Subrecursive Programming Languages, Part I: Efficiency and Program Structure

ROBERT L. CONSTABLE

Cornell University, Ithaca, New York

AND

ALLAN B. BORODIN

University of Toronto, Toronto, Ontario, Canada

ABSTRACT. The structural complexity of programming languages, and therefore of programs as well, can be measured by the subrecursive class of functions which characterize the language. Using such a measure of structural complexity, we examine the trade-off relationship between structural and computational complexity.

Since measures of structural complexity directly related to high level languages interest us most, we use abstract language models which approximate highly structured languages like Algol.

KEY WORDS AND PHRASES: programming languages, universal machine, computational complexity, complexity class, subrecursive hierarchy, program size, primitive recursive functions, elementary functions, Loop language, Grzegorczyk hierarchy, speed-up theorem

CR CATEGORIES: 4.20, 4.22, 5.22, 5.24

1. *Introduction*

It is widely accepted that the theory of computing can be organized on the basis of conservation principles or trade-off relationships. Such relationships hold among quantities characterizing computation (such as logical complexity, structural complexity, resource expenditure, etc). Some important exchange relationships are well known. For instance, the universal machine involves a trade-off of machine structure for size and computational complexity. Structural complexity in this example is a quantity like the "state symbol product" for Turing machines.

The structural complexity of programming languages, and therefore of programs as well, can be measured by the subrecursive class of functions which characterize the language. Using such a measure of structural complexity, we examine the trade-off relationship between structural and computational complexity.

Since measures of structural complexity directly related to high level languages

Authors' addresses: R. L. Constable, Computer Science Department, Upson Hall, Cornell University, Ithaca, NY 14850; A. Borodin, Department of Computer Science, U. of Toronto, Toronto 181, Ont., Canada.

CONTENTS

interest us most, we use abstract language models which approximate highly structured languages like Algol.

Our attention to programming language models is also motivated by concern for a thesis (somewhat like Church's or Turing's thesis), implicitly known in the literature, that all functions actually used in computing are a subset of the primitive recursive functions. This thesis implies that the subrecursive programming languages considered here are adequate for actual computing. Furthermore, these languages have advantages over universal (general recursive) languages; among them are: all programs halt on all inputs, the run time of any program can be bounded above from its syntax, and mathematical expressions can be uniformly assigned to programs in a natural manner.

But are these advantages free? Not entirely. Blum [3] has shown that one cost is economy of program size. The subrecursive languages will always be very uneconomical in the sense that for every recursive function $f(\ )$ there will be functions $k(\ )$ whose shortest subrecursive programs, π, satisfies

$$f(|\tau|) < |\pi|$$

where $|\ |$ measures size and τ is a general recursive program for $k(\ )$.

It was conjectured that a price was paid for run time as well as for size, at least by certain interesting subrecursive languages and formalisms, such as [19]. We show that the conjecture is false and that in fact these subrecursive run times are, given the right basic operations, within a linear factor of general recursive run times.

The case for subrecursive languages is supported further by the observation in

Constable [8] that the uneconomically long subrecursive programs known from Blum must also be computationally very complex (at least on a finite set).

The advantages of these languages over general recursive languages should be explored more carefully, especially in regard to such problems as equivalence and correctness of programs and especially with attention to their exchange relationships with other properties of programs.

In this paper we examine the exchange between structure and efficiency for specific subrecursive languages for the primitive recursive functions. The languages presented here are all based on existing languages. They are selected with several criteria in mind. One is to point out their expressive power as support for the "implied thesis." Another is to facilitate definitions of structural complexity. A third is to relate our languages to the most elegant examples in the literature. From each language, one acquires a better "feel" for the primitive recursive functions and their apparent "naturalness."(1)

2. *General Recursive Languages*

Simple abstract models of numerical programming languages are now common in the literature (see [9, 25, 26]). These models characterize the core of most high level programming languages (like Algol, Fortran, and PL/I). We shall use modifications of such models to study the relationship between program structure and computational complexity for the specific task of computing functions from N^n into $N = \{0, 1, 2, \cdots\}$ or $N^+ = \{1, 2, 3, \cdots\}$.(2)

The languages we study can be described in terms of a set of statement types (assignment, conditional, go to, and iterative) where the statements are composed of arithmetic expressions (or terms) and relations. For simplicity, only binary and unary terms and relations are used.

2.1. TERMS AND RELATIONS. Using BNF we present the syntactic categories used to form the programming languages.

$\langle\text{variable}\rangle ::= X \mid \langle\text{variable}\rangle X$
$\langle\text{constant}\rangle ::= 0 \mid 1 \mid 2 \mid \cdots$
$\langle\text{argument}\rangle ::= \langle\text{variable}\rangle \mid \langle\text{constant}\rangle$
$\langle\text{1-operator}\rangle ::= O_0^1 \mid O_1^1 \mid O_2^1 \mid \cdots$
$\langle\text{2-operator}\rangle ::= O_0^2 \mid O_1^2 \mid O_2^2 \mid \cdots$
$\langle\text{terms}\rangle ::= \langle\text{argument}\rangle \mid \langle\text{1-operator}\rangle\ (\langle\text{argument}\rangle) \mid$
$\qquad\qquad \langle\text{2-operator}\rangle\ (\langle\text{argument}\rangle, \langle\text{argument}\rangle)$

We will use customary abbreviations and let X_i denote $\underbrace{X \cdots X}_{i \text{ times}}$, thus $X_1 = X$, $X_2 = XX$, $\cdots$. We let v_i denote variables, and also 0, v, $v + 1$, $v \dot{-} 1$ abbreviate $O_0^1(v)$, $O_1^1(v)$, $O_2^1(v)$, $O_3^1(v)$, respectively, and $v_1 + v_2$, $v_1 \dot{-} v_2$, $v_1 \cdot v_2$, $v_1 \div v_2$ abbreviate $O_0^2(v_1, v_2)$, $O_1^2(v_1, v_2)$, $O_2^2(v_1, v_2)$, $O_3^2(v_1, v_2)$, respectively.

(1) Another implicit problem in the literature of recursive function theory and the theory of computing is to explain the apparent naturalness of $\mathcal{R}^1$. Some authors interpret their results as denying naturalness [20], others go to lengths to affirm it [8].

(2) Other numerical tasks such as computing over the rationals (or the reals) can be naturally reduced to this one.

The interpretation is that the O_i^n are n-argument functions. Thus $O_0^1(x) = 0$, $O_1^1(x) = x$, $O_2^1(x) = x + 1$, and

$$O_3^1(x) = \begin{cases} 0 & \text{if } x = 0, \\ x - 1 & \text{otherwise} \end{cases}$$

are the common mathematical expressions for the functions denoted by O_i. Among the O_i the only infrequently seen definition is $O_3^2(x, y) = x \div y =$ greatest integer less than or equal to x/y if $y > 0$ else 0 (also read $\lfloor x/y \rfloor =$ *floor* of x/y).

A class of relations is defined by

$\langle$1-relator$\rangle$ ::= $P_0^1 \mid P_2^1 \mid \cdots$
$\langle$2-relator$\rangle$ ::= $P_0^2 \mid P_1^2 \mid \cdots$
$\langle$relation$\rangle$ ::= $\langle$1-relator$\rangle$ ($\langle$argument$\rangle$) |
$\langle$2-relator$\rangle$ ($\langle$argument$\rangle$, $\langle$argument$\rangle$)

The common abbreviations are P_0^1 for $= 0$, P_1^1 for $\neq 0$, P_0^2 for $=$, and P_1^2 for $\neq$.

The interpretation is again standard: $P_i^1(v)$ denotes a predicate on N^1 and $P_i^2(v_1, v_2)$ denotes a predicate on N^2. The standard predicates are $P_0^1(x)$ iff $x = 0$, $P_1^1(x)$ iff $x \neq 0$, $P_0^2(x, y)$ iff $x = y$, $P_1^2(x, y)$ iff $x \neq y$. We could also add $\leq, \geq, <, >$ in the same manner.

Terms and relations are used in building statements. The statement types are listed below with brief informal interpretations. They are so common that a formal semantics would only be an exercise in formalism.

2.2. LABELS. For the purposes of describing the relationship between statements, these languages will use statement labels. (We shall see that they are dispensable.) The simplest labels are the positive integers, N^+, and the simplest labeling convention is that all statements are labeled, giving programs a linear structure.

$\langle$label$\rangle$ ::= $1 \mid 2 \mid 3 \mid \cdots$

2.3. STATEMENTS. Terms and relations are used to form the following statement types.

(1) Assignments. The *general assignment* statement is
(i) $\langle$variable$\rangle \leftarrow \langle$term$\rangle$
but we also consider the *special assignments* of the form
(ii) $v \leftarrow f_i^1(v)$
(2) go to's. The basic go to is
(i) go to + $\langle$label$\rangle$ or
(ii) go to − $\langle$label$\rangle$
In addition we consider the *computed go to's*:
(ci) go to + $\langle$variable$\rangle$
(cii) go to − $\langle$variable$\rangle$

The signs $+$, $-$ indicate the direction in which the label must be; the plus sign indicates that control goes forward in the program to a statement with a higher label than the go to itself.[3] The minus sign indicates that the label is the same or lower

[3] When the program structure is sufficiently simple (i.e. it contains conditionals and N^+ as labels); then, go to $\pm c$ can be interpreted as "go $\pm$ c statements from this one," i.e. either add c or subtract c from the label of the go to. Then adding 0 is not allowed but subtracting it is.

than the label of the go to itself. The signs are clearly dispensible. We only use them to emphasize the distinction.

In the computed go to, the content of the variable is the label. Here it is essential that labels be numbers; also if the computed label lies in the wrong direction from the go to, the statement is treated as a "no-op" (i.e. is not executed).

(3) Conditionals. The basic conditional is

(i) ⟨conditional⟩ :: = if ⟨relation⟩ then ⟨go to⟩ else ⟨go to⟩

But, the more complex form, (ii), is often useful.

(ii) ⟨nested-conditional⟩ :: = if ⟨relation⟩ then ⟨program⟩ else ⟨program⟩

where the syntactic variable ⟨program⟩ is defined below.

The interpretation of the conditional is completely standard, as in Algol. The nested conditional can be interpreted by first reducing it to a simple conditional.

A common abbreviation is, if ⟨relation⟩ then ±⟨label⟩, for, if ⟨relation⟩ then go to ±⟨label⟩ else go to "next statement." We call this the "one branch conditional."

(4) Input/output (I/O). The statements

(i) IN $v_1, \cdots, v_n$ and

(ii) OUT $w_1, \cdots, w_m$

are the only I/O commands. We will always use these commands in a simple manner. Each will appear only once in a program and it serves to indicate which variables are inputs and which are outputs. The command OUT $w_1, \cdots, w_p$ will mean that the program halts and the output is to be found in $w_1, \cdots, w_p$. The command IN $v_1, \cdots, v_n$ means that variables $v_1, \cdots, v_n$ are loaded with the input values.

(5) Iterative statements. The basic iterative is

```
(i) DO $v$
      $\pi$
    END
```

where π is a program; in BNF we can write

⟨iterative⟩ :: = DO⟨variable⟩;⟨program⟩;END

We also allow

```
(ii) DO WHILE $P$
       $\pi$
     END
```

where π is a program and P is a relation; in BNF:

⟨while iterative⟩ :: = DO WHILE ⟨relation⟩;⟨program⟩;END

The interpretation here is described simply in terms of the previous statement types. Namely, occurring in a program $\hat{\pi}$,

```
DO $v$        is      $\bar{v} \leftarrow v$
$\pi$               1 if $\bar{v} = 0$ then 2
END                   $\pi$
                      $\bar{v} \leftarrow \bar{v} \dot{-} 1$
                      go to 1

                    2 ________
```

where $\bar{v}$ does not appear in π or $\hat{\pi}$ ($\bar{v}$ is called the *loop control variable*). Also

```
DO WHILE $P$
$\pi$
END
```

is

```
1 if $\neg P$ then 2
  $\pi$
  go to 1

2 ______
```

(Notice that for every relation, P, its negation, $\neg P$, is also a relation.)

