Subrecursive Programming Languages III,[†]
The Multiple Recursive Functions, $\mathcal{R}^n$

Robert L. Constable
Cornell University

71 - 99
Technical Report

June, 1971

Department of Computer Science
Cornell University
Ithaca, New York 14850

Subrecursive Programming Languages III,[†]

The Multiple Recursive Functions, $\mathcal{R}^n$

Robert L Constable
Cornell University

ABSTRACT

The characterization of program structure is an elusive aspect of the theory of programming languages. Contingent on such a characterization is an interesting study of the trade-off relationships between program structure and efficiency, between structure and intelligibility, between structure and size, and so forth.

One promising definition of program structure is suggested by the studies of the structure of recursion schemata pursued by logicians in the 1930's. In this paper we carry over some of these ideas to programming and present new computer-oriented definitions of certain well-known classes of recursive functions, namely the multiple recursive, $\mathcal{R}^n$, function of Péter. The treatment will facilitate discussions of ultra-complex functions when the need arises. Surprisingly, the need to discuss them does arise frequently at least for

---

[†]Many of the resulats reported here were discovered in 1966, by D Ritchie while engaged in research under the direction of P Fischer at Harvard. They were written down but not published, and then rediscovered by this author in 1968 and circulated in technical report form. From the reaction to these reports the author was made aware of Ritchie's work.

Since there is little propsect that Ritchie's work will be available, and since these ideas are a logical sequel to Subrecursive Programming Languages I, by Constable & Borodin, a number of people felt that this version should be published with appropriate historical commentary.

Part of this work was supported by NSF contract GJ-579.

doubly recursive functions when teaching results about primitive recursive functions and the various languages for computing them. Students want examples of non-primitive recursive functions and computations. Theoreticians also want simple examples of functions which have high structural as well as computational complexity. $\mathcal{R}^2$ functions and the languages given here serve these purposes.

CONTENTS Page

§ Introduction

In the last ten 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 [16] [3], [15]. They have discovered new and significant results about them, [3], [14], [15], 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], [15].

This paper will present new computer-oriented definitions of certain well-known classes of recursive functions, namely the multiple recursive functions, $\mathcal{R}^n$, of Péter. Hopefully this will make these classes more accessible to various theoreticians in computer science. 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 need simple examples of functions which have very high structural as well as computational complexity because certain phenomena show up only at such levels. $\mathcal{R}^2$ functions and the language given here serve these purposes.

§2 Preliminaries

A model for program language semantics

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 (Random Access Stored Program machine), which in the way used here, is a very natural model whose characteristics will be familiar even to the computer scientist 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 $k(a)$ or $k(\ ): A \to B$. 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_0.

The operation of a RASP is determined by defining the effect of an instruction on the memory configuration, $k(\ ) \varepsilon K$, 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 instructions which generate its change of memory and change of control mappings.

Definition 2.1:

<u>mnemonic form</u>	<u>machine form</u>	<u>meaning</u>
TRN(v,n)	$2^0 \cdot 3^v \cdot 5^n$	<u>tr</u>ansfer the <u>n</u>umber n to the location having address v.
TRV(v,w)	$2^1 \cdot 3^v \cdot 5^w$	<u>tr</u>ansfer the <u>v</u>alue in location w to the location having address v.
AN(v,w)	$2^2 \cdot 3^v \cdot 5^w$	<u>a</u>dd 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 0 in v; in either case leave the value of w unchanged.
TRI(v,w)	$2^4 \cdot 3^v \cdot 5^w$	<u>tr</u>ansfer <u>i</u>ndirectly 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 even locations. The odd 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)	$2^s \cdot 3^w \cdot 5^b$	if the value of w is non-zero (positive) then the next instruction is taken from address b otherwise from address a+2.

These instructions will generate the transition functions H_1, H_2 of the RASP. Thus by the terminology of [8] the abstract machine is $<\mathbb{N}, \mathbb{N}, 0, K_0, H_1, H_2>$ where K_0 is the set of functions of finite support in $\mathbb{N}^{\mathbb{N}}$.

The language based on these instructions 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 [8] or [24] is available to the reader if difficulties arise.

A higher level universal language

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 higher level language (e.g. an assembly language) as an intermediate step.

