DRAFT

Program Structure and Computational Complexity

Dennis M. Ritchie

SYNOPSIS

The major purpose of this thesis is to show that when the language in which computations are described is restricted suitably, there can be an effective relationship between the complexity of a program and that of the computation it describes. We give two examples.

The first example is that of Loop programs. A Loop program is a finite sequence of instructions for manipulating non-negative, unbounded integers stored in registers; the instructions allow incrementing registers by unity, setting registers to zero, and moving the contents of registers. The only control instructions consist of Loops; there is a kind of Loop for each number $n \geq 1$. A Loop with $n = 1$ causes the execution of a portion of the program to be repeated a predetermined number of times equal to the current contents of a register. Loops may be nested, one inside another, to any fixed depth; but Loops with $n > 1$ are defined so as to make a Loop of type $n+1$ equivalent to a variable depth of nesting of Loops of type n.

Each Loop program is assigned an ordinal α, where $0 \leq \alpha < \omega^{\omega}$, which is intended to be the measure of complexity of the program. The ordinal assigned to a program depends effectively on the program, and measures the depth of nesting of the various kinds of Loops.

The idea of Loop programs whose only Loops have $n = 1$, although original with the author, is not unique to him; for example Minsky [17, pp. 212-215] discusses briefly the same idea. Some results of

the theory of such Loop programs have been announced by the author [22] and published by Meyer and the author [15,16]. The generalization with Loop instructions for each $n > 1$ is believed to be entirely new.

For each ordinal α, $0 \leq \alpha < \omega^{\omega}$, we define a function f_{α}. The function is recursive, strictly increasing, and if $\alpha > \beta$, f_{α} majorizes f_{β}. The definition of f_{α} for finite ordinals α is the same as the f_n of [15,16] and in general is a modification of the function W_{α} used by Robbin [25] for much the same purposes. The major results on Loop programs can be stated as follows: for each Loop program $\underset{\sim}{P}$ assigned ordinal α there is a number ~~of~~ p effectively derived from $\underset{\sim}{P}$ such that $\underset{\sim}{P}$ with inputs $x_1,\ldots,x_n$ requires no more than $f_{\alpha}^{(p)}(\max\{x_1,\ldots,x_n\})$ steps to halt (Theorem (3.6)). The notation $f_{\alpha}^{(p)}$ means f_{α} composed with itself p times. There are some programs $\underset{\sim}{P}$ assigned ordinal α which do in fact require $f_{\alpha}^{(p)}(x)$ steps to halt when given input x (Theorem (4.7)). A precise definition of the number of steps used by a program is a by-product of a formalization of Loop programs presented in §2.

Further results on Loop programs, and much of the rest of the thesis, use heavily the notion of computation-time closure. A set of functions is computation-time closed when both of the following are true: if a function f is in the set, a function b is in the set where b bounds the time required to compute f on a Turing machine; if b is in the set and b bounds the time required to compute f on a Turing machine, f is in the set.

If $\mathfrak{L}_\alpha$ is the class of functions computable by programs assigned an ordinal less than or equal to α, each $\mathfrak{L}_\alpha$ for $\alpha \geq 2$ is computation-time closed. This allows us to show the following: each class $\mathfrak{L}_\alpha$ for $\alpha \geq 2$ is closed under limited recursion (Theorem (6.8)); each class $\mathfrak{L}$ for $\alpha \geq 2$ can be characterized in arithmetic terms, without reference to Turing machines or Loop programs (Theorem (6.3)); if a program $\underset{\sim}{P}$ assigned ordinal α requires only $f_\alpha^{(p)}$ steps as a function of its inputs where $\beta < \alpha$, then $\underset{\sim}{P}$ can be rewritten effectively to yield a program $\underset{\sim}{P}'$ which is equivalent to $\underset{\sim}{P}$ but is assigned ordinal β. However, it is in general undecidable whether these hypotheses hold for $\underset{\sim}{P}$ (Theorem (12.6)).

The second example of a restricted program language is that describing the multiple recursive functions [19,21]. Each multiple recursive function can be defined by a formal system of equations which can effectively be assigned an ordinal $\alpha < \omega^\omega$. If $\mathfrak{R}_\alpha$ is the class of functions defined by systems of equations assigned ordinal α, then $\bigcup_{\alpha<\omega^n} R_\alpha$ is the class of n-recursive functions; Péter shows [21] that the 1-recursive functions are the same as the primitive recursive functions. Much the same theorems are proved for $\mathfrak{R}_\alpha$ as for $\mathfrak{L}_\alpha$. In particular, $\mathfrak{R}_\alpha$ is computation-time closed for $\alpha \geq 2$ (Theorem (9.3)); if $f \in \mathfrak{R}_\alpha$, $f(x_1,\ldots,x_n)$ can be computed by a Turing machine in $f_{1+\alpha}^{(p)}(\max\{x_1,\ldots,x_n\})$ steps for some p which is effectively found from the recursion equations defining f (Theorem (9.1)); $f_{1+\alpha} \in \mathfrak{R}_\alpha$ (Theorem (8.3)). These facts alone show: for $\alpha \geq 2$, $\mathfrak{L}_{1+\alpha} = \mathfrak{R}_\alpha$ (Theorem (10.1)).

The same kind of techniques are applied to the hierarchies of Axt [2], Grzegorczyk [9] and Robbin [25]. All of these hierarchies are shown to be identical to a portion of the $\mathfrak{L}_\alpha$ and $\mathfrak{R}_\alpha$ hierarchies, and thus to each other. Specifically, if $\mathfrak{P}_\alpha$, $\alpha < \omega$, are the Axt classes, $\mathfrak{L}_\alpha = \mathfrak{P}_\alpha$ for $\alpha \geq 4$ (Theorem (10.4)); if $\mathfrak{E}^G_\alpha$, $\alpha < \omega$, are the Grzegorczyk classes, $\mathfrak{L}_\alpha = \mathfrak{E}^G_{\alpha+1}$ for $2 \leq \alpha < \omega$ (Theorem (10.9)); if $\mathfrak{E}_\alpha$, $\alpha < \omega^\omega$, are a trivial modification of the Robbin classes, $\mathfrak{L}_\alpha = \mathfrak{E}_\alpha$ for $\alpha \geq 2$ (Theorem (10.6)). All of these results are straightforward using computation-time closure. Not all are new, however. According to a personal communication, Axt showed $\mathfrak{P}_\alpha = \mathfrak{E}^G_{\alpha+1}$ for $\alpha \geq \alpha_o$, $\alpha_o \approx$ but used a different method. Meyer showed the same thing independently [14], using a method like ours. Robbin [25] showed that $\bigcup_{\alpha<\omega^n} \mathfrak{E}_\alpha$ is the same as the class of n-recursive functions; however, he did not subdivide the latter class after the manner of our $\mathfrak{R}_\alpha$. It should be mentioned as well that Robbin established the identity of the n-recursive functions and those functions defined by ordinal recursion over certain "standard" well-orderings of type ω^{ω^n}, and also the classes of functions occurring in a restricted version of the Kleene subrecursive hierarchy [13]. It seems likely that by closer study equality of these classes could be established at each ordinal.

Chapters II, III, and IV studied Loop programs and multiple recursive functions; Chapter V contains three applications of the tools developed in the earlier chapters. The most important of these, as we have indicated, is the idea of computation-time closure. An early

appearance of this idea, without an explicit name, was in R. W. Ritchie [23], who used it to characterizing classes which form a hierarchy of elementary functions. Cobham [6] pointed out how each Grzegorczyk [9] class could be characterized in terms of the property, after the manner of our Theorem (6.2), which states $\mathcal{L}_\alpha$ is precisely the class of functions computable by a Turing machine in a time bounded by $f_\alpha^{(p)}$ for some p. As we mentioned, Meyer [14] and also Robbin [25] used the idea as well.

Chapter V, §13, discusses unnested and bounded n-recursion [20, 21] and their relation to the $\mathcal{L}_\alpha$ classes, thus strengthening some theorems of Péter [20,21]. §14 examines the properties of computation-time closed classes of functions in general; its major results are that each $\mathcal{L}_\alpha$ includes a sequence of classes, all computation-time closed and closed under limited recursion and substitution, which is densely ordered under set inclusion (Theorem (14.14)); also, $\mathcal{L}_\alpha$ includes an infinite sequence of classes with the same closure properties but pairwise incomparable under set inclusion (Theorem (14.15)). These two results were obtained in collaboration with Albert R. Meyer. §15 applies Lemma (14.13) to obtain a strengthened version of the Super Speed-up theorem of Blum [4]. Among the consequences of our Theorem (15.3) is that there are functions lying very low in the $\mathcal{L}_\alpha$ hierarchy whose computation can be sped up, in Blum's sense, very considerably.

I. INTRODUCTION

§0. Predicting how long a digital computer with a given program will require to process its inputs is sometimes impossibly difficult. This difficulty can be partially explained as a manifestation of the theorem that there is no effective method for bounding the computation time of a Turing machine by inspection of the machine, or for bounding the running time of a program written in any language capable of describing all recursive functions.

In other words, any formalism which can describe all terminating computations must describe some nonterminating computations, and there is no generally effective way of distinguishing the description of a terminating from a nonterminating computation. In consequence, there can be no satisfactory way of relating the complexity of a program in a sufficiently powerful language to the complexity of the operations it carries out. This fact is borne out most strongly by the existence of a universal Turing machine: a fixed program, actually quite small, whose behavior is as difficult to predict as that of any possible program.

Although bounding the length of a computation by inspection of a program for the computation is in general impossible, this problem, in common with many other unsolvable problems, has interesting special cases which can be treated. One approach which has yielded fruitful results is fairly common. It essentially involves a refusal to consider computations which take too long. Among the best known examples

of this method are the linear bounded automata of Myhill [18], the T(n) countable sequences of Hartmanis and Stearns [10], and the predictably computable functions of R. W. Ritchie [23]. Each of these theories considers computations by a Turing machine where there is a bound on the time (or the storage space) allowed for computation. The bound is imposed from outside simply by restricting attention to those computations which satisfy the bound.

By contrast, the approach of this thesis is to restrict the language in which programs for computations are expressed so that infinite computations are no longer possible. The first result of this restriction is that there are indeed effectively calculable bounds on the describable computations, but the important fact is that the existence of these bounds becomes a theorem not a postulate about the computations. It also becomes possible to do for these special kinds of programs what is impossible for programs in general, namely to relate the complexity of a program to the complexity of the calculation it describes; both kinds of complexity, of course, have to be taken in the proper sense.

The major part of this thesis is the study of two examples of the technique of restricting the language in which computations are described; the remainder consists of several ~~appreciations~~ applications of the tools developed in the first part. Before going into the specifics of the two examples, we should discuss the possible forms of an answer to the question: how does the complexity of a program relate to the complexity of the computation described by it?

It is not enough to say merely that there is an effective means of going from a program and its input to a number bounding the time required to run the program with that input. For if we know the program eventually does halt, the effective method is simply the following: run the program on the given input and measure the time required. This method is not only foolish in a practical sense but (far worse, from our point of view) uninteresting mathematically. A better way is to give the answer in terms of a known function. Thus if we had a program with a single input parameter, we might be satisfied to know that for input x, the program would halt within x^2 seconds. This is the kind of result given for the program considered in this thesis.

On the other hand, even this kind of answer has many practical defects. The trouble is that many simple programs can run for a long time. Consider the following pseudo-FORTRAN program.

```
      READ N
      J = 1
      DO  1 I = 1, N+1
1     J = 2**J
      PRINT J
```

The third and fourth lines mean that 2^J is to replace J, N+1 times. We assume that the storage registers associated with the variables of the program are of unlimited size. This program is an extremely simple one. Yet when N = 4, several pages of paper are required to write down the resulting J; when N = 5, the known universe is totally

insufficient to contain the volume of paper required to write down J. Thus the function of N which predicts the running time of the program must be very large. In fact, it is proportional to

$$2^{\cdot^{\cdot^{\cdot^{2^{2}}}}} \quad \text{height } N$$

This example indicates that we must accept one of two things: either that we agree to treat programs whose running times are so incredibly long as to preclude any practical application of the results developed, or that we must throw out means of expression, like those in the program above, which programmers could hardly do without. In either case the fact must be faced that there can be no direct practical applications of the theory. In the latter there is another difficulty. When programs are restricted severely enough to make every program halt in a rather short time, the exact means of expression allowed to begin to have a major effect on the time required: it matters a great deal, for example, whether multiplication is allowed as an elementary operation or must be done in steps by means of repeated addition. In the case of real computers, of course, this is an important consideration. But we have already given up real applications by treating only programs which no programmer would write, so it would be improper to claim practical significance for our work merely because of this feature.

On the other hand, the mathematical significance of the theory can only be enhanced when it is not model-dependent; that is when the details of the basic definitions have little effect on the theorems. Thus, in the programs described below and studied in the sequel it would make little difference if addition or multiplication were added as elementary operations. We study two major examples of ways of defining computations in such a manner that from a program one can go effectively to a function which bounds the length of the computation. The two examples are Loop programs and definition of functions by multiple recursion equations; both involve computations far beyond the capabilities of real computers, but in return give rise to interesting mathematical structures.

Loop programs exemplify the approach to the theory of computability introduced by Turing [28] in that a Loop program may be regarded as a set of instructions to be executed by a sort of digital computer. The Turing approach is typified by the use of simplified models of real computers; it is probably the one most frequently found.

A distinct although equivalent version of the theory of computability is the one based on systems of Herbrand-Gödel-Kleene recursion equations, as presented by Kleene in [11] and [12, §54]. Our second example, that of definition of functions by multiple recursion, bears exactly the same relationship to definition by unrestricted recursion equations as do Loop programs to programs in general: in each case the forms of expression are weakened in such a way that infinite computations become impossible.

A Loop program is a sequence of instructions for manipulating non-negative integers stored in registers; each register is capable of storing an arbitrarily large number, and the number of registers to which a program refers is fixed but unlimited. There are instructions for moving the contents of registers, for incrementing by unity, and for setting registers to zero. The flow of control in a Loop program normally passes from one instruction to the next in sequence, and the only way of affecting the normal flow is through the use of Loops. A Loop is introduced by a LOOP instruction and terminated by an END instruction. Together these indicate that the section of the program between the two instructions is to be executed repeatedly some number of times. There is a variety of LOOP instructions, one for each number $n \geq 1$; these are written LOOP(1), LOOP(2), etc.

Each kind of LOOP instruction names a register whose contents control the looping. In the case of the instruction "LOOP(1) X", for example, X may be any register name. This instruction causes the portion of the program between itself and its matching END to be repeated a number of times equal to the contents of X at the time the LOOP is encountered; subsequent changes to X do not affect the number of times the repetition occurs. Thus a Loop introduced by LOOP(1) is entirely comparable to the DO loop of FORTRAN and to the most usual cases of the for of Algol and the THROUGH of MAD. The similarity is not accidental, for part of the motivation for the study of Loop programs is to study the power of this construction.

Loops may contain other Loops; that is, Loops may be nested to any fixed depth. This is the motivation for the existence of LOOP(n) instructions for $n > 1$: the effect of LOOP(n + 1) is defined so as to make such a Loop equivalent to a variable depth of nesting of LOOP(n) Loops. In particular, the program

```
LOOP(n + 1) X
  Q
END
```

where $n \geq 1$, X is a register name, and $\underset{\sim}{Q}$ is a program, equivalent to the program

```
LOOP(n) X   }
  :         } x
LOOP(n) X   }
  Q
END         }
  :         } x
END         }
```

where x is the number in X initially; that is, we have a nest of LOOP(n) Loops of depth x. There are no constructions in real programming languages comparable to LOOP(n) where $n > 1$.

To each Loop program an ordinal α is assigned, where $\alpha < \omega^{\omega}$. The ordinal is derived directly from the depth of nesting of the various kinds of Loops: for a program without Loops, $\alpha = 0$; if a program is the concatenation of two programs with ordinals β, γ, the ordinal assigned is $\alpha = \max(\beta, \gamma)$; if program $\underset{\sim}{Q}$ is assigned ordinal β, then program $\underset{\sim}{P}$ =

LOOP(n+1) X

$\mathcal{Q}$

END

for $n \geq 0$ and X a register name, is assigned $\alpha = \beta + \omega^n$. Then, for example, in a program which uses only LOOP(1) instructions is assigned a finite ordinal equal to the greatest depth of nesting of Loops in the program. The ordinal assigned to a program is the measure of complexity of the program.

The notion of computation by Loop program can be formalized; a by-product of the formalization is a precise definition of the running time of a given program as a function of its inputs. The running time measures the number of individual instruction executions required to complete a program and in a sense the justification for introducing the somewhat opaque formalism is to make reasonable the claim that the complexity of a calculation is measured accurately by its running time.

The basic result on Loop programs is the Bounding Theorem (3.6). We introduce for each ordinal α, $\alpha < \omega^\omega$, a function f_α as follows: if $\alpha = 0$,

$$f_\alpha(x) = \begin{cases} x+1 & \text{if } x \leq 1 \\ x+2 & \text{if } x > 1 \end{cases}$$

If α is a successor ordinal, $\alpha = \beta + 1$,

$$f_\alpha(x) = f_\beta^{(x)}(1)$$

where the notation $f^{(x)}(y)$ means $f(f(\ldots f(y)\ldots))$; there are x compositions of f. That is, $f_{\beta+1}$ is defined from f_β by <u>iteration</u>. If α is a limit ordinal, let β be the least ordinal so $\alpha = \beta + \omega^{n+1}$, where $n \geq 0$. Then

$$f_\alpha(x) = f_{\beta+\omega^{\overset{n}{\omega}}\cdot x}(x)$$

Thus at limit ordinals, f_α is defined by diagonalization over a certain sequence $\{f_{\beta_i}\}$ of functions where $\beta_0 < \beta_1 < \cdots$ and $\sup\{\beta_i\} = \alpha$. The first few f_α are easy to describe: $f_1(x) = \min(1, 2x)$; $f_2(x) = 2^x$;

$$f_3(x) = 2^{\left.\begin{matrix}2\\ \cdot\\ \cdot\\ \cdot\end{matrix}\right\} x}$$

The details of the definition of f_α are unimportant. For finite ordinals, $\alpha = n$, f_α is the same as the f_n used in [15] and [16]; at limit ordinals, the definition is the same as that used by Robbin [25] for his functions W_α, which play the same role as our f_α. What is important is that the f_α are easily defined and have pleasant properties: each f_α is a strictly increasing function, and if $\alpha > \beta$, f_α majorizes (bounds almost everywhere) the function f_β.

Given the function f_α, the Bounding Theorem is: if $\underset{\sim}{P}$ is a program assigned ordinal α, there is a fixed number p, effectively found from $\underset{\sim}{P}$, such that the running time of $\underset{\sim}{P}$ with inputs $x_1, \ldots, x_n$ is bounded by $f_\alpha^{(p)}(\max\{x_1, \ldots, x_n\})$.

By fixing upon one or more registers for input and a register for output, we associate with a Loop program a function computed by that program; the class of functions computable by Loop programs assigned ordinals less than or equal to α is called $\mathcal{L}_\alpha$. It is an immediate consequence of the Bounding Theorem that every function $f \in \mathcal{L}_\alpha$ has a p so $f(x_1,\ldots,x_n) \leq f_\alpha^{(p)}(\max\{x_1,\ldots,x_n\})$. Also, for each $\alpha < \omega^\omega$ there is a function $\hat{f}_\alpha \in \mathcal{L}_\alpha$ so $\hat{f}_\alpha(x) \geq f_\alpha(x)$; it is immediate that the classes $\mathcal{L}_\alpha$ form a hierarchy, for it is easily shown that if $\alpha > \beta$, $f_\alpha(x) > f_\beta^{(c)}(x)$ for each c and almost all x. Already several of the goals looked for in the study of Loop programs have been achieved, for it follows first that every program assigned ordinal α consumes no more than $f_\alpha^{(p)}(\max\{x_1,\ldots,x_n\})$ steps when given input $x_1,\ldots,x_n$, and second that there are some programs assigned ordinal α which actually do require this many steps to halt. Thus the ordinal assigned a program is a reasonable measure of the (potential) complexity of the computation described by the program.

The further study of Loop programs, and in fact much of the remained of the thesis, is heavily concerned with the property of <u>computation-time closure</u> of a set of functions defined as follows: first, when a function is in the set, it can be computed by a Turing machine in a number of steps which is bounded, as a function of the inputs, by another function in the set; and second, if a bound on the computation time of a function is in the set, the function itself is in the set. Each class $\mathcal{L}_\alpha$ for $\alpha \geq 2$ is computation-time closed. The first requirement is met by combining the Bounding Theorem with a demonstration that

a Turing machine can simulate an arbitrary Loop program while consuming a number of steps which is an $\mathcal{L}_2$ function of the running time of the Loop program; the second by finding a Loop program which simulates a Turing machine calculation carried out for a given number of steps, and then substituting the known bound on the length of the computation into the simulation program.

The computation-time closure of $\mathcal{L}_\alpha$ leads immediately to several theorems; for example, if it is known that a program assigned ordinal α actually has a running time bounded by $f_\beta^{(c)}$ where $2 \leq \beta < \alpha$, the program can be effectively rewritten so it is assigned ordinal β. It is also shown that each class $\mathcal{L}_\alpha$, $\alpha \geq 2$, is closed under the operation of limited recursion (see Grzegorczyk [9]); that each class $\mathcal{L}_\alpha$, $\alpha \geq 2$, can be characterized in purely arithmetic terms, without reference either to Turing machines or Loop programs; and that every primitive recursive function is in $\mathcal{L}_\alpha$ for some finite ordinal α.

Our second example is that of the multiple recursive functions. These are, for our purposes, precisely those functions definable by certain formal systems of equations. We imagine a language containing symbols for constants, variables, function letters, and appropriate punctuation, combined in such a way as to represent definitions of effectively computable functions. This language is simply a formal version of the informal definition of functions by means of various kinds of recursion, including, for example, primitive recursion. Unlike Kleene [11, 12] however, we place certain restrictions on the form of the systems of equations. In particular, an equation defining

a function in terms of already-defined functions must be an instance of one of several schemata, namely those of substitution and n-recursion for some fixed integer $n \geq 1$. Substitution simply means obtaining a new function by means of explicit transformation or composition of other functions. The schema of n-recursion allows defining a function $f(x_1,\ldots,x_n)$ in terms of known functions and values of f itself at arguments $z_1,\ldots,z_n$ such that the n-tuple $z_1,\ldots,z_n$ is lexicographically less than $x_1,\ldots,x_n$. The very form of the schema of n-recursion is such as to ensure that the set of equations constituting an instance of n-recursion actually does define a function effectively.

An ordinal $\alpha < \omega^\omega$ can be effectively attached to each formal system of equations satisfying certain purely syntactic requirements. Letting $\mathfrak{R}_\alpha$ be the class of functions definable by systems of equations with ordinals less than or equal to α, another hierarchy results which is equivalent to the following: $\mathfrak{R}_0$ consists of the closure under substitution of the constant and identity functions; $\mathfrak{R}_\alpha$ for every $\alpha > 0$ consists of the closure under substitution of all functions f for which there exist β and n so $\alpha = \beta + \omega^n$ and f is definable by $(n+1)$-recursion from functions in $\mathfrak{R}_\beta$.

For each $n \geq 1$ the functions in $\bigcup_{\alpha<\omega^n} \mathfrak{R}_\alpha$ are called n-recursive; functions which are n-recursive for some n constitute the multiple recursive functions. The notion of multiple recursive function is a generalization of that of primitive recursive function, which was introduced explicitly by Gödel [8]; as Péter [21] shows, the 1-recursive

functions are identical to the primitive recursive functions. Ackermann [1] first introduced a 2-recursive (also called double recursive) function and used it to show that there are effectively computable functions which are not primitive recursive. Péter [19,20,21] studied the whole class of multiple recursive functions.

Our examination of the multiple recursive functions uses much the same methods as those applied to Loop programs. A Bounding Theorem for $\mathcal{R}_\alpha$ establishes that each function in $\mathcal{R}_\alpha$ is bounded by $f_{1+\alpha}^{(p)}$ for some p which can be found effectively from the formal system of equations defining the function; on the other hand, $f_{1+\alpha} \in \mathcal{R}_\alpha$ for $\alpha \geq 1$. Likewise, each class $\mathcal{R}_\alpha$ for $\alpha \geq 2$ is computation-time closed; this is established by considering the number of steps a Turing machine would require to carry out the evaluation of a function from its defining equations. Then the theorem $\mathcal{L}_{1+\alpha} = \mathcal{R}_\alpha$ for $\alpha \geq 2$ is immediate. For if $f \in \mathcal{L}_{1+\alpha}$, $f(x_1,\ldots,x_n)$ can be computed on a Turing machine in no more than $f_{1+\alpha}^{(p)}(\max\{x_1,\ldots,x_n\})$ steps; but the latter function is in $\mathcal{R}_\alpha$, and so by the computation-time closure of $\mathcal{R}_\alpha$, $f \in \mathcal{R}_\alpha$. The converse argument is identical.

It is here that the concept of computation-time closure is most important. For to show directly that $\mathcal{L}_{1+\alpha} = \mathcal{R}_\alpha$ is quite difficult. In particular, if $\alpha \leq \omega$ then to construct an equivalent Loop program with ordinal $1+\alpha$ directly from the equations defining an $\mathcal{R}_\alpha$ function is quite hard. But given that the $\mathcal{R}_\alpha$ function can be computed by a Turing machine in $f_{1+\alpha}^{(p)}$ steps, one need only write a program which

computes any function at least as large as $f^{(p)}_{1+\alpha}$ and insert it into a program to simulate the Turing machine.

The same kind of methods are also applicable to three other hierarchies, those of Grzegorczyk [9], Axt [2], and Robbin [25]. The first two classify the primitive recursive functions and the third all the multiple recursive functions. The point of interest is that each of these hierarchies is identical to a corresponding portion of the $\mathcal{E}_\alpha$ and $\mathcal{R}_\alpha$ hierarchies; the classes of functions eventually become the same.

The idea of computation-time closure, which plays a major role in our work, was used by R. W. Ritchie [23] without an explicit name; its value in characterizing the Grzegorczyk hierarchy was pointed out by Cobham [6]. Some of the results of Robbin [25] make implicit use of the idea.

The usefulness of the notion is that the particular functions in a computation-time closed set of functions depend merely on the approximate size of the functions in the set; that is, a function is in the set if and only if a sufficiently large function is in the set. For example, suppose $\mathcal{C}$ and $\mathcal{D}$ are two computation-time closed sets of functions, and that $\mathcal{D}$ contains both a function which grows at least exponentially and a function which majorizes every function of $\mathcal{C}$ Then it can be shown not only that $\mathcal{D}$ contains $\mathcal{C}$ properly, but that $\mathcal{D}$ contains a function universal for $\mathcal{C}$: a function $U \in \mathcal{D}$ so that for each $f \in \mathcal{C}$, $f(x) = U(e,x)$ for some e.

The secondary goal of this thesis is to study the application of computation-time closure and other tools developed in the pursuit of the primary goal. The most important application, of course, is the study of the classes $\mathfrak{L}_\alpha$ and $\mathfrak{R}_\alpha$, which arise from Loop programs and multiple recursive functions. There are three others: the effects of various restrictions on the schema of n-recursion; the extent to which computation-time closure characterizes a set of functions (which leads to an impressive refinement of the $\mathfrak{L}_\alpha$ hierarchy); and the existence of functions whose computation can be sped up very greatly.

For the most part this thesis is self-contained. The only requirement is a knowledge of the elementary theory of Turing machines: what they are, and a few of the tricks that they can perform in order to carry out intuitively simple kinds of operations. Familiarity with the first few chapters of Davis [7] is more than enough background.

The mathematical notation in the thesis is generally standard. We use a bar over a letter to indicate a sequence of elements: "$\bar{x}_n$" is the same as "$x_1,\ldots,x_n$". In each case the first subscript in the sequence is 1 and the last is the same as that on the barred letter. Variables and constants, usually indicated by small letters of the alphabet, all range over N, which is the class of non-negative integers; functions, often small letters f, g, h, are always functions from N^n into N for some n; sets of such functions are usually denoted by capital script letters. Small Greek letters from the beginning of the alphabet are used for ordinal numbers. Functional composition is often denoted

by juxtaposition, especially with one-place functions: $fg(x)$ is the same as $f(g(x))$. Finally "$\subset$" means strict set theoretic containment.

II. LOOP PROGRAMS

§1. A Loop program is a finite sequence of instructions for manipulating non-negative integers stored in registers. There is no limit to the size of an integer stored in a register, nor to the number of registers to which a program may refer; but a given program refers only to a fixed set of registers. We will use upper case English letters, sometimes with subscripts, as register names, and abbreviate a sequence $X_1,\ldots,X_n$ of register names by $\bar{X}_n$. Boldface capitals (identified by a wiggly underscore) stand for Loop programs, and if $\underset{\sim}{P}$ is a program $\underline{\text{Reg}}\ (\underset{\sim}{P})$ is the set of register names used by $\underset{\sim}{P}$.

The instructions of a Loop program are of five types:

(1) X = 0
(2) X = X + 1
(3) X = Y
(4) LOOP(n) X where n is a fixed integer, $n \geq 1$
(5) END

Here "X" and "Y" may be replaced by any names for registers, and the "0" of "X = 0" is to be read "zero".

(1.1) Definition. The class L of <u>Loop programs</u> is $\cup L_\alpha$, where α ranges over ordinals $< \omega^\omega$, and where L_α is the smallest class satisfying

(i) If $\alpha = 0$, L_α is the class of finite sequences of type (1), (2), and (3) instructions,

(ii) If $\underset{\sim}{P} \in L_\beta$ and $\beta < \alpha$, then $\underset{\sim}{P} \in L_\alpha$,

(iii) If $\underset{\sim}{Q}$, $\underset{\sim}{R} \in L_\alpha$ and $\underset{\sim}{P}$ is $\underset{\sim}{Q}$ concatenated with $\underset{\sim}{R}$, then $\underset{\sim}{P} \in L_\alpha$,

(iv) If $\underset{\sim}{Q} \in L_\beta$ and $\alpha = \beta + \omega^n$ for some n, $0 \leq n < \omega$, then $\underset{\sim}{P} \in L_\alpha$, where $\underset{\sim}{P}$ is

```
LOOP(n+1) X
   Q
END
```

and X is any register name.

By (1.1.iv), type (4) and (5) instructions occur in pairs, like parentheses in a well-formed formula, so that the LOOP-END pairs in a program are unambiguously determined.

The first three types of instruction have the interpretation suggested by their appearance. "X = 0" means that the contents of register X are to be replaced by zero; "X = X + 1" means that the contents of register X are to be incremented by one; "X = Y" means that the contents of register Y are to be copied into register X, destroying the old contents of X but leaving Y unchanged. These are the only instructions which affect the registers.

Instructions of types (1), (2), and (3) are executed sequentially in the order in which they appear in the program. Type (4) and (5) instructions affect the normal order by indicating that the execution of the block of instructions between the LOOP and its matching END is to be repeated zero or more times.

The effect of a LOOP(n) instruction is defined by induction on n. Specifically suppose that $\underset{\sim}{P}$ is a Loop program, and that x is stored in register X initially. Then the program

```
LOOP(1) X
   P
END
```

means that $\underset{\sim}{P}$ is to be repeated x times in succession before the next instruction (if any) after the END is executed. Changes in the contents of X by $\underset{\sim}{P}$ do not affect the number of times $\underset{\sim}{P}$ is executed; and if x is zero initially $\underset{\sim}{P}$ is not executed at all.

(1.2) Example. The L_1 program

```
LOOP(1) X
X = X+1
END
```

doubles the contents of register X.

(1.3) Example. If the initial contents of X and Y are x and y, the L_2 program

```
LOOP(1) Y
   A = 0
   LOOP(1) X
      X = A
      A = A +1
   END
END
```

leaves $x \dot{-} y$ in X, where $x \dot{-} y$ (pronounced "x monus y") equals $x - y$ if $x \geq y$, 0 otherwise.

Suppose now that the interpretation of the effect of a LOOP(n) - END pair has been given for some $n \geq 1$, and $\underset{\sim}{P}$ is a Loop program. Say that the initial contents of register X are $x \geq 1$. Then we interpret the program

```
LOOP(n + 1) X
  P
END
```

as being identical to

```
LOOP(n) X      }
LOOP(n) X      } x
   :           }
LOOP(n) X      }
  P
END            }
  :            } x
END            }
END            }
```

where the LOOP(n) - END pairs are nested to depth x. If x is zero initially, the effect is the same as

```
LOOP(1) X
  P
END
```

That is, $\underset{\sim}{P}$ is not executed at all.

(1.4) Example. Suppose we have the L_ω program

```
LOOP(2) X
X = X+1
END
```

and X contains 2. Then the program is equivalent to

```
LOOP(1) X
LOOP(1) X
X = X+1
END
END
```

and execution of the program would leave 8 in register X. Notice that the depth of nesting is not affected by changes to X.

(1.5) Example. If the initial contents of register X are 2, the L_{ω^2} program

```
LOOP(3) X
X = X+1
END
```

is equivalent to the program

```
LOOP(2) X
LOOP(2) X
X = X+1
END
END
```

which is in turn equivalent to

```
      LOOP(1) X
      LOOP(1) X
      LOOP(2) X
Q     X = X+1
      END
      END
      END
```

Now when the program $\underset{\sim}{Q}$ indicated above is executed, the contents of X will change to 8, by Example (1.4). But then the next time $\underset{\sim}{Q}$ is executed, $\underset{\sim}{Q}$ will be equivalent to

```
LOOP(1) X
  :          } depth 8
LOOP(1) X

X = X+1

END
  :          } depth 8
END
```

Thus the expansion of a LOOP(n+1) - END pair in terms of LOOP(n) - END depends on the contents of the associated register at the time the LOOP is encountered.

Finding the number left in register X by the program of (1.5) is left as an exercise for the persistent reader.

§2. Although it would be possible to characterize formally the notion of computation by Loop program directly in terms of the informal discussion above, the examples, especially (1.5), should have convinced the reader that such a characterization would tend to be quite complicated; more seriously, the individual steps in a computation by a Loop program would in themselves involve considerable computation. This is undesirable because we will be attempting to measure the computational complexity of a function by the number of steps required to compute it. If the individual steps turn out to be nearly as complicated as the function itself, this measure can hardly be claimed to have much significance.

We will circumvent this kind of objection by giving a definition of computation by Loop program whose individual steps are quite elementary. The price that must be paid for this characterization is that it is no longer clear from the definition that Loop programs behave as outlined in §1; thus, a theorem must be proved which states in effect that Loop programs operate as desired. The proof, unfortunately, is rather tedious; but given the theorem, we can select whichever version of computation is more appropriate to the case at hand.

To begin this alternate characterization, associate with each program $\underset{\sim}{P}$ not only the registers $\underline{\text{Reg}}\ (\underset{\sim}{P})$, but also a switch and a pushdown store; the latter are used by LOOP and END instructions.

(2.1) Definition. A pushdown store is either the single object (0) or the pair (t,p) where t is an n-tuple of integers and p is a pushdown store. If a pushdown store is (0) it is empty. The depth of (0) is (0), and if p is a pushdown store whose depth is m, the depth of (t,p) is m+1.

For the remainder of this section, let $\underset{\sim}{P}$ be a Loop program with $\underline{\text{Reg}}\ (\underset{\sim}{P}) = \{\bar{X}_r\}$ and let $\underset{\sim}{P}$ consist of the sequence $I_1, I_2, \ldots, I_e$ of instructions where $e > 0$. There is of course no loss of generality in restricting $\underline{\text{Reg}}\ (\underset{\sim}{P})$ in this way.

(2.2) Definition. A state of $\underset{\sim}{P}$ is an $(r+3)$-tuple $(\bar{x}_r, i, \ell, p)$ where $x_j \geq 0$ for $1 \leq j \leq r$, where $1 \leq i \leq e+1$, where $0 \leq \ell \leq 1$, and where p is a pushdown store. A state is initial if $i = 1$ and is final if $i = e+1$.