(6) Function procedures. Certain programs can be selected which compute functions $f(\ ): N^n \to N$ (or *vector functions*, $\langle f(\ )\rangle : N^n \to N^p$ where $\langle f(\ )\rangle = \langle f(\ )_1, \cdots, f(\ )_p\rangle$ each $f(\ )_i : N^n \to N$). Briefly, these are programs with n input variables, one output (p outputs). They will be defined more precisely below. Function procedures are ways to introduce new operations by definition within the program. The syntactic baggage required is the following.

$\langle n\text{-ary function variables}\rangle ::= f_0^n \mid f_1^n \mid \cdots$
$\langle n\text{-ary function}\rangle ::= \langle n\text{-ary function variable}\rangle\ (\langle\text{variable}\rangle, \cdots, \langle\text{variable}\rangle)$

The class of $\langle$function definitions$\rangle$ is defined by equations of the form

(i) (a) $f_i^n(x_1, \cdots, x_n) = \langle n$-argument function computing program not involving the function variable $f_i^n\rangle$.

These function definitions are used to expand the class of terms. Namely an $\langle f$-term$\rangle$ is $f_i^n(a_1, \cdots, a_n)$ for a_i an $\langle$argument$\rangle$. Then $\langle f$-assignments$\rangle$ are defined as

(b) $\langle\text{variable}\rangle \leftarrow \langle f\text{-term}\rangle$

The interpretation is that the program in (i) defines the function letter f_i^n (non-recursively) and $w \leftarrow f_i^n(v_1, \cdots, v_n)$ is interpreted to be the code

```
$\Pi_{f_i^n}(v_1, \cdots, v_p)$
$w \leftarrow u$
```

where $\Pi_{f_i^n}(v_1, \cdots, v_p)$ is the program defining the f-term, $f_i^n(v_1, \cdots, v_p)$ with $v_1, \cdots, v_p$ as input values and u as output (see Section 3 for details).

(ii) An important subclass of program function definitions is made up of those which can be given *explicitly* in terms of compositions of other functions or *substitution* of variables and constants for other variables. These operators are called the *operations of substitution*, abbreviated *Os*, and they are the most basic kinds of function definition. The class of functions explicitly definable from functions $f_1(\ ), \cdots, f_p(\ )$ is denoted $[f_1(\ ), \cdots, f_p(\ ); Os]$.

When function definition is required to be explicit, we have a statement category like the Fortran *function statement*. The concept of explicit definition is basic for the usual notion of recursion in mathematics. We briefly mention recursion in programming below.

(7) Recursive (function) procedures:

(i) When the condition that f_i^n not appear on the rhs (right-hand side) of (i-a) is removed in (6), then (6) defines the classes of $\langle$recursive function definition$\rangle$, $\langle$recursive f-term$\rangle$; and $\langle$recursive f-term assignments$\rangle$.

The interpretation in this case is more difficult. One can use the mechanism of Algol recursive procedures. We shall not go into this in detail. We include (7) only for completeness; it is not needed in what follows.

(ii) When f_i^n can be explicitly defined by terms allowing f_i^n, then we have the definition of *general recursion* in mathematics (see Kleene [14]).

In Section 3 we shall extend these statement types to include subrecursion of two kinds, (8-i) a specialization of (7-i) to subrecursive programs, and (8-ii) a specification of (7-ii) to certain types of recursion schemes, for example the primitive recursion scheme.

We summarize the statement types using BNF:

```
⟨statement⟩ ::= ⟨assignment⟩ |
                ⟨go to⟩ | ⟨computed go to⟩ |
                ⟨conditional⟩ | ⟨nested conditional⟩ |
                ⟨input⟩ | ⟨output⟩ |
                ⟨iterative⟩ | ⟨while iterative⟩ |
                ⟨function definition⟩ | ⟨recursive function definition⟩ |
                ⟨f-assignment⟩ | ⟨recursive f-assignment⟩
```

2.4. Programs and Languages. A *program* is a finite sequence of uniquely labeled statements. For definiteness, the labels 1 to n are used in a program of n statements) and any ⟨go to⟩ in the program refers to only labels 1 to $n + 1$, where $n + 1$ is used to designate a halt. The following are specimens of programs.

Example 1.
```
1  IN X
2  DO X
3  X ← X + 1
   END
4  OUT X
```

Example 2.
```
1  IN X
2  DO X; X ← X + 1; END
3  OUT X
```

Example 3.
```
1  IN X₁, X₂
2  if X₁ ≠ X₂ then if X₁ ≠ 0 then go to 5 else go to 3 else go to 6
3  Y ← X₁ + X₂
4  go to 8
5  DO X₁ ; X₁ ← X₁ + X₂ ; END
6  Y ← X₁
7  DO WHILE Y ≠ 0; X₂ ← X₂ + 1; Y ← Y ∸ 1; END
8  OUT Y
```

We also prohibit branching into the scope of a DO.

Various specific programming languages are defined by selecting subsets of the possible statement types and subsets of the operations and relations. We will define (below) the following language types: Algol-R, GR, GRμ, and G_3. The *language* is the collection of all programs whose statements come from the types allowed in the language base.

For convenience in describing the multitude of possible languages, we adopt the following abbreviations.

arithmetic operations	
unary	$+1, \dot{-}1, 0, (\ )$[4]
binary	$+, \dot{-}, \cdot, \div$
relations	
unary	$= 0, \neq 0$
binary	$=, \neq$
go to + ⟨label⟩	$\downarrow$
go to − ⟨label⟩	$\uparrow$
go to + ⟨variable⟩	$(\downarrow)$
go to − ⟨variable⟩	$(\uparrow)$
conditional	$\Diamond$
nested conditional	$(\Diamond)$
one-branch conditional	$\Diamond_1$
iterative	DO END
while iterative	DOWH END
function procedures	E
explicit function statements	E_0
recursive function procedures	R
recursive explicit function statement	R_0

(R_0 is also referred to as explicit recursive definition.)

Letting $A_8 = \{+1, -1, 0, (\ ), +, \dot{-}, \cdot, \div\}$ and $P_4\{= 0, \neq 0, =, \neq\}$, the basic languages are:

(1) Algol-R: $[A_8, P_4, \downarrow, \uparrow, (\downarrow), (\uparrow), (\Diamond), \overset{\text{DO}}{\text{END}}, \overset{\text{DOWH}}{\text{END}}, E, R]$

(2) GRμ: $[A_8, P_4, \downarrow, \uparrow, (\downarrow), (\uparrow), \Diamond_1, \Diamond, \overset{\text{DO}}{\text{END}}, \overset{\text{DOWH}}{\text{END}}, E]$

(3) GR: $[A_8, P_4, \downarrow, \uparrow, (\downarrow), (\uparrow), \Diamond_1, \Diamond, \overset{\text{DO}}{\text{END}}, E]$

(4) G_3: $[+1, \dot{-}1, \downarrow, \uparrow, \Diamond_1]$

Remark. The only difference between GR and GRμ is the DO WHILE statement. This statement allows direct implementation of the *least number operator*, μ, defined as

$$\mu y[P(\bar{x}, y)] = \text{least } y \text{ such that } P(\bar{x}, y) \text{ for } \bar{x} \in N^n, y \in N$$

The direct implementation is

```
$Y \leftarrow 0$
DO WHILE $\neg P(X_1, \cdots, X_n, Y)$
$Y \leftarrow Y + 1$
END
```

[4] These unary operators appear only in special assignments, $v \leftarrow v + 1$, $v \leftarrow v \dot{-} 1$, except for the identity operator, ( ), which appears as $v \leftarrow w$.

as long as $\neg P(\ )$ is expressible. Normally predicates other than those in P_4 will be represented by their characteristic function $fP(\ )$ such that

$$fP(x_1, \cdots, x_n) = \text{if } P(x_1, \cdots, x_n) \text{ then } 1 \text{ else } 0$$

In this case the implementation is

```
$Y \leftarrow 0$
DO WHILE $S \neq 1$
$S \leftarrow fP(X_1, \cdots, X_n, Y)$
$Y \leftarrow Y + 1$
END
```

2.5. Computations, Functions, and Run Times. Programs are intended to define computations. For a simple language like G_3 it is easy to be precise about how. For GR it is more difficult and for Algol-R still more difficult. We shall treat the latter by reducing them to G_3.

It is not difficult to see that Algol-R, and hence GRμ and GR, can be translated into G_3. In fact, the definition of each of the seven instruction types, except the computed go to, included a reference to a G_3 interpretation of it. All that remains for a complete reduction of these types to G_3 is a translation of the arithmetic operations and a treatment of procedures. The former will be given below (Theorem 3.2), and for the latter we have referred the reader to discussions of actual programming languages (such as Wegner [27]). The translations

T_1: Algol-R $\rightarrow G_3$
T_2: GR$\mu \rightarrow G_3$
T_3: GR $\rightarrow G_3$

will be used to define the semantics for these languages by the rule that the meaning of Π is the meaning of $T_i(\Pi)$, $i = 1, 2, 3$.

The programs we interpret are those which compute functions. These can be singled out syntactically. A program ϕ is a *function program* iff the first statement is IN $v_1, \cdots, v_n$ and the last is OUT $w_1, \cdots, w_p$ for v_i, w_j variables of ϕ, and no other I/O statements occur in ϕ.

To describe the (partial) function, $\phi(\ )$, which ϕ computes we would define a computation of ϕ and a *terminating computation* of ϕ. However, this matter is treated extensively in the literature (e.g. [9, 25]), and we refer the reader to these sources for precise definitions. Suffice it to say that a *computation*, as defined on a RASP for example, is a sequence of *states*, $\delta_0, \delta_1, \cdots, \delta_n, \cdots$. Each state is a pair, $\delta_i = \langle a_i, M_i\rangle$, where M_i is a list of values of all of the variables of ϕ and a_i is the label of a statement (the statement in *control* at that moment of the computation). The program ϕ takes one state into the next, $\delta_i \underset{\phi}{\Rightarrow} \delta_{i+1}$, iff the change in memory from M_i to M_{i+1} is the result of executing the statement labeled a_i, and if a_{i+1} is the label of the next statement to be executed.

The sequence $a_0, a_1, \cdots$ of labels is the *flow of control* and $M_0, M_1, \cdots$ is the sequence of *memory configurations*. A finite computation is said to be *terminating* and we write $\delta_0, \delta_1, \cdots, \delta_n$ for $n < \omega$. If a_n is a halt statement (OUT v), then the computation is *normally terminating* and the program ϕ_i is said to *halt*, abbreviated $\phi_i(x_1, \cdots, x_n) \downarrow$.