The language, called G, is characteristic of the core of many numerical programming languages. It is based on assignment statements and conditionals which test predicates and reference a linearly ordered set of statement labels.

Definition 2.2: Syntax for G-type languages
Variables over integers are denoted by subscripted capital Latin letters, A, B, ..., X, Y, X, A_1, B_1,... Operations on integers are denoted by f_i so that <u>assignments</u> have the form $w \leftarrow f(v_1,\ldots,v_n)$ where w, v_i denote variables. The usual operators will be +1, $\dot{-}1$, +, $\cdot$.

The relations are denoted p_i and predicates are denoted $p(v_1,\ldots,v_n)$.
ℓ The positive integers are used as labels and are denoted ℓ_i. A conditional has the form "if $p(v_1,\ldots,v_n)$ then ℓ_i." The usual relations
$\neq$ are $=0$, $\neq 0$, $=$, $\neq$.

The language also uses go to statements, go to $+\ell_i$, go to $-\ell_i$. The semantics for a specific G-type language are given below.

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

Definition 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 a constant >0 ; a_i is the address of s_i; i.e. $\mathrm{addr}(s_i)=a_i$. Also n must be positive.
s_i	go to -n	$C(p_1,a_i-2n)$ and n must be positive.
s_i	if $v\neq 0$ then b	C(addr(v),b') where b' is the address of

The Loop Language

A very elegant language for expressing primitive recursive functions has been defined independently by M Minsky [16] & D Ritchie [15]. Following Ritchie, the language is called the "Loop Language". The syntax for a version of Loop is presented below using a context-free grammar.

Definition 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>;<program>
```

Definition 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 variable.

Loop statement or program	Interpretation in G
$v \leftarrow 0$	$v \leftarrow 0$
$v \leftarrow w + 1$	$v \leftarrow w + 1$
if $\pi,\alpha,\beta,$ are programs	is interpreted as the program
α DO v π END β	$\overline{\alpha}$ $\overline{v} \leftarrow v$ 1 if $\overline{v}=0$ then 2 $\overline{\pi}$ $\overline{v} \leftarrow \overline{v} \dot{-} 1$ go to 1 2 $\overline{\beta}$
	where 2 labels the first statement of $\overline{\beta}$ and where $\overline{v}$ does not occur in $\overline{\pi},\overline{\alpha}$ or $\overline{\beta}$. the translations of π,α,β.

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

The Loop language as originally defined using the more restricted syntax of "$v \leftarrow v + 1$", but the languages have the same computational power.

If fact, one can easily define the Loop programs which will carry out the following G instructions, for v, w_i any variables, $n \in \mathbb{N}$. $v \leftarrow n$; $v \leftarrow w$; $v \leftarrow w-1$; $v \leftarrow w_1 + w_2$; $v \leftarrow w_1 \cdot w_2$; $v \leftarrow w_1 \div w_2$.[1] For example $x \leftarrow n$ is accomplished by the n + 1 statements; $X \leftarrow 0$; $X \leftarrow X+1$; ...; $X \leftarrow X+1$. The instruction $X \leftarrow X \dot{-} 1$ is defined by the program. S←0; DO X; X←S; S←S+1; END.

It is shown in Meyer & Ritchie [15] and in [16] 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. $\mathcal{R}^1 \subset \mathcal{R}^2 \subset \ldots \subset \mathcal{R}^n$ ..., called the Péter hierarchy [18]. The class $\mathcal{R}^n$ is called the class of (nested) n-fold recursive functions. 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 stack 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:

Definition 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 releasing 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

---

[1] assign to v the integer part of w_1/w_2.

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

(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 $P_2(\ )$, $s^2_2(\ )$ are specified. The following suffice:

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

$s^2_1(z)$ = largest u which divides z+1

$s^2_2(z) = ([z+1/2^{s^2_1(z)}]-1)/2$. 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

```
LOOPSTACK  X
π
END
```

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   ⎫
 :    ⎬ X times
END   ⎪
END   ⎭
```

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

Definition 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

* DO W   ⎫                                 (perform the inner loop (* refers to
  π      ⎬                                  a key location described later).)
  END    ⎭

2 if STACK EMPTY then exit                 (see remarks following program)