(2.3) Definition. If s and s' are states of $\underset{\sim}{P}$ with $s = (\bar{x}_r, i, \ell, p)$ and $s' = (\bar{x}'_r, i', \ell', p')$, then s' is the next state of s under $\underset{\sim}{P}$ if $i \neq e+1$, $\bar{x}'_r = \bar{x}_r$ except as provided in (i), (ii), (iii) below, and one of the following holds for some k, n:

(i) If I_i is "$X_k = 0$" then $x'_k = 0$, $i' = i+1$, $\ell = \ell' = 0$, and $p' = p$;

(ii) If I_i is "$X_k = X_k + 1$" then $x'_k = x_k + 1$, $i' = i+1$, $\ell = \ell' = 0$, and $p' = p$;

(iii) If I_i is "$X_k = X_j$" then $x'_k = x_j$, $i' = i+1$, $\ell = \ell' = 0$, and $p' = p$;

(iv) If I_i is "LOOP(n) X_k" and the matching END is I_m, then $i' = m$, $\ell = \ell' = 0$, and $p' = (t,p)$ where $t = (a_1,\ldots,a_n;1)$ and for all j with $1 \leq j < n$, $a_j = x_k \dot{-} 1$, $a_n = x_k$;

(v) For the remaining five cases let I_i be "END", $p = ((a_1,\ldots,a_n;a), q)$, and let the matching LOOP instruction be I_m = "LOOP(n) X_k". If $a_n = 0$, $a = 1$, $\ell = 0$, then $i' = i+1$, $\ell' = 0$, $p' = q$; or

(vi) If $a_n = 0$, $a = 0$, $\ell = 0$ then $i' = i$, $\ell' = 0$, $p' = q$;

(vii) If for all j with $1 \leq j < n$, $a_j = 0$, but $a_n \neq 0$ and $\ell = 0$ then $i' = m+1$, $\ell' = 0$, $p' = ((a_1,\ldots,a_{n-1},a_n - 1;a), q)$;

(viii) If for some u with $1 \leq u < n$, $a_u \neq 0$, and $a_n \neq 0$, $\ell = 0$, then $i' = i$, $\ell' = 1$, $p' = ((a_1,\ldots,a_{n-1},a_n-1;a),q)$;

(ix) If for some u with $1 \leq u < n$, $a_u \neq 0$, and for all j with $u < j < n$, $a_j = 0$, and $\ell = 1$, then $i' = i$, $\ell' = 0$, $p' = ((a'_1,\ldots,a'_n;0), p)$ where for $1 \leq j < u$, $a'_j = a_j$, $a'_u = a_u - 1$, and for $u < j < n$, $a'_j = x_k \dot{-} 1$, and $a_n = x_k$.

(2.4) Definition. Let $\underset{\sim}{P} = I_1, I_2, \ldots, I_e$ be a Loop program.

A sequence $s_1, \ldots, s_m$ of states of $\underset{\sim}{P}$ is an *execution* of $\underset{\sim}{P}$ whenever

(i) s_1 is initial, and

(ii) s_m is final, and

(iii) The pushdown stores of s_1 and s_m are the same, and

(iv) For each i, $1 \leq i < m$, s_{i+1} is the next state of s_i under $\underset{\sim}{P}$.

If the pushdown store of s_1 is empty, the execution is *proper*.

(2.5) Definition. If there is a unique execution of $\underset{\sim}{P}$ of length m beginning with $(\bar{x}_r, 1, 0, p)$ and ending with $(\bar{x}'_r, e+1, 0, p)$ then for $1 \leq i \leq r$, x'_i is the *integer left in* X_i *by* $\underset{\sim}{P}$ *when* Reg $(\underset{\sim}{P})$ *initially contain* $\bar{x}_r$. Also $m-1$ is the *running time*.

(2.6) Definition. If for each $\bar{x}_r$ there is a unique proper execution of $\underset{\sim}{P}$ beginning with $(\bar{x}_r, 1, 0, (0))$, then let $T_{\underset{\sim}{P}}(\bar{x}_r)$ be the running time of the execution beginning with $(\bar{x}_r, 1, 0, (0))$.

Definition (2.2) may seem complicated, but its complexity lies in the multitude of clauses rather than in the clauses themselves. A more comprehensible description of the execution of a Loop program can be given as follows.

(i)-(iii) If the current instruction is an instruction of type (1), (2) or (3), carry out the instruction in the obvious way and go on to the next instruction.

(iv) If the current instruction is "LOOP(n) X_k" put the $(n+1)$-tuple $(x_k \dot{-} 1, \ldots, x_k \dot{-} 1, x_k; 1)$ on the pushdown store. (If n = 1, put $(x_k; 1)$ on the pushdown store.) Then go to the matching END instruction.

(v) If the current instruction is "END", and if the top of the pushdown store is $(a_1, \ldots, a_n; 1)$ with $a_n = 0$, and $\ell = 0$, pop up the pushdown store and go on to the next instruction.

(vi) If the current instruction is "END", and if the top of the pushdown store is $(a_1, \ldots, a_n; 0)$ with $a_n = 0$, and $\ell = 0$, pop up the pushdown store and do this instruction again.

(vii) If the current instruction is "END", and if the top of the pushdown store is $(a_1, \ldots, a_n; a)$ with $a_i = 0$ for all $i < n$ but $a_n \neq 0$, and $\ell = 0$, subtract 1 from a_n and go to the instruction following the matching LOOP.

(viii) If the current instruction is "END", and if the top of the pushdown store is $(a_1, \ldots, a_n; a)$ with $a_n \neq 0$ and $a_u \neq 0$ for some $u < n$, and $\ell = 0$, subtract 1 from a_n and set $\ell = 1$; then do this instruction again.

(ix) If the current instruction is "END", and if the top of the pushdown store is $(a_1, \ldots, a_u, 0, \ldots, 0, a_n; a)$ with $1 \leq u < n$ and $a_u \neq 0$, and $\ell = 1$, and if the matching LOOP instruction is "LOOP(n) X_k", then set $\ell = 0$ and put the $(n+1)$-tuple $(a_1, \ldots, a_{u-1}, a_u \dot{-} 1, x_k \dot{-} 1, \ldots, x_k \dot{-} 1, x_k; 0)$ on the pushdown store; then do the END instruction again. This exhausts the cases which can possibly arise.

Examination of the various cases of (2.3) should convince the reader that the next state of a given state is unique if it exists at all, and thus that there is at most one execution with a given initial state. The possibilities do arise that a state has no next state yet is not final, or that there is never a final state; but the theorem about to be proved has among its consequences that from any initial state there is exactly one execution, and thus that the running time $T_{\underset{\sim}{P}}$ and the integer left in X_i are well-defined functions from N^r into N.

(2.7) Definition. Two programs $\underset{\sim}{P}$ and $\hat{\underset{\sim}{P}}$ are <u>equivalent</u> if given any initial state of $\underset{\sim}{P}$ and $\hat{\underset{\sim}{P}}$ there are unique executions of $\underset{\sim}{P}$ and $\hat{\underset{\sim}{P}}$ whose final states are the same except perhaps in the third from last ("instruction counter") component.

(2.8) Theorem. Let $\underset{\sim}{P}$ be a Loop program using r registers.

(i) If $s = (\bar{x}_r, 1, 0, p)$ is an initial state of $\underset{\sim}{P}$, there is a unique execution of $\underset{\sim}{P}$ beginning with s; furthermore, the running time and the integers left in <u>Reg</u> $(\underset{\sim}{P})$ are independent of p, the initial pushdown store.

(ii) If $\underset{\sim}{P}$ is of the form

LOOP(1) X
$\underset{\sim}{Q}$
END

where $\underset{\sim}{Q}$ is a Loop program and X is a register name, let X contain x initially; then $\underset{\sim}{P}$ is equivalent to $\hat{\underset{\sim}{P}}$ =

$$\left.\begin{array}{l} \underset{\sim}{Q} \\ \underset{\sim}{Q} \\ \vdots \\ \underset{\sim}{Q} \end{array}\right\} x$$

and $T_{\underset{\sim}{P}}(\bar{x}_k) = T_{\hat{\underset{\sim}{P}}}(\bar{x}_k) + x + 2$.

(iii) If $\underset{\sim}{P}$ is of the form

LOOP(n+1) X
$\underset{\sim}{Q}$
END

for $\underset{\sim}{Q}$ a Loop program, $n \geq 1$, and X a register name, let X contain x initially. Then if $x > 0$ $\underset{\sim}{P}$ is equivalent to $\hat{\underset{\sim}{P}}$ =

$$\left.\begin{array}{l} \text{LOOP(n) X} \\ \text{LOOP(n) X} \\ \vdots \\ \text{LOOP(n) X} \end{array}\right\} x$$

$$\left.\begin{array}{l} \underset{\sim}{Q} \\ \text{END} \\ \vdots \\ \text{END} \\ \text{END} \end{array}\right\} x$$

and if $x = 0$, $\underset{\sim}{P}$ is equivalent to $\hat{\underset{\sim}{P}}$ =

LOOP(1) X
$\underset{\sim}{Q}$
END

In both cases $T_{\hat{\underset{\sim}{P}}}(\bar{x}_r) = T_{\underset{\sim}{P}}(\bar{x}_r)$.

Proof. The proof is by transfinite induction on Definition (1.1) of L_α.

If $\underset{\sim}{P} \in L_\alpha$ by (1.1.i), so that $\alpha = 0$ and $\underset{\sim}{P}$ contains no LOOP instructions, (i) of the theorem is obvious and (ii) and (iii) are vacuous. If $\underset{\sim}{P} \in L_\alpha$ by (1.1.ii), so that $\underset{\sim}{P} \in L_\beta$ with $\beta < \alpha$, the theorem is immediate by the induction hypothesis. If $\underset{\sim}{P} \in L_\alpha$ by (1.1.iii) so that $\underset{\sim}{P}$ is $\underset{\sim}{Q}$ concatenated with $\underset{\sim}{R}$, any final state of $\underset{\sim}{Q}$ corresponds in an obvious way with an initial state of $\underset{\sim}{R}$; the details are omitted.

Now assume that $\underset{\sim}{P} \in L_\alpha$ by (1.1.iv) with $n = 0$; that is, $\underset{\sim}{P}$ is

LOOP(1) X
$\underset{\sim}{Q}$
END

for some $\underset{\sim}{Q} \in L_\beta$ where $\alpha = \beta + 1$ and X is some register name. Let there be e instructions in $\underset{\sim}{P}$ and say x is the initial contents of X; as an induction hypothesis assume that $\underset{\sim}{Q}$ satisfies (2.8.i). Consider the initial state $(\bar{x}_r,1,0,p)$. By (2.3.iv) the unique next state is $(\bar{x}_r,e,0,((x;1),p))$; the next state after this, by (2.3.vii), is $(\bar{x}_r,2,0,((x-1;1),p))$ if $x > 0$. But this is essentially an initial

state in an execution of $\underset{\sim}{Q}$; by the induction hypothesis the next several states consist of an execution of $\underset{\sim}{Q}$ which ends with $(\bar{x}'_r,e,0,((x-1;1),p))$ for some $\bar{x}'_r$. Then the next state is $(\bar{x}'_r,2,0,((x-2;1),p))$ if $x > 1$; repeating the argument leads, after x executions of $\underset{\sim}{Q}$, to the state $(\bar{x}''_r,e,0,((0;1),p))$. By (2.3.v) the next state is $(\bar{x}''_r,e+1,0,p)$ which is final. Counting the number of states not involved in the executions of $\underset{\sim}{Q}$ yields (2.8.ii) and thus (2.8.i).

The remaining possibility is that $\underset{\sim}{P} \in L_\alpha$ by (1.1.iv) with $n > 0$, so that $\underset{\sim}{P}$ is

I_1: LOOP(n+1) X

$\underset{\sim}{Q}$

I_e: END

Let the final END instruction be the e-th instruction of $\underset{\sim}{P}$, as indicated above. We have to show first that the program $\hat{\underset{\sim}{P}}$ =

I_1: LOOP(n) X
I_2: LOOP(n) X
⋮ ⋮
I_x: LOOP(n) X
} x

$\underset{\sim}{Q}$

$I_{\hat{e}}$: END
⋮ ⋮
$I_{\hat{e}+x-2}$: END
$I_{\hat{e}+x-1}$: END
} x

where $x > 0$ is the initial contents of X, is equivalent to $\underset{\sim}{P}$, and that $T_{\underset{\sim}{P}}(\bar{x}_r) = T_{\hat{\underset{\sim}{P}}}(\bar{x}_r)$. As indicated, we let $I_{\hat{e}}$ be the first END instruction of $\hat{\underset{\sim}{P}}$ after $\underset{\sim}{Q}$. The method is to consider an execution of $\hat{\underset{\sim}{P}}$ and show that each state of this execution corresponds in an appropriate sense to a state in the execution of $\underset{\sim}{P}$; the correspondence includes the requirements that the registers be the same, and that the pushdown stores be "similar". Since $\hat{\underset{\sim}{P}} \in L_{\beta+\omega^{n-1}x}$, and $\alpha > \beta + \omega^{n-1}x$, the induction hypothesis for $\hat{\underset{\sim}{P}}$ will yield the result desired.

In the definition and lemma that follow, we use a consistent notation: letters without hats refer to the program $\underset{\sim}{P}$, and those with hats refer to $\hat{\underset{\sim}{P}}$; for example, s and $\hat{s}$ are states of $\underset{\sim}{P}$ and $\hat{\underset{\sim}{P}}$ respectively. Also, a primed letter refers to the next state of a given state; so if, for example, $\hat{s}$ is a state of $\hat{\underset{\sim}{P}}$, $\hat{s}'$ is the next state of $\hat{s}$ under $\hat{\underset{\sim}{P}}$. Finally, x is the initial contents of register X. We assume that $x > 0$.

(2.9) Definition. For a pushdown store p let $\sigma_j p$ be the object at the j-th level of p; that is, if $p = (q_1,(q_2,\ldots(q_k,(0))\ldots))$, then $\sigma_j p = q_j$ for $1 \leq j \leq k$; if $j > k$, $\sigma_j \rho = 0$. Two pushdown stores p and $\hat{p}$ occurring in states of $\underset{\sim}{P}$ and $\hat{\underset{\sim}{P}}$ are <u>similar</u> if for each j one of the following holds:

(i) $\sigma_j p = \sigma_j \hat{p}$, or

(ii) $\sigma_j p = (y,a_1,\ldots,a_n;0)$ and $\sigma_j \hat{p} = (a_1,\ldots,a_n;0)$ and $\sigma_{j+1} p = (y,b_1,\ldots,b_n;b)$ for some y with $0 \leq y < x$; or

(iii) $\sigma_j p = (y,a_1,\ldots,a_n;0)$ and $\sigma_i \hat{p} = (a_1,\ldots,a_n;1)$ and $\sigma_{j+1} p = (y+1,b_1,\ldots,b_n;b)$ where for $1 \leq k < n$, $b_k = 0$, and $0 \leq y < x$; or

(iv) $\sigma_j p = (x-1,a_1,\ldots,a_n;1)$ and $\sigma_j \hat{p} = (a_1,\ldots,a_n;1)$.

(2.10) Lemma. Let $\hat{s}_1,\ldots,\hat{s}_m$ be an execution of $\hat{\underset{\sim}{P}}$. Then there is an execution $s_1,\ldots,s_m$ of $\underset{\sim}{P}$ such that $s_1 = \hat{s}_1$ and for each pair $s_j = (\bar{x}_r,i,\ell,p)$ and $\hat{s}_j = (\hat{\bar{x}}_r,\hat{i},\hat{\ell},\hat{p})$ we have $x_j = \hat{x}_j$ for $1 \leq j \leq r$, p is similar to $\hat{p}$, and one of the following holds:

(i) $1 < i < e$ and $\hat{i} = i + x - 1$; or

(ii) $i = \hat{i} = 1$, and $\ell = \hat{\ell} = 0$; or

(iii) $\sigma_1 p = (y,0,\ldots,0,a_n;a)$ with $0 < y < x$, $i = e$, $\ell = 1$, $\hat{\ell} = 0$, $\hat{i} = x - y + 1$; or

(iv) $i = e+1$, $\hat{i} = \hat{e} + x$

(v) $\sigma_1 p = (y,a_1,\ldots,a_n;a)$ and $i = e$, $\hat{i} = \hat{e} + y$ with $0 \leq y < x$, and $\ell = \hat{\ell}$.

Proof of Lemma. Let $s = \hat{s} = (\bar{x}_r,1,0,p)$ be an initial state of $\underset{\sim}{P}$ and $\hat{\underset{\sim}{P}}$. Then s and $\hat{s}$ satisfy (2.10.ii), and $p = \hat{p}$ so p is similar to $\hat{p}$ by (2.9.i). Now assume that $s = (\bar{x}_r,i,\ell,p)$ and $\hat{s} = (\bar{x}_r,\hat{i},\hat{\ell},\hat{p})$ are states of $\underset{\sim}{P}$ and $\hat{\underset{\sim}{P}}$ satisfying (2.10); we prove that s' and $\hat{s}'$ also satisfy (2.10). The proof consists in considering the cases that arise.

Case 1. s and $\hat{s}$ satisfy (2.10.i). Then $\underset{\sim}{P}$ and $\hat{\underset{\sim}{P}}$ are executing the same instruction of $\underset{\sim}{Q}$, and the result follows from an induction hypothesis on $\underset{\sim}{Q}$.

Case 2. s and $\hat{s}$ satisfy (2.10.ii), so $i = \hat{i} = 1$, $\ell = \hat{\ell} = 0$. Then (2.3.iv) applies to both s and $\hat{s}$: $s' = (\bar{x}_r, \hat{e} + x - 1, 0, ((x \dot{-} 1, \ldots, x \dot{-} 1, x; 1), \hat{p}))$ and $s' = (\bar{x}_r, e, 0, ((x \dot{-} 1, x \dot{-} 1, \ldots, x \dot{-} 1, x; 1), p))$. Then s' and $\hat{s}'$ satisfy (2.10.v) and p' and $\hat{p}'$ remain similar by (2.9.iv).

Case 3. s and $\hat{s}$ satisfy (2.10.iii), so $i = e$, $\hat{i} = x - y + 1$, $\ell = 1$, $\hat{\ell} = 0$. Then (2.3.iv) applies to $\hat{s}$, so if x_k is the current contents of register X, $\hat{s} = (\bar{x}_r, \hat{e} + y - 1, 0, ((x_k \dot{-} 1, \ldots, x_k \dot{-} 1, x_k; 1), \hat{p}))$. Also, (2.3.ix) applies to s, so $s' = (\bar{x}_r, e, 0, ((y-1, x_k \dot{-} 1, \ldots, x_k \dot{-} 1, x_k; 0), p))$. Now p' and $\hat{p}'$ remain similar by (2.9.iii); s' and $\hat{s}'$ satisfy (2.10.v).

Case 4. s and $\hat{s}$ satisfy (2.10.iv), so $i = e + 1$, $\hat{i} = \hat{e} + x$. The s and $\hat{s}$ are both final and neither has a next state.

Case 5. s and $\hat{s}$ satisfy (2.10.v), so $i = e$, $\hat{i} = \hat{e} + y$ where $0 \leq y < x$, $\ell = \hat{\ell}$, $\sigma_1 p = (y, a_1, \ldots, a_n; a)$, and by similarity, $\sigma_1 \hat{p} = (a_1, \ldots, a_n; \hat{a})$. There are several subcases corresponding to various possibilities for $\hat{s}$.

Subcase 5.1. (2.3.v) applies to $\hat{s}$: $\hat{a} = 1$, $a_n = 0$, $\hat{\ell} = 0$. Then by (2.3.v) $\hat{s}' = (\bar{x}_r, \hat{e} + y + 1, 0, \hat{q})$. First say $\sigma_1 p$ and $\sigma_1 \hat{p}$ satisfy (2.9.iii); then $\sigma_1 p = (y, a_1, \ldots, a_n; 0)$ and since $a_n = 0$, (2.3.vi) applies to s, so $s' = (\bar{x}_r, e, 0, q)$. But by (2.9.iii), $\sigma_2 p = (y + 1, b_1, \ldots, b_n; b)$ so s' and $\hat{s}'$ satisfy (2.10.v). On the other hand, if $\sigma_1 p$ and $\sigma_1 \hat{p}$ satisfy (2.9.iv), then by (2.3.v), $s' = (\bar{x}_r, e + 1, 0, q)$. Also by (2.3.v), $\hat{s}' = (\bar{x}_r, \hat{e} + x, 0, \hat{q})$ and so $\hat{s}$ and $\hat{s}'$ satisfy (2.10.iv).

Subcase 5.2. (2.3.vi) applies to $\hat{s}$: $\hat{a} = 0$, $a_n = 0$, $\hat{\ell} = 0$, and so $\hat{s}' = (\bar{x}_r, \hat{e}+y, 0, \hat{q})$. (2.3.vi) must also apply to s, so s' = $(\bar{x}_r, e, 0, q)$ and s' and $\hat{s}'$ satisfy (2.10.v).

Subcase 5.3. (2.3.vii) applies to $\hat{s}$: $\hat{\ell} = 0$ and $a_j = 0$ for $1 \leq j < n$ but $a_n \neq 0$. Then $\hat{s}' = (\bar{x}_r, x-y+1, 0, ((a_1, \ldots, a_{n-1}, a_n \dot{-} 1; \hat{a}), \hat{q}))$ by (2.3.vii). If $y = 0$, (2.3.vii) also applies to s and s' = $(\bar{x}_r, 2, 0, ((0, a_1, \ldots, a_{n-1}, a_n \dot{-} 1; a), q))$ so s' and $\hat{s}'$ satisfy (2.10.i). If $y > 0$, then (2.3.viii) applies to s; s' = $(\bar{x}_r, e, 1, ((y, a_1, \ldots, a_{n-1}, a_n \dot{-} 1; a), q))$. Then s' and $\hat{s}'$ satisfy (2.10.iii).

Subcase 5.4. (2.3.viii) applies to $\hat{s}$: $\hat{\ell} = 0$, $a_n \neq 0$, and for some u with $1 \leq u < n$, $a_u \neq 0$. Then $\hat{s}' = (\bar{x}_r, \hat{e}+y, 1, ((a_1, \ldots, a_{n-1}, a_n \dot{-} 1; \hat{a}), \hat{q}))$. By similarity, (2.3.viii) also applies to s, so s' = $(\bar{x}_r, e, 1, ((y, a_1, \ldots, a_{n-1}, a_n \dot{-} 1; a), q))$ and s' and $\hat{s}'$ satisfy (2.10.v).

Subcase 5.5. (2.3.ix) applies to $\hat{s}$: $\hat{\ell} = 1$, for some u with $1 \leq u < n$, $a_u \neq 0$, and for all j with $u < j < n$, $a_j = 0$. Then if x_k is the current contents of X, $\hat{s}' = (\bar{x}_r, \hat{e}+y, 0, ((a_1, \ldots, a_{u-1}, a_u \dot{-} 1, x_k \dot{-} 1, \ldots, x_k \dot{-} 1, x_k; 0), \hat{p}))$. By similarity (2.3.ix) applies also to s, and so s' = $(\bar{x}_r, e, 0, ((y, a_1, \ldots, a_{u-1}, a_u \dot{-} 1, x_k \dot{-} 1, \ldots, x_k \dot{-} 1, x_k; 0), p))$. Then p' and $\hat{p}'$ remain similar by (2.9.ii), and s' and $\hat{s}'$ satisfy (2.10.v). This concludes the proof of Lemma (2.10).

We have thus shown that given an execution of $\hat{\underset{\sim}{P}}$, there is an identical-length execution of $\underset{\sim}{P}$ with the same initial state and such that in each corresponding state the registers are identical. Also, by the

similarity of the pushdown stores, the execution of $\underset{\sim}{P}$ ends with the pushdown store the same as it was initially; $\underset{\sim}{P}$ and $\hat{\underset{\sim}{P}}$ are then equivalent.

The sole remaining case is that x, the initial contents of X, is zero. But then the following is an execution of $\underset{\sim}{P}$:

$$s_1 = (\bar{x}_r, 1, 0, p)$$

$$s_2 = (\bar{x}_r, e, 0, ((0,0,\ldots,0;1),p)) \qquad \text{by (2.3.iv)}$$

$$s_3 = (\bar{x}_r, e+1, 0, p) \qquad \text{by (2.3.v)}$$

This proves (2.8.iii); (2.8.i) is immediate by the induction hypothesis for $\hat{\underset{\sim}{P}}$ and Theorem (2.8) is proved.

In view of (2.8.i) the distinction between executions and proper executions (in which the pushdown store is initially empty) is unnecessary, since the initial contents of the pushdown store do not affect the quantities of interest, the final contents of the registers and the running time.

§3. The previous section showed that the running time function $T_{\underset{\sim}{P}}$ for any program $\underset{\sim}{P}$ is totally defined. It should also be intuitively clear that $T_{\underset{\sim}{P}}$ is effectively computable. Thus the claim that the running time of a Loop program is bounded *a priori* is trivially true, provided that the claim simply means that given a program with its initial state, there is an effective method of finding a number that bounds the number of steps required for the program to halt. For since any Loop program with any input eventually does halt, an "effective method" simply consists of running the program and counting the steps.

Of course, bounding the running time of $\underset{\sim}{P}$ by $T_{\underset{\sim}{P}}$ is not very informative, for it amounts to "predicting" that $\underset{\sim}{P}$ will run as long as it runs. One would at least hope for bounding functions which are in some sense sufficiently comprehensible that they provide more information than the previous tautology. An inevitable difficulty is that the bounding functions must grow at such extraordinary rates that their sizes can hardly be called comprehensible. Nevertheless, the functions f_α defined below have such simple definitions and useful properties that our Theorem (3.6) below has intuitive appeal as well as technical usefulness.

(3.1) Definition. If $g: N \to N$ is a function, the function $h: N^2 \to N$ is called the *iterate* of g (or, h *is defined by iteration from* g) whenever h satisfies

$$h(0,z) = z$$
$$h(y+1,z) = g(h(y,z))$$

Often, we will write the iterate $h(y,z)$ as $g^{(y)}(z)$. Thus, $g^{(y)}(z) = g(g(\ldots\ g(z)\ \ldots))$, the composition being taken y times.

(3.2) Definition. For $\alpha < \omega^{\omega}$ an ordinal, the function f_α is defined as follows:

(i) if $\alpha = 0$, $f_\alpha(x) = x+1$ if $x \leq 1$; $f_\alpha(x) = x+2$ if $x > 1$;

(ii) if $\alpha = \beta+1$, $f_\alpha(x) = f_\beta^{(x)}(1)$;

(iii) if α is a limit ordinal and β is the least ordinal satisfying $\alpha = \beta + \omega^{n+1}$ for some $n \geq 0$, then $f_\alpha(x) = f_{\beta+\omega^n x}(x)$.

Thus if α is a successor, f_α is defined by iteration from its predecessor; if α is a limit, f_α is defined by diagonalization over a certain sequence $\{f_{\beta_i}\}$ of functions where $\underline{\sup}\ \{\beta_i\} = \alpha$.

In the proofs below we will use implicitly a number of elementary facts about the arithmetic of ordinals, and also the Normal Form Theorem for ordinals less than ω^{ω}: any ordinal $\alpha < \omega^{n+1}$ for some n, $0 \leq n < \omega$, may be written

$$\alpha = \omega^n \cdot a_n + \omega^{n-1} \cdot a_{n-1} + \cdots + \omega^0 \cdot a_0$$

where $0 \leq a_i < \omega$ and the a_i are unique. See, for example, Suppes' book [5].

(3.3) Definition. For $\alpha = \omega^n \cdot a_n + \cdots + \omega^0 \cdot a_0$ an ordinal, write

$$t_m(\alpha) = \Sigma_{i=0}^{m-1}\ a_i$$

for each $m \leq n+1$; if $m > n+1$, $t_m = t_{n+1}(\alpha)$. Also, $t_\omega(\alpha) = t_{n+1}(\alpha)$ if $\alpha < \omega^{n+1}$.

Notice that $t_0(\alpha) = 0$ for all α. The next lemma collects most of the information we require about the functions f_α.

(3.4) Lemma. For all $x, p \in N$, $\alpha, \beta < \omega^\omega$:

(i) $f_1(x) = 2x + (1 \dot{-} x)$

(ii) $f_1^{(p+1)}(x) = 2^p \cdot f_1(x) \geq 2^{p+1} \cdot x$

(iii) $f_2(x) = 2^x$

(iv) $f_\alpha(0) = 1$

(v) $f_\alpha(x) \geq x + 1$

(vi) $f_\alpha^{(p)}(x)$ is increasing in p, x

(vii) if $\alpha = \beta + \omega^n$, then $f_\alpha(x) \geq f_\beta(x)$ for $x \geq t_n(\beta)$

(viii) if $\alpha > \beta$, then $f_\alpha(x) \geq f_\beta(x)$ for $x \geq t_\omega(\beta)$

(ix) $2 \cdot f_\alpha^{(p)}(x) \leq f_\alpha^{(p+1)}(x)$ for $\alpha \geq 1$, $x + p \geq 1$

(x) $\left(f_\alpha^{(p)}(x)\right)^2 \leq f_\alpha^{(p+2)}(x)$ for $\alpha \geq 2$, $x + p \geq 2$.

Proof. (i) If $x = 0$, $f_1(0) = f_0^{(0)}(1) = 1 = 2 \cdot 0 + (1 \dot{-} 0)$. $f_1(1) = f_0^{(1)}(1) = 2 = 2 \cdot 1 + (1 \dot{-} 1)$. If for $x \geq 1$ $f_1(x) = 2x$, $f_1(x+1) = f_0 f_1(x) = 2x + 2 = 2(x+1) + (1 \dot{-} (x+1))$.

(ii) Immediate for $p = 0$. $f_1^{(p+1)}(x) = f_1 f_1^{(p)}(x)$ $= 2 \cdot f^{(p)}(x) = 2^p \cdot f_1(x) \geq 2^{p+1} \cdot x$.

(iii) $f_2(0) = f_1^{(0)}(1) = 1$. $f_2(x+1) = f_1^{(x+1)}(1) = 2^x \cdot f_1(1)$ $= 2^{x+1}$ by (ii).

(iv)-(vii) These will all be proved simultaneously by induction on α and x. All are immediate for $\alpha = 0$ by definition.

If $\alpha = \beta + 1$, then $f_\alpha(0) = f_\beta^{(0)}(1) = 1$ proving (iv). Also, $f_\alpha(0) \geq 0$, yielding (v). Now $f_\alpha(x+1) = f_\beta f_\alpha(x) > f_\alpha(x)$, using (v) for f_β. Then $f_\alpha^{(p+1)}(x) = f_\alpha f_\alpha^{(p)}(x) > f_\alpha^{(p)}(x)$, proving (vi). Also, $f_\alpha(x+1) = f_\beta f_\alpha(x) > (x+1)+1$, proving (v).

Now in (vii), n must be 0 since α is a successor. Since $f_\alpha(x) \geq x+1$, $f_\beta f_\alpha(x) \geq f_\beta(x+1)$, using (vi) for f_β. But $f_\beta f_\alpha(x) = f_\alpha(x+1)$ so $f_\alpha(x+1) \geq f_\beta(x+1)$ for all $x \geq 0 = t_0(\beta)$, proving (vii).

The next possibility is that α is a limit ordinal: let $\alpha = \beta + \omega^{n+1}$ where $n \geq 0$ and β is the least such ordinal. Then $f_\alpha(x) = f_{\beta+\omega^n x}(x)$. Now $f_\alpha(0) = f_\beta(0) = 1$, proving (iv). Also,

$$\begin{aligned} f_\alpha(x+1) &= f_{\beta+\omega^n(x+1)}(x+1) \\ &> f_{\beta+\omega^n(x+1)}(x) \quad \text{by (vi)} \\ &\geq f_{\beta+\omega^n x}(x) \quad \text{by (vii) since } t_n(\beta+\omega^n x) = 0 \\ &= f_\alpha(x) \end{aligned}$$

Then $f_\alpha^{(p+1)}(x) = f_\alpha f_\alpha^{(p)}(x) > f_\alpha^{(p)}(x)$, proving (vi). Also, $f_\alpha(0) = 1 \geq 0+1$ and $f_\alpha(x+1) > f_\alpha(x) > x+1$ proving (v). Finally, write $\beta = \beta' + \gamma$ where $t_{n+1}(\beta') = 0$ and $\gamma < \omega^{n+1}$. Then $\alpha = \beta + \omega^{n+1} = \beta' + \omega^{n+1}$, and by choice of β', $f_\alpha(x) = f_{\beta'+\omega^n x}(x)$. Since $\gamma < \omega^{n+1}$, if $x \geq t_{n+1}(\gamma)$ then $\omega^n x \geq \gamma$. So, using (vii) for $\beta' + \omega^n x$ and $\beta' + \gamma$,

$$f_\alpha(x) = f_{\beta'+\omega^n x}(x) \geq f_{\beta'+\gamma}(x) = f_\beta(x)$$

if $x \geq t_n(\gamma)$. But $x \geq t_{n+1}(\gamma) \geq t_n(\gamma)$, proving (vii). This completes the proof of (iv)-(vii).

(viii) If $\alpha > \beta$ there is a $\gamma > 0$ so $\alpha = \beta + \gamma$. Write $\gamma = \omega^n + \omega^n \cdot g_n + \cdots + \omega^o \cdot g_o = \omega^n + \gamma'$, so $\alpha = \beta + \omega^n + \gamma'$. By repeated applications of (vii) we have $f_\alpha(x) \geq f_{\beta+\omega^n}(x)$ for all x, since $t_n(\beta + \omega^n) = 0$. Also by (vii), $f_{\beta+\omega^n}(x) \geq f_\beta(x)$ if $x \geq t_\omega(\beta) \geq t_n(\beta)$. So $f_\alpha(x) \geq f_\beta(x)$ for $x \geq t_\omega(\beta)$.

(ix) $2 \cdot f_\alpha^{(p)}(x) = f_1 f_\alpha^{(p)}(x) \leq f_\alpha^{(p+1)}(x)$ if $\alpha \geq 1$ and $x + p \geq 1$ by (i) and (viii) since $f_\alpha^{(p)}(x) \geq 1$ if $x + p \geq 1$.

(x) Trivially, $z^2 \leq 2^{2^z} = f_2^{(2)}(z)$ for all z. Then

$$\left(f_\alpha^{(p)}(x)\right)^2 \leq f_2^{(2)} f_\alpha^{(p)}(x) \leq f_\alpha^{(p+2)}(x)$$

if $x + p \geq 2$ by (viii), since $x + p \geq 2$ implies $f_\alpha^{(p)}(x) \geq 2$. This completes the proof of (3.4)

(3.5) Definition. A function $g: N^m \to N$ is <u>bounded by</u> f: $N \to N$ whenever for all $\bar{x}_m$, we have $g(\bar{x}_m) \leq f(\max\{\bar{x}_m\})$, where $\max\{\bar{x}_m\}$ is the largest member of $\bar{x}_m$.

(3.6) Bounding Theorem. Let $\underset{\sim}{P}$ be a program in L_α. Then there is a number p, which can be found effectively from $\underset{\sim}{P}$, such that $f_\alpha^{(p)}$ bounds $T_{\underset{\sim}{P}}$, the running time of $\underset{\sim}{P}$.

Proof. The proof is by induction on α and Definition (1.1). Say $\underset{\sim}{P} \in L_\alpha$, let $\underset{\sim}{P}$ use k registers, and let <u>m</u> be an abbreviation for <u>max</u>$\{\bar{x}_k\}$ where $\bar{x}_k$ are the numbers initially in <u>Reg</u> $(\underset{\sim}{P})$. There are four main cases corresponding to the clauses of (1.1).

Case 1. $\alpha = 0$. Then $\underset{\sim}{P}$ has no loops and so $T_{\underset{\sim}{P}}$ is identically equal to the length of $\underset{\sim}{P}$; if $p > 0$ is this length, then

$$T_{\underset{\sim}{P}}(\bar{x}_k) = p \leq f_0^{(p)}(0) \leq f_0^{(p)}(\underline{m})$$

by (3.4.v), (3.4.vi).

Case 2. $\underset{\sim}{P} \in L_\alpha$ by (1.1.ii), so that $\underset{\sim}{P} \in L_\beta$ with $\beta < \alpha$. By the induction hypothesis we have have q so $T_{\underset{\sim}{P}}(\bar{x}_k) \leq f_\beta^{(q)}(\underline{m})$. But by (3.4.viii), if we let $p = t_\omega(\beta)$ and if $x \geq p$, then $f_\alpha(x) \geq f_\beta(x)$. By (3.4.v), $f_\alpha^{(p)}(\underline{m}) \geq \underline{m} + p$, so $f_\alpha^{(p+q)}(x) = f_\alpha^{(p)} f_\alpha^{(q)}(x) \geq T_{\underset{\sim}{P}}(\bar{x}_k)$.