A function program ϕ *computes the partial function* $\phi(\ ): N^n \rightarrow N$ iff when

$X_1, \cdots, X_n$ are the input variables and Y the output variable, then when X_i starts with value x_i, and all other variables have value 0, the computation of ϕ terminates iff $\phi(x_1, \cdots, x_n)$ is defined, and if ϕ terminates then Y has the value $\phi(x_1, \cdots, x_n)$.[5]

The *number* of *steps* in a terminating computation of ϕ on inputs $x_1, \cdots, x_n$ is denoted $t\phi_i(x_1, \cdots, x_n)$. The step counting function $t\phi_i(\ )$ can be syntactically defined from ϕ_i in a simple manner, as follows: pick a variable S not in ϕ_i; replace OUT Y by OUT S; after the input instruction of ϕ_i, place $S \leftarrow 0$; then after each statement of ϕ_i insert $S \leftarrow S + 1$; change all labels, L, in conditionals to L' (L' is a temporary new symbol not in the language) and for each label L' put the pair of statements

$$\begin{cases} L' : S \leftarrow S + 1 \\ \quad \text{go to } L \end{cases}$$

at the end of ϕ_i; then relabel the new program in order and call the result $t\phi_i$.[6]

2.6. Characterizing Languages. We can now speak precisely about the expressive power of programming languages. A programming language $\mathcal{L}$ is *capable of computing* $\phi: N^n \rightarrow N$ iff there is a program π of $\mathcal{L}$ which computes ϕ. The programming language is *characterized* by the class of partial number theoretic functions which it is capable of computing.

We use the following notation for the function classes:

$\mathcal{P}_n$	all n argument partial functions, $\alpha: N^n \rightarrow N$,
$\mathcal{PR}_n$	all n argument partial recursive functions, $\phi_i : N^n \rightarrow N$,
$\mathfrak{F}_n$	all n argument total functions, $f: N^n \rightarrow N$,
$\mathcal{R}_n$	all n argument total recursive functions.

When used without the subscript, the letters designate the union over all n, thus $\mathfrak{F} = \bigcup_{n=0}^{\infty} \mathfrak{F}_n$.

When discussing functions we follow Rogers [24] and let lower-case Greek letters α, β, γ denote partial functions and lower-case Latin letters f, g, h denote total functions. We frequently use the notation $\phi(\ ), f(\ )$ to distinguish the function (as a set of ordered pairs) from the rule ϕ, f describing the function.

Now we can state a well-known characterization.

(1) G_3 is characterized by $\mathcal{PR}$. This fact is established in Minsky [21] and in Shepherdson and Sturgis [26].

From the translation in Subsection 2.5 we know that

(2) Algol-R, GRμ, and GR are characterized by $\mathcal{PR}$. A language characterized by $\mathcal{PR}$ is called *universal* or *general recursive.* A language characterized by a subset of $\mathcal{R}$ is called *subrecursive.* We shall see some of them in Section 3.

2.7. Indexing Universal Languages and Abstract Computational Complexity. Some of the results in Section 3 can be treated very abstractly in terms of recursive function theory. In order to pursue that viewpoint we will present very briefly the formal apparatus needed. We emphasize that the following definitions

[5] One can drop this assumption on the other variables if he selects syntactically those programs in which all noninput variables (work variables) are initialized before use. This is perhaps more realistic but technically more tedious.

[6] Of course, the instructions $S \leftarrow 0$ and "go to L" are translated into their G_3 equivalents, and relabeling involves adjustment of the labels in their conditionals.

and theorems are included for reference only, with the understanding that the reader interested in this viewpoint is already familiar with them from sources such as [1, 13, 24].

We begin with a list (indexing) of all function computing programs of the language (in general of the formalism for expressing algorithms). Therefore, let $\phi_0, \phi_1, \phi_2, \cdots$ be an effective enumeration of all function computing programs.[7] The basic theorems needed about the list are the "universal machine theorem" and the "*S-m-n*" theorem (so called for Kleene's original formulation). We state these theorems for the simple case of one argument functions.

THEOREM 2.1 (Universal machine for one argument functions). *There is a* ϕ_u^2 *such that* $\phi_u^2(i, x) = \phi_i(x)$ *for all i and all x.*[8]

THEOREM 2.2. *There is a function* $\sigma(\ )$ *such that* $\phi_j(i, x) = \phi_{\sigma(j,i)}(x)$ *for all x, i.*

It turns out that these two simple theorems serve to characterize any list, $\{\phi_i(\ )\}$, of $\mathcal{PR}$ which arises from any formalism which can be recursively translated to G_3, and to which G_3 can be recursively translated. Such indexings of $\mathcal{PR}$ are called *acceptable*, i.e. a mapping ϕ, $\phi: N \to \mathcal{PR}$, is *acceptable* iff it satisfies Theorems 2.1 and 2.2 (generalized to n argument functions). See Rogers [24] for an account of these indexings.

The time measure of computational complexity, $T = \{t\phi_i(\ )\}$, is conveniently thought of as the list $\{t\phi_i(\ )\}$. Two critical properties of the list are the following.

THEOREM 2.3. $\phi_i(x) \downarrow$ *(is defined) iff* $t\phi_i(x) \downarrow$ *(is defined).*

THEOREM 2.4. *There is a recursive predicate* $M_t(\ )$ *such that*

$$M_t(i, x, y) \text{ iff } t\phi_i(x) = y.$$

These two theorems are left to the reader. The first one is trivial and the second says that to tell whether $t\phi_i(x) = y$ we need only use the universal program known from Theorem 2.1, ϕ_u^2, to run ϕ_i for y steps and determine whether the computation has halted.

It turns out that these two theorems serve to characterize the notion of computational complexity in a very fruitful manner. We call a list $\{m\phi_i(\ )\}$ an *abstract* (or *Blum*) *computational complexity measure* iff

A1. $\phi_i(x) \downarrow$ iff $m\phi_i(x) \downarrow$,

A2. There is a recursive predicate $M_m(\ )$ such that $M_m(i, x, y)$ iff $m\phi_i(x) = y$.

See Blum [1], Borodin [4], and Hartmanis and Hopcroft [13] for an account of this theory.

3. *Subrecursive Programming Languages*

3.1. LANGUAGE DEFINITION. We consider three subrecursive languages, Algol-R_0^1, SR, and Loop. For the first language, we need the mechanism of primitive recursion. Given functions $h(\ ) \in \mathfrak{F}_{n+2}$ and $g(\ ) \in \mathfrak{F}_n$ and $\vec{x} \in N^n$, then $f(\ )$ is

[7] We think of lists as including functions of any finite number of inputs, but we usually want only the *one* argument functions, (i.e. the ϕ_i have only one input variable specified, usually x). Therefore, we think of the list as containing n-argument functions for all n from which the sublist of n argument functions for fixed n can be effectively extracted, and we use the same notation for both lists unless this will be confusing, in which case we write ϕ_i^n indicating n argument.

[8] $\phi_i(x) = \phi_u^2(i, x)$ means $\phi_i(x) \downarrow$ iff $\phi_u^2(i, x) \downarrow$ and if $\phi_i(x) \downarrow$ then $\phi_i(x) = \phi_u^2(i, x)$.

defined from $g(\ )$, $h(\ )$ by the schema of *primitive recursion*, R_0^1, iff

$$R_0^1 \quad \begin{cases} f(\vec{X}, 0) = g(\vec{X}) \\ f(\vec{X}, n+1) = h(\vec{X}, n, f(\vec{X}, n)) \end{cases}$$

or, written as a conditional expression,

$$f(\vec{X}, n) = (\text{if } n = 0 \text{ then } g(\vec{X}) \text{ else } h(\vec{X}, n \dot{-} 1, f(\vec{X}, n \dot{-} 1)))$$

This schema R_0^1 is but one of the infinitely many possible ways to write an explicit recursive definition which is guaranteed to define a total function if $h(\ )$ and $g(\ )$ are total. It is, however, general enough to permit nearly all forms of recursion which arise naturally in mathematics.

We now define the subrecursive languages.

(1) Algol-R_0^1: $[A_8, P_4, \downarrow, (\downarrow), (\overset{\text{DO}}{\Diamond}), \text{END}, E, R_0^1]$

(2) SR: $[A_8, P_4, \downarrow, (\downarrow), \Diamond_1, \overset{\text{DO}}{\Diamond}, \text{END}, E]$

(3) Loop: $[+1, 0, (\overset{\text{DO}}{\ \ }), \text{END}]$

(4) $\text{Loop}_{\min}$: $[+1, 0, \overset{\text{DO}}{\text{END}}]$

The semantics for these languages is again given by regarding each statement type as an abbreviation for the equivalent G_3 program. These abbreviations were supplied in defining the statement types in Subsection 2.3.

The Loop languages are due independently to Ritchie [19] (Loop) and Minsky [21] ($\text{Loop}_{\min}$). They are based on ideas developed by the logician Robinson [23]. It is easy to prove that the Loop languages are characterized by the primitive recursive functions, $\mathcal{R}^1$.

$$\mathcal{R}^1 = [+1, 0, U_i^n; Os, R_0^1]$$

where $U_i^n(x_1, \cdots, x_n) = x_i$. For more on $\mathcal{R}^1$ see [11, 14, or 23].

THEOREM 3.1. *Loop and Loop_{min} are characterized by $\mathcal{R}^1$.*

PROOF. See [19 or 21]. We shall sketch a proof of this in Section 4.

Both SR and Algol-R_0^1 are also characterized by $\mathcal{R}^1$. We show the "hard part" of this (Theorem 3.2) and leave the other as an exercise. It would be interesting to formulate a natural version of subrecursive Algol which allowed full recursive procedures, rather than explicit recursive procedures, and which was still characterized by $\mathcal{R}^1$.

3.2. TRANSLATING INTO LOOP

THEOREM 3.2. *SR is characterized by $\mathcal{R}^1$.*

DISCUSSION OF PROOF. The idea is to show that SR can be translated into Loop in the sense that for every SR program π there is a Loop program, p_i, and

$$*: \pi(x_1, \cdots, x_n) = p_i(x_1, \cdots, x_n)$$

for all $x_1, \cdots, x_n$. Since the meaning of the statement types in both SR and Loop are given in terms of G_3, the precise verification of $*$ can be done in G_3 where the semantics are manageable (and standard in the literature, especially [9 or 21]). We shall often leave to the reader the final detail of verifying this G_3 level equivalence.

The proof is given in three parts. First the operations and predicates of SR are reduced. The reduced language has the form:

$$[+1, 0, \neq 0, =0, \downarrow, (\downarrow), \Diamond_1, \Diamond, \overset{\text{DO}}{\text{END}}, E]$$

Next the conditional $\Diamond$ is reduced to $\Diamond_1$, computed go to's are reduced to go to's and procedures are eliminated. This leaves the language

$$[+1, 0, \neq 0, =0, \downarrow, \Diamond_1, \overset{\text{DO}}{\text{END}}]$$

The final, and hardest, phase is the elimination of all go to's and conditionals. This is done using the $\overset{\text{DO}}{\text{END}}$ statement as a switch to "shut-off" statements under appropriate conditions.

We absorb a proof of correctness of the translation into the construction itself. This is done at the end of each phase.

PROOFS.

Phase I. We define all operations, A_8, in $[+1, 0, \overset{\text{DO}}{\text{END}}]$

(1) First $X \leftarrow Y$ becomes

```
$X \leftarrow 0$
DO $Y$
$X \leftarrow X + 1$
END
```

(2) Then $Z \leftarrow Y \dot{-} 1$ becomes

```
$S \leftarrow 0$
$Z \leftarrow 0$
DO $Y$
$Z \leftarrow S$
$S \leftarrow S + 1$
END
```

(3) So $Z \leftarrow X + Y$ becomes

```
$Z \leftarrow X$
DO $Y$
$Z \leftarrow Z + 1$
END
```

(4) And $Z \leftarrow Y \dot{-} X$ becomes

```
$Z \leftarrow Y$
DO $X$
$Z \leftarrow Z \dot{-} 1$
END
```

(5) Then $Z \leftarrow X \cdot Y$ becomes

```
$Z \leftarrow 0$
DO $Y$
$Z \leftarrow Z + X$
END
```

(6) And $Z \leftarrow X \div Y$ becomes

```
$S \leftarrow X$
$Z \leftarrow 0$
DO $X$
  $S \leftarrow S \dot{-} Y$
  $B \leftarrow 0$
    DO $S$
      DO $Y$
      $B \leftarrow 0$
      $B \leftarrow B + 1$
      END
    END
    DO $B$
    $Z \leftarrow Z + 1$
    END
END
```

(where $B = 1$ iff $S > 0$ so B is simply a "switch" which allows Z to increase only while Y can still be subtracted from S, thus Z counts the number of times Y can be subtracted from S as long as $Y > 0$).

