the RASP instruction set once the <u>region function</u>, $next_i($ and the pairing functions $P_2(\ ), s_1^2(\ ), s_2^2(\ )$ are specified. The following suffice:

$next_i(x) = 3^i \cdot 5^x$, $p_2(x,y) = (2^x \cdot (2y+1)-1$

$s_1^2(z) =$ largest u which divides z+1.

$s_2^2(z) = ([z+1/2^{s_1^2(z)}]-1)/2$. The $next_i(\ )$ function allows for the specification of ω-many stacks, and it leaves all of even addressed memory available for storage of programs.

§3 The Loopstack language.

The new language is obtained from the loop language by augmenting it with the instruction whose syntax is

$$\begin{array}{l} \text{LOOPSTACK}_i\ X \\ \pi \\ \text{END} \end{array}$$

where π is any program of the new language.

The semantics of the instruction will appear involved but the idea is actually quite simple. The instruction as given is intended to execute the program

```
DO X  ⎫
DO X  ⎬  X - times
 :    ⎪
DO X  ⎭
 π
END   ⎫
 :    ⎪
END   ⎬  X - times
END   ⎭
```

This is carried out by means of the following program in the language G augmented by the stack instructions as defined above.

Def 3.1: code   comments

```
  W←1
  if X=0 then go to 1
  N←1
  CLEAR STACK_1  ⎫        see remarks following program
  DO X           ⎬
  STACK_1←X      ⎪        load stack
  END            ⎭
1 W←STACK_1
```

	code	comments
*	DO W	perform the inner loop. (* refers to a key location described later).
	π	
	END	
2	if STACK EMPTY then exit	see remarks following program
	$Z \leftarrow STACK_1$	decrease first
	$Z \leftarrow Z \dot{-} 1$	non-zero stack
	$STACK_1 \leftarrow Z$	element
	if $STACK_1 \neq 0$ then 3	look for a non-zero entry in stack
	$N \leftarrow N+1$	keep track of depth of search for (
	$P \leftarrow STACK_1$	"pop stack" to remove 0 item.
	go to 2	
3	DO N	
	$STACK_1 \leftarrow X$	
	END	
	$N \leftarrow 1$	
	go to 1	

Remarks: The stack is "cleared" by setting the <u>base variable</u> to [0,0], i.e. contents 0 and next address 0, and setting the pointer (located at $3^i \cdot 5^0$) to have the base address, $3^i \cdot 5$. The fact that the address portion of the base is 0 allows easy recognition of the end of the stack.

Succintly written the language Loopstack or LS for short consists of {$v \leftarrow w$, $v \leftarrow w+1$, $v \leftarrow n$, DO, END, LOOPSTACK, ENI Each use of the pair DO, END results in the allocation of a new loop control register (the $\bar{v}$ in the def. p_). Similarly, each use of the LOOPSTACK, END pair results in the allocation of a new stack.

Def 3.2: The precise definition of the LS language involves

<stack iterative> ::= LOOPSTACK<variable>;<program>;END

and the program clause of Loop is replaced by

<program> ::= <assignment>|<iterative>|<stack iterative>
|<program> ; <program> .

computing functions

The programs of Loopstack, Loop or G will compute functions if inputs and outputs ara specified. Given a program π of any of these languages, let $Var(\pi) = \langle V_1, \ldots, V_n \rangle$ be the distinct variables of π in order of occurrence. Note the programs will be thought of in vertical form where spacing replaces ";" as the statement separator. The order of occurrence is then the top to bottom order. Note also that for Loop and LS programs the static order of occurrence (order in the program) is the same as the dynamic order of occurrence (order in which variables are used during execution) except for looping. This is of course not true in G.

Call a variable of π a <u>left</u> variable iff every possible <u>first</u> <u>occurrence</u> is on the left hand side (lhs) of an assignment statement but not on the right hand side (rhs).

Given a program π, let $I = \{X_1, \ldots, X_i\} \subseteq Var(\pi)$ and $O = \{Y\} \subseteq Var(\pi)$. The pair I/O is an input/output specification for π. The variables $Var(\pi)-I$ are called the <u>work</u> variables wrt (with respect to) I/O. If the work variables are all left variables, then <u>π</u> <u>with</u> I/O is a <u>well-defined</u> <u>function</u> <u>program</u> or simply function program for short.

Say that the function program <u>π</u> <u>with I/O</u> <u>computes</u> $f: \mathbb{N}^n \to \mathbb{N}$ iff when each X_i of $\{X_1, \ldots, X_n\} = I$ has initial value x_i the program π terminates with $f(x_1, \ldots, x_n) = y$ in Y, $\{Y\} = O$. A function f is <u>LS computable</u> iff there exists a $\pi \in LS$ such that π with I/O computes f.

For example,

```
Y←0
DO X
Y←Y+1
END
```

with $I = \{X\}$, $O = \{Y\}$

computes $f(x)=2x$. A simpler program for the same task is

```
DO X
X←X+1
END
```

$I = \{X\} = O$. An interesting example is the following program to compute $f(x,y) = x \dot{-} y$.

```
DO Y
    W←0
    DO W
      X←W
      W←W+1
    END
END
```

$I = \{X,Y\}$, $O=\{X\}$.

The variable W is a work variable and it is a left variable.

The simplest use of the LOOPSTACK instruction computes an enormous function. To understand what

```
LOOPSTACK X
X←X+1
END
```

computes requires examining a sequence of Loop programs.†

---

† If $Var(\pi) = V_1$ then the I/O is assumed to be $I=\{V\}= O$ for the purpose of discussing function programs.