  Z ← STACK_1    ⎫                         [ decrease first
  Z ← Z ∸ 1      ⎬                           non-zero stack
  STACK_1 ← Z    ⎭                           element ]
```

code	comments
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 0)
$P \leftarrow STACK_1$	("pop stack" to remove 0 item)
go to 2	
3 DO N	
$STACK_1 \leftarrow X$	
END	
$N \leftarrow 1$	
go to 1	

Remarks: A simple stack is "cleared" by setting the base variable 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. A similar technique works of multiple stacks.

Succintly written the language Loopstack or LS for short, consists of [$v \leftarrow w$, $v \leftarrow w+1$, $v \leftarrow n$, DO, END, LOOPSTACK, END]. Each use of the pair DO, END results in the allocation of a new loop control register (the $\overline{v}$ in the Definition 2.5). Similarly, each use of the LOOPSTACK, END pair results in the allocation of a new stack.

Definition 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 G, Loop or LS can be regarded as function-computing programs if inputs and outputs are specified. Details of doing this vary (see [5] and [8]). We adopt the convention of Constable & Borodin [5] which follow those new standards. This involves selecting a set, IN, of input variables and a set (here a unit set), OUT, of output variables. These sets constitute the I/O (input/output) of the program.

Say that the function program π __with I/O computes__ $f:\mathbb{N}^n \to \mathbb{N}$ 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 __LS computable__ iff there exists a $\pi\varepsilon LS$ such that π with I/O computes f.

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.[2]

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 functions (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+1st__ iterate of __f__.

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 __diagonalization__. That is, $f_\omega(x)=f_x(x)$, and $f_\omega(x)$ is simply

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

$$\text{X}\leftarrow\text{X+1}$$

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

Proving this statement requires a careful analysis of the semantic definitions of LOOPSTACK X. The general theorem is given by defining $DO_X X$; π;END to be the program

---

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

```
DO X  )
 :    }   X times
DO X  )
π
END   )
 :    }   X times
END   )
```

for π any program.[3]

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 theorem, an example of DO_X X; π;END translated into the base language is provided below.

Example: for X=3

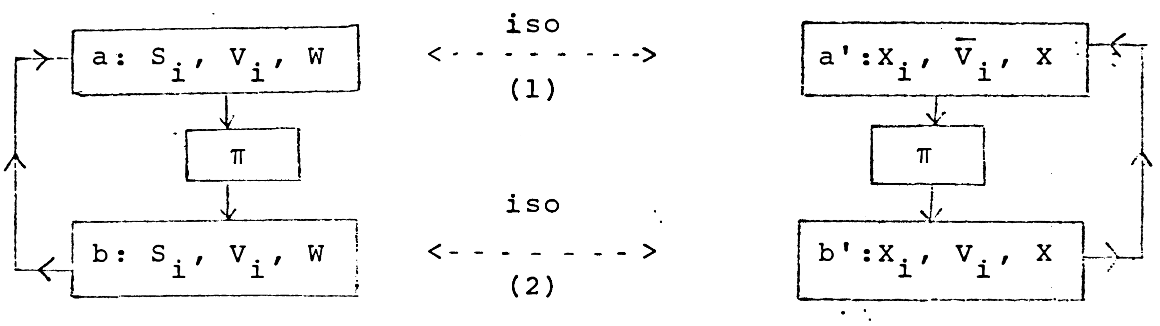
```
           X_1 ← X
      1    if X_1 = 0 then end_1
           X_2 ← X
      2    if X_2 = 0 then end_2
           X_3 ← X
      3    if X_3 = 0 then end_3
           π
           X_3 ← X_3 ∸ 1
           go to 3
   end_3   X_2 ← X_2 ∸ 1
           go to 2
   end_2   X_1 ← X_1 ∸ 1
           go to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

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

The translation of LOOPSTACK X on page 8 should be handy for comparison with the preceeding example.

The two programs can be proved equivalent by defining a type of <u>isomorphism</u> 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 registers 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 labelled 1 (DO W) and the one labelled 3 (DO N) (see * in code on page 8). Call these locations a and b respectively. Other important locations are $N \leftarrow N+1$ 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 <u>configuration</u> $\langle V_1,\ldots,V_n,S_1,\ldots,S_x\ W\rangle$ 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 outer most end_i instruction before control transfers to an $X_i \leftarrow X$ 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). 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,i.e. 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, and the proof is omitted here. q.e.d.