Case 3. $\underset{\sim}{P} \in L_\alpha$ by (1.1.iii), so that $\underset{\sim}{P}$ is $\underset{\sim}{Q}$ concatenated with $\underset{\sim}{R}$ and $\underset{\sim}{Q}$, $\underset{\sim}{R} \in L_\alpha$. By the induction hypothesis, let $f_\alpha^{(q)}$ and $f_\alpha^{(r)}$ bound $T_{\underset{\sim}{Q}}$ and $T_{\underset{\sim}{R}}$ respectively. After execution of $\underset{\sim}{Q}$, let the registers of $\underset{\sim}{P}$ contain $\bar{x}_k'$. Then $T_{\underset{\sim}{P}}(\bar{x}_k) = T_{\underset{\sim}{Q}}(\bar{x}_k) + T_{\underset{\sim}{R}}(\bar{x}_k')$; we have

$$T_{\underset{\sim}{P}}(\bar{x}_k) \leq f_\alpha^{(q)}(\underline{m}) + f_\alpha^{(r)}(\underline{\max}\{\bar{x}_k'\})$$

But after execution of $\underset{\sim}{Q}$, the largest integer in any register is at most $\underline{m} + f_\alpha^{(q)}(\underline{m})$, since each instruction execution can increase the largest register by 1 at most. But by (3.4.v) and (3.4.ix), $\underline{m} + f_\alpha^{(q)}(\underline{m}) \leq f_\alpha^{(q+1)}(\underline{m})$ since $\alpha \geq 1$, $q \geq 1$. Thus

$$\begin{aligned} T_{\underset{\sim}{P}}(\bar{x}_k) &\leq f_\alpha^{(q)}(\underline{m}) + f_\alpha^{(r)} f_\alpha^{(q+1)}(\underline{m}) \\ &\leq 2 \cdot f_\alpha^{(q+r+1)}(\underline{m}) && \text{by (3.4.vi)} \\ &\leq f_\alpha^{(q+r+2)}(\underline{m}) && \text{by (3.4.ix)} \end{aligned}$$

Case 4. $\underset{\sim}{P} \in L_\alpha$ by (1.1.iv), so that $\underset{\sim}{P}$ is

```
LOOP(n+1)  X
  Q
END
```

for some $\underset{\sim}{Q} \in L_\beta$ where $\alpha = \beta + \omega^n$, $n \geq 0$. We use the following

(3.7) Lemma. If $\underset{\sim}{Q} \in L_\beta$ and $T_{\underset{\sim}{Q}}$ is bounded by $f_\beta^{(q)}$, then the program $\underset{\sim}{P}$ =

```
LOOP(n+1)  X
  Q
END
```

has $T_{\underset{\sim}{P}}$ bounded by $f_{\beta+\omega^n}^{(q+b+4)}$, where $b = t_n(\beta)$.

Proof of Lemma. The proof is by induction on n. For $n = 0$, $t_n(\beta) = 0$ for all β. Then the lemma reduces to: if $\underset{\sim}{Q} \in L_\beta$ and $T_{\underset{\sim}{Q}}(\bar{x}_k) \leq f_\beta^{(q)}(\underline{m})$, then $\underset{\sim}{P}$ =

```
LOOP(1)    X
  Q
END
```

has $T_{\underset{\sim}{P}}(\bar{x}_k) \leq f_{\beta+1}^{(q+4)}(\underline{m})$. There are two possibilities; first take $\beta = 0$. Then $T_{\underset{\sim}{Q}}$ is identically equal to the length of $\underset{\sim}{Q}$, and the running time of $\underset{\sim}{P}$ is exactly $qx + x + 2$, where x is the initial contents of X, by (2.8.ii). But since by definition $x \leq \underline{m}$,

$$T_{\underset{\sim}{P}}(\bar{x}_r) = (q+1)x+2 \leq (q+1)\underline{m}+2$$
$$\leq 2^q \cdot \underline{m}+2$$
$$\leq f_1^{(q+2)}(\underline{m}) \quad \text{by (3.4.ii) and (3.4.v)}$$
$$\leq f_1^{(q+4)}(\underline{m}) \quad \text{by (3.4.vi)}$$

Now if $0 < \beta < \omega^\omega$, assume that $T_{\underset{\sim}{Q}}$ is bounded by $f_\beta^{(q)}$. If x is the number initially in X, $\underset{\sim}{P}$ is equivalent to

$$\left.\begin{matrix} \underset{\sim}{Q} \\ \underset{\sim}{Q} \\ \cdots \\ \underset{\sim}{Q} \end{matrix}\right\} x$$

By the same argument as for Case 3 of the theorem, the first execution of $\underset{\sim}{Q}$ requires at most $f_\beta^{(q)}(\underline{m})$ steps, the second $f_\beta^{(2q+2)}(\underline{m})$ steps, the third $f_\beta^{(3q+4)}(\underline{m})$ steps, ... , and by the obvious induction, the x-th requires at most $f_\beta^{(x(q+2)-2)}(\underline{m})$ steps. Thus, if $\underline{m} > 0$

$$T_{\underset{\sim}{P}}(\bar{x}_k) \leq \Sigma_{i=1}^{x} f_\beta^{(i(q+2)-2)}(\underline{m})+2$$
$$\leq x \cdot f_\beta^{(\underline{m}(q+2)-2)}(\underline{m})+2 \quad \text{by (3.4.vi)}$$
$$\leq \underline{m} \cdot f_\beta^{(\underline{m}(q+2)-2)} f_\beta^{(\underline{m})}(1)+2 \quad \text{by (3.4.v)}$$
$$\leq \underline{m} \cdot f_\beta^{(\underline{m}(q+3))}(1) \quad \text{by (3.4.v)}$$
$$= \underline{m} \cdot f_\alpha(\underline{m} \cdot (q+3)) \quad \text{by (3.2) since } \alpha = \beta+1$$
$$\leq \underline{m} \cdot f_\alpha f_1^{(q+1)}(\underline{m}) \quad \text{by (3.4.ii)}$$
$$\leq \underline{m} \cdot f_\alpha^{(q+2)}(\underline{m}) \quad \text{by (3.4.vii)}$$
$$\leq f_\alpha^{(q+4)}(\underline{m}) \quad \text{by (3.4.x)}$$

But even if $\underline{m} = 0$, $T_{\underset{\sim}{P}}(\bar{x}_k) = 2$, so $T_{\underset{\sim}{P}}(\bar{x}_k) \leq f_\beta^{(q+4)}(\underline{m})$ for all $\bar{x}_k$. This concludes the proof for $n = 0$.

Now we assume the lemma for some $n \geq 0$, and prove it for $n+1$. $\underset{\sim}{P}$ is then

```
LOOP(n+2)  X
  Q
END
```

which is equivalent to

```
LOOP(n+1)  X   }
   :           } x
LOOP(n+1)  X   }
  Q
END            }
   :           } x
END            }
```

where $x \geq 1$ is the number initially in X. If $x = 1$, $T_{\underset{\sim}{P}}$ is bounded by $f_{\beta+\omega^n}^{(q+b+4)}$; if $x = 2$, $T_{\underset{\sim}{P}}$ is bounded by $f_{\beta+\omega^n 2}^{(q+b+8)}$ since $t_n(\beta+\omega^n) = 0$; and by the obvious induction, for each $x \geq 1$,

$$T_{\underset{\sim}{P}}(\bar{x}_k) \leq f_{\beta+\omega^n x}^{(q+b+4x)}(\underline{m})$$

$$\leq f_{\beta+\omega^n x+1}(q+b+5\underline{m}+1) \qquad \text{by (3.4.vi) and (3.2)}$$

$$\leq f_{\beta+\omega^n(x+1)}(q+b+5\underline{m}+1) \qquad \text{by (3.4.vii)}$$

$$\leq f_{\beta+\omega^n(q+b+5\underline{m}+1)}(q+b+5\underline{m}+1) \qquad \text{by (3.4.vii)}$$

Now if $\beta = \omega^m b_m + \cdots + \omega^{n+1} b_{n+1} + \omega^n b_n + \cdots + \omega^o b_o$, let $\beta' = \omega^m b_m + \cdots + \omega^{n+1} b_{n+1}$. Then $\beta + \omega^n(q + b + 5\underline{m} + 1) = \beta' + \omega^n(q + b + b_n + 5\underline{m} + 1)$ and furthermore β' is the least ordinal with this property. Thus by (3.2),

$$\begin{aligned} T_{\underset{\sim}{P}}(\bar{x}_k) &\leq f_{\beta'+\omega^{n+1}}(q + b + b_n + 5\underline{m} + 1) \\ &= f_{\beta+\omega^{n+1}}(q + b + b_n + 5\underline{m} + 1) && \text{by definition of } \beta' \\ &\leq f^{(q+b+b_n+1)}_{\beta+\omega^{n+1}}(5\underline{m}) && \text{by (3.4.v)} \\ &\leq f^{(q+b+b_n+1)}_{\beta+\omega^{n+1}} f^{(3)}_1(\underline{m}) && \text{by (3.4.ii)} \\ &\leq f^{(q+b+b_n+4)}_{\beta+\omega^{n+1}}(\underline{m}) && \text{by (3.4.vii)} \end{aligned}$$

But even if $\underline{m} = 0$, $T_{\underset{\sim}{P}}(\bar{x}_k) = 2 \leq f^{(q+b+b_n+4)}_{\beta+\omega^{n+1}}(\underline{m})$ by (3.4.v). Since $t_{n+1}(\beta) = t_n(\beta) + b_n = b + b_n$, the lemma is proved, concluding Case 4.

Since in each of the four cases the p such that $f^{(p)}_\alpha$ bounds $T_{\underset{\sim}{P}}$ was found effectively, Theorem (3.7) is proved. We also have immediately

(3.8) Corollary. Let $\underset{\sim}{P} \in L_\alpha$ be a Loop program, and let $\underline{m}$ be the largest number initially in $\underline{\text{Reg}}\ (\underset{\sim}{P})$. Then there is a number p so that $f^{(p)}_\alpha(\underline{m})$ bounds all the numbers left in the registers of $\underset{\sim}{P}$ by execution of $\underset{\sim}{P}$.

Proof. Since each instruction execution can increase the largest register by 1 at most, the numbers left in $\underline{\text{Reg}}\ (\underset{\sim}{P})$ are all bounded by $\underline{m} + T_{\underset{\sim}{P}}(\bar{x}_k) \leq \underline{m} + f^{(p)}_\alpha(\underline{m})$. If $\alpha \geq 1$, by (3.4.ix) $\underline{m} + f^{(p)}_\alpha(\underline{m}) \leq f^{(p+1)}_\alpha(\underline{m})$. The proof for $\alpha = 0$ is obvious.

§4. If a set of registers is designated for input and output, a Loop program defines a function.

(4.1) Definition. Let $\bar{X}_m$ be distinct register names, and let P be a register name which need not be distinct from $\bar{X}_m$. If $\underset{\sim}{P}$ is a Loop program, the $(m+2)$-tuple $(\underset{\sim}{P}, \bar{X}_m, P)$ is called a <u>program with input and output</u>, $\bar{X}_m$ being the <u>input registers</u> and P the <u>output register</u>. The function $f: N^m \to N$ is <u>computed by</u> $(\underset{\sim}{P}, \bar{X}_m, P)$ providing $f(\underline{x}_m)$ equals the contents of P after execution of $\underset{\sim}{P}$ when $\bar{X}_m$ initially contain $\bar{x}_m$, and all other members of Reg $(\underset{\sim}{P})$ initially contain zero.

For example, if $\underset{\sim}{P}$ is the program of Example (1.3), then $(\underset{\sim}{P},X,Y,X)$ computes $x \dot{-} y$; $(\underset{\sim}{P},X,Y,X)$ computes the projection $p_{22}(x,y) = y$.

(4.2) Definition. $\mathfrak{L}_\alpha$ for $0 \leq \alpha < \omega^\omega$ is the set of functions computable by programs in L_α with input and output

$$\mathfrak{L} = \bigcup_{\alpha < \omega^\omega} \mathfrak{L}_\alpha.$$

Obviously, if $\alpha > \beta$ then $\mathfrak{L}_\alpha \supseteq \mathfrak{L}_\beta$ by (1.1.ii) of the definition of L_α. It is the task of this section to prove that if $\alpha > \beta$ the containment $\mathfrak{L}_\alpha \supsetneq \mathfrak{L}_\beta$ is proper.

(4.3) Definition. Let $\underset{\sim}{F}_0$ be the program

$$X = X + 1$$
$$X = X + 1$$

and if β is the least ordinal so $\alpha = \beta + \omega^n$, let $\underset{\sim}{F}_\alpha$ be the program

```
LOOP(n+1)  X
  F_β
END
```

It is immediate that $\underset{\sim}{F}_\alpha \in L_\alpha$ by Definition (1.1).

(4.4) Lemma. Let $\hat{f}_\alpha$ be the function computed by $(\underset{\sim}{F}_\alpha, X, X)$. Then if $x > 0$, $\hat{f}_\alpha(x) \geq f_\alpha(x)$. Also, $\hat{f}_\alpha \in \mathcal{E}_\alpha$.

Proof. $\hat{f}_0(x) = x+2 \geq f_0(x)$ for all x by definition. Say that $\alpha = \beta + 1$; then $\underset{\sim}{F}_\alpha$ is

```
LOOP(1)  X
  F_β
END
```

which is equivalent, when $x > 0$ is in X initially, to

$$\left.\begin{array}{l} \underset{\sim}{F}_\beta \\ \underset{\sim}{F}_\beta \\ \vdots \\ \underset{\sim}{F}_\beta \end{array}\right\} x$$

So $\hat{f}_\alpha(x) = \hat{f}_\beta^{(x)}(x) \geq f_\beta \hat{f}_\beta^{(x-1)}(x) \geq \cdots \geq f_\beta^{(x-1)} \hat{f}_\beta(x) \geq f_\beta^{(x)}(x)$ $\geq f_\beta^{(x)}(1) = f_\alpha(x)$ if $x > 0$.

Now if β is the least ordinal for which $\alpha = \beta + \omega^{n+1}$ and if $x > 0$ is in X, then $\underset{\sim}{F}_\alpha$ is equivalent to

```
LOOP(n+1)  X   }
  :            } x
LOOP(n+1)  X   }
  F_β
END            }
  :            } x
END            }
```

But this is exactly the program $\underset{\sim}{F}_{\beta+\omega^n x}$. So if $x > 0$, $\hat{f}_\alpha(x) = \hat{f}_{\beta+\omega^n x}(x) \geq f_{\beta+\omega^n x}(x) = f_\alpha(x)$; this concludes the proof of (4.4).

(4.5) Theorem. For $\alpha \geq 1$, $f_\alpha \in \mathcal{L}_\alpha$.

Proof. $f_1(x) = 2x + (1 \dot{-} x)$ is in $\mathcal{L}_1$ via the program $\underset{\sim}{F}$ =

```
LOOP(1)    X
  G = G+1
  G = G+1
END
F = F+1
LOOP(1)    X
  F = G
END
```

where $(\underset{\sim}{F},X,F)$ computes f_1. For $\alpha \geq 2$, we defer the proof until Chapter IV. The only facts we will need for the remainder of this chapter are given by Lemma (4.4). It is possible to construct a program for f_α in L_α, but a surprising amount of labor is involved.

(4.6) Lemma. If $\alpha > \beta$, then for any constant c, $f_\alpha(x) > f_\beta^{(c)}(x)$ for almost all x.

Proof. If $\alpha > \beta$, then $\alpha \geq \beta + 1$. First we establish the result for $f_{\beta+1}$ and f_β.

$f_{\beta+1}(x) = f_\beta^{(x)}(1)$; for $x \geq c$, $f_{\beta+1}(x) = f_\beta^{(c)} f_\beta^{(x-c)}(1)$.

But for large x, $f_\beta^{(x-c)}(1) = f_{\beta+1}(x - c) > x$ by (3.4.ii) and (3.4.viii). Thus $f_{\beta+1}(x) > f_\beta^{(c)}(x)$ for large x.

But now if $\alpha \geq \beta + 1$, for large x $f_\alpha(x) \geq f_{\beta+1}(x)$ by (3.4.viii); this yields the lemma.

(4.7) Hierarchy Theorem. If $\alpha > \beta$, $\mathfrak{L}_\alpha \supsetneq \mathfrak{L}_\beta$.

Proof. As remarked above, if $\alpha > \beta$, $\mathfrak{L}_\alpha \supseteq \mathfrak{L}_\beta$ by definition. If $\mathfrak{L}_\alpha = \mathfrak{L}_\beta$, the function $\hat{f}_\alpha$ of Lemma (4.4) would be a member of $\mathfrak{L}_\beta$; but for all c, $\hat{f}_\alpha(x) \geq f_\alpha(x) > f_\beta^{(c)}(x)$ for almost all x by (4.6) and (4.4). Then by (3.7), $\hat{f}_\alpha \notin \mathfrak{L}_\beta$. This proves (4.7).

The Bounding Theorem (3.6) and the Hierarchy Theorem (4.7) together provide the rigorous justification for the claim that the simple measure of the complexity of syntactic structure of a Loop program by Definition (1.1) is also an adequate measure of the power of the program; for the Bounding Theorem implies a maximal complexity on the functions of $\mathfrak{L}_\alpha$ by bounding the number of steps the computation of each such function can possibly consume. The Hierarchy Theorem yields a minimal complexity for $\mathfrak{L}_\alpha$ by exhibiting $\mathfrak{L}_\alpha$ functions which cannot be computed in fewer steps than the number implied by the structure of their programs.

It is convenient to introduce at this point a property of the classes $\mathcal{L}_\alpha$ which follows almost immediately from its definition.

(4.8) Definition. The operations of substitution consist of the following methods of obtaining a function f from given functions g, h:

(i) Substituting a constant: obtaining f from g where $f(\bar{x}_n) = g(\bar{x}_n, c)$ for some number c;

(ii) Permuting and identifying variables: obtaining f from g where $f(\bar{x}_n) = g(\xi_1, \ldots, \xi_m)$ and each ξ_i, $1 \leq i \leq m$, is one of the x_i;

(iii) Composition: obtaining f from g, h where $f(\bar{x}_n) = g(\bar{x}_n, h(\bar{x}_n))$.

Also, if $\mathcal{C}$ is a class of functions, $\mathcal{C}$ is closed under substitution whenever any function f obtained from functions in $\mathcal{C}$ by substitution is also a member of $\mathcal{C}$.

(4.9) Theorem. For all $\alpha < \omega^\omega$, $\mathcal{L}_\alpha$ is closed under substitution.

Proof. Say $(\underset{\sim}{G}, \bar{X}_n, H, G)$ computes g, $(\underset{\sim}{H}, \bar{X}_n, H)$ computes h, and $f(\bar{x}_n) = g(\bar{x}_n, h(\bar{x}_n))$. We assume that $\underset{\sim}{\text{Reg}}\ (\underset{\sim}{G}) \cap \underset{\sim}{\text{Reg}}\ (\underset{\sim}{H}) = \{\bar{X}_n, H\}$ and that neither $\underset{\sim}{G}$ nor $\underset{\sim}{H}$ uses registers $\bar{Z}_n$. These conditions can of course be brought about by changes in names of the registers used by $\underset{\sim}{G}$ and $\underset{\sim}{H}$. Let $\underset{\sim}{F}$ be the program

$$
\begin{array}{l}
Z_1 = X_1 \\
\quad \vdots \\
Z_n = X_n \\
\quad \underset{\sim}{H} \\
X_1 = Z_1 \\
\quad \vdots \\
X_n = Z_n \\
\quad \underset{\sim}{G}
\end{array}
$$

Then $(\underset{\sim}{F},\bar{X}_n,G)$ computes f. This proves that $\mathfrak{L}_\alpha$ is closed under composition; proofs for the other possibilities, substitution of a constant and permutation and identification of variables, are entirely analogous and are omitted.

§5. The preceding section showed that $\mathfrak{L}_\alpha$ contains some very large functions -- in fact, functions larger than any in $\mathfrak{L}_\beta$ if $\beta < \alpha$ -- but it is not yet at all clear that Loop programs can do anything much but run for a long time and eventually halt with rather large numbers in the registers. This section will demonstrate that even L_2 programs can perform quite complicated operations, and will lay the groundwork for showing among other things that each $\mathfrak{L}_\alpha$ contains very small functions more complicated than any functions in $\mathfrak{L}_\beta$ if $\beta < \alpha$.

In particular, (5.1) shows how to construct L_2 programs which simulate Turing machines; (5.2) shows how to construct Turing machines which simulate Loop programs. Theorem (5.1) is useful in relating Loop programs to other formalisms for computation, as is done in Chapter IV. Combining (5.1) with (5.2) yields Loop programs which simulate other Loop programs; §6 leans heavily on this possibility.

We assume that the reader is familiar with the elementary capabilities of Turing machines as discussed, for example, in Kleene [K] or Davis [D]. Our theorems would be true using any of the various formalisms for Turing machines; for definiteness, we give an informal definition of computation by Turing machine much like that of [K].

A Turing machine $\mathfrak{M}$ is determined by a finite set $Q_{\mathfrak{M}}$ of quintuples $\{(q_i, s_j, s_k, d, q_\ell)\}$, where d is either "L" or "R", and such

that no two quintuples of $Q_{\mathfrak{M}}$ have the same first two components. The first and last components of the quintuples of $Q_{\mathfrak{M}}$ comprise the states of $\mathfrak{M}$; the second and third components comprise the symbols of $\mathfrak{M}$. One of the states, q_0, is distinguished as the initial state, and one of the symbols, s_0, is called "blank" and is also written "B". Associated with the Turing machine is a tape, which consists of a two-way infinite sequence of squares; each square has printed on it one of the symbols of $\mathfrak{M}$. If the symbol printed on a square is s_0, the square is blank, and at any time almost all of the squares on a tape are blank.

One square on the tape is scanned by $\mathfrak{M}$. A situation consists of a particular printing of the squares of the tape, a particular square on the tape (the scanned square) and a particular state; the machine is in that state.

Given a situation, $\mathfrak{M}$ may perform a step as follows: if the machine is in state q_i and the symbol on the scanned square is s_j, and if $(q_i, s_j, s_k, d, q_\ell) \in Q_{\mathfrak{M}}$, then the symbol on the scanned square is replaced by s_k, the scanned square moves one square to the left or right according as d is "L" or "R", and the machine goes into state q_ℓ. If no quintuple of $Q_{\mathfrak{M}}$ begins with q_i, s_j then no act is performed and the machine has halted; in this case the situation is terminal.

The Turing machine is used by choosing some situation in which to start it; the machine then successively performs steps until it halts, and the contents of the tape in the terminal situation determine

the output. Specifically, let s_1 be the symbol "1". Represent the natural numbers 0, 1, 2,... by "1", "11", "111",..., so that in the representation of x there are x + 1 occurrences of "1". Also, represent an n-tuple $x_1,\ldots,x_n$ by juxtaposing the representations of the x_i separated by "B" so that the representation of (0,2,1,3), for example, is "1B111B11B1111".

A Turing machine computes the (partial) function $f: N^n \to N$ if when the Turing machine is started in state q_0 with the representation of $\bar{x}_n$ on its tape, which is blank otherwise, and with the square just to the right of the representation of $\bar{x}_n$ the scanned square, then the Turing machine eventually halts with a total of $f(\bar{x}_n)$ "1"s to the left of the scanned square in the terminal situation, providing $f(\bar{x}_n)$ is defined. If $f(\bar{x}_n)$ is not defined, the Turing machine does not halt.

For example, if a Turing machine computes x + y, when started in the situation

$$\begin{array}{l} \ldots\ \mathrm{B111B1111B}\ \ldots \\ \qquad\qquad\ \uparrow \\ \qquad\qquad\ q_0 \end{array}$$

it may halt in the situation

$$\begin{array}{l} \ldots\ \mathrm{B1B111B1BB111B}\ \ldots \\ \qquad\qquad\qquad\ \uparrow \\ \qquad\qquad\qquad\ q_i \end{array}$$

where no quintuple starts with q_i, B. The notation for situations should be obvious.

(5.1) Theorem. Let $\mathfrak{M}$ be a Turing machine which computes the function $f: N^n \to N$. Then there is a Loop program with input and output $(\underset{\sim}{TM}_{\mathfrak{M}}, \bar{X}_n, S, P)$ where $\underset{\sim}{TM}_{\mathfrak{M}} \in L_2$ which computes a function $TM_{\mathfrak{M}}: N^{n+1} \to N$ with the following property: if s exceeds the number of steps required to compute $f(\bar{x}_n)$ using $\mathfrak{M}$, then $f(\bar{x}_n) = TM_{\mathfrak{M}}(\bar{x}_n, s)$.

Proof. For simplicity, the theorem will be proved only for the case n = 1, and for $\mathfrak{M}$ a 2-symbol machine with symbols {B,1}. Exactly the same methods apply when n and the number of symbols of $\mathfrak{M}$ are unrestricted.

The heart of the construction is an L_1 program $\underset{\sim}{Step}_{\mathfrak{M}}$ which in effect carries out a single step in the Turing machine computation. $\underset{\sim}{Step}_{\mathfrak{M}}$ uses several main registers Q, T_L, T_S, T_R which contain respectively the number of the current state, and representations of the tape to the left of, on, and to the right of the scanned square. Suppose the non-blank portion of the tape is

$$\cdots B\ S_{-u}\ S_{-u+1} \cdots S_{-1}\ S_0\ S_1 \cdots S_{v-1}\ S_v\ B \cdots$$

where each S_i is "B" or "1" and S_0 is the scanned square. Then T_L contains

$$t_{-u} \cdot 2^{u-1} + t_{-u+1} \cdot 2^{u-2} + \cdots + t_{-1} \cdot 2^0$$

where each t_i is 0 or 1 according as S_i is "B" or "1". Likewise T_S contains t_0 and T_R contains

$$t_v \cdot 2^{v-1} + t_{v-1} \cdot 2^{v-2} + \cdots + t_1 \cdot 2^0$$

That is, T_L, T_S, T_R contain numbers whose binary representations are images of the corresponding portions of the tape.

Also suppose register Q contains a number q, where $0 \leq q \leq m$ and $\mathfrak{M}$ has $m+1$ states $\{q_0,\ldots,q_m\}$. Consider the program Decode$_{\mathfrak{M}}$

$C_{00} = 0$
$C_{01} = 0$
$C_{10} = 0$
$C_{11} = 0$
$\vdots$
$C_{m0} = 0$
$C_{m1} = 0$
$C_{00} = C_{00} + 1$
LOOP(1) Q
$C_{m0} = C_{m-1,0}$
$C_{m-1,0} = C_{m-2,0}$
$\vdots$
$C_{10} = C_{00}$
END

LOOP(1) T_S
$C_{01} = C_{00}$
$C_{00} = 0$
$C_{11} = C_{10}$
$C_{10} = 0$
$\vdots$
$C_{m1} = C_{m0}$
$C_{m0} = 0$
END

It is easy to see that if Q contains i and T_S contains j, then $C_{ij} = 1$, but $C_{k\ell} = 0$ for $i \neq k$ or $j \neq \ell$.

Now let the quintuples of $\mathfrak{M}$ be $\{m_1,\ldots,m_r\}$. Let $\underset{\sim}{\text{Quints}}_{\mathfrak{M}}$ be the program

```
Decode_M
L = 0
R = 0
M1
M2
 .
 .
 .
Mr
```

Here if m_i is the quintuple (q_i,s_j,s_k,d,q_ℓ) and d is "L" then $\underset{\sim}{M}_i$ is the program

```
LOOP(1)  C_ij
  T_S = s_k
  L   = 1
  Q   = ℓ
END
```

If d is "R", $\underset{\sim}{M}_i$ is the program

```
LOOP(1)  C_ij
  T_S = s_k
  R   = 1
  Q   = ℓ
END
```

Here we use the obvious abbreviation "$T_S = s_k$" for

$$T_S = 0$$

if s_k = "B", and

$$T_S = 0$$
$$T_S = T_S + 1$$

if s_k = "1". Likewise, "Q = ℓ" is an abbreviation for

$$\left.\begin{array}{l} Q = 0 \\ Q = Q+1 \\ \vdots \\ Q = Q+1 \end{array}\right\} \ell$$

Thus if the number of a state is in register Q and the contents of the scanned square are in register T_S, $\underset{\sim}{\text{Quints}}_{\mathfrak{M}}$ causes the next state to appear in Q and the new symbol for the scanned square to be placed in T_S. $\underset{\sim}{\text{Quints}}_{\mathfrak{M}}$ sets registers L and R so that L = 0 and R = 1 if a rightward move is to be made, while L = 1 and R = 0 if a leftward move is to be made. If the situation is terminal, Q and T_S remain unchanged and L = R = 0.

Given the interpretation above for the numbers in T_L, T_S, T_R, the effect of a rightward move of $\mathfrak{M}$ can be reflected by replacing T_L by $2 \cdot T_L + T_S$, replacing T_S by $rm(T_R,2)$ and replacing T_R by $T_R/2$. Here we use "T_L", for example, to refer both to the register and its contents. Also, rm(x,y) is the remainder upon division of x by y, and x/y is the integral quotient of x and y: the greatest integer z so $z \cdot y \leq x$. Arbitrarily, we set x/0 = 0.

These functions can be carried out in L_1. Consider the following program $\underset{\sim}{\text{RM}}$ ("rightward move").

```
T_LR = 0
LOOP(1)  T_L
  T_LR = T_LR + 1
  T_LR = T_LR + 1
END
```
$\Big\}$ $T_{LR} \leftarrow 2 \cdot T_L$

```
LOOP(1)  T_S
  T_LR = T_LR + 1
END
```
$\Big\}$ $T_{LR} \leftarrow 2 \cdot T_L + T_S$

```
T'_SR = 0
T'_SR = T'_SR + 1
T_SR = 0
LOOP(1)  T_R
  T  = T_SR
  T_SR = T'_SR
  T'_SR = T
END
```
$\Big\}$ $T_{SR} \leftarrow rm(T_R,2)$

```
T_RR = 0
T'_RR = 0
LOOP(1)  T_R
  T_RR = T_RR + 1
  T = R_RR
  T_RR = T'_RR
  T'_RR = T
END
```
$\Big\}$ $T_{RR} \leftarrow T_R/2$

RM places $2 \cdot T_L + T_S$ in T_{LR}, $rm(T_R,2)$ in T_{SR}, and $T_R/2$ in T_{RR} but does not change T_L, T_S, T_R. Of course there is a corresponding program LM which puts $2 \cdot T_R + T_S$ in T_{RL}, $rm(T_L,2)$ in T_{SL} and $T_L/2$ in T_{LL} without changing T_L, T_S, T_R and which thus simulates a leftward move.

Now let $\underset{\sim}{\text{Step}}_{\mathfrak{M}}$ be

```
Quints_M
LM
RM
LOOP(1)  L
  T_L = T_LL
  T_S = T_SL
  T_R = T_RL
END
LOOP(1)  R
  T_L = T_LR
  T_S = T_SR
  T_R = T_RR
END
```

$\underset{\sim}{\text{Step}}_{\mathfrak{M}}$ is an L_1 program; given the number of a state in Q and a tape configuration in T_L, T_S, T_R, execution of $\underset{\sim}{\text{Step}}_{\mathfrak{M}}$ leaves the next state in Q and the next tape configuration in T_L, T_S, T_R. But then if an initial situation is in Q, T_L, T_S, T_R, the L_2 program $\underset{\sim}{\text{Result}}_{\mathfrak{M}}$ =

```
LOOP(1)  S
  Step_M
END
```

leaves in T_L, T_S, T_R a representation of the tape configuration after s steps of M, where s is the number in S; if s exceeds the number of steps required for M to halt, the final tape is left in T_L, T_S, T_R. Thus the only remaining tasks are to find a program which, when given

an input number, produces the corresponding initial situation, and to find a program which, given a tape situation, yields an output number from the final tape representation.

According to the formalism agreed upon above, if the input number is x, the tape representation is a string of x+1 "1"s just to the left of the scanned square; in other words, we want T_L to contain $2^{x+1} - 1$ and T_S, T_R to contain zero. The job is done by the L_2 program Input:

```
Q = 0
T_L = 0
T_S = 0
T_R = 0
X  = X+1
LOOP(1) X
  LOOP(1)  T_L
    T_L  = T_L+1
  END
  T_L = T_L+1
END
```

} $T_L \leftarrow 2^{x+1} - 1$

Next, the output number is to be the total number of "1"s occurring in the binary expansion of T_L. The L_2 program Output =

```
P = 0
LOOP(1)  T_L
  T ← rm(T_L,2)
  T_L ← T_L/2
  LOOP(1)  T
    P = P+1
  END
END
```

} $P \leftarrow P+T$

leaves in P the correct number. We have used, for example, "$T \leftarrow rm(T_L,2)$" as an abbreviation for a program which puts $rm(T_L,2)$ into T without destroying the constants of T_L. The necessary programs appear as part of the program RM above.

Finally, let $\underset{\sim}{TM}_{\mathfrak{M}}$ be the L_2 program

Input
Result$_{\mathfrak{M}}$
Output

Then $(\underset{\sim}{TM}_{\mathfrak{M}},X,S,P)$ computes $TM_{\mathfrak{M}}$ with the properties required, and Theorem (5.1) is proved.

(5.2) Theorem. For each $n > 0$ there is a Turing machine $\mathfrak{LP}_n$ which computes a function $LP_n: N^{n+1} \to N$ with the following property: if $(\underset{\sim}{P},\bar{X}_n,P)$ is a Loop program with input and output which computes $f: N^n \to N$, then there is a number e so that $LP_n(e,\bar{x}_n) = f(\bar{x}_n)$. Furthermore, if $T_{\underset{\sim}{P}}$ is the running time function of $\underset{\sim}{P}$, then there is a constant c so that the total number of tape squares ever scanned in the computation of $LP_n(e,\bar{x}_n)$ is no more than $c\cdot(e+\max\{\bar{x}_n\}+T_{\underset{\sim}{P}}(\bar{x}_n))^3$.

Proof. We will not actually construct $\mathfrak{LP}_n$, but we will give enough details so that it should be clear to anyone with some familiarity with computation by Turing machines that $\mathfrak{LP}_n$ exists. Actually, the first part of the theorem is immediate from the intuitive computability of functions defined by Loop programs.

For each $(\underset{\sim}{P},\bar{X}_n,P)$ there must be an e so if $(\underset{\sim}{P},\bar{X}_n,P)$ computes f, then $LP_n(e,\bar{x}_n) = f(\bar{x}_n)$. Thus e should somehow encode $(\underset{\sim}{P},\bar{X}_n,P)$. When this is the case, it is usual to say that e is a Gödel number of $(\underset{\sim}{P},\bar{X}_n,P)$.

The encoding can be done in a variety of ways; the one suggested here is particularly simple. First, we may as well assume that Reg $(\underset{\sim}{P}) = \{\bar{X}_r\}$, that the input registers are $\bar{X}_n$, and that the output register is X_1, since clearly for any Loop program with n input registers and an output register, there is another program in the desired form. (The new program is obtained merely by making the proper changes in the names of the registers and possibly adding an instruction "$X_1 = X_k$" to put the answer into X_1.) So we need only consider programs like $(\underset{\sim}{P},\bar{X}_n,X_1)$ where Reg $(\underset{\sim}{P}) = \{\bar{X}_r\}$. Now, using an eight-symbol alphabet:

L E = X 1 / + 0

rewrite $\underset{\sim}{P}$ by placing "/" between the instructions, by changing "LOOP(n)" to "L1 ... 1", that is, to "L" followed by n "1"s, by changing "X_k" to "X1 ... 1", that is, to "X" followed by k "1"s, and by changing "END" to "E". Thus the program $\underset{\sim}{P}$ =

```
X2 = 0
LOOP(1)  X1
  X2 = X2 + 1
END
X = X2
```

would become

$$X11 = 0/L1X1/X11 = X11+ 1/E/X1 = X11$$

Since 8 different symbols can appear in this representation, the representation of any program $\underset{\sim}{P}$ can be interpreted as a base 9 number; take "L" to have digit value 1, "E" to have value 2,..., "0" to have value 8. We will let the blank "B" have digit value 0. Thus given any program $\underset{\sim}{P}$ there is a unique number e associated with it, and if e is written in a base 9 notation $\underset{\sim}{P}$ is recoverable immediately. On the other hand, not every number e has a corresponding program; for example, all those numbers which contain significant zeroes in their base 9 expansion.

Now we proceed to describe the operation of $\mathcal{UP}_n$ Recall that $\mathcal{UP}_n$ is given an $(n+1)$-tuple $(e,\bar{x}_n)$ consisting of $e+1$ occurrences of "1", followed by "B", followed by x_1+1 occurrences of "1",..., followed by "B", followed by x_n+1 "1"s. We write this initial tape as

$$\ldots B \underline{e} B \underline{x}_1 B \ldots B \underline{x}_n B \ldots$$

where the underlined letters $\underline{x}$ represent a string of $x+1$ "1"s. $\mathcal{UP}_n$ performs as follows: first go to the representation of e and rewrite e as its base 9 representation (which, as explained above, is an image of $\underset{\sim}{P}$). Call this sequence of symbols $\hat{e}$. Of course, the length of $\hat{e}$ is no longer than the length of $\underline{e}$; in fact the replacement can be done using no more tape than is consumed by $\underline{e}$ itself. The tape thus becomes

$$\ldots\ \hat{e}\ B\ \underline{x}_1\ B\ \ldots\ B\ \underline{\bar{x}}_n\ B\ \ldots$$

Then $\mathfrak{LP}_n$ checks $\hat{e}$ to make sure it represents a permissible Loop program; the checking consists of examining each instruction to make sure it is a legal instruction, and verifying that LOOPs and ENDs are nested properly. If $\hat{e}$ does not represent a syntactically correct Loop program, $\mathfrak{LP}_n$ erases its whole tape and stops. Thus, in effect, every number e will be associated with some function; those numbers e which cannot be associated with a syntactically correct Loop program will all represent the function which is identically zero.