Consider the sequence f_0 is X←X+1 and f_{n+1} is

```
DO X
f_n
END
```

These programs compute an interesting sequence of functio (written $f_n(\ )$ to distinguish them from their canonical programs f_n). $f_0(x)=x+1$, $f_1(x)=2x$, $f_2(x)=2^x\cdot x, \ldots$. The general rule for an algebraic definition of $f_n(\ )$ involves iterates. Given any f: $\mathbb{N}\to\mathbb{N}$ the iterates of f are defined by $f^{(0)}(x)=x$, $f^{(n+1)}(x) = f(f^{(n)}(x))$, which is called the n+1 st iterate of f.

The sequence $\{f_n(\ )\}$ is defined by $f_0(x) = x+1$ and $f_{n+1}(x) = f_n^{(x)}(x)$, i.e. by iteration of the previous function. The fact that the algebraic and computational definitions of $f_n(\ )$ are equivalent is easily proved by induction. Clearly the result holds at 0. Assume it holds at n, i.e. say that $f_n(x) = f_{n-1}^{(x)}(x)$. To prove it for $f_{n+1}(\ )$ note that

```
DO X
f_n
END
```

calculates $f_n(x)$ on the fi pass through the loop. Then since the input of f_n is th same as its output, namely X, on the second pass through the loop X has value $f_n(x)$ (though notice that the loop control register $\bar{X}$ has value X-1). Thus on the second p

$f_n(f_n(x)) = f_n^{(2)}(x)$ is computed. Proceeding by induction one can prove that on the m-th pass the value computed is $f_n^{(m)}(x) = \underbrace{f_n(\ldots f_n(f_n(x))\ldots)}_{\text{m-times}}$. The loop is executed x times, so $f_n^{(x)}(x)$ is computed.

The elegance of the DO, END instruction is that it describes iteration so intuitively.

One can appreciate the astronomical character of these functions by considering the value of $f_3(\ )$. Notice that $f_2(x) = 2^x \cdot x$ so that

$$f_3(3) = 2^{2^{2^{3}\cdot 3}\; 2^{3}\cdot 3} \cdot 2^{2^{3}\cdot 3} \cdot 2^{3}\cdot 3$$

$$> 2^{2^{24}} +24 +24 +4$$

$$> 10^{10^{6}}$$

So the number has more than a million <u>digits</u>. And since to produce the number 10^{10^6} by the f_3 program, at least 10^{10^6} add operations must be performed, it cen be estimated that at a processing speed of 10^{12} operations (one trillion) per second (one operation per pico-second) it would require 10^{10^6-12} seconds, or about 10^{10^6-22} <u>millennia</u> to compute this number.

The LOOPSTACK program

```
LOOPSTACK X
X←X+1
END
```

will be denoted f_ω . The function it computes is defined from the sequence $\{f_n(\ )\}$ by <u>diagonalization</u>. That is, $f_\omega(x) = f_x(x)$, and $f_\omega(x)$ is simply

$$\left.\begin{array}{l}\text{DO X}\\ \vdots\\ \text{DO X}\\ \text{X X+1}\end{array}\right\}\ \text{X-times}$$

$$\left.\begin{array}{l}\text{END}\\ \vdots\\ \text{END}\end{array}\right\}\ \text{X-times}$$

Proving this statement requires a careful analyses of the semantic definition of LOOPSTACK.X. The general theorem is given by defining

$$\boxed{\begin{array}{l}\mathrm{DO}_X^{\,Y}\\ \pi\\ \mathrm{END}\end{array}}$$

to be the program

$$\left.\begin{array}{l}\text{DO X}\\ \vdots\\ \text{DO X}\end{array}\right\}\ \text{X-times}$$

$$\pi$$

$$\left.\begin{array}{l}\text{END}\\ \vdots\\ \text{END}\end{array}\right\}\ \text{X-times}$$

for π any program.[†]

Theorem (Interpretation 3.1:

LOOPSTACK X
π
END

and

$DO_X X$
π
END

with IN=X, OUT=X compute the same function.

To facilitate understanding the proof, an example of

$DO_X X$
π
END

translated into the base language will be provided on the following page.

[†] In Constable [4] where program modification is used as the basic for computing functions like $f_\omega(x)$, this type of program is precisely defined in RASP environment. Here the interpretation is left informal.

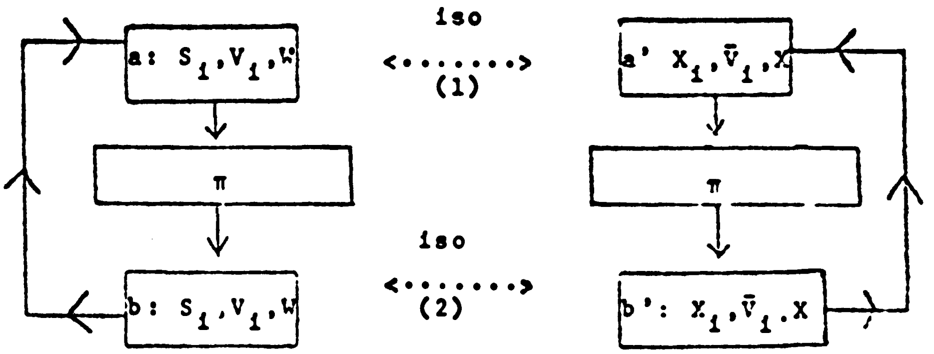
Example: for X=3

```
          X_1 ← X
 1        if X_1=0 then end_1
          X_2 ← X
 2        if X_2=0 then end_2
          X_3 ← X
 3        if X_3=0 then end_3
          π
          X_3 ← X_3 ∸ 1
          go to 3
 end_3    X_2 ← X_2 ∸ 1
          go to 2
 end_2    X_1 ← X_1 ∸ 1
          go to 1
 end_1    ________
```

The translator of LOOPSTACK X on p. 16 should be handy for comparison with the above example.