Prop 3.2: f_ω=[LOOPSTACK X; X←X+1; END]

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; $f_{\omega+n-1}$; END. Clearly the function $f_{\omega+n}(\ )$ is defined by

$$f_{\omega+n}(x)=f_{\omega+n-1}^{(x)}(x) \text{ for } n=1,2,\ldots$$

The LOOPSTACK instructions can be nested to define yet faster growing functions.

§4 Using Ordinal Notations to Describe Program Structure

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

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

Then $h_\omega(x)=h_x(x)=x+x=2x=f_1(x)$. The canonical program for $h_\omega(\ )$ is 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 the block b_i has been assigned notation γ_i, then 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 to 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 Cantor normal form (CNF) iff it appears as

$$\omega^{n_1}\cdot a_1+\omega^{n_2}\cdot a_2+\ldots+\omega^{n_p}\cdot a_p+a_{p+1} \quad \text{where} \quad n_i>n_{i+1} \quad \text{and}$$

$n_i, a_i \in \mathbb{N}$. As is well known, every ordinal has a unique CNF (see [25] 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 $c_1 \cdot \omega \cdot c_2 \cdot \omega \ldots c_n \cdot$ 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_i; END is $(\gamma_i) \cdot \omega^{\omega}$ where γ_i is the notation assigned to π_i. With this rule, ordinals can be assigned to all $\pi \varepsilon LS$.

The ordinals denoted by $\alpha(\pi)$ for $\pi \varepsilon LS$ 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 $a_i \varepsilon \mathbb{N}$ 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.

Definition: The <u>stack complexity</u> of an LS π is n iff the deepest nesting of LOOPSTACK instructions is n. The <u>pure (0) loop</u> complexity of π is n iff the deepest nesting of DO statements around a program α not containing a LOOPSTACK statement is n. The <u>m-loop</u> 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 $\pi \varepsilon LS$ 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: The ordinal $\alpha(\pi)$ is a measure of the structural complexity of π. In the next section we examine its relationship to computational complexity.

§5 Characterization of Loopstack

The Loopstack language can express only terminating programs.

Theorem 5.1: If $\pi \varepsilon LS$, then $\pi(x)\downarrow$ for all x, i.e. $\pi(x)$ halts.

Proof: Clear from Theorem 3.1.

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

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

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

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

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</u> <u>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) $f_\alpha(x)<f_\beta(x)$ e.f.s. (therefore $\exists p$ $f_\alpha(x)<f^{(p)}_\beta(x)$ for all x)[4]

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

Proof: elementary

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

Definition 5.2: Let $t_1 f_\alpha(x)$ for α in CNF denote the number of times the "$X \leftarrow X+1$" statement is executed in f_α.

Prop 5.3: For all $f_\alpha, 0\leq\alpha<\omega^2$ and α in CNF,

$$t_1 f_\alpha(x)+x = f_\alpha(x).$$

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

Prop 5.4: For α as above, $\exists_{c_\alpha} \varepsilon \mathbb{N}$ such that $tf_\alpha(x) \leq c_\alpha \cdot t_1 f_\alpha(x)$.

[4]The abbreviation e.f.s. means except on a finite set.

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

Definition 5.3: The instruction pair LOOPSTACK, END can be regarded as an operator, $0_{\omega^{\omega}}[f(\ )]$, on one argument functions f( ). Using a program f with input X and output X, $0_{\omega^{\omega}}[f(\ )](x)$ is output of LOOPSTACK X; f; END for X having initial value x. The iteration operator is denoted 0_{ω} and defined as $0_{\omega}[f(\ )](x)$ 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 monotone increasing then

(i) $0_{\omega}[f](x) < 0_{\omega}[g](x)$ for all x

(ii) $0_{\omega^{\omega}}[f](x) < 0_{\omega^{\omega}}[g](x)$ for all x

Proof: elementary

With these lemmata the theorem is eaily proved by induction. For similar type proofs in detail see [4], [5] or [22].