If on the other hand $\hat{e}$ represents a syntactically correct Loop program, $\mathfrak{LP}_n$ examines $\hat{e}$ to determine the registers $\bar{X}_r$ it uses, and then changes the tape to

$$\ldots\ B\ \hat{e}\ B\ \underline{x}_1\ B\ \ldots\ B\ \underline{x}_n\ B\ \underline{0}\ B\ \ldots\ B\ \underline{0}\ B\ \ldots$$

which represents $\hat{e}$ followed by the r-tuple $(\bar{x}_n,0,\ldots,0)$; that is, the initial contents of $\bar{X}_r$ since $X_{n+1},\ldots,X_r$ are zero. Continuing, $\mathfrak{LP}_n$ produces the tape

$$\underset{1}{\ldots\ B\ \hat{e}\ B}\Big|\underset{2}{\underline{x}_1\ B\ \ldots\ B\ \underline{x}_n\ B\ \underline{0}\ B\ \ldots\ B\ 0\ B}\Big|\underset{3}{\underline{1}\ B}\Big|\underset{4}{\underline{0}\ B}\Big|\underset{5}{B\ \ldots}$$

which, for convenient reference, we have divided into five regions. Region 1 contains $\hat{e}$, which represents the program $\underset{\sim}{P}$ being simulated; regions 2-5 together represent the initial state $(\bar{x}_r,1,0,(0))$ of $\underset{\sim}{P}$.

$\mathfrak{LP}_n$ is then ready to begin simulating $\underset{\sim}{P}$. In general, just before beginning a step in the simulation, $\mathfrak{LP}_n$ will have on its tape the following sequence of symbols, if the current state is $(\bar{x}_r, i, \ell, p)$.

$$\ldots B\hat{e} \,|\, B\underline{x}_1 B \ldots B\underline{x}_r \,|\, B\underline{i} \,|\, B\underline{\ell} \,|\, BBa_{11}Ba_{21}B\ldots Ba_{k_1 1}BB\ldots BBa_{1j}Ba_{2j}B\ldots Ba_{k_j j}BB\ldots$$
$$\quad 1 \qquad\quad 2 \qquad\quad 3 \quad 4 \qquad\quad 5$$

The representation of the state used in region 2-4 is obvious. The contents of region 5, which represents the pushdown store, are interpreted as follows. The object at the top of the pushdown store is $(a_{11}, a_{21}, \ldots, a_{k_1 1})$. More generally, the object at the m-th level of the pushdown store is $(a_{1m}, a_{2m}, \ldots, a_{k_m m})$. Tuples on the pushdown store are separated by double blanks, and members of a tuple are separated by single blanks.

What is the length of this representation of a state? The length of region 1 is no more than e + 1. Suppose the simulation has run for s steps, and let $\underline{m}$ be an abbreviation for $\max\{\bar{x}_n\}$, that is, the largest number initially in the registers. Then each of the x_i in region 2 is no more than $\underline{m} + s$. So the length of region 2 is no more than $r \cdot (\underline{m} + s + 2)$. But according to the encoding we have chosen, r is certainly less than e. So region 2 has a length of less than $e \cdot (\underline{m} + s + 2)$. Again, the number i represented in region 3 corresponds to the instruction about to be executed, which is a number certainly less than e, so region 3 has a length less than e + 2 squares. The ℓ of region 4 is either 0 or 1, and so the length of region 4 is 3 at most.

Whenever a tuple is placed on the pushdown store, all its members are bounded by the largest number in any register. Since nothing on the pushdown store ever becomes greater than the largest register, any single number anywhere in the pushdown store is bounded by $\underline{m}+s$. The largest tuple on the pushdown store has at most e components, since the number of components is a function only of $\underset{\sim}{P}$; the depth (number of tuples) of the pushdown store cannot exceed s, the number of instruction executions taken so far. Therefore region 5 has a length bounded by $s\cdot(1+e\cdot(\underline{m}+s+2))$.

Each of regions 1-5 has its length bounded by a polynomial of degree at most 3 in s, e, and $\underline{m}$. Therefore, the sum of the lengths of regions 1-5 is bounded by $c\cdot(e+\underline{m}+s)^3$ for some constant c.

The discussion so far has been fairly rigorous, except for the claim that the string of "1"s representing e could be turned into the string $\hat{e}$. The main portion of the construction whose details we will omit is that of showing that $\mathcal{TP}_n$ can transform the representation of a state as specified above into the representation of the next state, without using any tape squares other than those already used. We leave to the reader the task of convincing himself that this is possible, with the reminder that $\mathcal{TP}_n$ may use a large number of extra symbols to mark tape squares in which it has a special interest at some moment. We may also remark that all the theorems in the sequel which use this theorem would be unaffected if the polynomial bound $c\cdot(e+\underline{m}+s)^3$ were replaced by any exponential in e, $\underline{m}$ and s; and finally that the encoding we have chosen is actually rather inefficient

and that by using a binary encoding of the numbers making up a state, the present theorem would remain true with a bound on tape consumption of $d \cdot \log_2(1 + e + \underline{m} + s)$ for some constant d.

Granting that $\mathcal{LP}_n$ is able to replace the representation of a state by the representation of the next state without using more tape than is consumed by the states themselves, the theorem follows immediately. For $\mathcal{LP}_n$ simply keeps simulating $\underset{\sim}{P}$ until a final state is reached, then erases all of the tape but the portion containing x_1 and halts on the rightmost square of the representation of x_1. Thus $\mathcal{LP}_n$ has computed $(P, \bar{X}_n, X_1)$; and since the program runs for $T_{\underset{\sim}{P}}(\bar{x}_n)$ steps by definition, the total tape consumption is bounded by $c \cdot (e + \underline{m} + T_{\underset{\sim}{P}}(\bar{x}_n))^3 = c \cdot (e + \max\{\bar{x}_n\} + T_{\underset{\sim}{P}}(\bar{x}_n))^3$. This concludes the proof of (5.2).

(5.3) Theorem. For each $n > 0$ there is an $\mathcal{L}_2$ function M_n: $N^{n+2} \to N$ so that for any Loop program $(\underset{\sim}{P}, \bar{X}_n, P)$ which computes $f: N^n \to N$, there is an e such that $M_n(e, \bar{x}_n, s) = f(\bar{x}_n)$ provided $s \geq T_{\underset{\sim}{P}}(\bar{x}_n)$.

Proof. By (5.2) there is a Gödel number e for $(\underset{\sim}{P}, \bar{X}_n, P)$ so that $LP_n(e, \bar{x}_n) = f(\bar{x}_n)$, and LP_n is computable by a Turing machine $\mathcal{LP}_n$ whose total consumption of tape is no more than $c \cdot (e + \max\{\bar{x}_n\} + T_{\underset{\sim}{P}}(\bar{x}_n))^3$ squares. For brevity let this number of squares be t. Now say $\mathcal{LP}_n$ has q states and uses k symbols. Then the total number of distinct tapes appearing in the computation is no more than k^t, since each

tape square can have printed on it one of the k symbols. At each situation occurring in the computation the Turing machine is scanning one of the at most t squares, and is in one of the q states; therefore at most $q \cdot t \cdot k^t$ different situations can arise in the computation. If one of these situations is ever repeated, the whole computation must be caught in an endless loop; but this does not happen, so the Turing machine must halt within $q \cdot t \cdot k^t$ steps, that is, within a number of steps

$$q \cdot c \cdot (e + \max\{\bar{x}_n\} + T_{\underset{\sim}{P}}(\bar{x}_n))^3 \cdot k^{c \cdot (e + \max\{\bar{x}_n\} + T_{\underset{\sim}{P}}(\bar{x}_n))^3} = B(e, \bar{x}_n, T_{\underset{\sim}{P}}(\bar{x}_n))$$

Remembering that q, c, and k are fixed numbers, it is easy enough to show that B is actually a member of $\mathcal{L}_2$. Alternatively, it is easy to show

$$B(e, \bar{x}_n, T_{\underset{\sim}{P}}(\bar{x}_n)) \leq 2^{2^{(e + x_1 + \cdots + x_n + T_{\underset{\sim}{P}}(\bar{x}_n))}}$$

for large enough arguments. Since $f_2(x) = 2^x$ and $\hat{f}_2(x+1) \geq f_2(x)$, there is a constant b so

$$B(e, \bar{x}_n, T_{\underset{\sim}{P}}(\bar{x}_n)) \leq \hat{f}_2^{(2)}(e + x_1 + \cdots + x_n + T_{\underset{\sim}{P}}(\bar{x}_n) + b) = B'(e, \bar{x}_n, T_{\underset{\sim}{P}}(\bar{x}_n))$$

But B' is a member of $\mathcal{L}_2$ since it is obtained by substitution from members of $\mathcal{L}_2$. The function $x + y$, for example, is in $\mathcal{L}_1$ via the program $\underset{\sim}{A}$ =

```
LOOP(1)  X
  Y = Y+1
END
```

where $(\underset{\sim}{A},X,Y,Y)$ computes $x+y$.

Recall that the Turing machine $\mathcal{LP}_n$ of (5.2) is a particular, fixed machine. Apply (5.1) to this machine to get an $\mathcal{L}_2$ function $TM_{\mathcal{LP}_n}$ so that if z exceeds the number of steps required for $\mathcal{LP}_n$ to halt,

$$TM_{\mathcal{LP}_n}(e,\bar{x}_n;z) = LP_n(e,\bar{x}_n)$$

Then take $M_n(e,\bar{x}_n,s) = TM_{\mathcal{LP}_n}(e,\bar{x}_n,B'(e,\bar{x}_n,s))$. By the fact that B' is increasing, the proof of (5.3) is complete.

§6. All the investment in labor of §§2-5 now begins to pay off. We have several easy theorems which characterize the classes $\mathcal{L}_\alpha$ for $\alpha \geq 2$ in three ways, and which show each class $\mathcal{L}_\alpha$ for $\alpha \geq 2$ has two important closure properties. Finally, $\mathcal{L}_{\alpha+1}$ has a <u>universal function</u> for $\mathcal{L}_\alpha$, and $\mathcal{L}_{\alpha+1}$ has a very small function not in $\mathcal{L}_\alpha$.

(6.1) Theorem. For $\alpha \geq 2$, a function $f: N^n \to N$ is in $\mathcal{L}_\alpha$ if and only if there is a program $(P, \bar{X}_n, P)$ which computes f such that $T_{\underset{\sim}{P}}$ is bounded by $f_\alpha^{(p)}$ for some number p.

Proof. The "only if" part is simply the Bounding Theorem (3.6). Conversely, if $T_{\underset{\sim}{P}}(\bar{x}_n) \leq f_\alpha^{(p)}(\max\{\bar{x}_n\})$, then $T_{\underset{\sim}{P}}(\bar{x}_n) \leq \hat{f}_\alpha^{(p)}(x_1 + \cdots + x_n + 1)$. This latter function is in $\mathcal{L}_\alpha$. Then by (5.3) there is an e so $f(\bar{x}_n) = M_n(e, \bar{x}_n, \hat{f}_\alpha^{(p)}(x_1 + \cdots + x_n + 1))$. Since $M_n \in \mathcal{L}_2$, by substitution $f \in \mathcal{L}_\alpha$ for $\alpha \geq 2$.

This theorem is interesting because it shows that if we have any program $\underset{\sim}{P}$ which computes f, no matter how deeply the loops of $\underset{\sim}{P}$ are nested, so long as the running time of $\underset{\sim}{P}$ is bounded by $f_\alpha^{(p)}$ then $\underset{\sim}{P}$ can be rewritten as an L_α program.

(6.2) Theorem. For $\alpha \geq 2$, $\mathcal{L}_\alpha$ is the class of functions which are computable by a Turing machine where either the running time of the Turing machine or its consumption of tape is bounded by $f_\alpha^{(p)}$ for some number p.

Proof. Immediate by (5.1), (5.3), and the argument of (6.1).

Theorems (6.1) and (6.2) provide further evidence for our basic claim that the complexity of a function can be measured by the ordinal assigned to its Loop program. In particular, (6.2) assures us that the hierarchy of sets $\mathcal{L}_\alpha$ does not arise because of some peculiarity in the definition of Loop program, but that in fact if some function f is in $\mathcal{L}_\alpha$ but not in $\mathcal{L}_\beta$ (where $\alpha > \beta$) then f is more difficult to compute than any function in $\mathcal{L}_\beta$ even if the computation is done by the familiar Turing machine.

(6.3) Theorem. The n-argument functions of $\mathcal{L}_\alpha$ are precisely the functions expressible in the form

$$f(\bar{x}_n) = M_n(e,\bar{x}_n,f_\alpha^{(p)}(\max\{\bar{x}_n\}))$$

for some numbers e, p, and where M_n is a particular function in $\mathcal{L}_2$.

Proof. That each f is expressible in the required way is an immediate consequence of (6.1) and the Bounding Theorem (3.6). The converse follows from Theorem (4.5) and the closure of $\mathcal{L}_\alpha$ under substitution.

Theorem (6.3) characterizes $\mathcal{L}_\alpha$ in a purely arithmetic manner, without reference to Loop programs or Turing machines. Notice, however, that we have not yet proved Theorem (4.5) which shows that $f_\alpha \in \mathcal{L}_\alpha$; thus to avoid circularity we will refrain from using (6.3) until (4.5) is proved. Theorems (6.1) and (6.2) do not depend on (4.5).

(6.4) Definition. A class $\mathcal{C}$ of functions is <u>computation-time closed</u> if whenever $f \in \mathcal{C}$, there is a function $s_f \in \mathcal{C}$ such that s_f pointwise bounds the number of steps required to compute f on a Turing machine, and if conversely whenever there is an $s_f \in \mathcal{C}$ which bounds the number of steps required to compute some function f, then $f \in \mathcal{C}$.

(6.5) Theorem For $\alpha \geq 2$, $\mathcal{L}_\alpha$ is computation-time closed.

Proof. Immediate, using (6.2) and the fact that $\hat{f}_\alpha \in \mathcal{L}_\alpha$ and $\hat{f}_\alpha(x) \geq f_\alpha(x)$ for $x > 0$.

It can be proved that every class of functions which is closed under substitution, computation-time closed, and containing a sufficiently large function is also closed under the operation of <u>limited recursion</u> defined below; we will use another, more direct method to show each $\mathcal{L}_\alpha$ is closed under limited recursion. The proof yields a corollary which indicates the power of the classes $\mathcal{L}_\alpha$ for $\alpha < \omega$.

(6.6) Definition. If f obeys the conditions

$$f(\bar{x}_n, 0) = g(\bar{x}_n)$$
$$f(\bar{x}_n, y+1) = h(\bar{x}_n, y, f(\bar{x}_n, y))$$

then f is said to be defined by <u>primitive recursion from</u> g <u>and</u> h. We allow the case $n = 0$, so that g may be a function of 0 variables, that is, a constant.

(6.7) Definition. If $f: N^{n+1} \to N$ is defined by primitive recursion from functions g and h, and if in addition we have $f(\bar{x}_n,y) \leq b(\bar{x}_n,y)$ for some function b and all $\bar{x}_n$, y, then f is said to be defined by <u>limited recursion</u> from g, h, and b.

(6.8) Theorem. For $\alpha \geq 2$, $\mathfrak{L}_\alpha$ is closed under limited recursion. That is, if f is defined from g, h, $b \in \mathfrak{L}_\alpha$ by limited recursion, then $f \in \mathfrak{L}_\alpha$.

Proof. We have

$$f(\bar{x}_n,0) = g(\bar{x}_n)$$
$$f(\bar{x}_n,y+1) = h(\bar{x}_n,y,f(\bar{x}_n,y))$$
$$f(\bar{x}_n,y) \leq b(\bar{x}_n,y)$$

where $g,h,b \in \mathfrak{L}_\alpha$. Let $(\underset{\sim}{G},\bar{X}_n,G)$ be a program for g where $\underset{\sim}{G} \in L_\alpha$ and $\underset{\sim}{G}$ does not destroy registers $\bar{X}_n$ and Y. Let $(\underset{\sim}{H},\bar{X}_n,Z,F,H)$ be a program for h where again $\underset{\sim}{H} \in L_\alpha$ and $\underset{\sim}{H}$ does not destroy the contents of $\bar{X}_n$, Z, F. We also assume that the registers of $\underset{\sim}{G}$ and $\underset{\sim}{H}$ do not overlap except for $\bar{X}_n$. Such programs are easily found given any programs for g and h. Then let $\underset{\sim}{F}$ be the program

```
  G
  ~
F = G
Z = 0
LOOP(1)  Y
    H
    ~
  F = H
  Z = Z+1
END
```

Then $(\underset{\sim}{F},\bar{X}_n,Y,F)$ computes f. For say y = 0; then the instructions within the Loop are not executed, and after execution F contains $g(\bar{x}_n) = f(\bar{x}_n,0)$. If $y > 0$, after the first execution of the instructions in the Loop the contents of F are $h(\bar{x}_n,0,g(\bar{x}_n)) = f(x,1)$; by induction, after the y executions of the instructions within the Loop, the contents of F are $h(\bar{x}_n,y-1,f(\bar{x}_n,y-1)) = f(\bar{x}_n,y)$. By counting the steps required to execute $\underset{\sim}{F}$,

$$T_{\underset{\sim}{F}}(\bar{x}_n,y) = T_{\underset{\sim}{G}}(\bar{x}_n) + \Sigma_{z=0}^{y-1}[2 + T_{\underset{\sim}{H}}(\bar{x}_n,z,f(\bar{x}_n,z))] + y + 4$$

By the Bounding Theorem (3.6), if we let $\underline{m} = \max\{\bar{x}_n\}$,

$$T_{\underset{\sim}{F}}(\bar{x}_n,y) \leq f_\alpha^{(p)}(\underline{m}) + \Sigma_{z=0}^{y-1}\,[2 + f_\alpha^{(q)}(\max\{\underline{m},z,f(\bar{x}_n,z)\})] + y + 4$$

By (3.8) since $b \in \mathcal{L}_\alpha$,

$$T_{\underset{\sim}{F}}(\bar{x}_n,y) \leq f_\alpha^{(p)}(\underline{m}) + \Sigma_{z=0}^{y-1}[2 + f_\alpha^{(q)}(\max\{\underline{m},z,f_\alpha^{(r)}(\max\{\underline{m},z\})})] + y + 4$$

$$\leq f_\alpha^{(p)}(\underline{m}) + (y\dot{-}1)\cdot[2 + f_\alpha^{(q)}(\max\{\underline{m},y,f_\alpha^{(r)}(\max\{\underline{m},y\})})] + y + 4$$

Then by using Lemma (3.4) repeatedly, exactly as in (3.7), there is a number s so that $T_{\underset{\sim}{F}}(\bar{x}_n,y) \leq f_\alpha^{(s)}(\max\{\underline{m},y\})$. But then by (6.1), $f \in \mathcal{L}_\alpha$. This concludes (6.8).

The method yields two corollaries.

(6.9) Corollary. If f is defined by primitive recursion from $g \in \mathcal{L}_{\alpha+1}$, $h \in \mathcal{L}_\alpha$, then $f \in \mathcal{L}_{\alpha+1}$.

Proof. If f is defined from g and h exactly as in the theorem, except that the requirement $f(\bar{x}_n,y) \leq b(\bar{x}_n,y)$ is dropped and we now allow $g \in \mathfrak{L}_{\alpha+1}$, then the program for f given in the proof of (6.8) still works; by Definition (1.1) of $L_{\alpha+1}$, $\underset{\sim}{F} \in \mathfrak{L}_{\alpha+1}$.

(6.10) Definition. $\mathfrak{P}$, the class of <u>primitive recursive functions</u>, is the smallest class of functions containing the successor function $s(x) = x+1$, the identity function $i(x) = x$, and closed under substitution and primitive recursion.

(6.11) Theorem. The class $\bigcup_{\alpha<\omega} \mathfrak{L}_\alpha$ contains the primitive recursive functions.

Proof. $\mathfrak{L}_0$ contains s(x) and i(x). By (4.9) and (6.9), each primitive recursive function is in $\mathfrak{L}_\alpha$ for some $\alpha < \omega$.

(6.12) Theorem. For each $\alpha \geq 2$, $\mathfrak{L}_{\alpha+1}$ contains a <u>universal function</u> for $\mathfrak{L}_\alpha$; that is, a function U_α: $N^2 \to N$ so that if $f: N \to N$ is a function in $\mathfrak{L}_\alpha$, there is an e so $U_\alpha(e,x) = f(x)$, and conversely for each fixed e, $U_\alpha(e,x)$ is an $\mathfrak{L}_\alpha$ function.

Proof. Given a function g, its iterate $g^{(y)}(x)$ is defined by a special case of primitive recursion (see Definition (3.1)). Thus in particular the function $\hat{f}_\alpha^{(y)}(x+1)$ is in $\mathfrak{L}_{\alpha+1}$. Take

$$U_\alpha(e,x) = M_1(e,x,\hat{f}_\alpha^{(e)}(x+1))$$

For each fixed e, $U_\alpha \in \mathcal{L}_\alpha$. Also, each function in $\mathcal{L}_\alpha$ must have an infinite number of Gödel numbers; for example, an arbitrarily large number of (useless) "X = X" instructions may be prefixed to any program. Thus by (6.1), for every $f \in \mathcal{L}_\alpha$, there is an e so $f(x) = U_\alpha(e,x)$.

Notice that although we used (6.9) in this proof, the theorem follows essentially from the computation-time closure of $\mathcal{L}_\alpha$ and the fact that $\mathcal{L}_{\alpha+1}$ contains a function which bounds every function of $\mathcal{L}_\alpha$.

(6.13) Corollary. For each $\alpha \geq 2$, $\mathcal{L}_{\alpha+1}$, contains a characteristic function (that is, a function whose range is $\{0,1\}$) which is not in $\mathcal{L}_\alpha$.

Proof. It is immediate that the function $1 \dot{-} x$ is in $\mathcal{L}_1$ and hence in $\mathcal{L}_{\alpha+1}$. Take $g(x) = 1 \dot{-} U_\alpha(x,x)$. Then by Cantor's diagonal method, if $g \in \mathcal{L}_\alpha$, g must have a Gödel number e: $g(x) = U_\alpha(e,x)$. But then

$$1 \dot{-} U_\alpha(e,e) = g(e) = U_\alpha(e,e)$$

which is absurd.

III. MULTIPLE RECURSIVE FUNCTIONS

§7. This chapter studies the theory of the multiple recursive functions. Many of the results in this theory have exact counterparts in the theory of Loop programs developed in Chapter II; it also turns out that the methods of proof of the corresponding theorems are often quite analogous. In large measure the similarity in the development of the two theories occurs simply because the theories are, in fact, very similar; it is also due to a conscious attempt to draw the appropriate parallels. This attempt is made in the belief that both the author and the reader benefit from the technical economy achieved by using a few tools rather than a large collection. Finally, we believe the methods used here and in Chapter II are of great utility in the characterization of sets of computable functions; support for such a claim can only come from successful use of these methods.

The theory of Loop programs may be regarded as an attempt to examine the result of restricting the notion of program in such a way that the structure of a program controls the complexity of the operations the program performs. The theory of Loop programs is thus in the tradition of the Turing-computable functions: those functions computable by Turing machines. Here we take "Turing machine" in the broad sense of referring to all the various theoretical machines which serve as models for digital computers. But it is well-known that several quite different ways of defining

"effectively computable" all lead to exactly the same class of functions. Chief among these alternative approaches is the definition of functions by Herbrand-Gödel-Kleene recursion equations.

We summarize this approach, following Kleene [K, §54]. Imagine a formal language built up from several basic symbols: = (equals), ' (successor), 0 (zero), (,) (left and right parentheses), $\underline{f}$, $\underline{g}$, $\underline{h}$, $\underline{f}_1$, $\underline{g}_1$, $\underline{h}_1$,..., (function letters), $\underline{x}$, $\underline{y}$, $\underline{z}$, $\underline{x}_1$, $\underline{y}_1$, $\underline{z}_1$,..., (variables for natural numbers), and , (comma). From these symbols are constructed several kinds of formal expressions. The numerals are 0, 0', 0",...; these stand for the natural numbers 0, 1, 2,... . The formal expression which is a numeral for a number x we write $\nu(x)$. Terms are 0, any variable letter, expressions of the form t' where t is a term, and $f(t_1,\ldots,t_n)$ where f is a function letter and $t_1,\ldots,t_n$ are terms.

Next we have equations of the form t = s where t and s are terms. Systems of equations are finite sequences $e_1,\ldots,e_n$ of equations. The systems of equations are the basic objects of study.

A system of equations may have a principal function letter: the first (left-most) function letter of the last equation of the system. From a system of equations formal deductions may be made. The deductions are precisely analogous to deductions in formal logic from a set of postulates. There are two rules of inference:

(R1) From an equation containing a variable letter, we may pass to the equation obtained by substituting

a particular numeral for every occurrence of the variable letter.

(R2) From an equation of the form $f(\nu(x_1),\ldots,\nu(x_n)) = \nu(x)$ and another equation $r = s$, we may pass to the equation which results by substituting $\nu(x)$ for one or more occurrences of $f(\nu(x_1),\ldots,\nu(x_n))$ in the equation $r = s$.

Then a deduction of an equation e from a system of equations E is a sequence of equations, each of which is either one of the equations of E or obtained from one (or two) of the earlier equations of the deduction by an application of R1 (or R2).

A system of equations E defines the function φ recursively whenever the following holds: f is the principal function letter of E, and for all $x_1,\ldots,x_n$ the equation $f(\nu(x_1),\ldots,\nu(x_n)) = \nu(x)$ is deducible from E if and only if $\varphi(x_1,\ldots,x_n) = x$. If a (total) function has a system of equations which defines it recursively, the function is called general recursive. Kleene shows that the class of general recursive functions is precisely the same class as the functions computable by a Turing machine.

The class of multiple recursive functions may be defined in an analogous way; we will instead use a slightly different approach, and then discuss its relationship with the Kleene formulation.

(7.1) Definition. For some $n \geq 1$ and $m \geq 0$, suppose the function $f:N^{n+m} \to N$ satisfies the 2^n equations:

$$f(0,\ldots,0,\bar{y}_m) = F_1$$
$$f(0,\ldots,0,x_n+1,\bar{y}_m) = F_2$$
$$f(0,\ldots,0,x_{n-1}+1,0,\bar{y}_m) = F_3$$
$$\vdots$$
$$f(x_1+1,\ldots,x_n+1,\bar{y}_m) = F_{2^n}$$

where $F_1,\ldots,F_{2^n}$ are formulas built up from constants, variables $\bar{x}_n$, $\bar{y}_m$, and functions $g_1,\ldots,g_r$ by substitution. Suppose also that F_1 contains no occurrences of f, and in each other equation

$$f(\xi_1,\ldots,\xi_n,\bar{y}_m) = F_j$$

where each ξ_i is either "x_i+1" or "0", each occurrence of f in F_j has a k, $1 \leq k \leq n$, so f appears in the context $f(\xi_1,\ldots,\xi_{k-1},x_k,T_{k+1},\ldots,T_n,\bar{S}_m)$ where ξ_k is "x_k+1", and $T_{k+1},\ldots,T_n,\bar{S}_m$ are terms (i.e. formulas) built up from variables $\bar{y}_m$ and those x_i for which ξ_i = "x_i+1" by application of $g_1,\ldots,g_r$ and f. Then f is said to be <u>defined</u> by n-recursion from $g_1,\ldots,g_r$.

(7.2) Example. f is defined by 2-recursion from $g_1,\ldots,g_r$ if f satisfies

$$f(0,0,y) = g_1(y,3)$$
$$f(0,x_2+1,y) = f(0,x_2,g_2(y))$$
$$f(x_1+1,0,y) = f(x_1,g_3(f(x_1,x_1,y+1)),g_4(y))$$
$$f(x_1+1,x_2+1,y) = g_5(f(x_1,f(x_1+1,x_2,y),y))$$

(7.3) Definition. For each ordinal $\alpha < \omega^{\omega}$, $\mathcal{R}_{\alpha}$ is the least class of functions satisfying

(i) If $\alpha = 0$, $\mathcal{R}_{\alpha}$ contains the successor function $s(x) = x+1$ and the identity function $i(x) = x$

(ii) If $\beta < \alpha$, $\mathcal{R}_{\beta} \subseteq \mathcal{R}_{\alpha}$

(iii) $\mathcal{R}_{\alpha}$ is closed under substitution

(iv) If $\alpha = \beta + \omega^{n}$ for some $n \geq 0$, and f is defined by $(n+1)$-recursion from $g_1, \ldots, g_r \in \mathcal{R}_{\beta}$, then $f \in R_{\alpha}$.

We will call $\mathcal{R} = \bigcup_{\alpha < \omega^{\omega}} \mathcal{R}_{\alpha}$ the <u>multiple recursive functions</u>. Also, for each $n \geq 1$, $\bigcup_{\alpha < \omega^{n}} \mathcal{R}_{\alpha}$ is the class of <u>n-recursive functions</u>.

It will be seen that if a function f is defined by n-recursion from well-defined, total functions $g_1, \ldots, g_n$, then f is in fact a well-defined, total function. The proof is by induction on the well-ordering of n-tuples of integers under the lexicographical ordering.

(7.4) Definition. The n-tuple $\bar{x}_n$ is <u>lexicographically less than</u> the n-tuple $\bar{y}_n$ (in symbols, $(\bar{x}_n) < (\bar{y}_n)$) whenever there is a u such that $x_u < y_u$ and for all $i < u$, $x_i = y_i$.

Notice that this relation is a well-ordering of order type ω^{n} by the mapping

$$(x_n) \longleftrightarrow \omega^{n-1} \cdot x_1 + \cdots + \omega^{0} \cdot x_n$$

(7.5) Theorem. If f is defined from total functions $g_1,\ldots,g_r$ by n-recursion, f is a total, well-defined function.

Proof. We have the equation $f(0,\ldots,0,\bar{y}_m) = F_1$. By the definition, F_1 cannot contain any occurrences of f; so $f(0,\ldots,0,\bar{y}_m)$ is uniquely defined for all $\bar{y}_m$. Now suppose $f(\bar{z}_n,\bar{y}_m)$ is uniquely defined for all $\bar{y}_m$ and all $\bar{z}_n$ with $(z_n) < (x_n)$. Then $f(\bar{x}_n,\bar{y}_m) = F_j$, where F_j is a formula built up from (some of) $g_1,\ldots,g_r$ and occurrences of f of the form $f(\bar{T}_n,\bar{S}_m)$ where $T_1,\ldots,T_n,S_1,\ldots,S_m$ are terms and, by definition, $(\bar{T}_n) < (\bar{x}_n)$. Thus $f(\bar{x}_n,\bar{y}_m)$ is uniquely defined.

Now by Definition (7.3) each function $f \in \mathfrak{R}_\alpha$ is defined by a sequence of equations, each of which defines a new function used in the definition of f. The initial equations in the sequence define functions from the initial functions s(x) and i(x); and each equation in the sequence is either an instance of substitution which defines a new function from functions defined earlier, or part of an instance of the schema of n-recursion from functions defined earlier. These equations can of course be translated into the formal equations of Kleene; this is really nothing more than a one-for-one replacement of the informal symbols of the defining equations ~~for~~ by the formal symbols of the recursion equations. Conversely, it should be obvious that each system of formal equations which obeys a few purely syntactic rules defines a multiple recursive function. The rules are: each equation e is either of the form $f(x_1,\ldots,x_n) = T$, where T is a term containing no function letters, or is of the form

$f(x_1,\ldots,x_n) = T$, where T is a term containing function letters defined by earlier equations (formal substitution), or is part of the (formal) scheme of n-recursion corresponding to the (informal) Definition (7.1). We also require that each system of equations be consistent: that it not define the same function letter twice, nor use the same function letter with varying numbers of arguments. Again, this restriction is purely syntactic. We may also attach an ordinal α to each function letter used in such a restricted system of equations: if a function letter f is defined by (formal) substitution from function letters $f_1,\ldots,f_r$, attach to f the ordinal $\alpha = \max\{\alpha_1,\ldots,\alpha_r\}$ where $\alpha_1,\ldots,\alpha_r$ are the ordinals attached to $f_1,\ldots,f_r$; or if r = 0, so f is defined by substitution from the empty set of functions, $\alpha = 0$. Also, if f is defined by (formal) (n+1)-recursion from $g_1,\ldots,g_r$, assign f the ordinal $\alpha = \max\{\alpha_1,\ldots,\alpha_r\} + \omega^n$. Then assign to a system of equations the ordinal of its principal function letter, and let R_α be the set of those systems of equations with ordinal less than or equal to α. The point is that the systems of equations in R_α have a purely syntactical definition; furthermore, given a sequence of formal symbols, we can effectively test whether the sequence is in R_α. Finally, each member of R_α is a system of equations in the Kleene sense, so deductions may be made from such a system in exactly the same way as they are from the more general systems of equations. It should be clear that a function f is in R_α if and only if there

is a system of equations in R_α which defines f recursively.

Other writers use definitions of n-recursion somewhat different from ours. Péter [~~P1~~[19], ~~P2~~[21]], for example, uses a slightly less general scheme in which f obeys

$$f(\bar{x}_n,\bar{y}_m) = g(\bar{y}_m) \qquad \text{if } x_1 \cdot \ldots \cdot x_n = 0$$
$$f(\bar{x}_1 +1,\ldots,x_n +1,\bar{y}_m) = F \qquad \text{otherwise}$$

where each occurrence of f in F has the form $f(x_1 + 1,\ldots,x_i + 1,x_{i+1},T_{i+2},\ldots,T_n,\bar{y}_m)$. Our development could just as easily have been carried out in this way. Robbin [~~JR~~[25]] uses a more general scheme.

$$f(\bar{x}_n,\bar{y}_m) = F_o \qquad \text{if } (\bar{x}_n) = (0,\ldots,0)$$
$$f(\bar{x}_n,\bar{y}_m) = F \qquad \text{if } (\bar{x}_n) \neq (0,\ldots,0)$$

where F_o is a formula not containing f, and every occurrence of f in F is of the form $f(T_1,\ldots,T_n,S_1,\ldots,S_m)$ where $T_1,\ldots,T_n,S_1,\ldots,S_m$ are formulas and for all $(\bar{x}_n) \neq (0,\ldots,0)$, $(T_n) < (\bar{x}_n)$. The only problem with this scheme for our purposes is that given a pair of equations in the above form, it is not clear from their syntactic structure that f is properly defined, because the condition $(T_n) < (\bar{x}_n)$ is not a syntactical property, but depends on the values of the functions involved. In fact, given a pair of equations like the above, it is effectively undecidable to determine in general whether the condition $(T_n) < (\bar{x}_n)$ is met. All of these approaches have the common property

that a function is defined by induction on the lexicographical well-ordering of n-tuples. As we will discover, all the variations are equivalent in that they lead to the same classes of functions.

§8. This section corresponds to §§3-4 of Chapter II in that it establishes the rate of growth of the largest functions of each class $\mathcal{R}_\alpha$. There is a Bounding Theorem for $\mathcal{R}_\alpha$, much like Theorem (3.6), showing that each function in $\mathcal{R}_\alpha$ is bounded by $f^{(p)}_{1+\alpha}$ for some p; and a Hierarchy Theorem for $\mathcal{R}_\alpha$, which proves the inequality $\mathcal{R}_\alpha \supset \mathcal{R}_\beta$ for $\alpha > \beta$ by demonstrating that $f_{1+\alpha} \in \mathcal{R}_\alpha$ for $\alpha \geq 1$. Thus the Bounding Theorem for $\mathcal{R}_\alpha$ is different from that for $\mathcal{L}_\alpha$, in that the former limits the size of the functions of $\mathcal{R}_\alpha$, whereas the latter bounds the computation time of functions of $\mathcal{L}_\alpha$. The bound on the functions of $\mathcal{L}_\alpha$ came as a corollary to the bound on computation time; the ~~reserve~~ reverse will be true of $\mathcal{R}_\alpha$.

(8.1) Bounding Theorem for $\mathcal{R}_\alpha$. If $f:N^n \to N$ is a function in $\mathcal{R}_\alpha$, there is a p such that $f(\bar{x}_n) \leq f^{(p)}_{1+\alpha}(\max\{\bar{x}_n\})$; p depends effective on the recursion equations defining f.

Proof. Like that of the Bounding Theorem for Loop programs, this proof is by induction on Definition (7.3) of $\mathcal{R}_\alpha$. There are four cases corresponding to the four clauses of (7.3) which exhaust the ways by which a function f may be a member of $\mathcal{R}_\alpha$.

Case 1. $f(x) = x+1$ or $f(x) = x$. We have immediately that $f(x) \leq f^{(1)}_0(x) \leq f^{(1)}_1(x)$.

Case 2. $f \in \mathcal{R}_\beta$ and $\beta < \alpha$. Then we have a p so that $f(\bar{x}_n) \leq f^{(p)}_{1+\alpha}(\max\{\bar{x}_n\})$ by the induction hypothesis for $\mathcal{R}_\beta$ and (3.4.viii).

Case 3. f is defined by substitution from functions $g_1,\ldots,g_r \in \mathcal{R}_\alpha$. The theorem is immediate by Lemma (3.4).

Case 4. f is defined by $(n+1)$-recursion from functions $g_1,\ldots,g_r$ in $\mathcal{R}_\beta$, where $\alpha = \beta + \omega^n$. This case is proved by induction on n. Suppose F is a formula built up by substitution. We define the depth of F by induction on its structure as follows: the depth of a variable or a constant is zero; the depth of $g(F_1,\ldots,F_m)$ where $F_1,\ldots,F_m$ are formulas is $\max\{\text{depth}(F_j)\}+1$.