The two programs will be proved equivalent by defining a type of isomorphism between their computations. This constitutes a rather strong statement of equivalence. A computation for a RASP-like machine is defined in [8]. Of particular concern in this case are: the resgisters of π as they appear in each computation, (say $V_1,\ldots,V_n$ in LOOPSTACK and $\bar{V}_1,\ldots,\bar{V}_n$ in DO_X) the loop control registers, X_i, of the DO_X-program, and the locations on the stack which are loaded during the load stack operation, denoted by $S_1, S_2, \ldots, S_x$ (reading from top down so that S_1 is the first location loaded). Consider two key locations in the control of LOOPSTACK, the instruction after the one labeled 1 (DO W) and the one labeled 3 (DO N) (see * in code on p. 16). Call these locations a and b respectively. Other important locations are N←N+1 and the end_i of DO_X.

The first type of correspondence which holds is this. The V_i correspond to $\bar{V}_i, S_i$ correspond to X_i and W to X_3. Then two facts are important. (1) at location a of one program the configuration $\langle V_1, \ldots, V_n, S_1, \ldots, S_{x_1} W\rangle$ has the same value as its image in the other program at the "if $X_n = 0$ then end_i" instruction of the inner most DO-loop (call this location a'). (2) Also at location b the configuration has the same value that its image in the DO_X program has when it reaches the outermost end_i instruction before control transfers to an X_i←X after leaving the inner loop (call this location b', b_j' would be clearer notation since b' is a function of the number of steps; but b' will be adequate). These facts will be established below. The following diagram may clarify the situation.



The programs each execute π between a,b and a',b'. The LOOPSTACK program will recycle from point b only if not all $S_i=0$. Likewise DO_X will recycle from b' only if not all X_i are 0.

If either halts at point b, does not recycle in the above diagram, then either changes the registers of π. So given that (1) and (2) above are true, the programs are equivalent.

To conclude the proof of the theorem facts (1) and (2) must be verified. A careful inspection of the programs will quickly reveal this, but a proof is offered.

The facts are proved by induction for all X. Consider first the extreme cases X=0,X=1. For X=0 the result of LOOPSTACK X should simply be π (0 DO statements). Clearly it is. Also for X=1 the result is clearly simply

```
DO W
  π
END
```

for W=1.

For any X>1 when control first reaches a,a' the X_1=X and S_1=X and all values of π are initial, W=X. So (1) is true after no executions of π . The first time (2) is reached

```
DO W
  π
END
```

is executed and

```
DO X_x
  π
END
```

is executed

```
DO X_3
  π
END
```

is an example)

Thus W=0, X_3=0 and S_{x-1}=x-1. Also at the outermost end_1 as specified in defining (2), X_{x-1}=X-1. Clearly the registers of π are the same in each program.

Suppose that (1) and (2) hold after m steps, to show that they hold after m+1 steps. Between b and a the stack is loaded. Each 0 location in the stack is filled in (from S_j up to S_x) with the current value of X. Between b' and a' each 0 loop control register is filled with the value of X. No other changes occur. Thus upon reaching a and a', the configurations are the same, proving (1) for m+1.

To reach b from a, DO W is executed, leaving W=0. Also to reach b' from a' DO X_x is executed leaving X_x=0. In both cases the registers of π are the same after the loop. Since these are not effected again until a is reached again, the result is proved at least for those registers. Since the configurations are assumed equal at a,a', it is only necessary to show that the same actions are taken on S_i and X_i. First, the exit (at 2) is taken only when the stack is poped (P←STACK) at least x times, and control leaves DO_x only after the conditionals "if $X_i \neq 0$ then_" have been checked x-times. This can happen in each case only when all S_i=0 and all X_i=0. Therefore if exit is taken, then (2) is true. Assume the exit is not taken, then the LOOPSTACK procedure transfer to 3 only when some $S_j \neq 0$ j>x. Let j_0 be the largest such j. Let k_0 be the largest k>x for which $X_k \neq 0$. Claim $j_0 = k_0$. Note at a,a' all values are the same, and the value of S_{j+1} determines whether the value of S_j is changed. Likewise in DO_x, the value of X_{k+1} determines whether X_k is changed. The changes only occur when S_{j+1}=0 and X_k=0. Thus both X_{j_0+1}=0 and X_{k_0+1}=0. But this contradicts the definition of k_0,j_0 unless $j_0+1=x=k_0+1$.

In this case. if $S_{x-1}\neq 0$ then $S_{x-1}=X_{x-1}\neq 0$. After statement 2 in LOOPSTACK $S_{x-1}=S_{x-1}-1$ and likewise after end_x in $DO_x, X_{x-1}=X_{x-1}-1$. Therefore upon reaching statement 3 of LOOPSTACK, the corresponding values in each configuration are still the same.

Q.E.D.

Prop 3.2:

```
LOOPSTACK X
X←X+1
END
```

computes $f_\omega(\ )$.

Proof:

By definition $f_\omega(x)=f_x(x)$ and by the Interpretation Theorem the program computes $f_x(x)$

q.e.d

The program $f_{\omega+m}$ are defined for n=1,2...

by

```
DO X.
  f_{ω+n-1}
END
```

Clearly the function $f_{\omega+n}(\ )$ is defined by $f_{\omega+n}(x)=f^{(x)}_{\omega+n-1}(x0$ for n=1,2,...

The LOOPSTACK instructions can be nested to define yet faster growing functions. Define $f_{\omega\cdot(m+1)}$ for m=0,1,2, by

LOOPSTACK X
$f_{\omega\cdot m}$
END

For exemple, $f_{\omega\cdot 2}$ is

```
LOOPSTACK X
LOOPSTACK X
X←X+1
END
END
```

In such a program, two separate stacks are being used (just as in

```
DO X
DO X
X←X+1
END
END
```

two separate loop control regis

are used). The function computed by $f_{\omega\cdot(m+1)}$ is also defined by $f_{\omega\cdot(m+1)}(x)=f_{\omega\cdot m+x}(x)$.