The reader can verify that these Loop programs are equivalent to the usual interpretation of the operations (given in Subsection 2.3).

We next show that the relations $v = w$ and $v \neq w$ can be replaced by relations $u = 0$, $u \neq 0$.

Notice, $X = Y$ iff $(X \dot{-} Y) + (Y \dot{-} X) = 0$ so whenever $X = Y$ or $X \neq Y$ occurs compute:

```
(7) $S_1 \leftarrow X \dot{-} Y$
    $S_2 \leftarrow Y \dot{-} X$
    $S \leftarrow S_1 + S_2$
```

and test $S = 0$ or $S \neq 0$ respectively.

The replacements described in phase I are used in translation as follows. Given $\pi \in$ SR, find variables S, S_1, S_2, B not in π and replace each assignment statement of π by the Loop code. Before each conditional which tests $v = w$ $(v \neq w)$ place the Loop code (7) and move the label of the conditional to the first statement of (7), and change the relation to $S = 0$ $(S \neq 0)$.

Phase II. Whenever, if p then s else s_2, is encountered it can be replaced by

```
(1) if $p$ then $s_1$
    $s_2$
```

So, the new program obtained this way is equivalent to the original.

A computed go to, go to $+X$, can be replaced by a sieve of go to's. First notice that only finitely many of the values of X can make sense, say $1, \cdots, m$. Therefore, go to $+X$ can be replaced by

```
(2) if $X = 1$ then $+1$
    if $X = 2$ then $+2$
    ⋮
    if $X = m$ then $+m$
```

where $X = i$ is provided for by setting a certain number of variables to constants at the beginning of the program, e.g. set $N_m = m$ by

$$\begin{array}{l} N_m \leftarrow 0 \\ \left.\begin{array}{l} N_m \leftarrow N_m + 1 \\ \vdots \\ N_m \leftarrow N_m + 1 \end{array}\right\} m \text{ times} \end{array}$$

Function procedures are treated as "macros." That is, whenever an f-assignment statement, $v \leftarrow f(v_1, \cdots, v_n)$, is encountered in a program ϕ is replaced by the sequence of statements

$$\begin{array}{ll} (3) & \hat{\Pi}_f \\ & v \leftarrow w \\ & B_f \end{array}$$

The code $\hat{\Pi}_f$ is obtained from the function definition of f, $f(x_1, \cdots, x_n) = \Pi_f$, as follows. Delete the I/O commands. If $w_1, \cdots, w_p$ are the variables of Π_f, with IN $w_{i_1}, \cdots, w_{i_n}$, and $v_1, \cdots, v_n$ are the arguments of f in the f-assignment statement (in ϕ), then put the n statement $w_{i_j} \leftarrow v_j$, $j = 1, \cdots, n$, at the beginning of Π_f, and make all other variables of Π_f disjoint from those of ϕ. Let w be the output variable of Π_f. Move the label of $v \leftarrow f(v_1, \cdots, v_n)$ in ϕ to the first statement of $\hat{\Pi}_f$.

Since we are assuming that all work variables of function programs are initialized to zero from outside the program (see Footnote 4), we must restore the work variables of Π_f before we leave it. That is what the instructions of B_f do.

Now when control reaches $v \leftarrow f(v_1, \cdots, v_n)$ in ϕ, the computation $\Pi_f(v_1, \cdots, v_p)$ occurs and its value is placed in v. No other variables of ϕ are changed. In the new program, when control reaches the first statement of $\hat{\Pi}_f$ it causes the computation of $\hat{\Pi}_f$. This computation does not affect any values of ϕ other than v and before any statement of ϕ can cause a return to $\hat{\Pi}_f$, all work variables are initialized so that $\hat{\Pi}_f(x_1, \cdots, x_n) = \Pi_f(x_1, \cdots, x_n)$ for all $x_1, \cdots, x_n$.

This argument can be made precise by appealing to formal semantics for G_3 and proceeding as in Elgot and Robinson [9]. However, we feel that this informal treatment does not omit the essential ideas.

Phase III. Our goal is to remove all go to's and conditionals. It is by no means obvious how this can be done, but we can simplify the matter by concentrating on one type, conditionals. Notice that by using a variable like N_1 (recall $N_1 = 1$) we can replace go to $+c$ by "if $N_1 \neq 0$ then $+ c$" (abbreviate this by $\Diamond_{+1}$); furthermore, we can restrict consideration to conditionals with the predicate $v \neq 0$. Simply replace

$$\underline{l \text{ if } v = 0 \text{ then } +c} \quad \text{by} \quad \begin{array}{l} l \text{ if } v \neq 0 \text{ then } +d \\ \underline{\text{go to } +c} \end{array}$$

where $+d$ refers to the statement after go to $+c$.

We are now concerned with showing that $\text{Loop}^+ = [+1, 0, \Diamond_{+1}, \overset{\text{DO}}{\text{END}}]$ is equivalent to $[+1, 0, \overset{\text{DO}}{\text{END}}] = \text{Loop}_{\text{min}}$. Given Π in Loop^+, suppose there are m conditionals. For each of them pick a variable, H_i, not in Π. We form a switch, SW (H_i), to put around certain statements of Π. The switch around statement s,

```
┌SW(H)
└s
```

is defined by

```
S ← 0
DO H
S ← 0
S ← S + 1
END
DO S
s
END
```

The operation of the switch is simple; if $H = 0$, then s is not executed because S is 0. If $H > 0$ then $S = 1$ and s is executed exactly once.

The conditionals of Π are removed in steps. Let their order of occurrence in Π be

```
$l_1$ : if $v_1 \neq 0$ then $+ c_1$
$l_2$ : if $v_2 \neq 0$ then $+ c_2$
⋮
$l_m$ : if $v_m \neq 0$ then $+ c_m$
```

Replace l_1 by

```
DO $v_1$
$H_1 \leftarrow 0$
END
```

If statement $\bar{s}_1$ is at location $+ c_1$, then replace $\bar{s}_1$ by the pair $H_1 \leftarrow H_1 + 1; \bar{s}_1$, and move the label of $\bar{s}_1$ to $H_1 \leftarrow H_1 + 1$.

Now put the switch SW(H_1) around every statement of Π and every statement in the scope of all DO-loops in Π except the two just modified. Move labels of statements to first statement of SW(H_1). Call the resulting program Π_1.

Now treat Π_1 as Π and repeat the process for l_2. Call the resulting program Π_2. Notice that all original statements of Π except l_1 and $\bar{s}_1$ have around them a double switch,

```
 SW(H_2)
┌SW(H_1)
└[
```

The statements $\bar{s}_1$ and DO v_1 ; $H_1 \leftarrow 0$; END only have the single switch SW(H_2).

Continue in this manner producing $\Pi_1, \Pi_2, \cdots, \Pi_m$. Now prefix to each Π_i $2m$ statements which initialize the switches:

$$H_1 \leftarrow 0; \quad H_1 \leftarrow H_1 + 1; \quad H_2 \leftarrow 0; \cdots ; H_m \leftarrow H_m + 1.$$

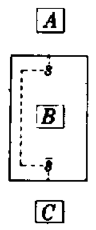
Still call the resulting programs Π_i. Notice that Π_m belongs to Loop. We claim that the Π_i are all equivalent. The intuition behind the equivalence is that when a conditional causes control to "branch" or "jump," the switches are all turned off (set to 0) and no statement of Π is executed until they are turned on again. This happens only at the location to which control branched. It is important that this location be accessible from the conditional without the use of further branches because they are all turned off. Because the branch point lies below the conditional, it is so accessible.

Notice that the theorem cannot be true for the language GR which differs from SR only by the inclusion of backwards branches, $\uparrow$ and $(\uparrow)$. We see from the above intuitive argument why this translation would fail for GR, namely there is no way to reach the location which turns the switches back on again.

We now offer a more precise proof that Phase III is correct. We prove for all n that Π_n is equivalent to Π_{n+1}. Let "if $v \neq 0$ then $+c$" be the first conditional of Π_n (if there is none, then $\Pi_n = \Pi_{n+1}$ and we are finished). Call it s. Let $\bar{s}$ be the statement referred to by $+c$.

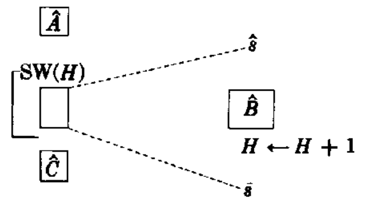
The two statements, s and $\bar{s}$, can be separated by 0 or $m > 0$ END statements not matched with DO's (m indicates the depth from which l is branching). We prove the result by induction on m.

For $m = 0$ the program Π_n has the form



where A, B, C are blocks of statements and where the box around s, $\bar{s}$ indicates the outer boundary of any loops containing them (this description is not vital to the proof, but hopefully it is helpful).

The program Π_{n+1} has the form



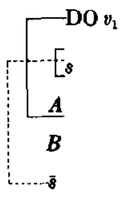
enlargement of switch portion

where $\hat{A}$, $\hat{B}$, $\hat{C}$ are the blocks A, B, C with the switch, SW(H), installed.

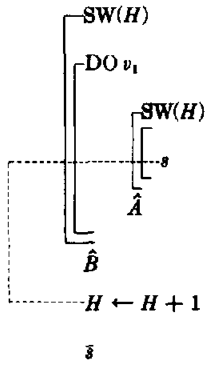
Before control in Π_n reaches s, the programs Π_n and Π_{n+1} behave identically since $H > 0$ and the switches have no effect. When Π_n reaches s, if the branch is not taken, then H remains 1 and the programs operate identically. If the branch is taken, then Π_n jumps to $\bar{s}$ skipping over B. Control in Π_{n+1} proceeds through B downward to $H \leftarrow H + 1$. But no statements of B are executed since they are all protected by switches which are set to 0. Once $H \leftarrow H + 1$ is executed, the switches are restored and $\bar{s}$ is executed. So Π_n and Π_{n+1} are operating identically again and the result of each computation on the variables of Π_n between s and $\bar{s}$ is the same. Thus the computations produce identical results on outputs (on all variables of Π_n), and the case $m = 0$ is proved.

Assume the theorem is true up to m. We prove it for $m + 1$. We actually assume the stronger statement that the computation is identical on the (explicit) variables of Π_n (but not necessarily on the loop control variables of Π_n).

The critical piece of Π_n has the form



and Π_{n+1} has the form



As in the $m = 0$ case, as long as $H > 0$ the computations run identically. Suppose the branch is taken at s. Then in Π_n, control skips to $\bar{s}$ leaving all the loop control variables and explicit variables of Π_n unchanged. In Π_{n+1}, H is set to 0 but control continues in the loops until it eventually exits to the outermost loop whose END-statement separates s and $\bar{s}$. At this point, the variables of Π_n and Π_{n+1} have the same values (by the induction assumption).

When control reaches DO v_1 (the test, if $\bar{v}_1 = 0$ then ___ in the implementation of the DO-loop), no statements in the scope of the DO v_1 are executed because they are all protected by the switch, SW(H). Thus after at most v_1 iterations control passes to $\hat{A}$ and then to $H \leftarrow H + 1$ without changing any (explicit) variables of Π_n. At location $\bar{s}$ the H is restored to 1 and the programs Π_n and Π_{n+1} have had the same effect on the variables of Π_n. The only difference is that all loop control variables of Π_{n+1} in the scope of DO v_1 are at zero but those of Π_n have positive values.