Now consider the base of the induction, $n = 0$. Then $\alpha = \beta + 1$ and f is defined by 1-recursion from $g_1,\ldots,g_r \in \mathcal{R}_\beta$. We have

$$f(0,\bar{y}_m) = F_1$$
$$f(x+1,\bar{y}_m) = F_2$$

Let a be the greater of the depths of F_1 and F_2, and let b be sufficiently large so $f_{1+\beta}^{(b)}$ bounds each of $g_1,\ldots,g_r$, and also all the constants occurring in F_1 and F_2. Then

$$f(0,\bar{y}_m) \leq f_{1+\beta}^{(ba)}(\max\{\bar{y}_m\})$$

Suppose for each $z \leq x$ where $x \geq 0$ we have

$$f(z,\bar{y}_m) \leq f_{1+\beta}^{(ba^{z+1})}(\max\{z,\bar{y}_m\})$$

By definition, $f(x+1,\bar{y}_m) = F_2$. But since each occurrence of f in F_2 is of the form $f(x,\bar{T}_m)$, by the increasing property of $f_{1+\beta}$ and the hypotheses on F_2 and f,

$$f(x+1,\bar{y}_m) \leq f_{1+\beta}^{(ba^{x+2})}(\max\{x+1,\bar{y}_m\})$$

Thus, if we write $\underline{m}$ for $\max\{x,\bar{y}_m\}$,

$$\begin{aligned} f(x,\bar{y}_m) &\leq f_{1+\beta}^{(ba^{x+1})}(\underline{m}) \\ &\leq f_{1+\beta}^{(ba^{\underline{m}+1})}\, f_{1+\beta}^{(\underline{m})}(1) \\ &= f_{1+\alpha}(ba^{\underline{m}+1}+\underline{m}) \\ &\leq f_{1+\alpha}\, f_2^{(ba+1)}(\underline{m}) \\ &\leq f_{1+\alpha}^{(ba+2)}(\underline{m}) \end{aligned}$$

We have thus proved the following for $n = 0$:

(8.2) Lemma. If f is defined by $(n+1)$-recursion from $g_1,\ldots,g_r$, and if the greatest depth of the formulas $F_1,\ldots,F_{2^{n+1}}$ defining f is a, and $f_{1+\beta}^{(b)}$ bounds all of $g_1,\ldots,g_r$ as well as all the constants of $F_1,\ldots,F_{2^{n+1}}$, then f is bounded by $f_{1+\alpha}^{(ba+t+2)}$, where $\alpha = \beta+\omega^n$ and $t = t_n(\beta)$.

Proof. The basis $n = 0$ has already been done, so we will assume the lemma for some $n \geq 0$ and prove it for $n+1$. Thus, a function $f(x,x_0,\ldots,x_n,\bar{y}_m)$ is being defined by $(n+2)$-recursion. For each fixed x, let $f_{(x)}(x_0,\ldots,x_n,\bar{y}_m) = f(x,x_0,\ldots,x_n,\bar{y}_m)$. On examining the 2^{n+2} equations defining f, it is found that the first 2^{n+1} of them constitute a definition of $f_{(0)}$ by $(n+1)$-recursion, for these

equations specify the value of $f(x,x_0,\ldots,x_n,\bar{y}_m)$ when $x = 0$. Thus by the induction hypothesis, $f_{(0)}(x_0,\ldots,x_n,\bar{y}_m) \leq f^{(ba+t+2)}_{1+\beta+\omega^n}(\underline{m})$, where $\underline{m}$ is $\max\{x_0,\ldots,x_n,\bar{y}_m\}$ and $t = t_n(\beta)$. Suppose for some x that

$$f_{(x)}(x_0,\ldots,x_n,\bar{y}_m) \leq f^{\left((ba+t+2)^{x+1}\right)}_{1+\beta+\omega^n(x+1)}(\underline{m})$$

Again, by the definition (7.1) of n-recursion, $f_{(x+1)}$ is defined by $(n+1)$-recursion from $g_1,\ldots,g_r$ and $f_{(x)}$. The depth of the defining formulas is still a, and by (3.4.viii) and the induction hypothesis for $g_1,\ldots,g_r$, the function

$$f^{\left((ba+t+2)^{x+1}\right)}_{1+\beta+\omega^n(x+1)}$$

bounds all of $g_1,\ldots,g_r$, $f_{(x)}$. Thus, now letting $\underline{m}$ be $\max\{x+1,x_0,\ldots,x_n,\bar{y}_m\}$,

$$\begin{aligned} f_{(x+1)}(x_0,\ldots,x_n,\bar{y}_m) &\leq f^{\left((ba+t+2)^{x+1}\cdot a+t+2\right)}_{1+\beta+\omega^n(x+2)}(\underline{m}) \\ &\leq f^{\left((ba+t+2)^{x+2}\right)}_{1+\beta+\omega^n(x+2)}(\underline{m}) \end{aligned}$$

Thus, we have shown where $\underline{m}$ is $\max\{x,x_0,\ldots,x_n,\bar{y}_m\}$,

$$\begin{aligned} f(x,x_0,\ldots,x_n,\bar{y}_m) &\leq f^{\left((ba+t+2)^{x+1}\right)}_{1+\beta+\omega^n(x+1)}(\underline{m}) \\ &\leq f^{\left((ba+t+2)^{\underline{m}}+\underline{m}\right)}_{1+\beta+\omega^n(\underline{m}+1)}(1) \\ &\leq f_{1+\beta+\omega^n(\underline{m}+1)+1}\left((ba+t+2)^{\underline{m}+2}\right) \\ &\leq f_{1+\beta+\omega^n\cdot A}(A) \qquad \text{where } A = (ba+t+2)^{\underline{m}+2} \end{aligned}$$

Now if $\beta = \omega^s \cdot b_s + \cdots + \omega^{n+1} \cdot b_{n+1} + \cdots + \omega^0 \cdot b_0$, let $\beta' = \omega^s \cdot b_s + \cdots + \omega^{n+1} \cdot b_{n+1}$. Thus β' is the least ordinal so $\alpha = \beta' + \omega^{n+1}$, and $1 + \beta + \omega^n \cdot (ba + t + 2)^{\underline{m+2}} = 1 + \beta' + \omega^n \cdot ((ba + t + 2)^{\underline{m+2}} + b_n)$. Then

$$\begin{aligned} f(x, x_0, \ldots, x_n, \bar{y}_m) &\leq f_{1+\beta'+\omega^n \cdot (A+b_n)}(A + b_n) \\ &= f_{1+\alpha}((ba + t + 2)^{\underline{m+2}} + b_n) \\ &\leq f_{1+\alpha}^{(ba+t+b_n+2)}(\underline{m}) \end{aligned}$$

But since $t_{n+1}(\beta) = t + b_n$ by definition of t, this proves Lemma (8.2) and thus Theorem (8.1).

Unfortunately, the somewhat more attractive conjecture that $f_\alpha^{(p)}$ bounds the functions of $\mathcal{R}_\alpha$ fails. This matter will be discussed after (8.3).

(8.3) Theorem. For each $\alpha \geq 1$, $f_{1+\alpha} \in \mathcal{R}_\alpha$.

Proof. Consider the function $h_{\beta,n}: N^{n+2} \to N$ defined by $(n+2)$-recursion from f_β:

$$\begin{aligned} h_{\beta,n}(\xi_0, \ldots, \xi_n, 0) &= 1 \\ h_{\beta,n}(0, \ldots, 0, x+1) &= f_\beta(x+1) \\ h_{\beta,n}(\xi_0, \ldots, \xi_{n-1}, x_n+1, x+1) &= h_{\beta,n}(\xi_0, \ldots, \xi_{n-1}, x_n, h_{\beta,n}(\xi_0, \ldots, \xi_{n-1}, x_n+1, x)) \\ h_{\beta,n}(\xi_0, \ldots, \xi_{n-2}, x_{n-1}+1, 0, x+1) &= h_{\beta,n}(\xi_0, \ldots, \xi_{n-2}, x_{n-1}, x+1, x+1) \\ &\vdots \\ h_{\beta,n}(x_0+1, 0, \ldots, 0, x+1) &= h_{\beta,n}(x_0, x+1, 0, \ldots, 0, x+1) \end{aligned}$$

Each equation containing a ξ is schematic in that it represents all the equations obtained by replacing ξ_i by "$x_i + 1$" or "0". We show that when β is of the form $\beta = \beta' + \omega^n$ for some β', then $h_{\beta,n}(x_0,\ldots,x_n,x) = f_{\beta+\gamma}(x)$ where $\gamma = \omega^n \cdot x_0 + \cdots + \omega^0 \cdot x_n$. The induction is on γ. If $\gamma = 0$, so $x_o = \cdots = x_n = 0$, then $h_{\beta,n}(0,\ldots,0,x)$ $= f_\beta(x)$ by the first and second equations. If γ is a successor, so $\gamma = \delta + 1$ where $\delta = \omega^n \cdot x_0 + \cdots + \omega^0 \cdot x_n$, the third equation applies:

$$h_{\beta,n}(\xi_0,\ldots,\xi_{n-1},x_n+1,x+1) = h_{\beta,n}(\xi_0,\ldots,\xi_{n-1},x_n,h_{\beta,n}(\xi_0,\ldots,\xi_{n-1},x_n+1,x))$$

By the induction hypothesis for δ and the first equation, we have

$$h_{\beta,n}(x_0,\ldots,x_{n-1},x_n+1,0) = 1$$
$$h_{\beta,n}(x_0,\ldots,x_{n-1},x_n+1,x+1) = f_{\beta+\delta}(h_{\beta,n}(x_0,\ldots,x_{n-1},x_n+1,x))$$

But for fixed $x_0,\ldots,x_n$, these are the same equations defining $f_{\beta+\delta+1}$ $= f_{\beta+\gamma}$, by Definition (3.2). Finally, if γ is a limit, so $\gamma = \delta + \omega^{n-m}$ where $m < n$ and $\delta = \omega^n \cdot x_0 + \cdots + \omega^{n-m+1} \cdot x_{m-1} + \omega^{n-m} \cdot x_m$, we have

$$h_{\beta,n}(x_0,\ldots,x_{m-1},x_m+1,0,\ldots,0,0) = 1$$
$$h_{\beta,n}(x_0,\ldots,x_{m-1},x_m+1,0,\ldots,0,x+1) = h_{\beta,n}(x_0,\ldots,x_{m-1},x_m,x+1,0,\ldots,0,x+1)$$

Combining the equations and using the induction hypothesis for δ,

$$h_{\beta,n}(x_0,\ldots,x_{m-1},x_m+1,0,\ldots,0,x) = f_{\beta+\delta+\omega^{n-m-1}\cdot x}(x)$$
$$= f_{\beta+\gamma}(x)$$

by Definition (3.2).

Now consider the equations

$$f(0,y) = y+1$$
$$f(x+1,y) = f(x,f(x,y))$$

which are an instance of 1-recursion. We show that $f(x,y) = y+2^x$. This is clearly true for $x = 0$; if it is true for x,

$$\begin{aligned} f(x+1,y) &= f(x,\ f(x,y)) \\ &= f(x,y)+2^x \\ &= y+2^x+2^x \\ &= y+2^{x+1} \end{aligned}$$

So $f \in \mathcal{R}_1$ and $f(x,0) = f_2(x)$.

Now let α be an ordinal, $1 < \alpha < \omega^\omega$, and assume that $f_{1+\beta} \in \mathcal{R}_\beta$ for $1 \leq \beta < \alpha$. If α is a successor, $\alpha = \beta+1$, then $f_{1+\alpha}$ is obtained from $f_{1+\beta}$ by iteration (Definition (3.1)), which is a special case of 1-recursion, so $f_{1+\beta} \in \mathcal{R}_\alpha$. If α is a limit ordinal, let β be the least ordinal so $\alpha = \beta+\omega^{n+1}$. By definition, $h_{1+\beta,n}$ is obtained by $(n+2)$-recursion from $f_{1+\beta}$, and so by the induction hypothesis, $h_{1+\beta,n} \in \mathcal{R}_{1+\beta+\omega^{n+1}}$. But $h_{1+\beta,n}(x,0,\ldots,0,x) = f_{1+\beta+\omega^n\cdot x}(x) = f_{1+\alpha}(x)$, so by closure under substitution, $f_{1+\alpha} \in \mathcal{R}_\alpha$. This concludes (8.3).

The rather unpleasant need to use $f_{1+\alpha}$ to bound $\mathcal{R}_\alpha$, rather than f_α, stems from the difference between 1-recursion and primitive recursion. The equations above,

$$f(0,y) = y+1$$

$$f(x+1,y) = f(x, f(x,y))$$

which make $f(x,y) = y+2^x$, are not an instance of primitive recursion, because in the latter scheme the parameters must remain fixed, not variable, in the defining formulas. In other words, the schema of primitive recursion may be written

$$f(0,\bar{y}_m) = F_1$$

$$f(x+1,\bar{y}_m) = F_2$$

where F_1 does not contain f, and where every instance of f in F_2 is of the form $f(x,\bar{y}_m)$; here 1-recursion would have $f(x,\bar{T}_m)$ where $\bar{T}_m$ are formulas. The difference is between "nested" and "unnested" formulas. This matter will be discussed more fully in Chapter V. Notice, incidentally, that if $\alpha \geq \omega$, $1+\alpha = \alpha$.

The above results give

(8.4) Hierarchy Theorem for $\mathcal{R}_\alpha$. If $\alpha > \beta$, $\mathcal{R}_\alpha \supset \mathcal{R}_\beta$.

Proof. Immediate by (8.1), (8.3), and (4.6).

§9. The task of this section is to establish the computation-time closure of $\mathcal{R}_\alpha$ for each $\alpha \geq 2$. The path we take is essentially the same as that followed for $\mathcal{L}_\alpha$: show that the computation time of each function in $\mathcal{R}_\alpha$ is bounded by another function in $\mathcal{R}_\alpha$, and then find a function in $\mathcal{R}_2$ which mimics the actions of an arbitrary Turing machine for a given number of steps. We base the proof for the first half of the result on the use of deductions from the formal recursion equation defining a function in $\mathcal{R}_\alpha$. This method is by no means the only way to carry out the proof, but it seems to offer the fewest technical difficulties and will be applicable as well to later work.

(9.1) Theorem. For each $\alpha < \omega^\omega$, if $f \in \mathcal{R}_\alpha$ then f can be computed by a Turing machine in such a way that the number of steps required to compute $f(\bar{x}_n)$ is bounded by $f^{(p)}_{1+\alpha}(\max\{\bar{x}_n\})$ for some p.

Proof. We will show that for each $f \in \mathcal{R}_\alpha$ there is a set of equations E defining f recursively and a number q so that $f^{(q)}_{1+\alpha}(\max\{\bar{x}_n\})$ bounds the number of equations in a certain deduction of the equation $f(\nu(x_1),\ldots,\nu(x_n)) = \nu(x)$ from E. Then we will arrange for a Turing machine to perform the deduction and conclude the theorem.

If $f \in R_o$, then $f(\bar{x}_n) = x_i + c$ or $f(\bar{x}_n) = c$ for some constant c. Thus f is definable by one of the equations

$$f(\bar{x}_n) = x_i'\cdots'$$

or

$$f(\bar{x}_n) = 0'''\cdots'$$

A deduction of the equation $f(\nu(x_1),\ldots,\nu(x_n)) = \nu(x)$ simply consists of the $n+1$ equations which start with the original defining equation and have the variables $x_1,\ldots,x_n$ successively replaced by $\nu(x_1),\ldots,\nu(x_n)$. Thus the number of equations is bounded by a constant, $n+1$, and a fortiori by $f_1^{(n+1)}(\max\{\bar{x}_n\})$.

Now suppose $f \in \mathcal{R}_\alpha$ where $\alpha \geq 1$. If $f \in \mathcal{R}_\alpha$ because $f \in \mathcal{R}_\beta$ with $\beta < \alpha$, the claim is trivial by Lemma (3.4.viii). If f is defined by substitution from functions in $\mathcal{R}_\alpha$, the proof follows from arguments similar to, but simpler than, those used for the next case. We omit the details.

There remains the case in which f is defined by n-recursion from functions in $\mathcal{R}_\beta$, where $\alpha = \beta + \omega^{n-1}$ for some $n \geq 1$. We have the 2^n equations

$$f(\bar{\xi}_n,\bar{y}_m) = F_j , \qquad 1 \leq j \leq 2^n$$

where each equation is obtained by allowing each ξ_i to be either "$x_i + 1$" or "0". The functions $g_1,\ldots,g_r$ appearing in the formulas F_j are all bounded by $f_{1+\beta}^{(q)}$ for some q by Theorem (8.1). Define for each i, $1 \leq i \leq r$, a function $\ell_{g_i}:N^{n_i} \to N$ such that $\ell_{g_i}(x_1,\ldots,x_{n_i})$ bounds the number of equations in the deduction of the equation $g_i(\nu(x_1),\ldots,\nu(x_{n_i})) = \nu(x)$.

How do we deduce the equation $f(\overline{\nu(x_n)}, \overline{\nu(y_m)}) = \nu(x)$? (We have written $\overline{\nu(x_n)}$ for $\nu(x_1),\ldots,\nu(x_n)$.) First select the applicable equation on the basis of which x_i are zero:

$$f(\bar{\xi}_n,\bar{y}_m) = F_j$$

and then substitute the desired numerals for the $\bar{\xi}_n$ to get

$$f(\overline{\nu(x_n)}, \overline{\nu(y_m)}) = \nu(F_j)$$

where $\nu(F_j)$ is F_j with a numeral substituted for each corresponding variable in F_j. This requires $n+m+1$ equations. Then replace one of the innermost function letters by the numeral which is its value. This will require a subsidiary deduction of the proper equation. Then, similarly, replace one of the remaining innermost function letters by making a second subsidiary deduction, continue until all the function letters are removed from $\nu(F_j)$; we then have

$$f(\overline{\nu(x_n)}, \overline{\nu(y_m)}) = \nu(x)$$

for $\nu(x)$ a numeral. Thus the total number of equations is no more than

$$n+m+1+\Sigma[\ell_{h_k}(T_1,\ldots,T_{s_k})+1]$$

equations, where the sum ranges over all literal appearances in F_j of a function letter h_k in the form $h_k(T_1,\ldots,T_{s_k})$ and where $T_1,\ldots,T_{s_k}$ are formulas. Notice that we include f itself in this census of function letters, so terms of the form $\ell_f(T_1,\ldots,T_{n+m})$ will appear

in the expression above; this function letter represents the number of equations required to deduce f.

Thus we arrive at the 2^n equations

$$\ell_f(\bar{\xi}_n,\bar{y}_m) = \Sigma_j$$

These define the function ℓ_f by n-recursion from $g_1,\ldots,g_r$, $\ell_{g_1},\ldots,\ell_{g_r}$, f, and addition. Each Σ_j is a formula $n+m+1+\Sigma[\ell_{h_k}(T_1,\ldots,T_{s_k})+1]$ like the one derived above. Now consider the following modified equations:

$$\ell^*_h(\bar{\xi}_n,\bar{y}_m) = \Sigma^*_j$$

Here Σ^*_j is the formula arrived at by replacing each occurrence of $g_i(T_1,\ldots,T_{s_i})$ by $f^{(q_i)}_{1+\beta}(T_1+\cdots+T_{s_i})$, where q_i is chosen so the latter function bounds the former; likewise, $\ell_{g_i}(T_1,\ldots,T_{s_i})$ is replaced by its bound $f^{(p_i)}_{1+\beta}(T_1+\cdots+T_{s_i})$. That such bounds exist is guaranteed by Theorem (8.1) and the induction hypothesis for $\ell_{g_1},\ldots,\ell_{g_r}$. Finally, replace each occurrence of $f(T_1,\ldots,T_{n+m})$ in Σ_j by $\ell^*_f(T_1,\ldots,T_{n+m})$. By the way in which the formulas Σ^*_j were defined, ℓ^*_f is thus obtained by n-recursion from the functions $x+y$ and $f_{1+\beta}$; so by Lemma (8.2), ℓ^*_f is bounded by $f^{(q)}_{1+\beta+\omega^{n-1}}$ for some q. But we also have $\ell^*_f(\bar{x}_n,\bar{y}_m) \geq \ell_f(\bar{x}_n,\bar{y}_m)$ and $\ell^*_f(\bar{x}_n,\bar{y}_m) \geq f(\bar{x}_n,\bar{y}_m)$, for ℓ^*_f is defined from increasing functions which bound those defining ℓ_f and f, and the formulas defining ℓ^*_f are of equal or greater depths. Thus the deduction of $f(\overline{\nu(x_n)}, \overline{\nu(y_m)}) = \nu(x)$ contains no more than $f^{(q)}_{1+\beta+\omega^{n-1}}(\max\{\bar{x}_n,\bar{y}_m\})$ equations.

Next, it should be clear that there is a c so the t-th equation in the deduction will contain no more than $c^{t+\max\{\bar{x}_n,\bar{y}_m\}}$ characters. For substituting a numeral $\nu(x)$ in an equation can increase its length by at most $d\cdot\nu(x)$ for some fixed d; and each numeral which is substituted is either one of the $\bar{x}_n$, $\bar{y}_m$ or already appears as part of an earlier equation. Since $f_2(x) = 2^x$, there is an s so the total number of characters in a deduction, namely

$$f^{(q)}_{1+\alpha}(\max\{\bar{x}_n,\bar{y}_m\})\cdot c^{f^{(q)}_{1+\alpha}(\max\{\bar{x}_n,\bar{y}_m\})+\max\{\bar{x}_n,\bar{y}_m\})}$$

is bounded by $f^{(s)}_{1+\alpha}$.

Now a Turing machine can certainly carry out the deduction we have outlined. Given input $\bar{x}_n$, $\bar{y}_m$, it simply forms the equation $f(\bar{x}_n,\bar{y}_m) = F_j$, and proceeds to derive the succeeding lines of the deduction exactly as suggested above. Even if none of the deduction is erased from the tape, the total number of tape squares used need be no more than

$$x_1+\cdots+x_n+n+y_1+\cdots+y_m+m+f^{(s)}_{1+\alpha}(\max\{\bar{x}_n,\bar{y}_m\})$$

Then by exactly the same argument as that given in (5.3), the total number of steps required is no more than $f^{(p)}_{1+\alpha}(\max\{\bar{x}_n,\bar{y}_n\})$ for some p, so long as $\alpha \geq 1$. Even if $\alpha = 0$, the theorem remains true; for suppose $f(\bar{x}_n) = x_i + c$. Then f can be computed as follows: move to the left over the representation of $x_1,\ldots,x_n$, erasing the tape, until x_i is reached; pass over x_i, and then add c - 1 "1"s to its

left. Continue to the left, erasing $x_{i-1},\ldots,x_1$. Then move right again until $x_i + c$ has been passed, and stop. The total number of steps is no more than $f_1^{(p)}(\max\{\bar{x}_n\})$, for suitable p. This concludes the proof of Theorem (9.1).

A fuller discussion of the use of Turing machines to carry out deductions from recursion equations is given by Kleene [K, §69]; readers who mistrust our sketch of such mechanized deductions should consult this work.

Theorem (9.1) constitutes half of the proof that $\mathcal{R}_\alpha$, $\alpha \geq 2$, is computation time closed; the other half follows from the next theorem.

(9.2) Theorem. Let $\mathfrak{M}$ be a Turing machine which computes the function $f:N^n \to N$. Then there is an $\mathcal{R}_2$ function $TM_{\mathfrak{M}}:N^{n+1} \to N$ with the following property: if s exceeds the number of steps required to compute $f(\bar{x}_n)$ using $\mathfrak{M}$, then $f(\bar{x}_n) = TM_{\mathfrak{M}}(\bar{x}_n, s)$.

Proof. This proof can be made by giving a direct construction of $TM_{\mathfrak{M}}$, but a simpler method is to show that $\mathcal{R}_\alpha \supseteq \mathcal{L}_\alpha$ for $\alpha < \omega$, and then use Theorem (5.1) to conclude (9.2).

As we have remarked, $\mathcal{L}_0 = \mathcal{R}_0$, for each function in both classes can be written in one of the forms $f(\bar{x}_n) = x_i + c$ or $f(\bar{x}_n) = c$ for some constant c. Now suppose ~~$\mathcal{L}_\alpha \supseteq \mathcal{R}_\alpha$~~ $\mathcal{R}_\alpha \supseteq \mathcal{L}_\alpha$ for some α, $0 \leq \alpha < \omega$, and let $\underset{\sim}{P} \in L_\alpha$ be a Loop program with $\underline{Reg}(\underset{\sim}{P}) = \{X_1,\ldots,X_n\}$.

For each i, $1 \leq i \leq n$, let $f_i : N^n \to N$ be the function computed by $(\underset{\sim}{P}, \bar{X}_n, X_i)$. By definition, each $f_i \in \mathfrak{L}_\alpha$. Now consider the function

$$f_i^*(\bar{x}_n, 0) = x_i$$
$$f_i^*(\bar{x}_n, z+1) = f_i^*(f_1(\bar{x}_n), \ldots, f_n(\bar{x}_n), z)$$

which is defined by 1-recursion from $f_1, \ldots, f_n$; by the hypothesis on $f_1, \ldots, f_n$, $f_i^* \in \mathfrak{R}_{\alpha+1}$. Let $\underset{\sim}{P}^*$ be the program

```
LOOP(1) Z
  P
END
```

Now we assert that $f_i^*(\bar{x}_n, z)$ is the function computed by $(\underset{\sim}{P}^*, X_n, Z, X_i)$. This is certainly the case when $z = 0$; for then $\underset{\sim}{P}^*$ is equivalent to the empty program. If the assertion is true for initial contents of $Z = z$, let the initial contents of Z be $z+1$, and the initial contents of $\bar{X}_n$ be $\bar{x}_n$. $\underset{\sim}{P}^*$ is thus equivalent to

$$\underset{\sim}{P}_z \left\{ \begin{array}{c} \underset{\sim}{P} \\ \underset{\sim}{P} \\ \vdots \\ \underset{\sim}{P} \end{array} \right\} z .$$

The program $\underset{\sim}{P}$ leaves $f_1(\bar{x}_n), \ldots, f_n(\bar{x}_n)$ in registers $X_1, \ldots, X_n$; and by hypothesis, if the contents of $\bar{X}_n$ are $\bar{y}_n$ at the beginning of the execution of the program $\underset{\sim}{P}_z$ above, then $\underset{\sim}{P}_z$ leaves $f_i^*(y_1, \ldots, y_n, z)$

in register X_i. Thus when the initial contents of Z are z+1, $\underset{\sim}{P}^*$ leaves $f_i^*(f_1(\bar{x}_n),\ldots,f_n(\bar{x}_n),z) = f_i^*(\bar{x}_n,z+1)$ in register X_i; so $(\underset{\sim}{P}^*, \bar{X}_n, Z, X_i)$ computes $f_i^*(\bar{x}_n,z)$. If register Z is one of the X_i, say Z is register X_j, then $(\underset{\sim}{P}^*,\bar{X}_n,X_j,X_i)$ computes $f_i^*(\bar{x}_n,x_i)$.

The foregoing establishes our claim that $\mathfrak{L}_\alpha \subseteq \mathfrak{R}_\alpha$ for $\alpha < \omega$ for the functions of $\mathfrak{L}_\alpha$ computed by programs of the form

```
LOOP(1) X
  Q
END
```

When we have a program of the form

```
P
Q
```

the claim follows from the closure of $\mathfrak{R}_\alpha$ under substitution.

Thus for $\alpha < \omega$, $\mathfrak{L}_\alpha \subseteq \mathfrak{R}_\alpha$; in particular by Theorem (5.1), the desired function $TM_{\mathfrak{M}} \in \mathfrak{R}_2$ and Theorem (9.2) is proved.

Theorems (9.1), (9.2), (8.1) and (8.3) give immediately

(9.3) Theorem. For each $\alpha \geq 2$, $\mathfrak{R}_\alpha$ is computation-time closed.

IV. IDENTICAL HIERARCHIES

§10. The following very important result is now straightforward.

(10.1) Theorem. If $2 \leq \alpha < \omega^{\omega}$, $\mathcal{R}_\alpha = \mathcal{L}_{1+\alpha}$.

Proof. If $f \in \mathcal{R}_\alpha$, the time required to compute f using a Turing machine is bounded by $f^{(p)}_{1+\alpha}$ for some p. By (4.4), $\hat{f}^{(p)}_{1+\alpha}(x_1 + \cdots + x_n + 1) \geq \hat{f}^{(p)}_{1+\alpha}(\max\{\bar{x}_n\})$, and $f^{(p)}_{1+\alpha}(x_1 + \cdots + x_n + 1) \in \mathcal{L}_{1+\alpha}$. Then by the computation-time closure of $\mathcal{L}_{1+\alpha}$, $f \in \mathcal{L}_{1+\alpha}$. Conversely, if $f \in \mathcal{L}_{1+\alpha}$, the computation time of f is bounded by $f^{(q)}_{1+\alpha}$ for some q; but $f^{(q)}_{1+\alpha}(x_1 + \cdots + x_n) \in \mathcal{R}_\alpha$, so by the computation-time closure of $\mathcal{R}_\alpha$, $f \in \mathcal{R}_\alpha$.

Notice that this gives

Proof of Theorem (4.5) concluded. We showed $f_1 \in \mathcal{L}_1$ directly; $f_2 \in \mathcal{L}_2$ follows by (6.9); (8.3) and (10.1) give $f_\alpha \in \mathcal{L}_\alpha$ for $\alpha \geq 3$, yielding the theorem.

Theorem (10.1) follows from just two important characteristics of each $\mathcal{L}_{1+\alpha}$ and $\mathcal{R}_\alpha$: First, each class (for $\alpha \geq 2$) is substitution and computation-time closed; second, the two classes contain functions of the same size, in that any function in the one class is bounded by some function in the other. Thus it appears that any class of functions which has these two closure properties is essentially characterized by the size of the functions it contains.

This same approach using computation-time closure is applied below to three examples of other hierarchies mentioned in the literature; we show that each of these hierarchies is identical to a portion of the $\mathcal{L}_\alpha$ hierarchy. Not all the theorems are proved solely on the basis of computation-time closure -- sometimes ad hoc methods are easier -- but mostly we make use of this powerful closure property.

A hierarchy similar to the $\mathcal{R}_\alpha$ hierarchy where $\alpha < \omega$ was defined by Axt [~~A1, A2~~ 2]. We have

(10.2) Definition (Axt). For each α, $0 \leq \alpha < \omega$, let $\mathcal{P}_\alpha$ be the smallest class of functions satisfying

(i) The successor function $s(x) = x+1$ and the identity function $i(x) = x$ are in $\mathcal{P}_\alpha$,

(ii) If $\alpha > \beta$, $\mathcal{P}_\alpha \supseteq \mathcal{P}_\beta$,

(iii) $\mathcal{P}_\alpha$ is closed under substitution,

(iv) If f is defined by primitive recursion from functions $g,h \in \mathcal{P}_\beta$, then $f \in \mathcal{P}_\alpha$ where $\alpha = \beta+1$.

It is obvious that $\mathcal{P}$, the class of primitive recursive functions, is precisely

$$\bigcup_{\alpha < \omega} \mathcal{P}_\alpha$$

See Definition (6.10). The difference between the $\mathcal{R}_\alpha$ hierarchy for $\alpha < \omega$ and the $\mathcal{P}_\alpha$ hierarchy is that where $\mathcal{R}_\alpha$ is defined using 1-

recursion, $\mathcal{P}_\alpha$ is defined using the less general schema of primitive recursion.

It should be clear intuitively that the function $TM_{\mathfrak{M}}$ which mimics Turing machines is primitive recursive. In fact, this result follows from proofs of the Kleene Normal Form Theorem; see, for example, Kleene [K, §58] or Davis [D, p. 63]. This fact alone would put $TM_{\mathfrak{M}}$ in $\mathcal{P}_\alpha$ for each $\alpha > \alpha_o$, where α_o is a fixed ordinal less than ω. The next lemma, therefore, is of interest only because it shows α_o to be no greater than 4.

(10.3) Lemma. The function $TM_{\mathfrak{M}}$ of Theorems (5.1) and (9.2) is in $\mathcal{P}_4$. Also, each function used in the definition of $TM_{\mathfrak{M}}$ is bounded by $f_2^{(p)}$ for some p.

Proof. The proof of the lemma consists merely of an enumeration of the definitions of various functions, concluding with that for $TM_{\mathfrak{M}}$; this together with a verification that the function so enumerated have the properties ascribed to them. The verification is left mostly to the reader. Instead of giving the details here we segregate them in §11, since, as remarked above, the real content of the lemma is already obvious: that $TM_{\mathfrak{M}} \in \mathcal{P}_\alpha$ for some $\alpha < \omega$, and therefore that $TM_{\mathfrak{M}}$ can be defined using functions bounded by $f_\alpha^{(p)}$ for some α and p.

(10.4) Theorem. For $4 \leq \alpha < \omega$, $\mathcal{L}_\alpha = \mathcal{P}_\alpha$.

Proof. By Corollary (6.9) and the closure of $\mathcal{L}_\alpha$ under substitution, $\mathcal{L}_\alpha \supseteq \mathcal{P}_\alpha$ for all $\alpha > \omega$. On the other hand, since $f_1 \in \mathcal{P}_1$ and $f_{\alpha+1}$ is defined from f_α by a special case of primitive recursion, $f_\alpha \in \mathcal{P}_\alpha$ for each $\alpha \geq 1$; thus by (6.3) and (10.3), $\mathcal{P}_\alpha \supseteq \mathcal{L}_\alpha$ for $4 \leq \alpha < \omega$.

We remark that the first half of this proof, that $\mathcal{L}_\alpha \supseteq \mathcal{P}_\alpha$, could have been shown as follows: prove that each function in $\mathcal{P}_1$ is bounded by $f_1^{(p)}$ for some p. Then by Lemma (8.2), each function in $\mathcal{P}_\alpha$ is bounded by $f_\alpha^{(p)}$ for some p. Finally, Theorem (9.1) applies, *a fortiori*, to $\mathcal{P}_\alpha$ as well as $\mathcal{R}_\alpha$, since primitive recursion is a special case of 1-recursion; thus each function in $\mathcal{P}_\alpha$ can be computed in fewer than $f_\alpha^{(p)}$ steps. Then by the computation time closure of $\mathcal{L}_\alpha$, $\mathcal{L}_\alpha \supseteq \mathcal{P}_\alpha$.

Other hierarchies may be obtained by starting with a fixed set of functions and closing under substitution and limited recursion. The next example is essentially the one studied by Robbin [~~JR~~ 25]; his initial function was 2^x rather than f_0, but otherwise he used functions like f_α.

(10.5) Definition (Robbin). For each ordinal α, $\alpha < \omega^\omega$, let $\mathcal{E}_\alpha$ be the smallest class of functions satisfying

(i) $\mathcal{E}_\alpha$ contains the successor function, the function $\max(x,y)$, and f_α,

(ii) $\mathcal{E}_\alpha$ is closed under substitution,

(iii) $\mathcal{E}_\alpha$ is closed under limited recursion.

(10.6) Theorem. For $2 \leq \alpha < \omega^{\omega}$, $\mathcal{E}_\alpha = \mathcal{L}_\alpha$.

Proof. Say $\alpha \geq 2$. Then $\mathcal{L}_\alpha$ contains all the starting functions of $\mathcal{E}_\alpha$, and by (6.8) and (4.9), $\mathcal{L}_\alpha$ is closed under limited recursion and substitution. Thus $\mathcal{L}_\alpha \supseteq \mathcal{E}_\alpha$. Conversely, if $f \in \mathcal{L}_\alpha$, by Theorem (6.3) f may be written

$$f(\bar{x}_n) = M_n(e,\bar{x}_n,f_\alpha^{(p)}(\max(x_1,\ldots,\max(x_{n-1},x_n)\ldots)))$$

for some e and p. Since M_n is obtained by substitution from $TM_{\mathfrak{M}}$ for some $\mathfrak{M}$, by closure under substitution and Lemma (10.3), $TM_{\mathfrak{M}} \in \mathcal{E}_2$; for all the recursions defining $TM_{\mathfrak{M}}$ in (10.3) are bounded by $f_2^{(p)}$. Then by (6.3), $f \in \mathcal{E}_\alpha$.

Grzegorczyk [7] studied a similarly defined hierarchy $\{\mathcal{E}_\alpha^G: \alpha < \omega\}$. His starting functions, however, are somewhat different.

(10.7) Definition. For each α, $0 \leq \alpha < \omega$, let g_α be the function defined as follows:

$$g_0(x,y) = y+1$$
$$g_1(x,y) = x+y$$
$$g_2(x,y) = (x+1)\cdot(y+1)$$

For $\alpha \geq 2$,

$$g_{\alpha+1}(0,y) = g_\alpha(y+1,y+1)$$
$$g_{\alpha+1}(x+1,y) = g_{\alpha+1}(x,g_{\alpha+1}(x,y))$$

We remark that these functions were somewhat simplified by R. W. Ritchie [~~RWR2~~ 25].