§4 Using ordinal notations to describe program structur

A more refined set of functions can be used in place of the $f_n(\ )$ to yield sharper results. Let $h_1(x)=x+1$ and $h_n(x)=x+n$. The canonical program h_n will be

$$h_n \qquad \left.\begin{array}{l} X\leftarrow X+1 \\ X\leftarrow X+1 \\ \vdots \\ X\leftarrow X+1 \end{array}\right\} \text{n-times}$$

Then $h_\omega(x)=h_x(x)=x+x=2x=f_1(x)$. The canonical program for $h_\omega(\ )$ is f_1

```
DO X
X←X+1
END
```
.

Considering h_ω as a "limit" program suggests that the DO-block be represented in some way by the ordinal notation ω. Following this suggestion leads to a useful assignment of ordinal notations to programs.

The statement <variable>←<variable>+1 is assigned the numeral 1. Each pair (LOOPSTACK,END) (DO,END) determines a block, namely all statements in their scope. Every LS program π has the outer structure $b_1;b_2;\ldots;b_n$ where each b_i is an assignment statement or a block. An ordinal notation β_i is assigned to each block, b_i, and the notation $\beta_1+\beta_2+\ldots+\beta_n$ is assigned to π . To assign a notation to a block, assume that the program π_i which is the scope of th block b_i has been assigned notation γ_i, then the notation $(\gamma_i)\cdot\omega$ is assigned to DO<variable>; π_i;END. An assignment to LOOPSTACK<variable> ;π_i;END is considered below.

For example,

```
X←X+1
X←X+1
DO X
X←X+1
END
DO X
DO X
X←X+1
X←X+1
END
END
```

is assigned the ordinal notation

$(1+1) + 1\cdot\omega+(2\cdot\omega)\cdot\omega$.

The notation $(1+1) + (1)\cdot\omega +(2\cdot\omega)\cdot\omega$ simplifies to $2+\omega+2\cdot\omega^2$, and the ordinal it names is the same as that named by ω^2, therefore in terms of meaning, $2+\omega+2\cdot\omega^2=\omega^2$ (just as in arithmetic $(1+1)\cdot 3=6$). In this paper, the notation will be the main object of interest as opposed t the ordinal itself. This is indicated by writing "the notation $2\cdot\omega^2$" (just as in arithmetic one writes "the expression $(1+1)\cdot 3$").

An ordinal notation $<\omega^\omega$ will be in <u>Cantor</u> <u>normal</u> <u>form</u> (CNF) iff it appears as

$$\omega^{n_1}\cdot a_1+\omega^{n_2}\cdot a_2+\ldots+\omega^{n_p}\cdot a_p+a_{p+1}$$

where $n_i, a_i \in \mathbb{N}$. As is well known, every ordinal has a unique CNF. (see [20] for a proof.).

Theorem 4.1 If $\pi\in$ Loop and $\alpha(\pi)$ is the ordinal notation associated with π, then the maximum exponent of ω in the CNF of $\pi(\alpha)$ is the maximum depth of nesting of the DO-blocks(loop-complexity).

Proof: The only way that ω^n can appear in the CNF is if $c_1\cdot\omega\cdot c_2\cdot\omega\ldots c_n\cdot\omega$ appears in $\pi(\alpha)$ which means that DO-END pairs are nested n-deep. q.e.d.

The ordinal notation assigned to a LOOPSTACK-block, LOOPSTACK <variable>;π_1;END is $(\gamma_1)\cdot\omega^\omega$ where γ_1 is the notation assigned to π_1. With this rule, ordinals can be assigned to all $\pi\in$LS.

Example:

```
X←X+1
DO X
LOOPSTACK X
DO X
X←X+1
END
END
END
LOOPSTACK X
LOOPSTACK X
X←X+1
X←X+1
END
END
```

is assigned the ordinal notation

$1+ (\omega\cdot\omega^{\omega})\cdot\omega + (2\cdot\omega^{\omega})\cdot\omega^{\omega}$

which simplifies to

$1+\omega\cdot\omega^{\omega+1} + 2\cdot\omega^{\omega\cdot 2}$

The ordinals denoted by $\alpha(\pi)$ for $\pi\in LS$ are less than ω^{ω^2}. The CNF for these ordinals is

$$\omega^{\beta_1}\cdot a_1 + \omega^{\beta_2}\cdot a_2 +\ldots+\omega^{\beta_n}\cdot a_n+a_{n+1}$$

where $a_i\in\mathbb{N}$ and $\beta_i > \beta_{i+1}$ and $\beta_i < \omega^2$ is in CNF.

From $\alpha(\pi)$ the structural complexity of π can be determined.

Def: The stack complexity of an LS π is n iff the deepest nesting of LOOPSTACK instructions is n. The pure (0) loop complexity of π is n iff the deepest nesting of DO statements around a program α not containing a LOOPSTACK statement is n. The m-loop complexity of π is the maximum depth of nesting around programs of stack complexity m.

Prop: The stack and m-loop complexity of a Loopstack program can be effectively determined from the program.

Prop 4.2: The stack complexity of $\pi \in LS$ is the maximum n such that $\omega \cdot n+m$ occurs as an exponent of ω in $\alpha(\pi)$. The n-loop complexity of π is the maximum m such that $\omega \cdot n+m$ occurs as an exponent of ω in $\alpha(\pi)$.

Proof: for the reader.

§5 $\mathcal{R}^2$ characterizes Loopstack

The Loopstack language can express only terminating programs.