This difference can have no effect because if control ever returns to DO v_1 in either Π_n or Π_{n+1}, all loop control variables in its scope will be reinitialized to values of the (explicit) variables of Π_n. If control does not return to DO v_1, then those loop control variables cannot affect the (explicit) variables. Hence, in either case, the (explicit) variables of Π_n and Π_{n+1} remain identical throughout the computation. Hence, Π_n and Π_{n+1} are equivalent. Q.E.D.

This concludes the proof that Phase III is correct. The entire translation of $\Pi \in$ SR to $T(\Pi) \in$ Loop has been broken down into phases;

$$T(\Pi) = T_3(T_2(T_1(\Pi))), \text{ SR} \ni \Pi \rightarrow T_1(\Pi) \rightarrow T_2(\Pi) \rightarrow T_3(\Pi) \in \text{Loop},$$

and each phase has been shown to preserve equivalence.

Thus $\Pi(x_1, \cdots, x_n) = T(\Pi)(x_1, \cdots, x_n)$ for all x_i. Q.E.D. (Theorem 3.2)

4. *Preliminary Theory*

In this section we prove a set of theorems which leads to a deeper understanding of the main results on efficiency in Section 5. The ideas and proof techniques used here are essential in Section 5, and the main purpose of this section is to present those ideas systematically. The final subsection, Subsection 4.5, connects our results to well-known facts about subrecursive hierarchies.

Central to the pertinent set of results is a simulation theorem. It has the flavor of those theorems which locate small universal Turing machines in the sense that it describes a simple structure for universality. The theorem is due to Meyer and Ritchie [19].

4.1. DEPTH OF NESTING IN LOOP PROGRAMS. Let L_n be the class of Loop programs having DO-loops nested to a depth of at most n. More precisely, L_0 has no DO-loops and for $\Pi \in$ Loop, $\Pi \in L_{n+1}$ iff $\Pi \in L_n$ or the only programs in the scope of iteratives are programs in L_n.

Thus if Π represents a program of L_0, B_i a program of L_1, and A_i a program of L_2, we see that any program in L_2 has the form $A_1; \cdots; A_n$ where A_i is of the form DO v; $B_1, \cdots, B_m$; END or of the form B_i or Π, and B_i has the form DO v; Π; END or the form Π.

The same rules can be used to define the classes SR_n and GR_n.

The class of functions computed by programs in L_n is denoted by $\mathcal{L}_n$.

4.2 SIMULATION. If α_i is an Algol-R program, then we know that it has an equivalent G_3 image under translation, say $\phi_{a(i)}$. Therefore the simulation theorem we state is applicable to Algol-R (hence GR) as well as G_3.

THEOREM 4.1. *If $\phi_i \in G_3$, then there is a $\phi_i^* \in L_1$ such that $\phi_{g(i)}$ as defined by (the translation of the GRμ program)*

```
H ← 1
DO WHILE H ≠ 0
φ_i*
END
```

is equivalent to ϕ_i, i.e. $\phi_i(\ ) = \phi_{g(i)}(\ )$.

DISCUSSION OF THE PROOF. Our proof is based on the ideas of Theorem 3.2. Because of the DO WHILE loop, the backward conditionals can be eliminated in the same manner as the forward conditionals. The variable H remains 1 until ϕ_i halts (if ever), so essentially the DO WHILE loop "runs a simulation of ϕ_i until it halts."

The only technical details are showing that all of the steps of Theorem 3.2 can be done with only one level of nesting. The switches and subtraction are the only steps that need to be modified from the approach of Section 3.

PROOF. Recall that the program ϕ_i halts by executing an output statement (or by branching to a nonexistent statement immediately after executing an output statement). Replace OUT w by the pair of statements $H \leftarrow 0$; OUT w and move the label of OUT w to $H \leftarrow 0$ (then relabel the entire program to keep it in standard form).

Now apply the Phase III translation of Theorem 3.2 to the modified ϕ_i and call the result ϕ_i^+. Observe that $H \leftarrow 1$; DO WHILE $H \neq 0$; ϕ_i^+; END is equivalent to ϕ_i

for the reasons given in the correctness proof for Phase III. In particular, for the case of a backward conditional, if $v \neq 0$ then $-c$, after setting the switch corresponding to this conditional, control will proceed downward to the END statement and then will return to the beginning of ϕ_i and flow down to the label c without changing the explicit variables of ϕ_i nor H.

To put ϕ_i^+ into L_1 we need to modify the use of switches. The nested switches

```
 SW(H_1)
[ SW(H_2)
  [ ⋮
    SW(H_n)
    [s
    ⋮
```

can be replaced by the single switch,

```
SW(H_1, ⋯ , H_n)
[s
```

defined by

```
P ← V(H_1, H_2, ⋯ , H_n)
S ← 0
DO P
S ← 0
S ← S + 1
END
DO S
s
END
```

where $V(H_1, \cdots, H_n)$ is 0 if any H_i is 0, and is positive otherwise. The code for one such function is:

```
P ← H_1
DO P
P ← H_2
END
⋮
DO P
P ← H_n
END
```

Thus the whole multiple switch, $\mathrm{SW}(H_1, \cdots, H_n)$, belongs to L_1. If the statement s is $v \leftarrow v + 1$, then $\mathrm{SW}(H_1, \cdots, H_n)$ is in L_1. But if it is $v \leftarrow v \dot{-} 1$ or if $v \neq 0$ then c, then the previously given Loop replacements cannot be used with this switch mechanism. Instead use the following. For

```
[SW (H_1, ⋯ , H_n)
[X̄ ← X ∸ 1
```

use

```
$\bar{X} \leftarrow 0$
[ SW$(H_1, \cdots, H_n)$
[ $\bar{X} \leftarrow X$
$Z \leftarrow 0$
DO $\bar{X}$
$X \leftarrow Z$
$Z \leftarrow Z + 1$
END
```

and for

```
[      SW$(H_1, \cdots, H_n)$
[ $l_1$ = if $X \neq 0$ then $c$
```

notice that it would, according to Theorem 3.2, become

```
$l_1$ : [ SW$(H_1, \cdots, H_n)$
        [ DO $X$
        [ $H_1 \leftarrow 0$
        [ END
```

It now becomes

```
$l_1$ : $\bar{X} \leftarrow 0$
      [ SW$(H_1, \cdots, H_n)$
      [ $\bar{X} \leftarrow X$
      DO $\bar{X}$
      $H \leftarrow 0$
      END
```

These are easily seen to have the same effect, and they are in L_1. Let ϕ_i^* be the program ϕ_i^+ with the above replacements. Then, since ϕ_i^* and ϕ_i^+ are equivalent, the result follows. Q.E.D.

An interesting corollary follows from this theorem. Suppose $t\phi_i(x_1, \cdots, x_n) \leq f(x_1, \cdots, x_n)$ for $f \in L_n$, where $n \geq 2$. Then the DO WHILE iterative in Theorem 4.1 can be replaced by a simple DO S, where S is larger than the time needed for ϕ_i^* to simulate ϕ_i. This simulation time is no more than the maximum of the input variables times a multiple of the length of ϕ_i, denoted $|\phi_i|$, plus a constant.[9] Thus for $d = a \cdot |\phi_i| + b$, the time to simulate is bounded by $d \cdot \max\{x_1, \cdots, x_n\} \cdot f(x_1, \cdots, x_n)$ which is bounded by $b(x_1, \cdots, x_n) = d \cdot \sum_{i=1}^{n} x_i \cdot f(x_1, \cdots, x_n)$. If $f(\ )$ is in L_n then so is $b(\ )$. Suppose it is computed by $B_f \in L_n$.

The idea now is to use a program like

```
$B_f$
$S \leftarrow F$
$H \leftarrow 1$
DO $S$
$\phi_i^*$
END
```

[9] The length of ϕ_i is simply the number of statements in ϕ_i.

to simulate ϕ_i. The only difficulty is that S is only an upper bound on $t\phi_i(x_1, \cdots, x_n)$ and $\phi_i{}^*$ may "run too long and damage the simulation." We correct this problem by setting a switch based on H. Then when $H = 0$, no statements of $\phi_i{}^*$ will be executed.

So let ϕ_i^{**} be $\phi_i{}^*$ with H as part of all switches, i.e. replace SW$(H_1, \cdots, H_p)$ by SW$(H, H_1, \cdots, H_p)$. By the same reasoning as given in detail before, the following program is equivalent to ϕ_i.

```
$B_f$
$S \leftarrow F$
$H \leftarrow 1$
DO $S$
$\phi_i^{**}$
END
```

But also the program belongs to L_n if $n \geq 2$ because the maximum nesting occurs in computing $f(x_1, \cdots, x_n)$. Thus the following result.

COROLLARY 4.1. *There is a* $p(\ ) \in \mathcal{R}_2$ *such that if* $\phi_i \in G_3$ *and* $t\phi_i(x_1, \cdots, x_n) \leq \phi_j(x_1, \cdots, x_n)$ *and* $\phi_j \in L_n$ *for* $n \geq 2$, *then there is a program* $\phi_{p(i,j)}$ *in* L_n *which is equivalent to* ϕ_i.

This corollary was first proved by Meyer and Ritchie [19]. For the language SR, the bound can be made tighter because the switches and the simulation of $\dot{-}1$ are unnecessary. Thus by the same reasoning as above.

THEOREM 4.2. *There is a* $p(\ ) \in \mathcal{R}_2$ *such that if* $\phi_i \in G_3$ *and* $t\phi_i(x_1, \cdots, x_n) \leq \phi_j(x_1, \cdots, x_n)$ *and* $\phi_j \in SR_n$ *for* $n \geq 1$, *then* $\phi_{p(i,j)} \in SR_n$ *and* $\phi_{p(i,j)}(\ ) = \phi_i(\ )$.

PROOF. We describe the construction of a function $\phi_i{}^+ \in$ SR$_0$ for which

```
IN $X$
$H \leftarrow 1$
DO WHILE $H \neq 0$
$\phi_i{}^+$
END
OUT $Y$
```

is equivalent to ϕ_i. From the construction and the arguments of the previous theorem, it will be clear how to prove this equivalence and complete the theorem. We now give the construction of $\phi_i{}^+$.

(i) Place at the beginning of ϕ_i a computed go to, go to $+\ G$, where G is not in ϕ_i. This will be used in executing backwards go to's of ϕ_i.

(ii) Assume Y is an output variable of ϕ_i and that OUT Y is the last statement of ϕ_i, and $|\phi_i| = l$, then replace OUT Y by the pair, $H \leftarrow 0$; $G \leftarrow l + 2$. These statements have labels $l + 1$, $l + 2$. The statement $G \leftarrow l + 2$ forces all subsequent executions of "go to $+\ G$" to branch back to $G \leftarrow l + 2$, thus setting up a loop which bypasses ϕ_i.

(iii) Replace every backwards go to, say "go to $-\ C$," by the pair $G \leftarrow m$; go to $+\ (l + 3)$, where m is the value needed to jump from go to $+\ G$ to the statement $-C$ (and $l + 3$ refers to the location beyond the last statement of the program ϕ modified by (i) and (ii)).

This is the best possible n for a result of this type because we can easily find ϕ_i for which $t\phi_i$ is bounded in SR_0, but there is no equivalent SR_0 program.

4.3. Structural Complexity Classes. The relationship between Theorems 4.1 and 4.2 is explained by the fact that SR_1 and L_2 compute the same class of functions. We say that two languages, $\mathfrak{L}$, $\bar{\mathfrak{L}}$, are *equivalent*, $\mathfrak{L} \equiv \bar{\mathfrak{L}}$, whenever they compute the same class functions. Thus we prove $\mathrm{SR}_n \equiv \mathrm{L}_{n+1}$.