(10.8) Definition (Grzegorczyk). For each α, $0 \leq \alpha < \omega$, let $\mathcal{E}^G_\alpha$ be the smallest class satisfying

(i) $\mathcal{E}^G_\alpha$ contains g_0 and g_α,

(ii) $\mathcal{E}^G_\alpha$ is closed under substitution,

(iii) $\mathcal{E}^G_\alpha$ is closed under limited recursion.

(10.9) Theorem. For $2 \leq \alpha < \omega$, $\mathcal{E}_\alpha = \mathcal{E}^G_{\alpha+1}$.

Proof. By definition,

$$g_3(0,y) = (y+2)^2$$
$$g_3(x+1,y) = g_3(x,g_3(x,y))$$

Abbreviate $(y+2)^2$ by $k(y)$. Then we assert that

$$g_3(x,y) = k^{(2^x)}(y)$$

The equation holds when $x = 0$; if $x \geq 0$,

$$\begin{aligned} g_3(x+1,y) &= g_3(x,g_3(x,y)) \\ &= k^{(2^x)}(k^{(2^x)}(y)) \\ &= k^{(2^{x+1})}(y) \end{aligned}$$

Now $k(y) = (y+2)^2 \leq y^4$ if $y \geq 2$. Therefore,

$$k^{(x)}(y) \leq y^{4^{2^x}}$$
$$\leq f_2^{(5)}(x+y) \qquad \text{if } y \geq 2$$

Then $f_2^{(7)}(x+y) \geq k^{(x)}(y)$ for all x, y. Thus $k^{(x)}(y) \in \mathcal{L}_2$, for it is definable by limited recursion (in fact limited iteration) from functions in $\mathcal{L}_2$. Then $g_3 \in \mathcal{L}_2$ by closure under substitution.

Now for $3 \leq \alpha < \omega$, $g_{\alpha+1}$ is obtained from g_α by 1-recursion. By Theorem (10.1) and the definition of $\mathcal{R}_\alpha$, $g_{\alpha+1} \in \mathcal{L}_{\alpha+1}$. This immediately proves $\mathcal{L}_\alpha \supseteq \mathcal{E}^G_{\alpha+1}$, since $\mathcal{L}_\alpha$ contains the starting functions of $\mathcal{E}^G_{\alpha+1}$ and has the same closure properties.

Now we show $g_{\alpha+2}(x,y) \geq f_\alpha^{(x)}(y)$ for $1 \leq \alpha < \omega$. For

$$g_3(0,y) = (y+2)^2 \geq f_1^{(0)}(y) = y$$
$$g_3(x+1,y) = g_3(x,g_3(x,y))$$
$$\geq f_1^{(x)}(g_3(x,y))$$
$$\geq f_1^{(2x)}(y)$$
$$\geq f_1^{(x+1)}(y) \qquad \text{if } x \geq 1$$

Even if $x = 1$, $g_3(1,y) = ((y+2)^2+2)^2 \geq f_1^{(1)}(y)$. For $1 \leq \alpha < \omega$,

$$g_{\alpha+3}(0,y) = g_{\alpha+2}(y+1,y+1) \geq f_\alpha^{(y+1)}(y+1) \geq f_{\alpha+1}^{(0)}(y)$$
$$g_{\alpha+3}(1,y) = g_{\alpha+3}(0,g_{\alpha+3}(0,y)) \geq f_\alpha^{(y+1)}(y) \geq f_{\alpha+1}^{(1)}(y)$$
$$g_{\alpha+3}(x+1,y) = g_{\alpha+3}(x,g_{\alpha+3}(x,y))$$
$$\geq f_{\alpha+1}^{(x)} f_{\alpha+1}^{(x)}(y)$$
$$\geq f_{\alpha+1}^{(x+1)}(y) \qquad \text{if } x \geq 1$$

So in particular, $g_{\alpha+1}(x,1) \geq f_\alpha(x)$. Since clearly $g_{\alpha+1}(x,y) \geq \max(x,y)$, there are functions in $\mathcal{E}^G_{1+\alpha}$ which bound $f_\alpha(\max(x,y))$. But since by Lemma (10.3), $TM_{\mathfrak{M}} \in \mathcal{E}^G_3$, by using Theorem (6.3) we have $\mathcal{E}^G_{\alpha+1} \supseteq \mathcal{L}_\alpha$ for $2 \leq \alpha < \omega$; this concludes (10.9).

§11. The major purpose of this section is merely to prove Lemma (10.3), which proof is, apparently of necessity, somewhat long-winded. A minor purpose is to demonstrate that a few other functions are in various classes $\mathcal{P}_\alpha$, so that these functions may be used in the sequel without further proof of their claimed properties.

Proof of Lemma (10.3). The construction is conceptually identical to that of (5.1), except that there a Loop program was written, and here a primitive recursive function is defined. The approach here constructs $TM_{\mathfrak{M}}$ directly, in contrast to that of Theorem (9.2), which showed that 1-recursions could perform the functions of LOOP(1) instructions, and concluded the theorem indirectly via (5.1). We remark that this latter method may, in fact, be used successfully to prove (10.3), but that without some complexities it succeeds only in showing that $TM_{\mathfrak{M}} \in \mathcal{P}_5$.

The following functions are all in $\mathcal{P}_1$.

$$x + 0 = x$$

$$x + (y + 1) = (x + y) + 1$$

For each fixed n, $n \cdot x = x + \cdots + x$

$$0 \dot{-} 1 = 0$$

$$(x + 1) \dot{-} 1 = x$$

$$p(x,0) = x$$
$$p(x,y+1) = 0$$

$$1 \dot{-} x = p(1,x)$$

We also write $\overline{sg}(x) = 1 \dot{-} x$ and $sg(x) = \overline{sg}(\overline{sg}(x))$.

Now if $g_1,\ldots,g_r$, $h_1,\ldots,h_{r+1}$ are given functions such that at most one of $g_1,\ldots,g_r$ is zero for any argument, the function f defined as follows is obtained from the given functions and $x+y$, $\overline{sg}$, and p by substitution:

$$f(\bar{x}_n) = \begin{cases} h_1(\bar{x}_n) & \text{if } g_1(\bar{x}_n) = 0 \\ \vdots & \vdots \\ h_r(\bar{x}_n) & \text{if } g_r(\bar{x}_n) = 0 \\ h_{r+1}(\bar{x}_n) & \text{otherwise} \end{cases}$$

Here f is said to be defined by cases. We have

$$f(\bar{x}_n) = p(h_1(\bar{x}_n),g_1(\bar{x}_n)) + \cdots + p(h_r(\bar{x}_n),g_r(\bar{x}_n)) + p(h_{r+1}(\bar{x}_n),\overline{sg}(g_1(x_n)) + \cdots + \overline{sg}(g_r(\bar{x}_n)))$$

Thus $\mathcal{P}_\alpha$ for $\alpha \geq 1$ is closed under definition by cases. The following functions are all defined by a single recursion and substitution from functions already defined, and thus are in $\mathcal{P}_2$:

$$x \cdot 0 = 0$$
$$x \cdot (y+1) = x \cdot y + x$$

$$x \dot{-} 0 = x$$
$$x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1$$

$$|x - y| = (x \dot{-} y) + (y \dot{-} x)$$

$$\tau(x,y) = (x+y)^2 + x$$

For each fixed $n \geq 0$,

$$n^0 = 1$$
$$n^{x+1} = n \cdot x^n$$

The following functions are all defined by a single recursion from functions already defined, and thus are in P_3;

$$rm(0, y) = 0$$
$$rm(x+1,y) = \begin{cases} 0 \text{ if } |rm(x,y)+1-y| = 0 \\ rm(x,y)+1 \text{ otherwise} \end{cases}$$

$$0/y = 0$$
$$(x+1)/y = \begin{cases} x/y+1 & \text{if } |(x/y+1)\cdot y - x - 1| \\ x/y & \text{otherwise} \end{cases}$$

$$\sqrt{0} = 0$$
$$\sqrt{x+1} = \begin{cases} \sqrt{x}+1 & \text{if } |(\sqrt{x}+1)^2 - x - 1| = 0 \\ \sqrt{x} & \text{otherwise} \end{cases}$$

$$\pi_1(x) = x \div (\sqrt{x})^2$$
$$\pi_2(x) = \sqrt{x} \div \pi_1(x)$$

The functions τ, π_1, π_2 are pairing functions with the properties $\tau(\pi_1(z), \pi_2(z)) = z$, $\pi_1(\tau(x,y)) = x$, $\pi_2(\tau(x,y)) = y$. Define, using substitutions from already-given functions,

$$(x)_Q = \pi_1(x)$$
$$(x)_L = \pi_1\pi_2(x)$$
$$(x)_S = \pi_1\pi_2\pi_2(x)$$
$$(x)_R = \pi_2\pi_2\pi_2(x)$$
$$E(x_1,x_2,x_3,x_4) = \tau(x_1,\tau(x_2,\tau(x_3,x_4)))$$

These last five functions provide the basis for the function about to be defined which mimics a Turing machine. If x_Q, x_L, x_S, x_R respectively represent the state of the Turing machine and its tape to the left of, on, and to the right of the scanned square, then $E(x_Q,x_L,x_S,x_R)$ will represent the whole current situation. Conversely, if z represents a situation, $(z)_Q$ represents the state in that situation; similarly for $(z)_L$, $(z)_S$, $(z)_R$. Let the Turing machine $\mathfrak{M}$ have u symbols $s_0,\ldots,s_{u-1}$ and v states $q_0,\ldots,q_{v-1}$; as before, the tape will be represented by a number which, in a base u notation, is an image of the corresponding portion of the tape.

Now let $Q_{\mathfrak{M}}(z)$ be that function, defined by cases, which is j whenever the quintuple $(q_{(z)_Q}, s_{(z)_S}, s_k, d, q_j)$ is a quintuple of $\mathfrak{M}$; $Q_{\mathfrak{M}}(z) = (z)_Q$ if such a quintuple does not appear. Likewise let $S_{\mathfrak{M}}(z)$ be the function which yields the next symbol to be placed on the scanned square, and let $D_{\mathfrak{M}}(s)$ be 0 if $\mathfrak{M}$ has halted, and 1 or 2 respectively if $\mathfrak{M}$ moves left or right. It should be clear that for each machine $\mathfrak{M}$, $Q_{\mathfrak{M}}$, $S_{\mathfrak{M}}$, and $D_{\mathfrak{M}}$ are defined by cases, and hence by substitution, from functions already given. Now define

$$\text{Step}_{\mathfrak{M}}(z) = \begin{cases} z & \text{if} \quad D_{\mathfrak{M}}(z) = 0 \\ E(Q_{\mathfrak{M}}(z),(z)_L/u,rm((z)_L,u),u\ (z)_R\ \ S_{\mathfrak{M}}(z)) & \text{if} \quad D_{\mathfrak{M}}(z) = 1 \\ E(Q_{\mathfrak{M}}(z),u\cdot(z)_L + S_{\mathfrak{M}}(z),rm((z)_R,u),(z)_R/u) & \text{if} \quad D_{\mathfrak{M}}(z) = 2 \end{cases}$$

Thus $\text{Step}_{\mathfrak{M}} \in \mathcal{P}_3$ and if z is the representation of a situation, $\text{Step}_{\mathfrak{M}}(z)$ is the representation of the next situation. Now say

$$\text{Result}_{\mathfrak{M}}(z,0) = z$$
$$\text{Result}_{\mathfrak{M}}(z,s+1) = \text{Step}_{\mathfrak{M}}(\text{Result}_{\mathfrak{M}}(z,s))$$

Then $\text{Result}_{\mathfrak{M}}(z,s) \in \mathcal{P}_4$; it is the situation resulting after s steps have been performed by $\mathfrak{M}$ when started with z. Define for a particular u

$$\text{Ones}(b,0) = u\cdot b + 1$$
$$\text{Ones}(b,x+1) = u\cdot 0 \quad (b,x) + 1$$

$\text{Ones} \in \mathcal{P}_2$, and when Ones(b,x) is written in base u notation, it consists of the digits of b followed by x+1 "1"s. Now let

$$\text{Input}_n(\bar{x}_n) = \text{Ones}(u\cdot\text{Ones}(\ldots u\cdot\text{Ones}(0,x_1);\ldots x_{n-1}),x_n)$$

so that, for example, $\text{Input}_2(x_1,x_2)$ consists, in base u notation, of x_1+1 "1"s, followed by 0, followed by x_2+1 "1"s. Then say

$$\text{Initial}_n(\bar{x}_n) = E(0,\ \text{Input}_n(\bar{x}_n),\ 0,\ 0)$$

$\text{Initial}_n(\bar{x}_n)$ is the encoding of the initial situation of $\mathfrak{M}$ with input $\bar{x}_n$.

Then define

$$\text{Output}^*(z,0) = 0$$

$$\text{Output}^*(z,x+1) = \begin{cases} 1 + \text{Output}^*(z,x) & \text{if } |\text{rm}(z/u^x,u) - 1| = 0 \\ \text{Output}^*(z,x) & \text{otherwise} \end{cases}$$

$$\text{Output}(z) = \text{Output}^*(z,z)$$

Output $\in \mathcal{P}_3$, and Output (z) is the number of "1"s occurring in the base u representation of z. Finally, define

$$TM_{\mathfrak{M}}(\bar{x}_n,s) = \text{Output}((\text{Result}_{\mathfrak{M}}(\text{Initial}(\bar{x}_n),s))_L)$$

$TM_{\mathfrak{M}}$ is the desired function. It should be obvious that all the functions used in the definition of $TM_{\mathfrak{M}}$ are bounded by $f_2^{(p)}$ for some p except perhaps $\text{Result}_{\mathfrak{M}}$. Even this is bounded, however; for $\text{Result}_{\mathfrak{M}}$ is in each case an encoding of four numbers. The encoding is a polynomial in the numbers encoded, and the numbers themselves represent tapes. But by the representation of a tape we have used, the size of the encoding of a tape is exponential in the length of the tape; and this length is linear in the number of steps taken. Thus $\text{Result}_{\mathfrak{M}}$ grows exponentially at worst; this makes it straightforward to show $\text{Result}_{\mathfrak{M}}$ is bounded by $f_2^{(p)}$ for some p, since $f_2(x) = 2^x$. Finally $TM_{\mathfrak{M}} \in \mathcal{P}_4$, so (10.3) is proved.

§12. Summarizing Theorems (10.1), (10.4), (10.6), and (10.8), we immediately

(12.1) Theorem. For $3 \leq \alpha < \omega$, $\mathfrak{L}_{\alpha+1} = \mathfrak{R}_\alpha = \mathfrak{P}_{\alpha+1} = \mathfrak{E}_{\alpha+1} = \mathfrak{E}^G_{\alpha+2}$.

For $2 \leq \alpha < \omega^\omega$, $\mathfrak{L}_{\alpha+1} = \mathfrak{R}_\alpha = \mathfrak{E}_{\alpha+1}$.

Therefore each of the theorems of §6 discussing $\mathfrak{L}_\alpha$ applies, mutatis mutandis, to the other classes as well. The following characterization is also interest.

(12.2) Theorem. For $\alpha \geq 2$, $\mathfrak{L}_\alpha$ is the closure under substitution of the (finite) set of functions $\{M_1, \tau, \pi_1, \pi_2, f_\alpha\}$.

Proof. τ, π_1, π_2 are the pairing functions defined in §11 with the properties $\tau(\pi_1(z),\pi_2(z)) = z$, $\pi_1(\tau(x,y)) = x$, $\pi_2(\tau(x,y)) = y$. §11 shows these functions are in $\mathfrak{E}_2$ and thus in $\mathfrak{L}_\alpha$ for $\alpha \geq 2$. Also, M_1 and f_α are in $\mathfrak{L}_\alpha$ by Theorems (5.2) and (4.5). Therefore, the closure of these functions is included in $\mathfrak{L}_\alpha$. Now if $f:N^n \to N$ is in $\mathfrak{L}_\alpha$, there is an $f^*:N \to N$ so $f^* \in \mathfrak{L}_\alpha$ and $f(\bar{x}_n) = f^*(\tau(x_1,\tau(x_2,\ldots,\tau(x_n,0)\ldots)))$; simply take $f^*(x) = f(\pi_1(x),\pi_1\pi_2(x),\ldots,\pi_1\pi_2^{(n-1)}(x))$. Then by Theorem (6.3),

$$f(\bar{x}_n) = M_1(e,\tau(x_1,\ldots,\tau(x_n,0)\ldots),f_\alpha^{(p)}(\tau(x_1,\ldots,\tau(x_n,0)\ldots))$$

for some e and p, since $\tau(x,y) \geq \max\{x,y\}$. This concludes (12.2).

Theorem (12.2) answers in the affirmative the question posed by Grzegorczyk [9, p. 41] whether his classes $\mathfrak{E}^G_\alpha$ were definable by substitution from a finite set of functions.

(12.3) Definition (Csillag-Kalmár). The class $\mathcal{E}$ of elementary functions is the least class such that

(i) $\mathcal{E}$ contains $x+y$, $x\dot{-}y$,

(ii) $\mathcal{E}$ is closed under substitution,

(iii) $\mathcal{E}$ is closed under the operations of limited sum and limited product: the operations which take $g:N^{n+1} \to N$ into $s:N^{n+1} \to N$, where

$$s(\bar{x}_n,y) = \Sigma_{i=0}^{y} g(\bar{x}_n,i)$$

and into $p:N^{n+1} \to N$ where

$$p(\bar{x}_n,y) = \prod_{i=0}^{y} g(\bar{x}_n,i)$$

Grzegorczyk was able to show that his class $\mathcal{E}_3^G$ is identical to the elementary functions [G, Theorem 4.4]. Thus, immediately,

(12.4) Theorem. $\mathcal{L}_2 = \mathcal{E}$.

Although the foregoing theorems show that all the hierarchies we have defined eventually become identical, we have not discussed much the relationships of the various classes at the bases of the hierarchies. Figure (12.5) depicts the known set-theoretic inclusions among these classes. The figure is to be read as follows. A vertical double line between two sets indicates that the set higher on the page is known to include properly the lower set, and that the proof of the inclusion is either given explicitly or follows immediately from explicit proofs. A double line one of whose members is dotted means

that there is a proper inclusion between the two sets but that we withhold the proof. The only such situations which require much thought are to show $\mathcal{P}_2 \supset \mathcal{L}_1$ and $\mathcal{L}_2 \supset \mathcal{P}_2$, especially the latter. A single solid line means an inclusion shown to exist but not known to be proper.

The horizontal dashed lines separate the sets into strata according to the functions whose rate of growth characterizes the sets in a stratum. Since each set in the stratum of f includes f, and each function in such a set has a p so $f^{(p)}$ bounds that function, it is impossible that a set in a lower stratum should include, properly or not, a set in a higher stratum. However, the inclusion relationships not explicitly indicated among the sets of a given stratum are uncertain. I conjecture that all the sets shown in the figure as incomparable are in fact incomparable, except that it seems likely that $\mathcal{R}_1 \subset \mathcal{P}_2$.

Granting that sets in different strata cannot be equal, why are all the sets in a given stratum not identical? The answer, of course, lies in their failure to be computation-time closed. This failure comes about in two ways, corresponding to the two parts of Definition (6.4). First, a function may fail to be in a class although the class contains a function bounding its computation time. This occurs because the particular functions $TM_{\mathfrak{M}}$ are not in the class; such is the case with, for example, $\mathcal{P}_0$, $\mathcal{P}_1$, $\mathcal{P}_2$ and (perhaps) $\mathcal{P}_3$. Second, there may be a function in the class whose computation time is not bounded in the class; this occurs with $\mathcal{E}_0$ and $\mathcal{E}_1$.

Conversely, given that above a certain point all the classes become computation-time closed, why should the hierarchies eventually become identical? After all, 1-recursion, for example, seems a considerably more powerful operation than primitive recursion: as we showed, with a single 1-recursion the function 2^x can be defined, while any function defined by a single 1-recursion is bounded by a linear function. This fact might lead us to suspect that one 1-recursion was worth two primitive recursions, and thus to the conjecture that $\mathcal{P}_{2\cdot\alpha} = \mathcal{R}_\alpha$ for $\alpha \geq \alpha_0$. The reason this does not occur is that while 1-recursion is more powerful than primitive recursion in terms of the size of functions definable, the functions definable by 1-recursion are larger by a fixed amount -- in fact, only exponentially larger. Once the class $\mathcal{P}_2$ is reached, functions of exponential growth are available and the advantage that 1-recursion has can be overcome by using substitution.

As we remarked in §7, there are variant definitions of the schema of n-recursion. Robbin [25] would allow a function f to be defined by

$$f(\bar{x}_n,\bar{y}_m) = F_0 \qquad \text{if } (\bar{x}_n) = (0,\ldots,0)$$
$$f(\bar{x}_n,\bar{y}_m) = F \qquad \text{if } (x_n) \neq (0,\ldots,0)$$

so long as each occurrence of f in F has the form $f(\bar{T}_n,\bar{S}_m)$, where $\bar{T}_n$, $\bar{S}_m$ are formulas and $(\bar{x}_n) > (\bar{T}_n)$. We rejected this scheme because it is in general impossible to determine by examination whether $(\bar{x}_n) > (\bar{T}_n)$ holds. On the other hand, perusal of Theorem (8.1) indicates that the only fact actually used about the occurrences of

the function being defined is that demanded by Robbin's definition: namely that the n-tuple of values occurring as the arguments of the definiendum on the right-hand side should be lexicographically less than its arguments on the left. Thus Theorem (8.1) holds as well if the definition of $\mathfrak{R}_\alpha$ is modified so that Robbin's, rather than our, use of the term n-recursion is used. Theorem (9.1) likewise does not depend on the particular form of our definition, but goes through as well with the more general one. (Actually, (9.1) needs to be supplemented with a little more argument, but we omit the details.) It follows that the modified $\mathfrak{R}_\alpha$ is identical to the actual $\mathfrak{R}_\alpha$, at least for $\alpha \geq 2$. (In order to make recursion possible at all, the initial function $x \dot{-} 1$, at least, has to be added. Otherwise it would be impossible to get off the ground, since there is no function $r \in \mathfrak{R}_0$ such that $x > r(x)$.)

On the other hand, neither do more restricted definitions of n-recursion affect the results. For example, we have allowed what Péter calls "replacement of parameters". In other words, in the schema of n-recursion $f(\bar{x}_n, \bar{y}_m)$ may be defined in terms of $f(\bar{T}_n, \bar{S}_m)$; the parameters $\bar{y}_m$ need not remain constant. It would make no difference if we required the occurrences of f on the right to be of the form $f(\bar{T}_n, \bar{y}_m)$; for in Theorem (10.3), $TM_{\mathfrak{M}}$, π_1, π_2, τ were defined without allowing replacement of parameters, and by Theorem (8.3), f_α may be defined without using parameters at all. Then by (12.2), the class $\mathfrak{R}_\alpha$ where n-recursion takes place without replacement of parameters is identical to the original $\mathfrak{R}_\alpha$. We could also require that on the right

hand side of the schema of n-recursion, the function letter being defined should not be nested within itself below the second level -- that is, that the defined letter, say f, may appear as part of an argument of f, but that these inner occurrences of f should not themselves contain f. Since in the proof of neither (8.3) nor (10.3) did we need to violate this condition, once again the classes $\mathcal{R}_\alpha$ would not be changed if the condition were imposed. However, we will show that the situation is different if no nesting whatever is allowed.

By Theorem (6.2), $\mathcal{L}_\alpha$ for $\alpha \geq 2$ is precisely the class of functions computable by a Turing machine in a number of steps bounded by $f_\alpha^{(p)}$ for some p. Consider any device or formalism whatever for computing functions, so long as this device has a notion of "step" which can be related to the steps of a Turing machine: in particular, that there are functions $k_1(x,s)$ and $k_2(x,s)$ so that if this device is given input x and halts within s steps, a Turing machine can produce the same output in $k_1(x,s)$ steps; and conversely, if some function is computed by a Turing machine, and if the function is computable at all by such a device, then when the Turing machine takes s steps for input x, the function can be computed by our device in no more than $k_2(x,s)$ steps.

It should be clear from the foregoing arguments that if $\mathcal{D}_\alpha$ is the class of functions computable by such a device within $f_\alpha^{(p)}$ of its steps, we will have the theorem $\mathcal{D}_\alpha = \mathcal{L}_\alpha$ for $\alpha > \alpha_0$ so long as k_1 and k_2 are bounded by some multiple recursive function. It seems unlikely

that any formalism for computation could be put forward seriously to which these considerations would not apply.

This reasoning above provides some justification for not giving in full detail the proofs of Theorems (5.2) and (9.1). The former theorem showed how to construct Turing machines to simulate the Loop programs, and the latter how to make Turing machines carry out deductions in the Herbrand-Gödel-Kleene formalism; in both cases, an unproved, though not unsupported, assertion was made that the simulation could be performed within a certain time. The essential content of each theorem is simply the fact that there is only a fixed time loss involved in transferring from the one formalism to the other, not what this loss factor actually is; thus verification that it is at most exponential is merely an interesting detail.

The original problem which motivated this thesis was that of relating the complexity of a program to the complexity of the function it computes. A final theorem will complete the investigation of the main question.

(12.6) Theorem. Say $\alpha \geq 2$. Given a program in L_α, or a set of recursion equations in R_α, it is effectively impossible to decide whether there is a $\beta < \alpha$ so that the program (or the equations) could be rewritten so as to give the same result, and yet be in L_β (or R_β).

Proof. A trivial modification of the constructions of §11 or Theorem (5.1) yields a function $C_{\mathfrak{M}}(x,s)$ which is one if Turing machine $\mathfrak{M}$ with input x halts in fewer than s steps, and is zero if it does not. Consider the derivations (in R_α) of the functions u_{y_0} for each y_0, where

$$u_{y_0}(x) = C_{\mathfrak{M}}(y_0,x)\cdot f_{1+\alpha}(x)$$

Let $\mathfrak{M}$ be a Turing machine such that the set $H = \{y_0 : \mathfrak{M}$ halts with input $y_0\}$ is non-recursive. If $y_0 \in H$, u_{y_0} is $f_{1+\alpha}$ almost everywhere; thus $u_{y_0} \notin \mathcal{R}_\beta$ for $\beta < \alpha$. If $y_0 \notin H$, $u_{y_0}(x) = 0$ for all x, so $u_{y_0} \in \mathcal{R}_0$. Then if we could decide whether the function u_{y_0} was in $\mathcal{R}_0$, we could decide whether $\mathfrak{M}$ halts with input y_0, and so H would be recursive, contrary to hypothesis. Clearly the same methods work also for programs in L_α.

We have thus established the following statements about Loop programs.

(1) Loop programs can compute a broad and interesting class of functions, namely the multiple recursive functions.

(2) Given a program, we can effectively find the least α for which the program is in L_α.

(3) For every program in L_α, we can effectively find a p so that with inputs $\bar{x}_n$, the program halts in fewer than $f_\alpha^{(p)}(\max\{\bar{x}_n\})$ steps.

(4) There are some programs in L_α which actually do run $f_\alpha^{(p)}$ steps.

(5) If we know a program requires fewer than $f_\alpha^{(p)}$ steps, we can effectively rewrite it so it is in L_α.

(6) However, it is in general impossible to determine whether an L_α program does in fact require at least $f_\alpha^{(p)}$ steps.

Exactly corresponding statements can be made for functions defined by multiple recursion equations. Statement (1) means that we have not proved impressive-looking theorems about an uninteresting class of objects. Statements (2)-(4) establish that the goal of relating the complexity of a program--as measured by the least α for which the program is in L_α -- to the time required to execute the program, is an aim successfully achieved. Moreover, (5) and (6) indicate, in an admittedly weak but nevertheless reasonable sense, that our measure of complexity is the best possible.

Finally, a word about practical applications. The fairest word, probably, is "none". It is true that if we restrict, say, FORTRAN by eliminating GO TO and IF statements the computation time could be predicted by examining the depth of nesting of DO loops. However, the prediction is likely to be impossibly pessimistic; for the rate of growth of even f_2 is quite large. To be told, say, that given input x, one's program will halt within

$$2^{2^{2^x}}$$

seconds, is not very useful if one wishes to use input 100 or even 2. Of course, by use of ad hoc methods the estimate could be improved,

but this is not very satisfying, since the whole point of the kind of analysis we have been doing is to avoid ad hoc methods and use a general method instead.

There is one further problem. Suppose examination of a program has revealed that the program with input x will halt within $f_5(x)$ (say) steps or seconds or whatever. We are interested in input 17 and therefore insist on inquiring as to the value of $f_5(17)$. To put it in recognizable form, we must compute $f_5(17)$ but to do this -- in fact even to write down the answer -- requires a time which is essentially $f_5(17)$ again! We would have been better off running the program itself; at least it had a chance of halting immediately.

V. RELATED TOPICS

§13. At the end of the last section several variant possibilities for a definition of n-recursion were mentioned and it was argued that all were essentially identical, in the sense that all would yield the same classes $\mathfrak{R}_\alpha$. This section studies two operations based on n-recursion which are strictly weaker than n-recursion: unnested n-recursion and limited n-recursion. We will be able to strengthen results of Péter on the two operations and to answer a question of Grzegorczyk on the latter one.

(13.1) Definition. The schema of unnested n-recursion is the same as the schema of n-recursion with the following additional restriction: if the function f is being defined, no occurrence of f on the right-hand side of the defining equations has another appearance of f in the formulas constituting its arguments.

Péter was able to show [P1, p.74] that the operation of unnested recursion does not lead out of the primitive recursive functions; that is, that the class $\mathfrak{P}$ is closed under this operation. Our analysis will confirm the result by showing in what class a function defined by unnested n-recursion from $g_1,\ldots,g_r$ must lie if $g_1,\ldots,g_r \in \mathfrak{L}_\alpha$.

(13.2) Definition. Call a 1-1 function $E_n : N^{n+1} \to N$ satisfactory for α, c if E_n is monotone increasing in each variable, and if for each i, $1 \leq i < n$, and all $\bar{x}_n$, y the following inequality holds:

$$E_n(x_1,\ldots,x_{i-1},x_i+1,x_{i+1},\ldots,x_n,y) > E_n(x_1,\ldots,x_{i-1},x_i,\underline{b},\ldots,\underline{b},y)$$

where $\underline{b} = f_\alpha^{(c)}(\max\{\bar{x}_n,y\})$.

Such an encoding E_n provides to a certain extent an order-preserving map from N^n into N for each value of the parameter y. Of course, E_n for $n > 1$ cannot be perfectly order-preserving, because the order type ω^n for $n > 1$ is strictly greater than the order type ω. A perfectly order-reserving map would have $\underline{b}$ arbitrarily large in Definition (13.2).

(13.3) Lemma. For each $n,c \geq 1$ and $\alpha \geq 2$, there is an $E_n \in \mathcal{E}_{\alpha+n-1}$ so E_n is satisfactory for α,c.

Proof. Induction on n. If $n = 1$, take $E_1(x_1,y) = 2^x \cdot 3^y$, and the lemma is immediate. When $n \geq 1$, let E_n be satisfactory for α, $c+2$ and assume $E_n(\bar{x}_n,y) \geq \max\{\bar{x}_n,y\}$; this is certainly the case when $n = 1$. Since $E_n \in \mathcal{E}_{\alpha+n-1}$, $E_n(\bar{x}_n,y) \leq f_{\alpha+n-1}^{(q)}(\max\{\bar{x}_n,y\})$ for some number q, by Theorem (8.1). Take $d = q+c+3$, write $\bar{x}_n + x$ for $x_1+x,\ldots,x_n+x$, and define

$$E_{n+1}(x,\bar{x}_n,y) \quad 2^x\cdot 3^y\cdot 5^{E_n(\bar{x}_n+x+y)^{\wedge}\cdot 7^{f^{(dx)}_{\alpha+n-1}(E_n(\bar{x}_n+x+y)^{\wedge})}}$$

Clearly $E_{n+1}(x,\bar{x}_n,y) \geq \max\{x,\bar{x}_n,y\}$, E_{n+1} is monotone increasing, and E_{n+1} is 1-1; also $E_{n+1} \in \mathfrak{E}_{\alpha+n}$ since $f^{(dx)}_{\alpha+n-1}$ is obtained from $f_{\alpha+n-1}$ by iteration. Let $\underline{b} = f^{(c)}_{\alpha}(\max\{x,\bar{x}_n,y\})$. For $1 \leq i < n$, the inequality

$$(*) \quad E_{n+1}(x,\bar{x}_{i-1},x_i+1,x_{i+1},\ldots,x_n,y) > E_{n+1}(x,\bar{x}_{i-1},x_i,\underline{b},\ldots,\underline{b},y)$$

holds. For let x be fixed. Then by hypothesis,

$$E_n(\bar{x}_{i-1}+x+y,x_i+x+y+1,x_{i+1}+x+y,\ldots,x_n+x+y,y)$$
$$> E_n(\bar{x}_{i-1}+x+y,x_i+x+y,\underline{b}^*,\ldots,\underline{b}^*,y)$$

where $\underline{b}^* = f^{(c+2)}_{\alpha}(\max\{\bar{x}_n+x+y\}) \geq f^{(c)}_{\alpha}(\max\{x,\bar{x}_n,y\})+x+y$. By definition of $\underline{b}$ and the monotonicity of E_n,

$$E_n(\bar{x}_{i-1}+x+y,x_i+x+y+1,x_{i+1}+x+y,\ldots,x_n+x+y,y)$$
$$> E_n(\bar{x}_{i-1}+x+y,x_i+x+y,\underline{b}+x+y,\ldots,\underline{b}+x+y,y)$$

Then by the monotone dependence of E_{n+1} on E_n, (*) holds for $1 \leq i < n$. It remains to be shown that

$$(**) \quad E_{n+1}(x+1,\bar{x}_n,y) > E_{n+1}(x,\underline{b},\ldots,\underline{b},y)$$

holds as well. By definition,

$$E_{n+1}(x+1,\bar{x}_n,y) = 2^{x+1}\cdot 3^y\cdot 5^{E_n(\bar{x}_n+x+y+1)^{\wedge}}\cdot 7^{f_{\alpha+n-1}^{(d(x+1))}(E_n(\bar{x}_n+x+y+1)^{\wedge})}$$

$$> 2^x\cdot 3^y\cdot 5^{E_n(\bar{x}_n+x+y)}\cdot 7^{f_{\alpha+n-1}^{(dx)}f_{\alpha+n-1}^{(d)}(E_n(\bar{x}_n+x+y+1)^{\wedge})}$$

Now for all $\bar{z}_n$, x, y

$$E_n(\bar{z}_n+x+y,y) \leq f_{\alpha+n-1}^{(q)}(\max\{\bar{z}_n+x+y\})$$

$$\leq f_{\alpha+n-1}^{(q+2)}(\max\{x,\bar{z}_n,y\})$$

Therefore, if $\underline{b} = f_{\alpha+n-1}^{(c)}(\max\{x,\bar{x}_n,y\})$, putting $\underline{b}$ for $z_1,\ldots,z_n$, we have

$$2\cdot E_n(\underline{b}+x+y,\ldots,\underline{b}+x+y)^{\wedge} \leq 2\cdot f_{\alpha+n-1}^{(q+c+2)}(\max\{x,\bar{x}_n,y\})$$

$$\leq f_{\alpha+n-1}^{(q+c+3)}(\max\{x,\bar{x}_n,y\})$$

using (3.4.iii). But since by definition $d = q+c+3$, and $\max\{x,\bar{x}_n,y\} \leq E_n(\bar{x}_n+x+y+1)$,

$$2\cdot E_n(\underline{b}+x+y,\ldots,\underline{b}+x+y)^{\wedge} \leq f_{\alpha+n-1}^{(d)}(E_n(\bar{x}_n+x+y+1)^{\wedge})$$

So by the above,

$$E_{n+1}(x+1,\bar{x}_n,y) > 2^x\cdot 3^y\cdot 5^{E_n(\bar{x}_n+x+y)^{\wedge}}\cdot 5^{f_\alpha^{(dx)}(2\cdot E_n(\underline{b}+x+y,\ldots,\underline{b}+x+y)^{\wedge})}$$

$$> 2^x\cdot 3^y\cdot 5^{E_n(\underline{b}+x+y,\ldots,\underline{b}+x+y)^{\wedge}}\cdot 7^{f_\alpha^{(dx)}(E_n(\underline{b}+x+y,\ldots,\underline{b}+x+y)^{\wedge})}$$

$$= E_{n+1}(x,\underline{b},\ldots,\underline{b})^{\wedge}$$

which is the inequality (**). Therefore E_{n+1} satisfies Definition (13.2) and (13.3) is complete.

(13.4) Lemma. Let E_n be the encoding function of Lemma (13.3). Then for each i, $1 \leq i \leq n$, the function π_i^n, where

$$\pi_i^n(E_n(\bar{x}_n, y)) = x_i$$

is in $\mathfrak{L}_2$.