Theorem 5.1 If $\pi \in LS$, then $\pi(x)\downarrow$ for all x.

Proof:

A proof involves two assertions: (1) if π terminates for all x, then

```
DO X
π
END
```

halts for all x. This is elementary.

(2) If halts for all x, then LOOPSTACK X

π

END

hlats for all x. This is clear by the Interpretation Theo since for any X it is equivalent to

DOX

X

DOX

π

END

X

END

which can be reduced to case (1) by induction.

Q.E.D.

As in the Loop language, the run-time of Loopstack programs can be estimated from the syntax of the program.

Def 5.1: Given Loopstack program π, the run-time of π on input x is denoted by $\sigma\pi(x)$ and is defined using run-time of the RASP program image of π.

Theorem 5.2: If $\pi \in LS$ and its stack complexity is m and the m-loop stack complexity is n, then there is a t such that

$$\sigma\pi(x) < f^{(t)}_{\omega\cdot m+n}(x) \quad \text{for all x.}$$

The proof of this theorem is best broken down into simple lemmata. Essential to the analysis are the following properties of the functions $f_{\omega\cdot n+m}(\ )$, called <u>spine functions</u> after the terminology of [6].

Lemma 5.1: If $\alpha<\beta<\omega^2$ and α,β are in CNF, then

(i) $f_\alpha(\ )$ is strictly increasing in x

(ii) $f_\alpha(x)<f_\beta(x)$ a.e.x (therefore $\exists p f_\alpha(x)<f_\beta^{(p)}(x)$ for all x)

(iii) $\exists q,t f_\alpha(f_\beta(x))<f_\beta^{(q)}(x)$ and $f_\beta(f_\alpha(x))<f_\beta^{(t)}(x)$ for all x.

Proof: elementary

The next two lemmata are not required for this proof, but they show that $\{f_\alpha\}$ forms a <u>spine</u> in the sense of [6].

Def 5.2: Let $\sigma_1 f_\alpha(x)$ for α in CNF denote the number of times the "X←X+1" statement is executed in f_α.

Prop 5.3: For all f_α , $< \alpha < \omega^2$ and α in CNF,

$$\sigma_1 f_\alpha(x)+x=f_\alpha(x).$$

Proof: elementary, simply write down the definitions and compare q.e.d.

Prop 5.4: For α as above, $\sigma f_\alpha(x) \leq c_\alpha \cdot \sigma_1 f_\alpha(x)$ $c_\alpha \in \mathbb{N}$.

Proof: It is easily verified that depending on the form of α , the "overhead" instructions executed in getting to the inner loop are bounded by a constant times the value of x . q.e.d.

To assist further with theorem 5.2 , define

Def 5.3: The instruction pair LOOPSTACK,END can be regarded as an operator, $O_{\omega^\omega}[f(\ )]$, on one argument functions f( ). Using a program f with inputX and output

$O_{\omega^\omega}[f(\ )](x)$ is output of

```
LOOPSTACK X
f
END
```

for X hav
initial v
x

The iteration operator is denoted O_ω and defined as

$O_\omega[f(\ )](x)$ output of

```
DO X
f
END
```

when X starts at x.

Lemma 5.2: If $f(x) < g(x)$ for all x, and g( ) is monotc increasing then

(i) $O_\omega[f](x) < O_\omega[g](x)$ for all x

(ii) $O_{\omega^\omega}[f](x) < O_{\omega^\omega}[g](x)$ for all x

Proof: elementary

With these lemmata the theorem is easily proved. It restated for convenience.

Theorem 5.2: If $\pi \in LS$ and $\alpha(\pi) = \omega \cdot m + n$, then $\exists t$ such that

$$\sigma\pi(x) < f^{(t)}_{\omega \cdot m+n}(x) \quad \text{for all } x.$$

Proof: by induction or m and n

For $m = o$ and $n \leq o$ the result is known from [13]. Assume the result for m and all n. Then for a proof by induction the result must be established for m+1 and all n. First for n=o, let

```
LOOPSTACK X
  π1
END
```

be the first occurrence of the m+1 deep stack instructions. Thus the stack complexity of π_1 is m and the m-loop complexity is some q. By assumption

$$\sigma\pi_1(x) < f^{(p)}_{\omega \cdot m+q}(x) \qquad \text{for all } x.$$

Since by Lemma 1 $f^{(p)}_{\omega \cdot m+q}(x)$ is monotone, Lemma 2 implies

$$0_{\omega^\omega}[\pi_1(\ )](x) < 0_{\omega^\omega}[f^{(p)}_{\omega \cdot m+q}](x) \quad \text{for all } x.$$

Notice that at this point it is almost trivial to observe that $f^{(p)}_{\omega \cdot m+q}(x) < f^{(p')}_{\omega(m+1)}(x)$ and consequently that

$$0_{\omega^\omega}[\pi_1(\ )](x) < 0_{\omega^\omega}[f^{(p)}_{\omega \cdot (m+1)}(x)](x) = f^{(p)}_{\omega \cdot (m+2)}(x) \quad \text{for all } x.$$

This will establish a slightly weaker theorem (with (m+1) for m in the right hand side of the inequality).

A more careful analyses shows that in fact

$O_{\omega^{\omega}}[\pi_1(\ )](x) < f^{(p^1)}_{\omega\cdot(m+1)}(x)$ for all x because

$O_{\omega^{\omega}}[f^{(p)}_{\omega\cdot m+q}](x) = f^{(p)}_{\omega\cdot m+q+x}(x)$, using the Interpretation Theorem, and

$$f^{(p)}_{\omega\cdot m+q+x}(x) < f^{(q)}_{\omega\cdot(m+1)}(x)$$

since $f^{(q)}_{\omega\cdot(m+1)} > f^{(p)}_{\omega\cdot m+b(x)}(x)$ whenever

$b(x) < f^{(q-1)}_{\omega\cdot(m+1)}(x)$.