Theorem 4.3. *$SR_n \equiv L_{n+1}$.*

Discussion of the Proof. This result depends on two critical bounding lemmas which estimate the growth rates of functions in terms of their depth of nesting and length. We shall state these lemmas here and prove them in the Appendix. To state them we need two recursive sequences of programs.[10]

(1) Let f_0 be $X \leftarrow X + 1$ and f_{n+1} be DO X; f_n ; END.

(2) Let g_0 be $X \leftarrow X \cdot X$ and g_{n+1} be DO X; g_n ; END.

Let $f_n(\ )$, $g_n(\ )$ be the functions computed by f_n and g_n respectively. Notice that $f_0(x) = x + 1, f_1(x) = 2 \cdot x, f_2(x) = 2^x \cdot x$ and $g_0(x) = x^2, g_1(x) = x^{2^x}$. The standard mathematical definition of these functions is in terms of *iteration*. Namely, for any $h(\ ) \in \mathfrak{F}_1$ define $h^{(0)}(x) = x$, $h^{(n+1)}(x) = h(h^{(n)}(x))$. Then notice that $f_{n+1}(x) = f_n^{(x)}(x)$ and $g_{n+1}(x) = g_n^{(x)}(x)$. (The notion of iteration is extended to vector valued functions in the Appendix.)

Also notice that $g_1(\ )$ and $f_2^{(2)}(\ )$ have the same order of growth (this will be made precise below). This is the essential reason that $\mathrm{SR}_1 = \mathrm{L}_2$. Likewise $g_n(\ )$ and $f_{n+1}^{(p)}(\ )$ for some p have the same order of growth.

Furthermore, $f_n \in \mathrm{L}_n$ and $g_n \in \mathrm{SR}_n$. The following additional facts about f_n and g_n are needed and are easily established by routine inductive arguments. They simply say that f_n, g_n are monotone when $x \geq 2$.

Lemma 4.1.

(a) *$f_n(x) \geq x$, $g_n(x) \geq x$ for all n, all x.*

(b) *$f_n(x) > x$, $g_n(x) > x$ for all n, all $x \geq 2$.*

(c) *$f_n(x) \leq f_m(x)$ for all $n \leq m$, all $x > 1$,*
$g_n(x) \leq g_m(x)$ for all $n \leq m$, all $x > 1$.

(d) *$f_n(x) < f_m(x)$ for all $n < m$, all $x \geq 2$,*
$g_n(x) < g_m(x)$ for all $n < m$, all $x \geq 2$.

(e) *$f_n(x) < f_n(x + 1)$ for all n, all x,*
$g_n(x) < g_n(x + 1)$ for all n, all x.

We now state the critical bounding lemmas.

Lemma 4.2 (Bounding for Loop). *If $\phi_i \in L_m$ and $\phi_i(\ ) : N^n \rightarrow N^p$, then for all $j = 1, \cdots, p, \phi_i(\bar{x})_j \leq f_n^{(|\phi_i|)}$ (max $\bar{x}$) if max $\bar{x} \geq 2$, where $\bar{x} \in N^n$, and max $\bar{x}$ = max $\{x_1, \cdots, x_n\}$.*

Lemma 4.3 (Bounding for SR). *If $\phi_i \in SR_m$ and $\phi_i(\ ) : N^n \rightarrow N^p$, then for all $j = 1, \cdots, p, \phi_i(\bar{x})_j \leq g_n^{(|\phi_i|)}$ (max $\bar{x}$) if max $\bar{x} \geq 2$.*

These lemmas simply state the fairly obvious fact that f_n and g_n are the *fastest growing programs* in L_n and SR_n *for their size*. The details, like using vector valued functions and requiring max $\bar{x} \geq 2$, fall out of the proof technique. The first is a

[10] Such sequences are examples of *spines* and are important in subrecursive hierarchy theory. They are discussed in [8].

convenience, the second is because when $x = 0$ the "loops do not work," and when $x = 1$, $x \cdot x = x$.

One more lemma is required before we can prove Theorem 4.3. We must point out that the classes L_n and SR_n are "closed under the step counting operations." More precisely, given $\phi_i \in \mathrm{L}_n$ we must show that $t\phi_i \in \mathrm{L}_n$. As presently defined, $t\phi_i$ does not have this property, so we modify the definition for $\phi_i \in$ Loop.

If $\phi_i \in$ Loop, then define $t\phi_i$ as follows:

(i) Pick a variable S not in ϕ_i and place $S \leftarrow 0$ before ϕ_i.

(ii) Place $S \leftarrow S + 2$ ($S \leftarrow S + 1$; $S \leftarrow S + 1$) after $S \leftarrow 0$ (this counts the steps needed to define the work variable of G_3 programs used to translate go to's).

(iii) After each assignment of ϕ_i, place $S \leftarrow S + 1$.

(iv) Before each DO v, place $S \leftarrow S + 1$ (to count the loop control variable initialization, $\bar{v} \leftarrow v$).

(v) After each DO v, place $S \leftarrow S + 1$ (to count the conditional, if $\bar{v} \neq 0$ then ____).

(vi) Before each END, add $S \leftarrow S + 1$ (to cover $\bar{v} \leftarrow v \dot{-} 1$ and the go to).

(vii) After each END, add $S \leftarrow S + 1$ (to cover the conditional branch, if $v \neq 0$ then ____, when exiting the loop).

(viii) Replace OUT w by OUT S (thus S, the number of steps, is the outuput).

Call the resulting program $t\phi_i$. To be precise we should prove that the new definition of $t\phi_i$ agrees with the old, but this should be clear from the construction and we accept it as proven.

Lemma 4.4. *If $\phi_i \in L_n$ then $t\phi_i \in L_n$.*

Proof. None of the steps (i)–(viii) increases the depth of nesting.

An easier construction can be given to define $t\phi_i$ for $\phi_i \in$ SR such that

Lemma 4.5. *If $\phi_i \in SR_n$ then $t\phi_i \in SR_n$.*

Proof. For the reader.

We are now ready to prove that $\mathrm{SR}_n \equiv \mathrm{L}_{n+1}$.

Proof of Theorem 4.3 for $n = 1$. We prove the result in detail only for $\mathrm{SR}_1 \equiv \mathrm{L}_2$. The general case follows by similar argument (slightly more complex treatment of $f_{n+1}(\ ) \leq g_n(\ )$ and $g_{n+1}(\ ) \leq f_n^{(2)}(\ )$).

(1) $\mathrm{SR}_1 \subseteq \mathrm{L}_2$. According to Corollary 4.1 we need only show that $\phi_i \in \mathrm{SR}_1$ implies $t\phi_i(x) \leq f(\vec{x})$ for $f \in \mathrm{L}_2$. But $\phi_i \in \mathrm{SR}_1$ implies $t\phi_i \in \mathrm{SR}_1$ by Lemma 4.5 and $t\phi_i(\vec{x}) \leq g_1^{(|\phi_i|)}(\max \vec{x})$.

But notice that $g_1(x) = x^{2^x} \leq 2^{2^x \cdot x} 2^x \cdot x = f_2^2(x)$ if $x \geq 2$. Thus

$$t\phi_i(\vec{x}) \leq f_2^{(2 \cdot |\phi_i|)}(\max \vec{x} + 2) < f_2^{(2|\phi_i|)}(x_1 + \cdots + x_n + 2).$$

The program for $f_2^{(2 \cdot |\phi_i|)}(x_1 + \cdots + x_n + 2)$ is simply

```
$X \leftarrow X_1 + \cdots + X_2 + 2$
$\left.\begin{array}{l} f_2 \\ \vdots \\ f_2 \end{array}\right\} 2 \cdot |\phi_i|$
```

where $X \leftarrow X_1 + \cdots + X_n + 2$ is implemented by

```
X ← 0
DO X_1
X ← X + 1
END
DO X_2
X ← X + 1
END
⋮
DO X_n
X ← X + 1
END
X ← X + 1
X ← X + 1
```

So $f_2^{(2)}(x_1 + \cdots + x_n + 2)$ can be done in L_2.

(2) $L_2 \subseteq SR_1$. We argue in an exactly parallel manner as for (1). In brief $\phi_i \in L_2$ implies $t\phi_i \in L_2$ implies $t\phi_i(\bar{x}) \le f_2^{(|\phi_i|)}$ (max $\bar{x} + 2$) but $f_2(x) \le g_1(x)$ for all $x \ge 2$. So

$$t\phi_i(\bar{x}) \le g_1^{(|\phi_i|)}(x_1 + \cdots + x_n + 2),$$

and the rhs is clearly in SR_1. Q.E.D.

This theorem suggests that the class of functions computed by SR_1 and L_2 might be fundamental. The class turns out to be one which is important in logic and which has been extensively studied, namely the elementary functions, $\mathcal{E}$.

4.4 The Elementary Functions, $\mathcal{E}$. This class is defined algebraically as follows. Introduce the operators for $\bar{x} \in N^n$.

(1)	$\sum$:	partial summation:	$s(\bar{x}, y) = \sum_{i=0}^{y} f(i, \bar{x})$.
(2)	$\prod$:	partial product:	$p(\bar{x}, y) = \prod_{i=0}^{y} f(i, \bar{x})$.
(3)	$\mu\le$:	bounded least number:	$b(\bar{x}, y) = \begin{cases} \mu z \le y[f(z, \bar{x}) = 0], \\ 0 \text{ if no such } z \text{ exists.} \end{cases}$

The functions $s(\ )$, $p(\ )$, $b(\ )$ are said to be obtained from $f(\ )$ by $\sum$, $\prod$, $\mu\le$ respectively.

Let $x{*}{*}y$ denote x^y and then in the notation of Subsection 2.3, item (6), define $\mathcal{E} = [+1, \dot{-}1, 0, +, \dot{-}, \cdot, \div, {*}{*}; Os, \mu\le]$.

It is proved in Grzegorczyk [11] that:

Theorem 4.4. $\mathcal{E} = [+1, \dot{-}1, 0, +, \dot{-}, \div; Os, \sum, \prod]$.

We can easily show:

Theorem 4.5. $\mathcal{E} \subseteq \mathcal{SR}_1$.

Proof. The base functions except for $**$ are in $\mathcal{SR}_1$ since there is a basic SR_1 instruction for each. For $Z \leftarrow X{*}{*}Y$ use $Z \leftarrow 1$; DO Y; $Z \leftarrow Z \cdot X$; END. The closure under Os is contained in $\mathcal{SR}_1$ because the operations of substitution correspond to composition of programs. The only remaining task is showing closure of $\mathcal{SR}_1$ under $\mu\le$. This *cannot* be done directly by simulating, $\mu\le$, with a DO-loop, because the loops cannot be nested. The strategy is to write G_3-programs for $\mu\le$ and show that the run time can be estimated by $g_1^{(p)}(x + q)$ for some p, q. Then apply the simulation theorem, Theorem 4.3.