Proof. Grzegorczyk [9, p.13] showed that the function $(x)_y$ is elementary, where $(x)_y$ is the exponent of the y-th prime in the prime-power decomposition of x. The 0-th prime is taken to be 2, so, for example, $(2^x \cdot 3^y)_0 = x$, $(2^x \cdot 3^y)_1 = y$. Then by Theorem (12.4), $(x)_y \in \mathfrak{L}_2$. Now $\pi_1^1(z) = (z)_0$ since $E_1(x,y) = 2^x \cdot 3^y$. If $\pi_1^n, \ldots, \pi_n^n$ are all in $\mathfrak{L}_2$,

$$\pi_1^{n+1}(z) = (z)_0$$

for $2 \leq i \leq n+1$, $\pi_i^{n+1}(z) = (\pi_{i-1}^n((z)_2) \div (z)_1) \div (z)_0$
So π_i^{n+1} is also in $\mathfrak{L}_2$.

(13.5) Theorem. Say $\alpha \geq 2$. If f is defined by unnested n-recursion from functions in $\mathfrak{L}_\alpha$, then $f \in \mathfrak{L}_{\alpha+n}$.

Proof. The function f satisfies the 2^n equations

$$f(\bar{\xi}_n, \bar{y}_m) = F_j$$

where for $1 \leq j \leq 2^n$, F_j is a formula. Each occurrence of f in one of the formulas F_j is of the form $f(\bar{S}_n, \bar{T}_m)$, where $\bar{S}_n$, $\bar{T}_m$ are formulas

not containing f. Thus these formulas represent functions in $\mathfrak{L}_\alpha$. Let c be great enough so $f_\alpha^{(c)}$ bounds all $\bar{S}_n$ appearing in any formula F_j in the context $f(\bar{S}_n,\bar{T}_m)$. Then by Lemma (13.3) choose an encoding E_n satisfactory for α, c, and let $\pi_1^n,\ldots,\pi_n^n$ be the decoding functions for E_n. Now consider the function $\hat{f}$ satisfying

$$\hat{f}(0,\bar{y}_m) = 0$$

$$\hat{f}(x+1,\bar{y}_m) = \begin{cases} \hat{F}_1 & \text{if } \pi_1^n(x+1) = \cdots = \pi_1^n(x+1) = 0 \\ \hat{F}_2 & \text{if } \pi_1^n(x+1) = \cdots = \pi_{n-1}^n(x+1) = 0,\ \pi_n^n(x+1) > 0 \\ & \vdots \\ \hat{F}_{2^n} & \text{if } \pi_1^n(x+1) > 0\ ,\ldots,\ \pi_n^n(x+1) > 0 \end{cases}$$

Here for each j, $1 \le j \le 2^n$, $\hat{F}_j$ is the formula which results from F_j by replacing each occurrence of x_i, $1 \le i \le n$, by $\pi_i^n(x+1) \div 1$, and replacing each occurrence of $f(\bar{S}_n,\bar{T}_m)$ by

$$\hat{f}(\min(E_n(\bar{S}_n,\ \max\{\bar{y}_m\})+1,x),\bar{T}_m)$$

Here, of course, min(a,b) is the smaller of a and b. We assert that these equations define a unique function $\hat{f}$, and that

$$f(\bar{x}_n,\bar{y}_m) = \hat{f}(E_n(\bar{x}_n,\max\{\bar{y}_m\}+1,\bar{y}_m)$$

The first half of the assertion is immediate by the form of the equations. For $\hat{f}(0,\bar{y}_m)$ is defined outright, and $\hat{f}(x+1,\bar{y}_m)$ is defined in terms of known functions and values of $\hat{f}$ of the form $\hat{f}(z,\bar{T}_m)$ where $z < x+1$, since on the right-hand side the first argument of $\hat{f}$ is always min(E,x) for some formula E, and $\min(E,x) \le x$.

The other half of the assertion ~~was~~ does the fact that E_n is satisfactory for α, c. We have

$$\hat{f}(E_n(0,\ldots,0,\max\{\bar{y}_m\})+1,\bar{y}_m) = \hat{F}_1$$

Since $\hat{F}_1$ contains no occurrences of $\hat{f}$ nor of x_i for any i, $\hat{F}_1 = F_1$ as a function of $\bar{y}_m$, so the assertion is true for $(\bar{x}_n) = (0,\ldots,0)$. Say for some $\bar{\xi}_n$ the assertion is true for all $(\bar{z}_n) < (\bar{\xi}_n)$ where each ξ_i is either "$x_i + 1$" or "0". Then

$$\hat{f}(E_n(\bar{\xi}_n,\max\{\bar{y}_m\})+1,\bar{y}_m) = \hat{F}_j$$

where $1 \leq j \leq 2^n$.

Now for all those ξ_i for which ξ_i = "$x_i + 1$", $\pi_i^n(E_n(\bar{\xi}_n,\max\{\bar{y}_m\}))-1 = x_i$; so $\hat{F}_j$ is the same formula as F_j, except that

$$\hat{f}(\min(E_n(\bar{S}_n,\max\{\bar{y}_m\})+1,E_n(\bar{\xi}_n,\max\{\bar{y}_m\})(+1),\bar{T}_m)$$

is substituted for $f(\bar{S}_n,\bar{T}_m)$. But since each S_i, as a function of $\bar{x}_n$, $\bar{y}_m$, is bounded by $f_\alpha^{(c)}$, and since by definition of n-recursion $\bar{S}_n < \bar{\xi}_n$, and finally since E_n is satisfactory for α, c, we have

$$E_n(\bar{S}_n,\max\{\bar{y}_m\}) < E_n(\bar{\xi}_n,\max\{\bar{y}_m\})$$

Thus those instances of $\hat{f}$ on the right might as well be of the form

$$\hat{f}(E_n(\bar{S}_n,\max\{\bar{y}_m\})+1,\bar{T}_m)$$

but since $(\bar{S}_n) < (\bar{\xi}_n)$, by the induction hypothesis the occurrences of $\hat{f}$ have the same value as

$$f(\bar{S}_n, \bar{T}_m)$$

Thus $\hat{f}(E_n(\bar{\xi}_n, \max\{\bar{y}_m\}) + 1, \bar{y}_m) = f(\bar{x}_n, \bar{y}_m)$ which complete the transfinite induction proving our assertion.

The schema of which the definition of $\hat{f}$ is an example is called course-of-values recursion with replacement of parameters. In the no-parameter case course-of-values recursion differs from primitive recursion by defining $f(x+1)$ not merely from the immediately preceding value $f(x)$, but also using several earlier values $f(r_1(x)),\ldots,f(r_k(x))$ where $r_1(x),\ldots,r_k(x) \leq x$. The term "replacement of parameters" is used because $\hat{f}(x+1, y_m)$ is defined using not only $\hat{f}(r_i(x), \bar{y}_m)$ where $r_i(x) \leq x$, but values of the form $\hat{f}(r_i(x), g_1(x, \bar{y}_m),\ldots,g_m(x, \bar{y}_m))$, so the parameters $\bar{y}_m$ do not stay fixed.

Péter [2!, §3, §5] shows how such kinds of recursions can be reduced to primitive recursion. The essential idea for course-of-values recursion can be demonstrated by an example. Let p_y be the y-th prime, where the 0-th prime is 2; as mentioned in the proof of (13.4), $(x)_y$ is the exponent of the y-th prime in the prime-power factorization of x. Say

$$g(0) = a$$

$$g(x+1) = h(x, g(r(x)))$$

where $r(x) \leq x$. Define a new function g* as follows:

$$g^*(0) = 2^a = p_0^{(a)}$$

$$g^*(x+1) = g^*(x) \cdot p_{x+1}^{h(x,(g^*(x))_{r(x)})}$$

Thus g^* is defined by primitive recursion. It should be clear that

$$g^*(x) = p_0^{g(0)} \cdot p_1^{g(1)} \cdot \ldots \cdot p_x^{g(x)}$$

and thus that

$$g(x) = (g^*(x))_x$$

Therefore if $\alpha \geq 2$, and g is defined by course-of-values recursion from functions in $\mathfrak{L}_\alpha$, $g \in \mathfrak{L}_{\alpha+1}$.

A similar argument can be applied when replacement of parameters takes place. Thus the function $\hat{f}$ defined above is in $\mathfrak{L}_{\alpha+n}$, since it is defined by course-of-values recursion with replacement of parameters from functions in $\mathfrak{L}_{\alpha+n-1}$. This completes the proof of Theorem (13.5).

(13.6) Definition. If f is defined by n-recursion from $g_1,\ldots,g_r$ and if in addition there is a function g_{r+1} so $f(\bar{x}_n,\bar{y}_m) \leq g_{r+1}(\bar{x}_n,\bar{y}_m)$, then f is said to be defined by <u>limited n-recursion</u> from $g_1,\ldots,g_r$, g_{r+1}.

Péter showed that limited n-recursion, like unnested n-recursion, does not lead out of the primitive recursive functions [~~P1~~ 21, p.113; ~~P3~~ 20].

(13.7) Theorem. Say $\alpha \geq 2$. If f is defined by limited n-recursion from functions in $\mathfrak{L}_\alpha$, then $f \in \mathfrak{L}_{\alpha+n}$.

Proof. In the proof of Theorem (9.1), which showed that each function f in $\mathcal{R}_\alpha$ could be computed by a Turing machine within time $f^{(p)}_{1+\alpha}$, we arrived at the following intermediate result: if f is defined by n-recursion from $g_1,\ldots,g_r$, the number of equations $\ell_f(\bar{x}_n,\bar{y}_m)$ required to deduce the equation $f(\overline{\nu(x_n)}, \overline{\nu(y_m)}) = \nu(x)$ is given by another n-recursion as follows

$$\ell_f(\bar{\xi}_n,\bar{y}_m) = \Sigma_j$$

where each Σ_j is a sum of the form

$$n+m+1 + \Sigma[\ell_{h_k}(T_1,\ldots,T_{s_k})+1]$$

and the sum ranges over literal appearances of function letters h_k in F_j. Now all the functions $g_1,\ldots,g_r$, $\ell_{g_1},\ldots,\ell_{g_r}$, f in each Σ_j are bounded by $f^{(c)}_{1+\alpha}$ for some c, so each function $h_k(T_1,\ldots,T_{s_k})$ occurring in each Σ_j may be replaced by $f^{(c)}_\alpha(T_1 + \cdots + T_{s_k})$. Here h_k ranges over $g_1,\ldots,g_r$, $\ell_{g_i},\ldots,\ell_{g_r}$, f; the function f can be included because of the bounding condition. But now observe that the function ℓ^*_f which results bounds ℓ_f, and ℓ^*_f is defined by an unnested n-recursion from functions in $\mathcal{E}_\alpha$ if $\alpha \geq 2$. Then ℓ^*_f is bounded by $f^{(d)}_{\alpha+n}$ for some d, by Theorems (13.5) and (8.1); the rest of Theorem (9.1) goes through unchanged, and if $\alpha \geq 1$, f can be computed by a Turing machine in time $f^{(e)}_{\alpha+n}$ for some e, so $f \in \mathcal{E}_{\alpha+n}$, and Theorem (13.7) is proved.

It might be thought that Theorems (13.5) and (13.7) are pessimistic; although we have shown that if f is defined by limited or

unnested n-recursion from functions in $\mathcal{L}_\alpha$, then $f \in \mathcal{L}_{\alpha+n}$, perhaps in fact we always have $f \in \mathcal{L}_\alpha$. This is not the case.

(13.8) Theorem. Say $\alpha \geq 2$. Then for each $n \geq 1$ there is a function $T \in \mathcal{L}_{\alpha+n} - \mathcal{L}_{\alpha+n-1}$ such that T is definable by a single instance of limited, unnested n-recursion from functions in $\mathcal{L}_\alpha$.

Proof. Recall from Theorem (5.2) that $M_1(e,y,z)$ is the function computed by the Loop program with Gödel number e, when the input is y and the program halts in fewer than z steps. $M_1 \in \mathcal{L}_2$ by Theorem (5.2). Now define by unnested n-recursion from sg, M_1, f_α:

$$T(0,\ldots,0,e,y,z) = sg(M_1(e,y,z))$$
$$T(\bar{\xi}_{n-1},x_n+1,e,y,z) = T(\bar{\xi}_{n-1},x_n,e,y,f_\alpha(z))$$
$$T(\bar{\xi}_{n-2},x_{n-1}+1,0,e,y,z) = T(\bar{\xi}_{n-2},x_{n-1},z,e,y,1)$$
$$T(\bar{\xi}_{n-3},x_{n-2}+1,0,0,e,y,z) = T(\bar{\xi}_{n-3},x_{n-2},z,0,e,y,1)$$
$$\vdots$$
$$T(x_1+1,0,\ldots,0,e,y,z) = T(x_1,z,0,\ldots,0,e,y,1)$$

As usual, the equations containing a ξ are schematic: $\bar{\xi}_r$ represents all the r-tuples obtained by letting each ξ_i be either "x_i+1" or "0". Then it is easy to verify that

$$T(\bar{x}_n,e,y,z) = sg(M_1(e,y,f_{\alpha+n-1}^{(x_1)}\, f_{\alpha+n-2}^{(x_2)} \cdots f_\alpha^{(x_n)}(z)))$$

We omit the details. Now, recalling that $sg(0) = 0$, $sg(x+1) = 1$, we have $T(\bar{x}_n,e,y,z) \leq 1$; so this is an instance of limited n-recursion.

Now let

$$U(e,y) = T(e,0,\ldots,0,e,y,1)$$
$$= sg(M_1(e,y,f^{(e)}_{\alpha+n-1}(y)))$$

Then, by the argument of (6.12), U is universal for the characteristic functions of $\mathfrak{L}_{\alpha+n-1}$; so U and hence T cannot be members of $\mathfrak{L}_{\alpha+n-1}$. But $T \in \mathfrak{L}_{\alpha+n}$ by Theorem (13.5) or by Theorem (.317). This completes (13.8).

Grzegorczyk [9, p.41] posed the question: does the operation of limited 2-recursion lead outside the class $\mathcal{E}^G_{\alpha+1}$? Since $\mathfrak{L}_\alpha = \mathcal{E}^G_{\alpha+1}$, Theorem (13.8) answers the question affirmatively.

Theorems (13.5), (13.7) and (13.8) have to be modified slightly when n-recursion takes place without replacement of parameters, and since this restriction is imposed by Péter and probably is implied by Grzegorczyk, the situation is worth some discussion. However, detailed proof will not be given.

In the case of limited n-recursion, the constructions may be modified as follows.

(13.9) Theorem. Say $\alpha \geq 2$. If f is defined by limited n-recursion without replacement of parameters from functions in $\mathfrak{L}_\alpha$, $f \in \mathfrak{L}_{\alpha+n-1}$; and for $n > 1$, there is an f so defined such that $f \in \mathfrak{L}_{\alpha+n-1} - \mathfrak{L}_{\alpha+n-2}$.

Proof. The first half follows by observing that the function $\hat{f}$ occurring in the proof of Theorem (13.5) is defined, in this case, by a limited course of values recursion without replacement of parameters from functions in $\mathfrak{L}_{\alpha+n-1}$. This can be converted to a limited recursion from functions in $\mathfrak{L}_{\alpha+n-1}$, and we know already by Theorem (6.8) that $\mathfrak{L}_{\alpha+n-1}$ is closed under this operation. It follows that $f \in \mathfrak{L}_{\alpha+n-1}$.

On the other hand, in the proof of Theorem (13.8) only the parameter z (the last argument of T) is subject to replacement. Thus the definition of T can be regarded as an (n+1)-recursion without replacement of parameters, simply by considering z a recursion variable rather than a parameter. Thus for $n > 1$ the function T can be defined by limited n-recursion, and $T \in \mathfrak{L}_{\alpha+n-1} - \mathfrak{L}_{\alpha+n-2}$. This completes (13.9).

The same method can be adapted to show

(13.10) Theorem. If for $\alpha \geq 2$ and $n > 1$ f is defined from functions $g_1,\ldots,g_r \in \mathfrak{L}_\alpha$ by unnested n-recursion without replacement of parameters, then $f \in \mathfrak{L}_{\alpha+n-1}$, and there is an f so defined such that $f \in \mathfrak{L}_{\alpha+n-1} - \mathfrak{L}_{\alpha+n-2}$.

The proof is omitted. The requirement $n > 1$ must be included since unnested 1-recursion without replacement of parameters is essentially primitive recursion, which is known to be capable of defining functions in $\mathfrak{L}_{\alpha+1} - \mathfrak{L}_\alpha$ from functions in $\mathfrak{L}_\alpha$.

§14. The study of the several hierarchies carried out in Chapters II-IV depended heavily on the properties of computation-time closure, closure under substitution, and in some cases closure under limited recursion. Since the same classes arose again and again in spite of the various ways in which the hierarchies were defined, it is natural to wonder to what extent the closure properties alone characterize a set of functions. Might it be, for example, that every class of multiple recursive functions with the above closure properties and containing (say) $\mathfrak{L}_2$ must be either one of the $\mathfrak{L}_\alpha$ or the whole class of multiple recursive functions? This possibility seems, if anything, enhanced by the existence of two ways of refining the $\mathfrak{L}_\alpha$ hierarchy studied by R. W. Ritchie and by Cleave.

Ritchie [~~RWR1~~ 24] defines a hierarchy $\{F_i : i \in N\}$ whose union he calls the predictably computable functions, and which turns out to be precisely the set of elementary functions; that is $\mathfrak{L}_2$. F_0 may be taken to be the linear functions; then F_{i+1} is defined as the smallest class of functions computable on a Turing machine whose ~~as~~consumption of tape is bounded by a function in F_i. The input and output of the Turing machine are by Ritchie's convention in a binary encoding; it can be shown that $2^x \in F_1 - F_0$, $2^{2^x} \in F_2 - F_1$, etc. The term "predictably computable" arises from the fact that if a function is in F_i, it can be computed using an amount of tape bounded -- that is, predictable -- by a function in F_{i-1}, which in turn is predictable by a function in F_{i-2}, and so forth.

In characterizing his classes F_i, Ritchie showed that each class had the property of computation-time closure. Each class F_i is closed also under "explicit transformations" -- equivalent to Definition (4.8), parts (i) and (ii) -- but, as the example above indicates, F_i fails to be closed under composition. However, F_i is closed under a certain limited form of composition which is sufficient to prove the desired results. The F_i individually fail also to be closed under limited recursion, although of course their union is closed.

An analogous hierarchy $\{E_\alpha : \alpha < \omega^2\}$ was considered by Cleave [5 CL]. He considers a kind of simple computer, the "unlimited register machine" of Shepherdson and Sturgis [26 SS]. The classes E_α arise by restricting the number of "transfer" or "jump" instructions carried out in a given computation. Thus E_0 is the class of functions computable in such a way that the number of transfer instructions executed is bounded by a constant; given E_α, $E_{\alpha+1}$ is the class of functions computable in such a way that the number of transfers is bounded by a function in E_α. The analogy here with the predictably computable functions is evident. At limit ordinals, the functions obtained so far are collected:

$$E_{\omega\cdot(r+1)} = \bigcup_{s \in N} E_{\omega\cdot r+s}.$$

Thus at limit ordinals, the effect is that of defining a new machine whose elementary operations consist of those functions definable in a class with a smaller ordinal.

Cleave is able to show that if the basic arithmetic operations of his machine allow addition, multiplication, and testing for zero, then $E_{\omega\cdot s} = \mathscr{E}^G_{s+2}$ for each $s \in N$, $s \geq 1$; that is, $E_{\omega\cdot s} = \mathfrak{L}_{s+1}$. Thus, part of the $\mathfrak{L}_\alpha$ hierarchy appears again; but once more the classes E_α fail in general to be closed under limited recursion and substitution. For a fixed s, the classes $E_{\omega\cdot s+r}$ are analogous in several ways to the Ritchie classes F_r, but apparently it is not true that $R_r = E_r$.

The work of Ritchie and of Cleave tends to reinforce the naturalness of the $\mathfrak{L}_\alpha$ in two ways. First, certain of the $\mathfrak{L}_\alpha$ classes reappear in each of these contexts; and second, both methods of refining the hierarchy result in classes which fail to have the attractive closure properties of the $\mathfrak{L}_\alpha$.

Nevertheless, the hierarchy $\mathfrak{L}_\alpha$ can be refined in such a way that the closure properties of $\mathfrak{L}_\alpha$ are retained. In fact, we will demonstrate the existence of an almost embarrassing richness of classes which are closed under limited recursion, substitution, and have the property of computation-time closure. There are several preliminary definitions and theorems.

We recall some useful notation common in the literature.

(14.1) Definition. If $\mathfrak{M}$ is a Turing machine, let e be the Gödel number of $\mathfrak{M}$. Then $\varphi_e : N^n \to N$ is the (partial) function computed by $\mathfrak{M}$ with input $\bar{x}_n$, and $\Phi_e : N^n \to N$ is the (partial) function giving the exact number of steps required for $\mathfrak{M}$ to halt with input $\bar{x}_n$. Also,

say that e is the index of f when f is the function φ_e.

This definition assumes an arithmetization of Turing machines which has not been carried out. However, the task has often been performed in the literature; see the remarks following Theorem (14.3).

(14.2) Definition. If P is a predicate, we will say that P is a member of a class of functions if a representing function for P is in the class; that is, a function f so $f(\bar{x}_n) = 1$ if $P(\bar{x}_n)$ is true, $f(\bar{x}_n) = 0$ if $P(\bar{x}_n)$ is false. If P is a predicate $[P(\bar{x}_n)]$ will denote the representing function of P.

Then, for example, $x = y$ is a predicate in $\mathfrak{L}_2$, because $[x = y] = \overline{sg}|x - y| = 1 \dot{-} |x - y|$.

(14.3) Theorem. The predicate given by $[\Phi_e(\bar{x}_n) = y]$ is in $\mathfrak{L}_2$ as a function of e, $\bar{x}_n$, and y; there is an $\mathfrak{L}_2$ function U_n so if $z \geq \Phi_e(\bar{x}_n)$, $U_n(e,\bar{x}_n,z) = \varphi_e(\bar{x}_n)$.

Proof. As we have mentioned, to consider statements of this type requires an arithmetization of Turing machines. It is well known, however, that there exists a Gödel numbering of Turing machines such that for each n, $T_n \in \mathfrak{L}_2$, where $T_n(e,\bar{x}_n,y) = 1$ if the Turing machine with Gödel number e, given input $\bar{x}_n$, halts in precisely y steps, and

$T_n(e,\bar{x}_n,y) = 0$ otherwise. Then, of course, $[\Phi_e(\bar{x}_n) = y] = T_n(e,\bar{x}_n,y)$. Likewise $U_n \in \mathcal{L}_2$; here U_n is precisely analogous to the function LP_n of Theorem (5.2). See, for example, Davis [D, pp.56-62]. Davis notes only that his construction yields primitive recursive functions, but since it is readily shown that all the recursions are bounded by $f_2^{(p)}$ for some p, it is immediate that T_n and U_n are in $\mathcal{L}_2$. Kleene [K, §§56-57] carries out a similar arithmetization for recursion equations.

A property of certain functions which is very important in the sequel is

(14.4) Definition. A recursive function f is <u>honest</u> whenever the number of steps required to compute f is bounded by an $\mathcal{L}_2$ function composed with f; that is, if $f(\bar{x}_n) = U_n(e,\bar{x}_n,r(\bar{x}_n,f(\bar{x}_n)))$ for some number e and some $r \in \mathcal{L}_2$.

The term "honest" is used because if f is honest, the value of $f(\bar{x}_n)$ accurately reflects the difficulty of computing $f(\bar{x}_n)$. No disapproval of functions which are not honest is implied. In fact highly dishonest functions, for example complicated characteristic functions, are rather more interesting than honest functions; much of the time required to compute an honest function is spent merely in writing down the result.

We note that a somewhat broader definition of honest was used by Robbin [R].

A useful alternate characterization of honesty is the following.

(14.5) Theorem. A recursive function f is honest if and only if $[f(\bar{x}_n) = y]$ is in $\mathcal{L}_2$.

Proof. First assume $[f(\bar{x}_n) = y]$ is in $\mathcal{L}_2$. Hence we have a Turing machine which computes $[f(\bar{x}_n) = y]$ within $f_2^{(c)}(\max\{\bar{x}_n, y\})$ steps for some constant c. Consider the following procedure to compute f: given input $\bar{x}_n$, write $\bar{x}_n, 0$ on the tape and use the given machine to compute $[f(\bar{x}_n) = 0]$; if this is 0, add 1 to the 0 at the end of the $\bar{x}_n$ and compute $[f(\bar{x}_n) = 1]$; if this too is zero, continue testing $[f(\bar{x}_n) = 2]$, etc. until a true predicate is found. This requires on the order of

$$\sum_{i=0}^{f(x)} f_2^{(c)}(\max\{\bar{x}_n, i\})$$
$$\leq (1 + f(x)) \cdot f_2^{(c)}(\max\{\bar{x}_n, f(\bar{x}_n)\})$$

steps. But the latter function is in $\mathcal{L}_2$ as a function of $\bar{x}_n$ and $f(\bar{x}_n)$, so f is honest.

Conversely, if f is honest, there exist $e \in N$ and $r \in \mathcal{L}_2$ so that

$$[f(\bar{x}_n) = y] = [U_n(e, \bar{x}_n, r(\bar{x}_n, y)) = y] \cdot \sum_{i=0}^{r(\bar{x}_n, y)} [\Phi_e(\bar{x}_n) = i]$$

where the right-hand side is in $\mathcal{L}_2$ because $\mathcal{E} = \mathcal{L}_2$ ($\mathcal{E}$ is the class of elementary functions) and by definition, $\mathcal{E}$ is closed under limited sum.

Although we have called computation-time closure a closure property, it differs from other such properties, for example, closure under limited recursion, in an important sense. When we speak of the least class of functions containing given functions and closed under limited recursion, we refer to a well defined entity, namely the intersection of all classes of functions which contain the given functions and which are closed under limited recursion. That this intersection is indeed closed under limited recursion follows from the fact that given three functions ~~this~~ those is at most one function defined from them by limited recursion.

On the other hand, it is not clear that there must be any smallest class containing given functions and having the property of computation-time closure. For if a function is in such a class, the class is required to contain also some bound on the computation time of the function. But there are many such bounds, corresponding to many ways to compute the function, and there is no guide to selecting which bound should be included in the class The problem is quite real; indeed, one of the results in the sequel implies that there are sets of functions such that there is no smallest computation-time closed set containing the given set.

The next theorem relates the notions of computation-time closure and closure under limited recursion; thus it allows us to generate computation-time closed classes having desired properties without encountering the problem just discussed. The theorem also

provides an alternative proof of the closure of the classes $\mathcal{L}_\alpha$ under limited recursion.

(14.6) Definition. If a class of functions is such that every member of the class is bounded by an increasing function in the class, the class is called monotone. Also, for brevity, a class which is closed under substitution and is computation-time closed is called fully closed.

(14.7) Theorem. Let $\mathcal{C}$ be a class of functions containing $\mathcal{L}_2$. Then $\mathcal{C}$ is monotone and fully closed if and only if $\mathcal{C}$ is the closure under limited recursion and substitution of a set of honest functions.

Proof. First assume $\mathcal{C}$ is monotone and fully closed, and say

$$f(\bar{x}_n,0) = g(\bar{x}_n)$$
$$f(\bar{x}_n,y+1) = h(\bar{x}_n,y,f(\bar{x}_n,y))$$
$$f(\bar{x}_n,y) \leq b(\bar{x}_n,y)$$

where $g,h,b \in \mathcal{C}$. Define

$$f^*(\ell,\bar{x}_n,0) = \min(\ell,U_{n+1}(e_g,\bar{x}_n,\ell))$$
$$f^*(\ell,\bar{x}_n,y+1) = \min(\ell,U_{n+2}(e_h,\bar{x}_n,y,f^*(\ell,\bar{x}_n,y),\ell))$$
$$f^*(\ell,\bar{x}_n,y) \leq \ell$$

where e_g and e_h are indices for g and h, and Φ_{e_g} and Φ_{e_h} are bounded by functions in $\mathcal{C}$. Notice that $f^* \in \mathcal{L}_2$. Now by the hypotheses on $\mathcal{C}$, let $b'(\bar{x}_n,y) \geq b(\bar{x}_n,y)$, and say b' is in $\mathcal{C}$ and increasing. Likewise, let $\ell \in \mathcal{C}$ be an increasing function with $\ell(\bar{x}_n,y) \geq \Phi_{e_g}(\bar{x}_n)$, $\ell(\bar{x}_n,y) \geq \Phi_{e_h}(\bar{x}_n,y,b'(\bar{x}_n,y))$, and $\ell(\bar{x}_n,y) \geq b(\bar{x}_n,y)$. Then it is easy to show that $f(\bar{x}_n,y) = f^*(\ell(\bar{x}_n,y),\bar{x}_n,y)$, so $f \in \mathcal{C}$. That is, $\mathcal{C}$ is closed under limited recursion; in fact, $\mathcal{C}$ is the closure under substitution and limited recursion of its honest functions.

Conversely, let $\mathcal{C}$ be the closure under limited recursion and substitution of any set of honest functions. If $f \in \mathcal{C}$, $m_f \in \mathcal{C}$ where

$$m_f(\bar{x}_n,0) = 0$$

$$m_f(\bar{x}_n,y+1) = \begin{cases} y+1 & \text{if } f(\bar{x}_n, m_f(\bar{x}_n,y)) \leq f(\bar{x}_n,y+1) \\ m_f(\bar{x}_n,y) & \text{otherwise} \end{cases}$$

$$m_f(\bar{x}_n,y) \leq y$$

This function has the property that $f(\bar{x}_n, m_f(\bar{x}_n,y))$ is not less than any of $f(\bar{x}_n,0),\ldots,f(\bar{x}_n,y)$; so $f(x_n, m_f(\bar{x}_n,y)) + y$ is in $\mathcal{C}$, is strictly increasing in y, and bounds f. By applying ~~of~~ the same technique to the other variables of f, one finds a function in $\mathcal{C}$ which bounds f and is strictly increasing in each variable; thus $\mathcal{C}$ is monotone.

Now since $\mathcal{C}$ contains $\mathcal{L}_2$, all the honest functions of $\mathcal{C}$ have computation times bounded by functions in $\mathcal{C}$. It is easy to show that if f is defined by substitution from functions whose computation times are bounded in $\mathcal{C}$ then the computation time of f is likewise

bounded in $\mathfrak{C}$. There remains the case in which f is defined by limited recursion from g, h, b as above.

Given $\bar{x}_n$, y, there is an obvious method for using a Turing machine to compute f: first compute $g(\bar{x}_n) = f(\bar{x}_n,0)$; use this result to compute $h(\bar{x}_n, 0, f(\bar{x}_n,0)) = f(\bar{x}_n,1)$; continue until $f(\bar{x}_n,y)$ has been computed. If e_g and e_h are indices for g and h, the number of steps is bounded by

$$\Phi_{e_g}(\bar{x}_n) + \Sigma_{i=1}^{y} \Phi_{e_h}(\bar{x}_n,i-1,f(\bar{x}_n,i-1)) + f_2^{(a)}(\max\{\bar{x}_n,y\})$$

where the last term is added to cover the cost of bookkeeping. Since $\mathfrak{C}$ contains $\mathfrak{L}_2$, and Φ_{e_g}, Φ_{e_h}, and f are bounded by monotone functions in $\mathfrak{C}$, this number of steps is less than some function of $\mathfrak{C}$. By the containment of $\mathfrak{L}_2$ in $\mathfrak{C}$, $\mathfrak{C}$ has the function $TM_{\mathfrak{M}}$ for each $\mathfrak{M}$; thus $\mathfrak{C}$ is computation-time closed. (A more detailed discussion of the use of Turing machines to compute functions defined by limited recursion is presented by Ritchie [~~RWR1~~ 24].)

(14.8) Theorem. If f is honest and increasing, the iterate $f^{(y)}(x)$ is also honest.

Proof. Define

$$k(y,z) = \prod_{i=0}^{y} p_i^z$$

That is,

$$k(y,z) = p_0^z \cdot p_1^z \cdot \ldots \cdot p_y^z$$

Then let

$$It(y,w) = \prod_{i=1}^{y} [f((w)_{i-1}) = (w)_i]$$

Then

$$[f^{(y)}(x) = z] = sg \sum_{w=0}^{k(y,z)} \{[(w)_0 = x] \cdot [(w)_y = z] \cdot It(y,w)\}$$

which shows $[f^{(y)}(x) = z]$ is in $\mathfrak{L}_2$ and $f^{(y)}(x)$ is thus honest.

(14.9) Theorem. Let f be a recursive function. Then there is an honest increasing function h so $h(x) \geq f(x)$; and if $\alpha \geq 2$ and $f \in \mathfrak{L}_\alpha$, h can be chosen so $h \in \mathfrak{L}_\alpha$.

Proof. With our conventions for input and output, if φ_e is any recursive function,

$$\varphi_e(x) \leq \Phi_e(x) + x + 1$$

Let e be an index of f; then use the construction of Theorem (14.7) to find an $\mathfrak{L}_2$ function m so $\Phi_e(m(x))$ is not less than any of $\Phi_e(0), \Phi_e(1), \ldots, \Phi_e(x)$. Take $h(x) = \Phi_e(m(x)) + x + 1$; h is increasing, and

$$[h(x) = y] = [y > x] \cdot \sum_{z \leq x} \{[z = m(x)] \cdot [\Phi_e(z) = (y \dot{-} x) \dot{-} 1]\}$$

so h is honest. Moreover, if $f \in \mathfrak{L}_\alpha$, e can be chosen so $\Phi_e \in \mathfrak{L}_\alpha$.

(14.10) Definition. If f is any strictly increasing function, $f:N \to N$, then the inverse of f, written f^{-1}, is the function defined by: $f^{-1}(x)$ is the largest y such that $f(y) \leq x$ if such a y exists; $f^{-1}(x)$ is 0 if y does not exist.

(14.11) Theorem. If $f:N \to N$ is a strictly increasing function, f^{-1} has the following properties:

(i) f^{-1} is nondecreasing, $f^{-1}f(x) = x$, and if $x \geq f(0)$, $ff^{-1}(x) \leq x$;

(ii) If f is recursive, f^{-1} is recursive;

(iii) If f is honest, $f^{-1} \in \mathfrak{L}_2$.

Proof. If $x \geq f(0)$, there exists a y so $f(y) \leq x$, by taking $y = 0$. Since f is increasing, there are at most finitely many y so $f(y) \leq x$, so f^{-1} is well-defined. Now $f^{-1}f(x) = x$, since $f^{-1}f(x)$ is the largest y so $f(y) \leq f(x)$; by the increasing property of f, $x = y$. Also $ff^{-1}(x) \leq x$ if $x \geq f(0)$. For in this case there is a y so $f(y) \leq x$; $ff^{-1}(x) \leq x$ is immediate by definition. Also, f^{-1} is nondecreasing; for by definition, $f(f^{-1}(x+1)+1) > x+1$. But if $f^{-1}(x+1)+1 \leq f^{-1}(x)$, since $f^{-1}(x) \leq x$ we have a contradiction. This completes (14.11.i).

If f is honest, $[f(y) = x]$ is in $\mathfrak{L}_2$. Say

$$f^{-1}(0) = 0$$

$$f^{-1}(x+1) = \begin{cases} f^{-1}(x)+1 & \text{if } [f(f^{-1}(x)+1) = x+1] \\ f^{-1}(x) & \text{otherwise} \end{cases}$$

$$f^{-1}(x) \leq x$$

Since f^{-1} is defined by limited recursion from functions in $\mathfrak{L}_2$, $f^{-1} \in \mathfrak{L}_2$. We omit the proof that f^{-1} so defined in the inverse of f. Even if f is only recursive, $[f(y) = x]$ is recursive and the above limited recursion defines f^{-1} effectively, so f^{-1} is recursive. This completes (14.11.ii) and (14.11.iii).

(14.12) Definition. Let r be an increasing, recursive function, and let f, g be functions. If for all y and x, $x \geq r(y)$ implies $f^{(y)}(x) < g(x)$, write $f \prec_r g$. If there exists an r so $f \prec_r g$, we will also say $f \prec g$.

It should be obvious that $\prec$ is a partial ordering on functions. It is easily shown that $f_\alpha \prec_r f_{\alpha+1}$ where $r(y) = 2 \cdot y + 1$. If f and g are recursive, it is an interesting question whether the proposition, "for all y, g majorizes $f^{(y)}$", implies the existence of a recursive r so $f \prec_r g$.

The next lemma shows $\prec$ provides a dense ordering on the multiple recursive functions; it is basic for the major results of both this section and §15.

(14.13) Lemma. Suppose f and h are increasing, honest functions and $f \prec h$. Then there exists an increasing, honest g so $f \prec g \prec h$.

Proof. Say $f \prec_r h$. By Theorem (14.9), take s honest, increasing, and such that $s(x) \geq hr(x^2)$. Let $t = s^{-1}$ and observe that $t(x) \leq \sqrt{(r^{-1}h^{-1}(x))}$. Now define g:

Need r recursive!

$$g(x) = f^{(t(x)+1)}(x)$$

Since t is nondecreasing and f is increasing, g is increasing; g is honest since $t \in \mathfrak{L}_2$ by Theorem (14.11.iii) and $[f^{(y)}(x) = z] \in \mathfrak{L}_2$ by (14.8).

Next, $f <_{r_1} g$ via $r_1 = s$. For if $x \geq s(y)$, $t(x)+1 > y$ so $f^{(y)}(x) < g(x)$.