The above analyses establishes that $f^{(p')}_{\omega(m+1)}(x)$ bounds the value of $O_{\omega^{\omega}}[\pi_1(\ )](x)$. To bound the run-time observe that the run time is proportional to

$$\sigma\pi_1(x)+\sigma\pi_1(\pi_1(x))+\ldots+\sigma\pi_1(\pi_1^{(b(x))}(x)) \quad \text{for some } b(x)$$

Indeed b(x) can be no larger than the maximum value of $O_{\omega^{\omega}}[\pi_1(\ )](x)$ (since b(x) measures the number of times through the inner most loop of $DO_X X$). Thus an extremely crude, but adequate estimate of the run time is

$$\sigma O_{\omega^{\omega}}[\pi_1(\ )](x) < f^{(p^1)}_{\omega\cdot(m+1)}(x) \cdot f^{(p)}_{\omega\cdot m+q}(f^{(b(x))}_{\omega\cdot m+q}(x)) \cdot$$

where $b(x) < f^{(p^1)}_{\omega\cdot(m+1)}(x)$.

Since as above $f^{(p)}_{\omega\cdot m+q}(x) < f^{(p^1)}_{\omega\cdot(m+1)}(x)$ it follows that there is a t such that

$$\sigma O_{\omega^{\omega}}[\pi_1(\ )](x) < f^{(t)}_{\omega\cdot(m+1)}(x) \text{ for all } x$$

as was to be shown.

This establishes the theorem for m+1, n=0. The case for n>0 is handled by first reducing the inner loop as above and then applying the standard arguments for the Loop case to obtain the bound $f_{\omega\cdot(m+1)+n}(x)$.

q.e.d.

characterization of the language

The Loopstack language can easily be characterized using the class $\mathcal{R}^2$ of doubly recursive functions. From Robbin [17] two critical facts are known:

Theorem 5.5: $\mathcal{R}^2 = \bigcup_{\alpha<\omega^2} \mathcal{E}(f_\alpha)$

Theorem 5.6: $\phi_1 \in \mathcal{R}^2$ iff $\sigma\phi_1(x) \leq f_\alpha(x)$ a.e.x for $\alpha<\omega^2$

From these two theorems

Theorem: $f \in \mathcal{R}$ is LS computable iff $f \in \mathcal{R}^2$

Proof:

(1) To show $\mathcal{R}^2 \subseteq$ LS: It is known by definition that $f_\alpha \in$ LS for all $\alpha<\omega^2$. It is also known that $\mathcal{E}(f_\alpha) = \mathcal{R}^1(f_\alpha)$, thus since $\mathcal{R}^1(f_\alpha) \subseteq$ LS for all $\alpha<\omega^2$, it follows by Thm 5.6 above that $\mathcal{R}^2 = \bigcup_{\alpha<\omega^2} \mathcal{E}_2(f_\alpha) \subseteq$ LS.

(2) To show that $LS \subseteq \mathcal{R}^2$ note that any Loopstack program π is a RASP program of a special sort. By thm 5.2 on estimating run times it is known that $\sigma\pi(x)<f_\alpha(x)$ for all x, and some $\alpha<\omega^2$. Thus by Thm 5.6 above, $\pi(\ )\in\mathcal{R}^2$

q.e.d.

<u>efficiency of the language</u>.

According to the last theorem, Loopstack can be interpreted as a language for $\mathcal{R}^2$. A general programming language, like G used earlier to define Loopstack, can also compute $\mathcal{R}^1$ and more. The added flexibility of G may make it possible to compute elements of $\mathcal{R}^2$ more efficien But there is a limit to the possible added efficiency as the following theorem shows.

Theorem 5.7: Let $g(x)=x^{(x)}$, then for all $f\in\mathcal{R}^2$ and for all $\phi_i(\ )=f(\ )$ such that $\sigma\phi_i(\ )<k(\ )\in\mathcal{R}^2$, there is an LS program $\alpha_j(\ )=f(\ )$ such that

$$\sigma\alpha_j(x) \le c\cdot a(\sigma\phi_i(x)).$$

Remarks: The theorem says that LS programs need not be more than exponentially inefficient. Granted that for practical purposes this is terrible, but things could be worse. For instance it might be that for any $h\in\mathcal{R}^2$ there is $f\in\mathcal{R}^2$ such that $\phi_i(\ )=f(\ )$ and for <u>all</u> $\alpha_j(\ )=f(\ )$,

$$h(\sigma\phi_i(x)) < \sigma\alpha_j(x) \text{ for all } x.$$

That situation would be really terrible.

Proof:

The theorem is proved by simulating ϕ_i in Loop and using $f_\alpha, \alpha<\omega^2$ as a clock to shut off the simulation. The clock can be computed in parallel and turned off when the simulation stops. The g-cost enters in estimating the "over run" on the clock. This is the time it requires for the clock to run down once the simulation program has sent the signal to stop (has stopped the clocks "feed back"). The details of such a procedure are given in Constable & Borodin [5].

The basic idea is represented by the following diagram. Let $\bar{\phi}_i$ be the program simulating ϕ_i and suppose for illustration that $\sigma\phi_i(x) < f_{\omega+2}(x)$ all x.

```
LOOPSTACK X
DO X
DO X
φ̄i
Z←Z+1
END
*
END
*
END
```

where * is the translation of the following line into Loop

* if S=0 then X←Z else X←0.

The variable S will remain 0 until the simulation, of $\bar{\phi}_i$, stops; then S is increased to 1 signalling the clock to stop (i.e. stopping the feed back of Z into X which is what allows computation of $f_{\omega+2}$).