Suppose for induction on the number of applications of $\mu\le$ that $\phi_f(\ )$ computes

$f(\ )$ and $t\phi_f(x) \leq g_1^{(p)}(x + q)$ for all x. Let $\phi_i(\vec{x}, y) = \mu z \leq y[f(\vec{x}, z) = 0]$. Let ϕ_f have inputs $x_1, \cdots, x_m$ Z and output W, then let ϕ_i be the G_3 program for

```
    IN $X_1, \cdots, X_n, Y$
    $Z \leftarrow 0$
    DO $Y + 1$
    $\phi_f$
    if $W = 0$ then 1
    $Z \leftarrow Z + 1$
    END
    $Z \leftarrow 0$
1   $Y \leftarrow Z$
    OUT $Y$
```

The run time, $t\phi_i(\ )$, is bounded by $\sum_{i=0}^{y} a \cdot t\phi_f(\vec{x}, i) + b$, thus by

$$\sum_{i=0}^{y} a \cdot g_1^{(p)} (\max \{\vec{x}, i\}) + b \leq (y + 1) \cdot a \cdot g_1^{(p)} (\max \{\vec{x}, y\}) + b.$$

Now $(y + 1) \cdot a < g_1^{(p+a)} (\max \{\vec{x}, y\})$, so clearly there exist p^1, q^1 such that

$$(y + 1) \cdot a \cdot g_1^{(p)} (\max \{\vec{x}, y\}) + b < g_1^{(p^1)} (\max \{\vec{x}, y\} + q^1).$$

Thus by induction, the run time of any ϕ_i computing a function in $\mathcal{E}$ is bounded in $\mathcal{SR}_1$, so $\phi_i(\ ) \in \mathcal{SR}_1$. Q.E.D.

The inverse inclusion, $\mathcal{SR}_1 \subseteq \mathcal{E}$, can easily be proved from a general principle well known in the literature. We summarize in the next section the treatment of these principles given in [8], which is a synthesis of Kleene [14], Cobham [6], and Ritchie [22], from the viewpoint of functionals (or relative subrecursion).

4.5. Elementary Indexings and Measures. The programming languages considered here can all be relativized to an arbitrary function $f(\ ) \in \mathfrak{F}$. We simply allow f as a new basic operation and interpret the assignment $w \leftarrow f(v_1, \cdots, v_n)$ as: w receives the value of $f(\ )$ applied to the values of $v_1, \cdots, v_n$. This simple mechanism for relative computability is one of the salient advantages of the language approach to computability.[11]

The concept of an acceptable indexing directly generalizes to relative computability, and the time measure, $\{t\phi_i(\ )\}$, generalizes directly by counting the assignment as a single step. We denote the relative indexing of a measure by $\{\phi_i^f(\ )\}$ and $\{t\phi_i^f(\ )\}$. General relativized $\{m\phi_i^f(\ )\} = \Phi^f$ measures are defined by requiring a relativized measure function, $M^f(\ )$, in the axioms of Subsection 2.7.[12]

We will now outline the approach to elementary measures given in [8]. First we define the (relativized) *computation predicate*, $T(\ )$. This is also known as the *Kleene T-predicate* (see [14]).

(1) $T_{n+2}^f (i, \vec{x}, y)$ iff y is the number of a terminating computation of program ϕ_i^f with $\vec{x} \in N^n$ as inputs. Also write *comp*^{f} $(i, \vec{x}) = \mu z T(i, \vec{x}, z)$, called the *computation function*.

[11] Notice that relative computability is of interest even in a constructive theory if one is not willing to accept Church's thesis. In the constructive setting, Church's thesis has the character of a reduction axiom.

[12] There is a good deal of interesting work to be done in generalizing the notion $\{m\phi_i^f(\ )\}$ correctly.

(2) Define $\mathcal{E}(f(\ )) = [+1, \dot{-}1, 0, +, \dot{-}, \cdot, \div, **, f(\ ); Os, \mu\leq]$, the functions *elementary in* $f(\ )$ (or elementary *relative to* $f(\ )$). A predicate $P(\ )$ is elementary in $f(\ )$ iff its characteristic function $ch_p(x) =$ if $P(x)$ then 1 else 0, belongs to $\mathcal{E}(f(\ ))$.

(3) The Kleene normal form theorem asserts that $\exists U(\ ) \in \mathcal{E}$ and $T^f_{n+2}(\ ) \in \mathcal{E}(f(\ ))$ such that $\phi_i^f(\bar{x}) = U(\mu y T^f_{n+2}(i, \bar{x}, y))$.

(4) An acceptable relativized indexing will be called *elementary* iff (3) holds for $\phi_i^f(\ )$. A measure, $\{m\phi_i^f(\ )\}$, is *elementary* iff

(a) $\exists h(\ ) \in \mathcal{E}(f(\ ))$ such that $comp^f\ (i, x) \leq h(m\phi_i^f(x))$ and

(b) $\phi_i^f(\ ) \in \mathcal{E}(f(\ ))$ implies $m\phi_i^f(\ ) \in \mathcal{E}(f(\ ))$.

(A third condition which is natural but unnecessary here is

(c) $M^f(\ )$, the measure function, belongs to $\mathcal{E}(f(\ ))$.)

It is now easy to verify the following critical principle. By way of abbreviation we use $\phi(\ ) \leq \psi(\ )$ iff $\phi(x) \leq \psi(x)$ for all x; and if $\mathcal{C} \subseteq \mathfrak{F}$ then write $\phi(\ ) \leq \mathcal{C}$ iff $\exists g(\ ) \in \mathcal{C}$ and $\phi(\ ) \leq g(\ )$.

THEOREM 4.6[13] (Ritchie-Cobham). *If* $\{\phi_i^f(\ )\}$ *and* $\{m\phi_i^f(\ )\}$ *are elementary, then*

$$m\phi_i^f(\ ) < \mathcal{E}(f(\ )) \quad \textit{iff} \quad \phi_i^f(\ ) \in \mathcal{E}(f(\ )).$$

PROOF. For simplicity, consider only $\phi_i(\ ) \in \mathcal{PR}_1$. The "if condition" is immediate from the definition. For the "only if" part, note that from (3): $\phi_i^f(x) = U(\mu y T^f(i, x, y))$, and since $\{\phi_i^f(\ )\}$ is elementary, $U(\ ), T^f(\ ) \in \mathcal{E}(f(\ ))$. Since $\{m\phi_i^f(\ )\}$ is elementary, the y above satisfies $y \leq h(m\phi_i^f(x))$ for $h(\ ) \in \mathcal{E}(f(\ ))$. Define $\bar{h}(x) = \sum_{i=0}^{x} h(i) + 1$; then $h(\ ) < \bar{h}(\ )$ and $\bar{h}(\ )$ is increasing. Also, $\bar{h}(\ ) \in \mathcal{E}(f(\ ))$ (recall Theorem 4.4). Since $m\phi_i^f(\ ) \leq \mathcal{E}(f(\ ))$, $\exists g(\ ) \in \mathcal{E}(f(\ ))$ such that $m\phi_i^f(x) \leq g(x)$ for all x. So $y \leq h(m\phi_i^f(x)) < \bar{h}(m\phi_i^f(x)) \leq \bar{h}(g(x))$ and $\bar{h}(g(\ )) \in \mathcal{E}(f(\ ))$. Define $s(i, x, y) = U\ (\mu z \leq y T^f(i, x, z))$; then $s(\ ) \in \mathcal{E}(f(\ ))$ and $s(-, -, \bar{h}(g(\ ))) \in \mathcal{E}(f(\ ))$. Since $\phi_i^f(x) = s(i, x, \bar{h}(g(x)))$, $\phi_i^f(\ ) \in \mathcal{E}(f(\ ))$. Q.E.D.

Remark. A class $\mathcal{C}$ satisfying the condition that $m\phi_i^f(\ ) \leq \mathcal{C}$ implies $\phi_i^f(\ ) \in \mathcal{C}$ is called *full* wrt Φ^f. If also $\phi_i^f(\ ) \in \mathcal{C}$ implies $\exists\phi_j^f(\ ) = \phi_i^f(\ )$ and $m\phi_j^f(\ ) \leq \mathcal{C}$, then $\mathcal{C}$ is called *closed* wrt Φ^f.

The statement of Theorem 4.6 is now that "the classes $\mathcal{E}(f(\ ))$ are full and closed wrt to any elementary measure Φ^f." It is an interesting open problem to find the *least* closed and full class wrt $\{t\phi_i^f(\ )\}$.

It is well known that all reasonable or natural formulations of abstract machine and language models (e.g. Turing machines, G_3-programs, etc.) are elementary as are the usual measures of computational complexity (time and tape, for instance) on them. Cobham [6] argues this explicitly for a subset $\mathcal{E}^2$ of $\mathcal{E}$. $\mathcal{E}^2$ are called the *primary functions*.

The reason for this is that the elementary functions allow most all functions and predicates which are used in combinatorial description. Furthermore, machine and language models are intentionally constructed by simple means from simple bases. We summarize this information as

THEOREM 4.7. *The relativized* G_3 *indexing and time measure,* $\{\phi_i^f(\ )\}$ *and* $\{t\phi_i^f(\ )\}$, *are elementary.*

[13] The historical origins of this theorem can be found in Kleene's treatment of primitive recursive functions. N. A. Routledge called the theorem "Kleene's principle." The exact version given here was first due to Ritchie [22] and later explicitly discussed by Cobham [6].

Proof. See Minsky [21] or Shepherdson and Sturgis [26].

This fact means that all the specific languages and measures considered here are elementary. From this we can easily prove:

Theorem 4.8. $\mathcal{E} = \mathcal{SR}_1$.

Proof. Again consider only one argument functions for simplicity. By Theorem 4.5 we need only show that $\mathcal{SR}_1 \subseteq \mathcal{E}$. But if $\phi_i(\ ) \in \mathcal{SR}_1$, then $t\phi_i(\ ) \in \mathcal{SR}_1$ and in fact $t\phi_i(x) \leq g_1^{(|t\phi_i|)}(x + q)$. But $g_1(\ )$ is defined in $\mathcal{E}$ by the expression $x{*}{*}(2{*}{*}x)$ (i.e. x^{2^x}), which is clearly elementary. Thus we have shown $\phi_i \in \mathcal{SR}$ implies $t\phi_i(\ ) \leq \mathcal{E}$. So by Theorems 4.6 and 4.7, $\phi_i(\ ) \in \mathcal{E}$. Q.E.D.

It is most natural to inquire about generalizing the equalities $\mathcal{E} = \mathcal{SR}_1 = \mathcal{L}_2$ to $\mathcal{E}(f(\ ))$, $\mathcal{SR}_1(f(\ ))$ and $\mathcal{L}_2(f(\ ))$. The classes are all well defined. Unfortunately the classes are not equal. This points up a deficiency in the L_2 and SR_1 definitions of $\mathcal{E}$. They do not provide a definition of the relativized classes.

The classes $\mathcal{SR}_1(f(\ ))$ and $\mathcal{L}_2(f(\ ))$ are not as interesting mathematically as $\mathcal{E}(f(\ ))$ because they are not as stable; they are not functionally closed, i.e. if $g(\ ) \in \mathcal{SR}_1$, it is not necessarily the case that $\mathcal{SR}_1(g(\ )) \subseteq \mathcal{SR}_1$.

We now verify the class inequalities.

Fact 4.1. $\mathcal{L}_2(f(\ )) \neq \mathcal{SR}_1(f(\ )) \neq \mathcal{E}(f(\ ))$ *for all* $f(\ ) \in \mathcal{F}_1$.

Take $f(\ ) = f_2(\ )$, then $f_4(\ ) \in \mathcal{L}_2(f_2(\ ))$ but $f_4(\ ) \notin \mathcal{E}(f_2(\ )), f_4(\ ) \notin \mathcal{SR}_1(f_2(\ ))$. Also $f_3(\ ) \in \mathcal{SR}_1(f_2(\ ))$ but $f_3(\ ) \notin \mathcal{E}(f_2(\ ))$. These observations all follow directly from the bounding theorems for L and SR and from the fact that $\mathcal{E}(f_2(\ )) = \mathcal{E} = \mathcal{SR} = \mathcal{L}_2$. The same observations yield:

Theorem 4.9. (Grzegorczyk hierarchy).