Characterization of the language

The Loopstack language can easily be characterized using the class $\mathcal{R}^2$ of doubly recursive functions. From Robbin [22] two critical facts are known:

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

Theorem 5.6: $\phi_i \in \mathcal{R}^2$ iff $t\phi_i(x) \leq f_\alpha(x)$ e.f.s. for $\alpha<\omega^2$

From these two theorems

Theorem 5.7: 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 $f_\alpha \varepsilon LS$ for all $\alpha<\omega^2$. It is also known that $\mathcal{E}(f_\alpha) \subset \mathcal{R}^1(f_\alpha)$, thus since $\mathcal{R}^1(f_\alpha) \subseteq LS$ for all $\alpha<\omega^2$, it follows by Thm 5.6 above that $\mathcal{R}^2 = \bigcup_{\alpha<\omega^2} \mathcal{E}(f_\alpha) \subseteq LS$.

(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 $t\pi(x) < f_\alpha(x)$ for all x, and some $\alpha<\omega^2$. Thus by Thm 5.6 above, $\pi(\ ) \varepsilon \mathcal{R}^2$. q.e.d.

Efficiency of the language

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 efficiently. But, there is a limit to the possible added efficiency as the following theorem shows.

Theorem 5.8: Let $g(x)=x^x$, then for all $f(\ ) \varepsilon \mathcal{R}^2$ and for all $\phi_i(\ )=f(\ )$ such that $t\phi_i(\ ) < k(\ ) \varepsilon \mathcal{R}^2$, there is an LS program $\alpha_j(\ )=f(\ )$ such that

$$t\alpha_j(x) \leq c \cdot g(t\phi_i(x)).$$

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 $h \varepsilon \mathcal{R}^2$ there is $f \varepsilon \mathcal{R}^2$ such that $\phi_i(\ )=f(\ )$ and for <u>all</u> $\alpha_j(\ )=f(\ )$,

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

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 "feedback"). The details of such a procedure are given in Constable & Borodin [5].

The basic idea is represented by the following diagram. Let $\overline{\phi}_i$ be the program simulating ϕ_i and suppose for illustration that $t\phi_i(x) < f_{\omega+2}(x)$ for all x. LOOPSTACK X; DO X; DO X; $\overline{\phi}_i$; 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 $\overline{\phi}_i$, stops; then S is increased to 1 signalling the clock to stop (i.e. stopping the feedback 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 $t\phi_i(x)$). Once S=Ø, LOOPSTACK runs until all stack elements are 0. Since there is no feedback, the process requires $c \cdot Z^Z$ steps because

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

K←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 5.8 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.

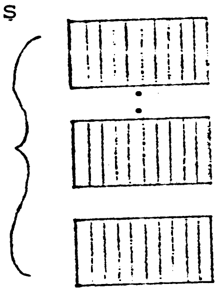
Then LOOPSTACK X is equivalent to: DO X; STACK_1←X; END; DO STACK_1←X; π; END.

§6 Conclusion

<u>Generalizations to n-fold recursion</u>

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

X



← (each turned on its side)

The command to

```
LOOPSTACK² X
π
END
```

would cause a two dimensional stack to be loaded as

$$\left.\begin{array}{c} \overbrace{\begin{array}{c} X,X,\ldots,X \\ X,X,\ldots,X \\ \vdots \qquad \vdots \\ X,X,\ldots,X \end{array}}^{X\text{-times}} \end{array}\right\} X \text{ times}$$

Then the top level stack would execute $\mathrm{LOOPSTACK}^2 X$. 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. Then the top stack would execute LOOPSTACK X again. The effect of continuing this is to perform

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

$$\pi$$

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

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

Clearly the whole process can be carried over to $\mathrm{LOOPSTACK}^n X$ where

$$\begin{array}{l}\mathrm{LOOPSTACK}^n X \text{ computes}\\ X \leftarrow X+1\\ \text{END}\end{array}$$

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

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

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

---

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

where $\omega^{\omega^{n_i}}$ describes the maximum depth of nesting of LOOPSTACK^{n_i} 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 enough 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.

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 [27]. In 1925 (published 1928) Ackermann [27] 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[11]) and number theory (Péter [18]) but there was no natural reason known for studying them.