When the clock receives the signal to stop, the largest value in the stack will be Z, a value equal to the number of steps of ϕ_i simulated (close to $\sigma\phi_i(x)$). Once S=∅, LOOPSTACK runs until all stack elements are 0. Since there is no feedback, this process requires $c \cdot Z^Z$ steps because

$$\left.\begin{array}{l}\text{DO } Z\\ \vdots\\ \text{DO } Z\end{array}\right\} Z \qquad \text{requires } Z^Z \text{ steps}$$

$$X \leftarrow K+1$$

$$\left.\begin{array}{l}\text{END}\\ \vdots\\ \text{END}\end{array}\right\} Z$$

q.e.d.

The cost of using LS programs can be reduced to a simple constant factor of the cost in G by having an instruction which allows immediate exit from any loop.

A sufficient instruction would be a "go to" of the form "go to + c" which means go to the instruction located c instructions below this one. With such an instruction the program of Thm 5.7 would be modified to use for * the following.

* if S=0 then X←Z else go to + c

where c causes a branch beyond the end of the program (causing a halt).

The language could also be made more flexible by breaking LOOPSTACK into two separate instructions:

(1) STACK←X, load stack

(2) DO STACK X, regard the stack as described in lines 8 through 24 of the definition of LOOPSTACK.

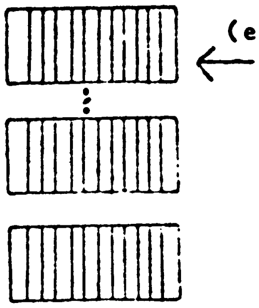
Then LOOPSTACK X is equivalent to

```
DO X
STACK_1←X
END
DO STACK_1 X
π
END.
```

§6 Conclusion

generalizations to $\mathcal{R}^n$

The LOOPSTACK type of instruction can be generalized to "multi-dimensional" stack-like lists. A two-dimensional stack can be regarded as a stack of stacks



(each turned on its side)

The command to

```
LOOPSTACK² X
  π
END
```

would cause a 2-dim stack to be loaded as

$$\overbrace{\begin{matrix} X,X,\ldots,X \\ X,X,\ldots,X \\ \vdots \quad\quad \vdots \\ X,X,\ldots,X \end{matrix}}^{X\text{-times}} \left.\vphantom{\begin{matrix} X \\ X \\ \vdots \\ X \end{matrix}}\right\} X\text{-times}$$

Then the top level stack would execute LOOPSTACK^{2}X. When it finished, the top most element on the next stack would be decreased and the top stack would be loaded again by the current value of X, which may cause a configuration like

$$\begin{array}{l} X^1,X^1,\ldots,X^1,X^1,\ldots,X^1 \quad . \\ X,\ X\ ,\ldots,X \\ \vdots \qquad\quad \vdots \\ X,\ X,\ldots\ \ X. \end{array}$$

Then the top stack would execute LOOPSTACK X again.
The effect of continuing this is to perform.

$$\left.\begin{array}{l} \text{LOOPSTACK X} \\ \qquad\vdots \\ \text{LOOPSTACK X} \end{array}\right\} \text{X-times}$$

$$\pi$$

$$\left.\begin{array}{l} \text{END} \\ \quad\vdots \\ \text{END} \end{array}\right\} \text{X-times.}$$

When π is $X \leftarrow X+1$ the function computed is $f_{\omega^2}(x)=f_{\omega\cdot x}(x)$.

A new hierarchy of functions, $f_{\omega^2}, f_{\omega^2\cdot 2},\ldots,f_{\omega^2\cdot n},\ldots$ is obtained by nesting the two-dimensional stack statements.

Clearly the whole process can be carried over to LOOPSTACK^n X where LOOPSTACK^n X computes

X←X+1

END

$f_{\omega^n}(x) = f_{\omega^{n-1}\cdot x}(x)$.† Using the results of Constable [25] it is easily shown as in §5 that LS^n is characterized by $\mathcal{R}^n$.

Moreover the structural complexity of the programs determines an uppor bound on the run-time as in Thm 5.2. The structural complexity can be characterized by an ordinal of the form

$$\alpha = \omega^{\beta_1}\cdot a_1+\omega^{\beta_2}\cdot a_2+\ldots+\omega^{\beta_n}\cdot a_n+a_{n+1} < \omega^{\omega^{\omega}}$$

where $\omega^{\omega^{n_1}}$ describes the maximum depth of nesting of LOOPSTACK^{n_1} instructions. The details of this are similar enough to the case ω^2 that they can be left to the interested reader. The case of $\mathcal{R}^2$ is general enoug to illustrate the ideas yet simple enough to provide an easily intelligible presentation. The case $\mathcal{R}^2$ is also the most needed for pedagogical purposes.

---

† For instant headache the reader might contemplate the value of $f_{\omega^3}(7)$.

historical note.

It is interesting to note the long line of research leading to a computational approach to $\mathcal{R}^1$ and $\mathcal{R}^n$ for n>1. The primitive recursive functions appeared as early as 1888 in the work of Dedekind [22]. In 1925 (published 1928) Ackermann [22] produced the first known non-primitive recursive function. Both R. Péter and S. Kleene studied primitive recursive functions extensively in the mid-1930's. They were useful in logic (Kleene [7]) and number theory (Péter [15]) but there was no natural reason known for studying them.

In 1947 R.M. Robinson [18] considerably simplified the methods of dealing with primitive recursion by reducing it to iteration. In 1953 Grzegorczyk [9] contributed rather deep insights into the structure of $\mathcal{R}^1$ by defining the Grzegorczyk hierarchy. This work stimulated P. Axt [1] and J.P Cleave [3] to investigate $\mathcal{R}^1$ still further. Cleave motivated by the work of R. Ritchie [16] in relating function classes (ε) to computational complexity, gave the first computational account of $\mathcal{R}^1$. Building on the work of Robinson & Axt, D. Ritchie and M. Minsky offered an elegant syntactical definition of $\mathcal{R}^1$ in terms of programming languages.