For typographical convenience, write $F(y,x)$ for $f^{(y)}(x)$. We assert that

$$g^{(y+1)}(x) \leq F((y+1)\cdot(tg^{(y)}(x)+1),x)$$

If $y = 0$, the assertion is immediate by definition of g. If $y \geq 0$, assume the assertion for y; then

$$\begin{aligned} g^{(y+2)}(x) &= gg^{(y+1)}(x) \\ &= F(tg^{(y+1)}(x)+1, g^{(y+1)}(x)) \\ &\leq F(tg^{(y+1)}(x)+1, F((y+1)\cdot(tg^{(y)}(x)+1),x)) \\ &\leq F(tg^{(y+1)}(x)+1, F((y+1)\cdot(tg^{(y+1)}(x)+1),x)) \\ &= F(tg^{(y+1)}(x)+1+(y+1)\cdot(tg^{(y+1)}(x)+1),x) \\ &= F((y+2)\cdot(tg^{(y+1)}(x)+1),x) \end{aligned}$$

and the assertion is proved. Take $r_2(y) = r((y+1)^2)$; now $f^{(0)}(x) = x = g^{(0)}(x) < h(x)$ if $x \geq r(0)$. Since $r_2(0) > r(0)$, $g^{(0)}(x) < h(x)$ if $x \geq r_2(0)$.

If $g^{(y)}(x) < h(x)$ whenever $y \geq r_2(y)$, by substitution in the inequality asserted above

$$
\begin{aligned}
g^{(y+1)}(x) &< F((y+1)\cdot(\mathrm{th}(x)+1),\ x) && \text{for } x \geq r_2(y) \\
&\leq F((y+1)\cdot(\sqrt{(r^{-1}(x))}+1),x) \\
&\leq F(r^{-1}(x),x) && \text{for } r \geq r_2(y+1) \\
&= f^{(r^{-1}(x))}(x)
\end{aligned}
$$

The third line follows since it is easily shown that $(y+1)\cdot(\sqrt{(r^{-1}(x))}+1) < r^{-1}(x)$ when $x \geq (y+2)^2$; but since r is increasing, $r_2(y+1) = r((y+2)^2) \geq (y+2)^2$. Then since $rr^{-1}(x) \leq x$ if $x \geq r(0)$, and since for all y $r_2(y) \geq r(0)$, $f^{(r^{-1}(x))}(x) < h(x)$ by the assumption on r. Therefore $g^{(y)}(x) < h(x)$ for $x \geq r_2(y)$; that is, $g <_{r_2} h$. Lemma (14.13) is proved.

(14.14) Theorem. Say $2 \leq \beta < \alpha < \omega^{\omega}$. Then there is a family D of classes of functions such that

(i) If $\mathcal{D} \in D$, $\mathcal{L}_\beta \subset \mathcal{D} \subset \mathcal{L}_\alpha$;

(ii) D has a dense, linear ordering under set inclusion;

(iii) If $\mathcal{D} \in D$, $\mathcal{D}$ is fully closed and closed under limited recursion;

(iv) If $\mathcal{D}_1$, $\mathcal{D}_2 \in D$ and $\mathcal{D}_1 \subset \mathcal{D}_2$, $\mathcal{D}_2$ contains a universal function for $\mathcal{D}_1$.

Proof. By Theorem (14.9), choose an honest, increasing function $t_\beta \in \mathcal{L}_\beta$ so $t_\beta(x) \geq f_\beta(x)$. Let $t_\alpha(x) = t_\beta^{(x)}(x)$; then t_α is increasing, $t_\alpha \in \mathcal{L}_\alpha$, and, by Theorem (14.8), t_α is honest. Finally, $t_\beta <_r t_\alpha$ via $r(y) = y+1$.

Proof. The construction of Theorem (14.14) yields an infinite set T of functions all of which are honest and increasing, and such that T is linearly ordered by $<$; also, $T \subset \mathfrak{L}_\alpha$ and each member of T increases faster than any member of $\mathfrak{L}_\beta$.

For each $t \in T$, let d_t be the function

$$d_t(x) = \begin{cases} t(x) & \text{if } x \in \underline{\text{range}}\ t \\ 0 & \text{otherwise} \end{cases}$$

Each d_t is honest, for

$$[d_t(x) = y] = \begin{cases} [t(x) = y] & \text{if } \Sigma_{i=0}^{x}[t(i) = x] \neq 0 \\ [y = 0] & \text{otherwise} \end{cases}$$

Then for each $t \in T$, let the set $\mathscr{J}_t$ be in I, where $\mathscr{J}_t$ is the closure under limited recursion and substitution of $\{d_t, f_\beta, \max, s\}$. As before, s is the successor function. (14.15.i) and (14.15.iii) are immediate.

Now consider a set $\mathscr{J}_t \in I$. We assert that each function $f \in \mathscr{J}_t$ has constants a_f and b_f so that $n_f(y) \leq f_\beta^{(b_f)} t^{-1}(y)$, where $n_f(y)$ is the function giving the number of n-tuples $(\bar{x}_n)$ with $\max\{\bar{x}_n\} \leq y$ and such that $f(\bar{x}_n) > f_\beta^{(a_f)}(y)$. That is, $n_f(y)$ is the cardinality of the set

$$\{\bar{x}_n : \max\{\bar{x}_n\} \leq y \ \&\ f(\bar{x}_n) > f_\beta^{(a_f)}(y)\}$$

Such constants certainly exist for f_β, max, and s; and the cardinality of

$$\{x: x \leq y \ \& \ d_t(x) > f_\beta(y)\}$$

is no more than $t^{-1}(y) + 1 \leq f_\beta t^{-1}(y)$. If f is defined by limited recursion from functions for which the assertion above holds, the assertion holds for f immediately by the bounding condition. If f is defined by substitution, f may be written

$$f(\bar{x}_n) = h(g_1(\bar{x}_n),\ldots,g_m(\bar{x}_n))$$

and where we may assume there are suitable constants $a_h, b_h, a_1, b_1, \ldots, a_m, b_m$ so that the assertion holds for $h, g_1, \ldots, g_m$. By taking some of $g_1, \ldots, g_m$ to be constant or identity functions, any instance of substitution may be written in this form.

Let $a_g = \max\{a_1,\ldots,a_m\}$, $b_g = \max\{b_1,\ldots,b_m\}$; and say $a = a_h \cdot a_g$. If $\max\{\bar{x}_n\} \leq y$, $f(\bar{x}_n) > f_\beta^{(a)}(y)$ only if all of $g_1(\bar{x}_n),\ldots,g_m(\bar{x}_n)$ are bounded by $f_\beta^{(a_g)}(y)$ but $h(g_1(\bar{x}_n),\ldots,g_m(\bar{x}_n)) > f_\beta^{(a)}(y)$, or one or more of $g_1(\bar{x}_n),\ldots,g_m(\bar{x}_n)$ exceeds $f_\beta^{(a_g)}(y)$. In other words, the number of n-tuples $(\bar{x}_n)$ with $\max\{\bar{x}_n\} \leq y$ and such that $f(\bar{x}_n) > f_\beta^{(a)}(y)$ is no more than $n_f(y)$, where

$$n_f(y) = f_\beta^{(b_h)} t^{-1} f_\beta^{(a_g)}(y) + \Sigma_{i=1}^{m} f^{(b_i)} t_\beta^{-1}(y)$$

Now by examination of the construction of the function $t \in T$ in Lemma (14.13), for each such t there is a non-decreasing function r so $t(x) = f_\beta^{(r(x))}(x)$. Then for any c,

$$\begin{aligned} f_\beta^{(c)} t(x) &= f_\beta^{(c+r(x))}(x) \\ &\leq f^{(c+rf_\beta^{(c)}(x))}(x) \\ &= t f_\beta^{(c)}(x) \end{aligned}$$

By applying t^{-1} to both sides of this inequality,

$$t^{-1}\, f_\beta^{(c)} t(x) \leq f_\beta^{(c)}(x)$$

Putting $t^{-1}(y)$ for x,

$$t^{-1}\, f_\beta^{(c)}(y) \leq f_\beta^{(c)} t^{-1}(y) \qquad \text{for} \quad y \geq t(0)$$

By choosing b sufficiently large, then

$$t^{-1}\, f_\beta^{(a_g)}(y) \leq f_\beta^{(b)} t^{-1}(y)$$

But then

$$n_f(y) \leq f_\beta^{(b_h+b)} t^{-1}(y) + m \cdot f_\beta^{(b_g)} t^{-1}(y) \leq f_\beta^{(b_f)} t^{-1}(y)$$

for suitable b_f; this concludes the proof of our assertion. The next step in (14.15) is to show that if t, u $\in$ T and $t \prec u$, there are no numbers a, b so n_{d_t} is bounded by $f_\beta^{(b)} u^{-1}$; we conclude that $d_t \notin \mathcal{J}_u$. Because $f_\beta \prec t$, for each number a there is a constant c so the cardinality of

$$\{x: x \leq y \;\&\; d_t(x) > f_\beta^{(a)}(y)\}$$

is greater than $t^{-1}(y) \dot{-} c$. Given any b, choose y_o so $u(y_o) \geq t^{(2)}(y_o)$ + c and $t(y_o) \geq f_\beta^{(b)}(y_o)$; this is possible because $f_\beta \prec t \prec u$. Then

$$\begin{aligned} n_{d_t}(u(y_o)) &\geq t^{-1}(u(y_o)) \dot{-} c \\ &\geq t(y_o) + c \dot{-} c \\ &\geq f_\beta^{(b)}(y_o) \\ &= f_\beta^{(b)} u^{-1}(u(y_o)) \end{aligned}$$

Therefore, for no a, b is n_{d_t} bounded by $f^{(b)}{}_u{}^{-1}$; hence $d_t \notin \mathcal{I}_u$. On the other hand, every function in $\mathcal{I}_t$ is bounded by $t^{(c)}$ for some c; but if $t \prec u$, d_u is not bounded by $t^{(c)}$ for any c. Thus $d_u \notin \mathcal{I}_t$; and so $\mathcal{I}_u$ and $\mathcal{I}_t$ are setwise incomparable, proving (14.15.ii). (14.15.iv) will follow immediately from the next theorem, which is interesting in its own right.

(14.16) Theorem. Let $\mathfrak{C}$ and $\mathfrak{D}$ be fully closed classes containing $\mathfrak{L}_2$ with $\mathfrak{C} - \mathfrak{D} \neq \emptyset$. Then there is a characteristic function in $\mathfrak{C} - \mathfrak{D}$.

Proof. Pick an arbitrary constant a and let $f^*(x,b)$ be the smallest number k so k is unequal to all of $U_1(0,x,f_2^{(a)}(\max\{x,b\}))$, $U_1(1,x,f_2^{(a)}(\max\{x,b\}))$,...,$U_1(x,x,f_2^{(a)}(\max\{x,b\}))$. It should be clear that $f^* \in \mathfrak{L}_2$ and $f^*(x,b) \leq x+2$.

Now take any function $g \in \mathfrak{C} - \mathfrak{D}$, and let $h \in \mathfrak{C}$ be a bound on the computation time of g. Then put $f(x) = f^*(x,h(x))$; $f \in \mathfrak{C}$ by closure under substitution. We assert that if e_1 is any index for f, $\Phi_{e_1}(x) > f_2^{(a)}(\max\{x,h(x)\})$ for almost all x. For if this is false, there is an $x \geq e_1$ so $\Phi_{e_1}(x) \leq f_2^{(a)}(\max\{x,h(x)\})$; then $f(x) \neq U_1(e_1,x,f_2^{(a)}(\max\{x,h(x)\}))$ by definition of f, but $f(x) = U_1(e,x,f_2^{(a)}(\max\{x,h(x)\}))$ by the properties of U_1. This is a contradiction.

Now let $c(x,y) = [f(x) = y]$; $c \in \mathfrak{C}$ is immediate. Consider the following procedure for computing f, given c: successively compute

$[f(x) = 0]$, $[f(x) = 1],\ldots,[f(x) = x+2]$; one of these must yield 1 as a result. Let $f(x)$ be the y for which $[f(x) = y] = 1$. If e_2 is an index for c, the number of steps required is bounded by

$$f_2^{(d)}(\max\{x, \sum_{y \leq x+2} \Phi_{e_2}(x,y)\})$$

for some fixed d. Then if $\Phi_{e_2}(x,y) < h(x)$ for infinitely many x, the number of steps required to compute f is less than $f_2^{(d)}(\max\{x,(x+3)\cdot h(x)\})$ for infinitely many x. But we showed above that any machine for f must require at least $f_2^{(a)}(\max\{x,h(x)\})$ steps for almost all x, where a was arbitrary; we conclude by this reductio that every index e_2 for c has $\Phi_{e_2}(x,y) > \overset{h}{H}(x)$ for almost all x. Then if $c \in \mathfrak{D}$, a function bounding h would also be in $\mathfrak{D}$ by the full closure property of $\mathfrak{D}$, and hence g would be in $\mathfrak{D}$; but $g \in \mathfrak{C} - \mathfrak{D}$, so $c \notin \mathfrak{D}$. Then also $c^* \in \mathfrak{C} - \mathfrak{D}$ where $c^*(x) = c(\pi_1(x),\pi_2(x))$, for $c(x,y) = c^*(\tau(x,y))$, which proves (14.16).

Theorems (14.14) and (14.15) may reasonably be interpreted as casting doubt on the naturalness of the classes $\mathfrak{L}_\alpha$. For if, as implied by Theorem (14.14), there is a dense, linearly ordered hierarchy of classes of functions whose union is the multiple recursive functions such that all the classes have the same strong closure properties as the $\mathfrak{L}_\alpha$, the $\mathfrak{L}_\alpha$ themselves no longer seem so significant. For example, given the dense hierarchy, we can find a subordering of any denumerable order type we please. Theorem (14.14) even implies the existence of uncountably many fully closed classes of multiple recursive functions with a linear set theoretic ordering. Likewise, Theorem (14.15) can

be extended to yield uncountably many incomparable classes which are fully closed.

One development is possible which would restore the importance of the classes $\mathfrak{L}_\alpha$. Suppose $\mathfrak{C}$ is any fully closed class of multiple recursive functions. Say $\mathfrak{C}[0] = \mathfrak{C}$; given $\mathfrak{C}[\alpha]$ for $\alpha < \omega^\omega$, let $\mathfrak{C}[\alpha + \omega^n]$ for $n \geq 0$ be the closure under substitution of $\mathfrak{C}[\alpha]$ and all functions obtainable by $(n+1)$-recursion from functions in $\mathfrak{C}[\alpha]$. Then it seems possible that for any such $\mathfrak{C}$, there are α, $\beta < \omega^\omega$ such that $\mathfrak{C}[\alpha] = \mathfrak{L}_\beta$; that is, by applying multiple recursion several times to any "in-between" class $\mathfrak{C}$, eventually one of the $\mathfrak{L}_\alpha$ classes is reached. This possibility has not been seriously investigated except by trying the few examples which suggested it.

§15. Blum has recently published some remarkable results on the complexity of recursive functions [4]. One of his theorems is the following.

(15.1) Speed-up Theorem (Blum). Let r be a total recursive function, $r: N^2 \to N$. Then there is a total recursive characteristic function f with the property that to every index i for f there corresponds another index j for f such that for almost all x, $\Phi_i(x) > r(x, \Phi_j(x))$.

Blum's theory is machine independent. For example, he does not demand of the step-counting function $\Phi_j(x)$ that it actually give the steps used by the j-th machine with input x, but merely that for each j and x that $\Phi_j(x)$ converge if and only if $\varphi_j(x)$ converges, and that the predicate $[\Phi_j(x) = z]$ be recursive. As we have seen, if Φ_j measures the actual number of steps taken by a Turing machine, $[\Phi_j(x) = z]$ is in $\mathfrak{L}_2$, that is, an elementary predicate.

The Speed-up Theorem implies, for example, that there is a recursive function f so if φ_i computes f, there is another index j for f so that $\Phi_j(x) < 2^{\Phi_i(x)}$ for almost all x; that is, given any machine for f there is another machine which computes f and halts in only about the logarithm of the number of steps required by the first machine. However, as Blum shows, the faster machines cannot in general be discovered effectively.

Blum also proved a more powerful version of the Speed-up Theorem which shows that the r of Theorem (15.1) can be as large as Φ_i itself.

(15.2) Super Speed-up Theorem (Blum). Let g be a total recursive function. Then there exists a recursive characteristic function f such that

(i) If i is an index for f, $\Phi_i(x) > g(x)$ for almost all x;

(ii) To any index i for f, there corresponds an index j for f such that $\Phi_i(x) > \Phi_j\Phi_j(x)$ for almost all x

This theorem has the Speed-up Theorem as an immediate consequence.

It might be thought that the function f whose computation can be sped up must be enormously more complex than the r of Theorem (15.1) or the g of Theorem (15.2). By agreeing that $\Phi_j(x)$ has its natural interpretation, the methods of Lemma (14.13) may be adopted to prove a stronger version of the Super Speed-up Theorem in which f is, in a reasonable way, only slightly more complex than g, and that there are functions lying very low in the $\mathcal{L}_\alpha$ hierarchy whose computation can be sped up quite considerably.

(15.3) Theorem. Let g be an honest, increasing function with $g(x) \geq 2^x$, and r be an unbounded, nondecreasing recursive function. Then there is a recursive characteristic function f such that:

(i) If i is any index for f, $\Phi_i(x) > g(x)$ for almost all x;

(ii) There is an index j for f such that
$\Phi_j(x) \leq g^{(r(x))}(x)$ for almost all x;

(iii) For each index i for f, there is another index j for f such that for all c, $\Phi_i(x) > \Phi_j^{(c)}(x)$ for almost all x.

Proof. The proof consists of a main Lemma (15.4), which is a strengthening of Blum's lemma for the Super Speed-up Theorem [B[4], p.330], then the construction of f, and finally several lemmas on the properties of f. Two of these latter are slightly modified versions of Lemmas 1 and 2 used by Blum [B[4], p.327].

(15.4) Lemma. Let g and r satisfy the hypotheses of Theorem (15.3). Then there is a function $q_s(x)$ such that

(i) For each s and all x, $q_s(x+1) > q_s(x)$;

(ii) For each s and all x, $q_{s+1}(x) \leq q_s(x)$;

(iii) For all s and c and almost all x, $q_{s+1}^{(c)}(x) < q_s(x)$;

(iv) For all s and almost all x, $g^{(r(x))}(x) \geq q_s(x) > g(x)$;

(v) As a function of s and x, $q_s(x)$ is honest.

Proof of Lemma. By (14.9), choose an honest increasing function b so for all x $b(x) \geq x^2$, $b(x) \geq g^{(x)}(x)$, $b(x) \geq g^{(r(x)+1)}(x)$, and such that $b^{-1}(x) < r(x)$ for almost all x. Then let $t_s(x) = b^{(2s+2)}(x)$. As a function of s and x, $t_s(x)$ is honest by Theorem (14.11). Then

$t_s^{-1}(x)$ is in $\mathfrak{L}_2$, where by $t_s^{-1}(x)$ we mean the greatest y so $t_s(y) \leq x$ if y exists; $t_s^{-1}(x) = 0$ if it does not. Then say

$$q_s(x) = g^{(t_s^{-1}(x)+1)}(x)$$

Parts (i), (ii), (iv), and (v) of the Lemma are immediate. Now if $x \geq t_s(y)$, $g^{(y)}(x) < g^{(y+1)}(x) \leq q_s(x)$; thus ~~$g < t_s q_s$~~. Since $t_{s+1}(x) = bbt_s(x) = bt_s b(x) \geq q_s t_s(x^2)$, by the argument of Lemma (14.13),

$$q_{s+1} \; ? \; q_s$$

where $r_s(y) = t_s((y+1)^2)$. This proves part (iii) and thus Lemma (15.4).

The proof of (15.3) now continues with the construction of f. First we define a function f_{uv} and an associated set K_{uv} each of which depend on the input x. Given x, compute $f_{uv}(x)$ and $K_{uv}(x)$ as follows.

Set $K_{uv}(-1) = \emptyset$

If $x \geq 0$, find the smallest k, $k \leq x$,

so that all of the following are true:

(a) $x < v$, or $x \geq v$ and $k \leq u$;

(b) $\Phi_k(x) \leq q_k(x)$;

(c) $k \notin K_{uv}(x-1)$.

If such a k exists, set $K_{uv}(x) = K_{uv}(x-1) \cup \{k\}$, and put $f_{uv}(x) = 1 \div \varphi_k(x)$; if no such k exists, put $K_{uv}(x) = K_{uv}(x-1)$, $f_{uv}(x) = 0$.

Then the function f of Theorem (15.3) is f_{00}. We can also construct f_{uv} more formally, so that it is clearer that it has the properties we ascribe to it. To simplify the presentation, we will use certain notations not yet introduced. If $P(\bar{x}_n,y)$ is a predicate, the predicates $(\exists\, y)_{< x} P(\bar{x}_n,y)$ and $(\forall\, y)_{< x}\, P(\bar{x}_n,y)$ are obtained from P by limited quantification; the meaning of the former, for example, is $(\exists\, y)\ (y < x\ \&\ P(\bar{x}_n,y))$. The predicates of $\mathfrak{L}_2$ are closed under limited quantification; this follows immediate from the closure of $\mathfrak{L}_2$ under limited sum and limited product. The predicates of $\mathfrak{L}_2$ are also closed under the Boolean operations &, $\vee$ and $\sim$. Finally, $\mathfrak{L}_2$ is closed under limited minimization: obtaining $\mu k_{< x}\, P(\bar{x}_n,k)$ from a predicate P, where the notation means the least k such that $k < x$ and $P(\bar{x}_n,k)$ is true; or zero if there is no such k. The closure of $\mathfrak{L}_2$ under this operation follows directly from the closure of $\mathfrak{L}_2$ under limited recursion. Grzegorczyk discusses all these operations more fully [G].

Construct functions c, K*, f* as follows.

$$\begin{aligned} c(u,v,b,K,x) = \mu k_{\leq x+1} \{ & ((x < v) \vee (x \geq v\ \&\ k \geq u)) \\ & \&\ (\exists\, y)_{\leq b}([q_k(x) = y]\ \&\ (\exists\, w)_{\leq y}[\Phi_k(x) = w]) \\ & \&\ (\forall\, i)_{\leq x}\, [(K)_i \neq k+1] \\ & \vee [k = x+1]\} \end{aligned}$$

$$K^*(u,v,b,0) = \begin{cases} 1 & \text{if } c(u,v,b,1,0) = 1 \\ 2 & \text{otherwise} \end{cases}$$

$$K^*(u,v,b,x+1) = \begin{cases} K^*(u,v,b,x) & \text{if } c(u,v,b,K^*(u,v,b,x),x+1) = x+2 \\ K^*(u,v,b,x) \cdot p_{x+1}^{1+c(u,v,b,K^*(u,v,b,x),x+1)} & \text{otherwise} \end{cases}$$

$$K^*(u,v,b,x) \leq \prod_{i \leq x} p_i^{x+1}$$

$$f^*(u,v,b,x) = \begin{cases} 0 & \text{if } (K^*(u,v,b,x))_x = 0 \\ 1 \dot{-} U_1((K^*(u,v,b,x))_x \dot{-} 1,x,b) & \text{otherwise} \end{cases}$$

$$f_{uv}(x) = f^*(u,v,q_0(v) + q_u(x),x)$$

If, in the informal algorithm, $K_{uv}(x) - K_{uv}(x-1) = \{k\}$, we will say φ_k is spoiled for x in K_{uv}. Notice that if φ_k is spoiled for x in K_{uv}, then $f_{uv}(x) = 1 \dot{-} \varphi_k(x) \neq \varphi_k(x)$. (Blum uses the term "cancelled".)

It is clear that f* defined above is elementary. It is not so clear that $f_{uv}(x) = f^*(u,v,q_0(v) + q_u(x),x)$; nevertheless, we will omit the detailed proof. The representation of K_{uv} used by K* is as follows: if φ_k has been spoiled for some $y \leq x$ in K_{uv}, then the prime-power decomposition of $K^*(u,v,q_0(v) + q_u(x),x)$ contains a factor p_y^{k+1} and no other prime in the factorization has an exponent k+1. If φ_k has not been spoiled for any $y \leq x$ in K_{uv}, the prime-power factorization of $K^*(u,v,q_0(v) + q_u(x),x)$ contains no prime with an exponent of k+1. The crucial fact which assumes that f* has the correct properties is that in the calculation of f_{uv} for $u \leq v$, we are called upon

to know the values of $q_u(x)$, $q_{u+1}(x),\ldots,q_x(x)$ if $x > v$, and $q_0(x)$, $q_1(x),\ldots,q_x(x)$ if $x \leq v$. In view of (15.4.i) and (15.4.ii), all of these are bounded by $q_0(v) + q_u(x) = b$. Then since $k \leq x$, the truth value of $(\exists y)_{\leq b}([q_k(x) = y] \;\&\; (\exists w)_{\leq y}[\Phi_k(x) = w])$ is the same as that of $\Phi_k(x) \leq q_k(x)$.

(15.5) Lemma (Blum). For each u there exists a v such that $f_{uv} = f_{00} = f$.

Proof. For each u there are only finitely many k with $k < u$, and in particular there are only finitely many φ_k with $k < u$ ever spoiled for any x in K_{00}. Choose $v > u$ so v bounds all x such that $k < u$ and φ_k is spoiled for x in K_{00}.

Now $K_{00}(-1) = K_{uv}(-1) = \emptyset$; assume $x \geq 0$ is the least number so $K_{00}(x) \neq K_{uv}(x)$. Then clauses (b) in the definitions of $K_{00}(x)$ and $K_{uv}(x)$ have identical truth values for each k; likewise for clauses (c). But then if $K_{uv}(x) \neq K_{00}(x)$, it must be that $x \geq v$ and there is a $k < u$ so $\Phi_k(x) \leq q_k(x)$ and $k \notin K_{uv}(x-1) = K_{00}(x-1)$. But then φ_k is spoiled for x in $K_{00}(x)$, and by choice of v, if $k < u$ and φ_k is spoiled for x in K_{00} then $v > x$. Since we proved above that $x \geq v$, we have a contradiction. Therefore, we have shown $K_{uv}(x) = K_{00}(x)$ for all x, and thus $f_{uv} = f_{00}$.

(15.6) Lemma (Blum). If $\varphi_i = f$, then $\Phi_i(x) > q_i(x)$ for almost all x.

Proof. Suppose for contradiction that there are infinitely many $x = x_0, x_1, \ldots$ such that $\Phi_i(x_j) \leq q_i(x_j)$. Since i is a fixed number there are only finitely many k with $k \leq i$; therefore, there must be a number x which bounds all those y for which there exists a $k < i$ such that φ_k is spoiled for y in K_{00}. If x_n is the least of $x_0, x_1, \ldots$ which exceeds this x, the conjunction of clauses (a), (b) and (c) in the definition of $f_{00} = f$ is true for $x = x_n$, $k = i$ and for no smaller k. Thus φ_i is spoiled for x_n. But then $\varphi_i(x_n) \neq f(x_n)$, a contradiction.

(15.7) Lemma. There is an increasing $\mathcal{L}_2$ function h so for each u, there is an index j for f such that

$$hq_u(x) > \Phi_j(x)$$

for almost all x.

Proof. Recall that $f_{uv}(x) = f^*(u,v,q_0(v) + q_u(x),x)$ and $f^* \in \mathcal{L}_2$. By the honesty of q_0 and q_u, there are $\mathcal{L}_2$ functions t_0 and t_1 so the computation times of q_0 and q_u are bounded by $t_0(v,q_0(v))$ and $t_1(u,x,q_u(x))$ respectively; also, the computation time of $f^*(u,v,z,x)$ is bounded by $t_f(u,v,z,x)$ and t_f is in $\mathcal{L}_2$. Thus there is an $\mathcal{L}_2$ function t so t is increasing and $t(u,v,q_0(v) + q_u(x),x)$ bounds the computation time of $f^*(u,v,q_0(v) + q_u(x),x)$. Let $h(z) = t(z,z,2\cdot z,z)$. Given u, use Lemma (15.5) to find a v so $f_{uv} = f_{00} = f$, and let j be the index of f_{uv}. Then

$$t(u,v,q_0(v) + q_u(x),x) > \Phi_j(x)$$

for all x. But for large x, $q_u(x)$ exceeds all of x, u, v, and $q_0(v)$; therefore for large x,

$$\begin{aligned} h(q_u(x)) &= t(q_u(x),q_u(x),2\cdot q_u(x),q_u(x)) \\ &\geq t(u,v,q_0(x)+ q_u(x),x) \\ &> \Phi_j(x) \end{aligned}$$

which completes Lemma (15.7).

Proof of Theorem (15.4) (concluded). By Lemma (15.6), if i is an index for f then for almost all x,

$$\Phi_i(x) > q_i(x)$$

By Lemma (15.4.iii), for every d and almost all x,

$$\Phi_i(x) > q_{i+1}^{(d)}(x)$$

Since by hypothesis $q_{i+1} > 2^x$, if h is any $\mathcal{L}_2$ function, d can be made large enough so

$$q_{i+1}^{(d)}(x) > q_{i+1}\, h\, q_{i+1}\, h \cdots q_{i+1} h(x)$$

In particular, if h is the function of Lemma (15.5), use the lemma to find an index j for f such that $h(q_{i+1}(x)) > \Phi_j(x)$ for almost all x; then

$$\Phi_i(x) > \Phi_j^{(c)}(x)$$

for each c and almost all x. This completes (15.3.iii). (15.3.i) follows from Lemmas (15.6) and (15.4.iv); (15.3.ii) follows from Lemmas (15.7) and (15.4.iv). Thus (15.3) is complete.

Theorem (15.3) is stronger than Blum's Super Speed-up Theorem in two ways: first, as mentioned, we have shown that functions capable of being sped up lie low in the $\mathcal{L}_\alpha$ hierarchy; for example in $\mathcal{L}_3$, by taking $g(x) = 2^x$ in (15.3). Second, given an index i for f, we have an index j for f so $\Phi_i(x) > \Phi_j^{(c)}(x)$ for every c and almost all

x; Blum's theorem had a j_1 so $\Phi_i(x) > \Phi_{j_1}^{(2)}(x)$, a j_2 so $\Phi_i > \Phi_{j_2}^{(3)}(x)$,... .

Thus, as an example, let $g(x) = 2^x$ in Theorem (15.3). Then there exists an f so if i is any index for f, there exists another index j for f such that all of the inequalities

$$\Phi_j(x) < \log \Phi_i(x)$$
$$\Phi_j(x) < \log\log \Phi_i(x)$$
$$\Phi_j(x) < \log\log\log \Phi_i(x)$$
$$\vdots$$

hold for almost all x. Also, $f \in \mathcal{L}_3$, and, in fact, if r is a non-decreasing, recursive, unbounded function, no matter how slowly increasing, then f can be computed in approximately $f_2^{(r(x))}(x)$ steps.

Then by Lemma (14.13), there is a set T of honest, increasing functions, all of which bound t_β, all of which are in $\mathfrak{L}_\alpha$, and which has a dense, linear ordering under $\prec$. For each function $t \in T$ with $t \neq t_\alpha$ and $t \neq t_\beta$, put $\mathfrak{D}_t$ in D, where $\mathfrak{D}_t$ is the closure under substitution and limited recursion of $\{t, s, \max\}$; here s is the successor function, $s(x) = x+1$. Each $\mathfrak{D}_t \in D$ is fully closed by Theorem (14.7). Clearly every function in $\mathfrak{D}_t$ is bounded by $t^{(c)}$ for some fixed c, so by definition of $\prec$, if $t_1 \prec t_2$ then $\mathfrak{D}_{t_1} \subset \mathfrak{D}_{t_2}$; thus D is densely ordered. Finally, if $\mathfrak{D} \in D$, $\mathfrak{L}_\beta \subset \mathfrak{D} \subset \mathfrak{L}_\alpha$ and for each $t \in T$, $t_\beta \prec t$.

Finally, if $\mathfrak{D}_{t_1}$, $\mathfrak{D}_{t_2} \in D$ and $\mathfrak{D}_{t_1} \subset \mathfrak{D}_{t_2}$, $t_1 \prec t_2$; thus $U_1(e,x,t_2(x)+e)$ is universal for the one-place functions of $\mathfrak{D}_{t_1}$, by exactly the same arguments as Theorem (6.12). This proves (14.14.iv).

(14.15) Theorem. Say $2 \leq \beta < \alpha < \omega^\omega$. Then there is an infinite family I of classes of functions such that

(i) If $\mathscr{I} \in I$, $\mathfrak{L}_\beta \subset \mathscr{I} \subset \mathfrak{L}_\alpha$;

(ii) The members of I are pairwise incomparable under set inclusion;

(iii) If $\mathscr{I} \in I$, $\mathscr{I}$ is fully closed and closed under limited recursion;

(iv) If $\mathscr{I}_1$, $\mathscr{I}_2 \in I$ and $\mathscr{I}_1 \neq \mathscr{I}_2$, there is a characteristic function in $\mathscr{I}_1 - \mathscr{I}_2$.

REFERENCES

[1] Ackermann, W., "Zum Hilbertschen Aufbau der reelen Zahlen" Math. Annalen 99 (1928), pp. 118-133.

[2] Axt, P., "Iteration of primitive recursion" Notices Amer. Math. Soc. 10, 1 (1963), Abstract 597-182.

[3] _____, "Enumeration and the Grzegorczyk hierarchy" Zeit. f. math. Logik u. Grundlagen d. Math. 9 (1963), pp. 53-65.

[4] Blum, M., "A machine-independent theory of the complexity of recursive functions" J. Assoc. Comp. Mach. 14, 2 (1967), pp. 322-336.

[5] Cleave, J. P., "A hierarchy of primitive recursive functions" Zeit. f. math. Logik u. Grundlagen d. Math. 9 (1963), pp. 331-345.

[6] Cobham, A., "The intrinsic computational complexity of functions" Proc. 1964 Cong. for Logic, Methodology, and Philosophy of Science, North-Holland, Amsterdam (1964).

[7] Davis, M., "Computability and Unsolvability" McGraw-Hill, New York (1958).

[8] Gödel, K., "Über die unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I" Monatshefte f. Math. u. Physik 37 (1931), pp. 349-360.

[9] Grzegorczyk, A., "Some classes of recursive functions" Rozprawy Matematyczne 4 (1953), pp. 1-45.

[10] Hartmanis, J. and Stearns, R.E., "On the computational complexity of algorithms" Trans. Amer. Math. Soc. 117, 5 (1965), pp. 285-306.

[11] Kleene, S.C., "General recursive functions of natural numbers" Math. Annalen 112 (1936) pp. 727-742.

[12] _____, "~~Instruction~~ Introduction to Metamathematics" Van Nostrand, Princeton, (1950).

[13] _____, "Extension of an effectively generated class of functions by enumeration" Colloquium Math. 6 (1958), pp. 67-78.

[14] Meyer, A. R., "Depth of nesting and the Grzegorczyk Hierarchy" Notices Amer. Math. Soc. 12, 3 (1965), Abstract 622-56.

[15] Meyer, A. R., and Ritchie, D.M., "Computational complexity and program structure" IBM Research Report RC-1817 (1967).

[16] _____, and _____, "The complexity of Loop programs" Proc. 22nd Nat. Conf. Assoc. for Comp. Mach., Thompson, Washington (1967), pp. 465-470.

[17] Minsky, M.L., "Computation: finite and infinite state machines" Prentice-Hall, Englewood Cliffs, N.J. (1967).

[18] Myhill, J., "Linear bounded automata" WADD Technical Note 60-165, Wright-Patterson AFB, Ohio (1960).

[19] Péter, R., "Über die mehrfache Rekursion" Math. Annalen 113 (1963), pp. 489-527.

[20] _____, "Die beschränkt-rekursiven Funktionen und die Ackermannsche Majorisierungsmethode" Publicationes Mathematicae Debreczen 4 (1956), pp. 367-375.

[21] _____, "Recursive functions" Academic Press, New York and London (1967). (Translation by István Földes of "Rekursive Funktionen", Akademischen Verlag, Budapest (1951).

[22] Ritchie, D.M., "Complexity classification of primitive recursive functions by their machine programs" Notices Amer. Math. Soc. 12, 3 (1965), Abstract 622-59.

[23] Ritchie, R. W., "Classes of predictably computable functions" Trans. Amer. Math. Soc. 106 (1963), pp. 139-173.

[24] _____, "Classes of primitive recursive functions based on Ackerman's function", Pacific J. Math. 12, 3 (1965) pp. 1027-1044.

[25] Robbin, J. W., "Subrecursive hierarchies" Doctoral Dissertation, Princeton (1965).

[26] Shepherdson, J.C., and Sturgis, H.E., "Computability of recursive functions" J. Assoc. Comp. Mach. 10 (1963), pp. 217-255.

[27] Suppes, P. "Axiomatic set theory" Van Nostrand, Princeton (1960).

[28] Turing, A.M., "On computable numbers, with an application to the Entscheidungsproblem" Proc. London Math. Soc., series 2, 42 (1936) pp. 230-265.