In 1947, R M Robinson [23] 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 [21] in relating function classes ($\mathcal{E}$) 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 syntactial definition of $\mathcal{R}^1$ in terms of programming languages. Finally, McCreight & Meyer [14] 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 literature. They appeared first in Péter's work, 1936. R M Robinson examined double recursion in 1948. A deep analysis of their structure (in relation to ordinal recursion) appeared by Tait [26] in 1961. This work inspired Robbin [22] 1965 to analyze their structure systematically in terms of Grzegorczyk's work on $\mathcal{R}^1$ and in terms of the Hartmanis & Stearns [10] approach to computational complexity. Building on the spade work of Robbin, D Ritchie first extended the Loop type approach to $\mathcal{R}^n$. Likewise using Robbin's results Constable, using program modification on a RASP, extended the Cleave approach to $\mathcal{R}^n$ and beyond to define a hierarchy up to ε_0. 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.

REFERENCES

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

[2] L. J. Bass and P. R. Young, Hierarchies Based on Computational Complexity and Irregularities of Class Determined Measured Sets, ACM Symposium on the Theory of Computing, Northhampton, Mass., (1970), 37-39.

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

[4] Robert L. Constable, Extending and Refining Hierarchies of Computable Functions, Computer Science Technical Report #25. University of Wisconsin, Madison, Wisconsin, (1968).

[5] Robert L. Constable and Allan Borodin, On the Efficiency of Programs in Subrecursive Formalisms, IEEE Conference Record of the 10th Annual Symposium of Switching and Automata Theory, Los Angeles, Calif., (1970), 60-67. (to appear in JACM as Subrecursive Programming Languages, I).

[6] Robert L. Constable, Subrecursive Programming Language II, On Program Size, (to appear in Special Issue JCSS, 1971).

[7] M. Davis, The Undecidable, Raven Press, Hewlett, N.Y., (1965).

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

[9] A. Grzegorczyk, Some Classes of Recursive Functions, Rozprawy Matematyczne, No. 4, (1953), 1-45.

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

[11] S. C. Kleene, Introduction to Metamathematics, D. Van Nostrand, Princeton, N.J., (1952).

[12] S. C. Kleene, Extension of an Effectively Generated Class of Functions by Enumeration, Colloq. Math G, (1958), 67-78.

[13] S. H. Lachlan, Multiple Recursion, Zeitschr. f Math Logik und Grund. d. Math 8, (1962), 81-107.

[14] E. M. McCreight and A. R. Meyer, Classes of computable functions defined by bounds on computations, ACM Symposium on Theory of Computing, Marina del Rey, Calif., (1969), 79-88.

[15] A. R. Meyer and D. M. Ritchie, The complexity of Loop programs, Proceeding 22 National ACM Conference, (1967), 465-470.

[16] M. Minksy, Computation, Finite & Infinite Machines, Prentice-Hall, Englewood Cliffs, N.J., (1967).

[17] C. Parsons, Hierarchies of Primitive Recursive Functions, Zeitschr Math. Logik Grund. Math, 14, (1968), 357-776.

[18] Roza Péter, Recursive Functions, 3rd ed. Academic Press, N.Y., (1967).

[19] D. M. Ritchie, Program Structure and Computational Complexity, Draft of Ph.D. Thesis, Harvard University, (1966).

[20] R. W. Ritchie, Classes of Recursive Functions Based on Ackermann's Function, Pacific Journal of Math, 15, (1965), 1027-44.

[21] R. W. Ritchie, Classes of Predictably Computable Functions, Trans. AMS, 106, (1963), 139-173.

[22] Joel Robbin, Subrecursive Hierarchies, Ph.D. Dissertation, Princeton,University, (1965).

[23] R. M. Robinson, Primitive Recursive Functions, Bulletin AMS, 53, (1947), 915-942.

[24] J. C. Shepherdson and H. E. Sturgis, Computability of Recursive Functions, JACM, 10, (1963), 217-255.

[25] Waclaw, Sierpinski, Cardinal and Ordinal Numbers, Warsaw, (1965).

[26] W. W. Tait, Nested recursion, Math. Ann. 143 (1961), 236-250.

[27] J. van Heijenoort, From Frege to Gödel, Harvard Univ. Press, Cambridge, Mass., (1967).