Finally McCreight & Meyer [12] established a relationship between $\mathcal{R}^1$ and a wide class of computational complexity measures Φ and their complexity classes, $\mathcal{R}^{\Phi}_t$.

At this point a good deal of deep knowledge has been accumulated about $\mathcal{R}^1$. It appears to be a natural and fundamental class, but there is as yet no satisfactory account of why this is so. Can one say that its importance is due to more than convenience and historical chance?

The investigation of $\mathcal{R}^n$ for $n>2$ has been more or less "academic". These classes have been interesting examples, but they have not been extensively "used" in logic, number theory or computing theory. Only a few workers have studied them in the litarature. They appeared first in Péter's work, 1936. R.M. Robinson examined double recursion in 1948. A deep anlysis of their structure (in relation to ordinal recursion) appeared by Tait [21] in 1961. This work inspired Robbin [17] 1965 to analyse their structure systemetically in terms of Grzegorczyk's work on $\mathcal{R}^1$ and in terms of the Hartmanis & Stearns [10] approach to computational complexity. Likewise the work of Cleave led Constable [4] 1968 to examine $\mathcal{R}^n$ from the viewpoint of the Ritchie, Cleave approach to computational complexity.

The work of Lachlan [11] 1962 or $\mathcal{R}^n$ and unpublished work of McCleary [Now, Notre Dame J. of Logic _] on $\mathcal{R}^1$ was helpful[1] to Constable, but the work of Robbin on $\mathcal{R}^n$ and [1... on $\mathcal{R}^1$ were available for use in the final version of [4].

The work of McCreight & Meyer 1969 is also applicable to an analysis of $\mathcal{R}^n$. Again for these classes one is curious about their apparent naturalness.

Open problems

Among the interesting problems suggested by this work are these.

(1) Discover why (or whether) the $\mathcal{R}^n$ classes are natural and seek the natural extension of $\mathcal{R}^n$ into the transfinite. Constable [4] is a beginning on the problem. There a hierarchy up to ε_0 is constructed. The question is how natural is that hierarchy?

(2) In pursuit of (1) it is desirable to establish relationships between the $\mathcal{R}^\alpha$ classes and all other known subrecursive hierarchy classes (see [4] for a more complete discussion of this).

(3) Discover a computer oriented definition for $\mathcal{E}$, the elementary functions, which has the same elegance as the Loop languages for $\mathcal{R}^1$.

In particular it is interesting to find relationships between subrecursive classes and properties of programming languages. Such relationships allow an interesting characterization of languages or segments of languages and a method for classifying and comparing them.

1]. Axt, Paul "Enumeration and the Grzegorczyk Hierarchy," Z. Math Logik Grund.Math 9 (1963), 53-65.

2]. Bass, L.J. and P.R. Young, "Hierarchies Based on Computational Complexity and Sneqularities of Class Determined Measured Sets," ACM Symposium on the Theory of Computing, 1970.

3]. Cleave, John P., "A Hierarchy of Primitive Recursive Functions," Zeitschr. F. Math Logik and Grund.D. Math., 9, (1963) 331-345.

4]. Constable, Robert L. Extending and Refining Hierarchies of Computable Functions. Comp. Sci. Tech. Report #25. Univ. of Wisc., 1968.

5]. Constable, Robert L. and Allan B. Borodin "On the efficiency of programs in subrecursive formalisms," Comp. Sci. Tech. Report 70-53, Cornell Univ., 1970.

6]. Constable, R.L. "On the Size of Programs in Subrecursive Formalisms," ACM Symposium on the Theory of Computing, 1970, 1-9.

7]. Davis, M. <u>The Undecidable</u>, Raven Press, 1965.

8]. Elgot, C. C. and A. Robinson "Random-Access Stored Program Machines, An Approach to Programming Languages," J.A.C.M., 11, (1964), pp. 365-399.

9]. Grzegorczyk, A. "Some Classes of Recursive Functions," Rozprawy Mathematcyzne, (1953), 1-45.

10]. Hartmanis, J. and R.E. Stearns "On the Computational Complexity of Algorithms," Trans. AMC, 117, 5, (1965), 285-306.

11]. Lachlan, S.H. "Multiple recursion," Zeitschr. f. math Logik und Grund. d. Math., 8, 1962, pp. 81-107.

12]. McCreight, E.M. and A. R. Meyer "Classes of computable functions defined by bounds on computations," ACM Symp. on Theory of Computing, 1969, 79-88.

13]. Meyer, A. R. and D. M. Ritchie "The complexity of Loop programs,". Proc. 22 National ACM Conf. (1967). 465-470.

14]. Minsky, M. Computation, Finite & Infinite, Prentice-Hall, 1967.

15]. Péter, Roze Recursive Functions, 3d ed., New York, 1967.

16]. Ritchie, Robert W. "Classes of Predictably Computable Functions, Trans." A.M.S., 106, 1963, pp. 139-173.

17]. Robbin, Joel Subrecursive hierarchies, Ph.D., Diss, Princeton, 1965.

18]. Robinson, R.M. "Primitive recursive functions," Bull AMS, 53, (1947), 915-942.

19]. Shepherdson, J.C. and H.E. Sturgis "Computability of Recursive Functions," J.A.C.M., 10, 1963, pp. 217-255.

20]. Sierpinski, Waclaw. Cardinal and Ordinal Numbers. Warsaw, 1965.

21]. Tait, W.W. "Nested recursion." Math. Ann., 143. 1961, pp. 236-250.

22]. van Heijenoort, J. From Frege to Godel, Cambridge, 1967.