(*i*) $\mathcal{E}(f_n(\ )) \subset \mathcal{E}(f_{n+1}(\ ))$ *for all* $n \geq 2$,

(*ii*) $\bigcup_{n=0}^{\infty} \mathcal{E}(f_n(\ )) = \mathcal{SR} = \mathcal{R}^1$.

It can also be shown by arguments similar to those in Theorem 4.3 that:

Theorem 4.10. *For all* $n \geq 1$, $\mathcal{E}(f_{n+1}(\ )) = \mathcal{SR}_n = \mathcal{L}_{n+1}$.

5. *Computational Efficiency and Program Structure*

5.1. Relative Efficiency. We know from the work of Blum that the availability of the "negative go to" allows a programmer to "compress his code." That is, GR programs can be much shorter than the *shortest* SR programs for some $\mathcal{R}^1$ functions. How does the "negative go to" effect computational efficiency measured in terms of running time?

The best result previously known (Meyer and Ritchie [19]) is that if ϕ_i denote G_3 programs and β_i denote Loop programs, then if $t\phi_i(\ ) < f_n^{(p)}(\ )$ there is a $\beta_i(\ ) = \phi_i(\ )$ and $t\beta_i(\ ) < f_n^{(p)}(\ )$.

There is, however, considerable latitude among run times bounded by $f_n^{(p)}(\ )$, and the previous best result leaves open the possibility that for *every* (increasing) primitive recursive function, $h(\ ) \in \mathcal{R}^1$, there exists some $\phi_i(\ ) \in \mathcal{R}^1$ such that every Loop program β_j for $\phi_i(\ )$ satisfies $h(t\phi_i(x)) < t\beta_j(x)$ for all $x > m$ for some m. That is, some G_3 programs may be arbitrarily more efficient than the best Loop programs for the same function, because there might be arbitrarily large primitive recursive gaps between G_3 and Loop run times for the same function. The main result of Theorem 5.1 shows that there are no such gaps.

To facilitate stating the results of this section, let $\{\alpha_i(\ )\}$ be an indexing of SR and $\{\beta_i(\ )\}$ be an indexing of Loop, and $\{\phi_i(\ )\}$ an indexing of G_3. As before, α_i, β_i will actually be the G_3 images of SR and Loop programs, so $\{\alpha_i(\ )\}$ and $\{\beta_i(\ )\}$ are sublists of $\{\phi_i(\ )\}$.

For simplicity, we state the main theorem only for one argument functions. The extension to multiple arguments is straightforward from the principles of the Loop and SR bounding theorems (Lemmas 4.2 and 4.3).

THEOREM 5.1 (Relative efficiency). *If $\phi_i \in G_3$ and $\phi_i(\ ) \in \mathcal{R}^1$ and $t\phi_i(x) \geq x$, then there exist $\alpha_{i_1} \in SR$, $\beta_{i_2} \in$ Loop and constants c_{i_1}, c_{i_2} such that*

(*a*) $t\alpha_{i_1}(x) \leq c_{i_1} \cdot t\phi_i(x)$ *and*
$\alpha_{i_1}(x) = \phi_i(x)$ *for all* x,

(*b*) $t\beta_{i_2}(x) \leq c_{i_2} \cdot (t_i(x))^2$ *and*
$\beta_{i_2}(x) = \phi_i(x)$ *for all* x.

The α_{i_1} and β_{i_2} can be found effectively. More precisely, there exist $p_j(\ )$ and $q_j(\ )$, $j = 1, 2$, such that if $t\phi_i(x) \leq f_n^{(m)}(x)$ for all x, then

(a') $t\alpha_{p_1(i,n,m)}(x) \leq q_1(i, n, m) \cdot t\phi_i(x)$ *and*
$\alpha_{p_1(i,n,m)}(x) = \phi_i(x)$ *for all* x *and*
$\alpha_{p_1(i,n,m)} \in SR_n$ *if* $n \geq 2$.

(b') $t\beta_{p_2(i,n,m)}(x) \leq q_2(i, n, m) \cdot t\phi_i(x)$ *and*
$\beta_{p_2(i,n,m)}(x) = \phi_i(x)$ *for all* x *and*
$\beta_{p_2(i,n,m)} \in L_{n+1}$.

COROLLARY 5.1. *The list $\{\alpha_i(\ )\}$ can also be taken to be an indexing of* [$+1, \dot{-}1, 0$, DO ( ), END] *and the results* (*a*) *and* (a') *hold with $\alpha_{p_3(i,n,m)}$ having depth of nesting $n + 1$.*

DISCUSSION OF THE THEOREM. The results say that restrictions on program structures in going from G$_3$ to SR cost only a multiplicative efficiency factor, and in going from G$_3$ to Loop the restrictions cost at most a square. Corollary 5.1 says that the nonlinearity results only from the cost of simulating subtraction.

DISCUSSION OF THE PROOF. The idea of the proof is simple. We will use a Loop program,

```
DO $S$
$\phi_i^{**}$
END
```

or an SR program,

```
DO $S$
$\phi_i^{+}$
END
```

(just as in Section 4), to simulate ϕ_i.

The key factor in controlling the efficiency of the simulation is calculation of the bound S "in parallel" with the simulation. That is, we construct a "clocklike mechanism"; the clock will run for exactly $f_n^{(p)}(x)$ steps unless it is shut off.[14] The simulation continues as long as the clock is running. If the simulation finishes before the clock, then it sends a signal to stop the clock. In the Loop language the clock cannot stop immediately, there is an overrun factor. Estimation of this factor is a critical step in the proof (Lemmas 5.3 and 5.4).

[14] The fact that there is a clock for $f_n{}^p(\ )$ depends on the "honesty" of the function $f_n(\ )$, i.e. its values are close to its run times.

The "clock-simulation" argument used here is typical of a certain class of diagonalization arguments in complexity theory (called downward diagonalization in [7] and discussed at length in [13]), but it is used for a different purpose here.

PROOF. In the first part of the proof we construct one part of the parallel procedure, the simulation of $\phi_i(\ )$, and show that it works correctly. In the second part we build the clock mechanism into the simulation and show why it works correctly.

Part I:

(1) Given ϕ_i, construct $\phi_i^{**} \in L_1$ just as in Corollary 4.1 of Subsection 4.2. Again assume that H does not appear in ϕ_i. The correctness of ϕ_i^{**} will follow from the arguments of Subsection 4.2. This finishes the simulation part for Loop.

(2) Construct an SR_0 program, ϕ_i^+, to simulate ϕ_i just as in Theorem 4.2, Subsection 4.2 (assume G does not occur in ϕ_i). The correctness of ϕ_i^+ will follow from the arguments of Subsection 4.2.

Notice $\phi_i^+ \in SR_0$ and $\phi_i^{**} \in L_1$. Now let $\bar{\phi}_i$ denote either ϕ_i^+ or ϕ_i^{**}. Then

```
IN $X$
$H \leftarrow 1$
DO WHILE $H \neq 0$
$\bar{\phi}_i$
END
OUT $Y$
```

is equivalent to ϕ_i. Our task in the next part is to simulate the DO WHILE iterative by a clocklike mechanism.

If $\phi_i \in G_3$ and $\phi_i(\ ) \in R^1$, then for purposes of the theorem we can assume that there exist n_i, p_i such that $t\phi_i(x) \leq f_{n_i}^{(p_i)}(x)$ for all x. This is because there must be some α_j which computes $\phi_i(\ )$, i.e. $\alpha_j(x) = \phi_i(x)$ for all x, and the G_3 program, ϕ_k, which simulates the execution of α_j and ϕ_i in parallel and stops as soon as one of them stops, has a run time which is less than $c \cdot \min(t\phi_i(x),\ t\alpha_j(x))$ for some constant c. So $t\phi_k(x) \leq c \cdot t\phi_i(x)$ and $\exists n_i, p_i\ t\phi_k(x) \leq f_{n_i}^{(p_i)}(x)$.

Part II:

(1) Suppose now that $t\phi_i(x) \leq f_{n_i}^{(p_i)}(x)$ for all x. The goal of this step is to describe a way to compute the clock in parallel with $\bar{\phi}_i$ and shut it off (without much "overrun") when ϕ_i halts.[15] The asterisk will indicate the critical statement needed.

```
*    if $H \neq 0$ then $X \leftarrow Z$ else $X \leftarrow 0$
```

In SR, * becomes

```
$l$ if $H \neq 0$ then $+\ (l+3)$        so we get    $l$ if $H \neq 0$ then $+\ (l+3)$
$l+1$   $X \leftarrow 0$                               $l+1$   $X \leftarrow 0$
$l+2$   go to $+\ (l+4)$                               $l+2$   go to $+\ (l+4)$
$l+3$   $X \leftarrow Z$                               $l+3$   $X \leftarrow Z$
```

In Loop * becomes

```
$X \leftarrow 0$
DO $H$
$X \leftarrow Z$
END
```

[15] We need not make the concept of a clock precise here. It is done in [13] and [8].

Now form a program δ_{i_1} (if $\bar{\phi}_i = \phi_i^{+}$) or δ_{i_2} (if $\bar{\phi}_i = \phi_i^{**}$) or δ for short.

```
DO $X$              ⎫
 .                  ⎪
    .               ⎪
       .            ⎬ $n_i$
  DO $X$            ⎪
    DO $X$          ⎪
      DO $X$        ⎭
      $\bar{\phi}_i$
      $Z \leftarrow Z + 1$
      END ⎫
        * ⎪
    END   ⎪
      *   ⎪
  END     ⎬ $n_i$
    *     ⎪
       .  ⎪
     .    ⎪
   .      ⎪
END       ⎭
```

Looking at the innermost loops we see the mechanism in more detail.

```
DO $X$
  DO $X$
  $\bar{\phi}_i$
  $Z \leftarrow Z + 1$
  END
  if $H \neq 0$ then $X \leftarrow Z$ else $X \leftarrow 0$
END
```

Observe that as long as $H \neq 0$ this program will compute $f_n(x)$ in variable Z, since the program is essentially

```
DO $X$          ⎫
 .              ⎪
   .            ⎬ $n_i$ times
     .          ⎪
  DO $X$        ⎪
    DO $X$      ⎭
    $Z \leftarrow Z + 1$
    END         ⎫
  $X \leftarrow Z$         ⎪
  END           ⎪
     .          ⎬ $n_i$ times
    .           ⎪
  .             ⎪
$X \leftarrow Z$           ⎪
END             ⎭
```

Furthermore, while $H \neq 0$ the program is simulating at least one-half step of ϕ_i every time $Z \leftarrow Z + 1$ is executed. Thus while $H \neq 0$ the value of Z indicates a lower bound on the number of steps of ϕ_i which $\bar{\phi}_i$ has "simulated."

To compute the final result, $\phi_i(x)$, form IN X; $H \leftarrow 1$; $\overbrace{\delta; *; \delta; *; \cdots; \delta}^{p_i \text{ times}}$; OUT Y. The result is either α_{i_1} or β_{i_1} depending on δ.

Now Z will potentially have the value of $f_{n_i}^{(p_i)}(x)$. Its actual value will depend on the value it has when H becomes zero, i.e. when $\bar{\phi}_i$ shuts off, i.e. when $\phi_i(x)$ halts. In the next step we determine how long α_{i_1} or β_{i_2} will run compared to $t\phi_i$.

(2) To calculate $t\alpha_{i_1}(x)$, $t\beta_{i_2}(x)$, four facts are needed about α_{i_1} and β_{i_2}. Let $D_1, \cdots, D_{n_i}$ be the loop control registers in δ (listed in order with the innermost loop first). The following hold for all inputs x.

LEMMA 5.1. *After executing the innermost DO, at every step of the computation,* $Z \geq D_i$ *for* $i = 1, \cdots, n_i$.

LEMMA 5.2. *After the first execution of the innermost loop,* $D_1 \leq Z \leq$ *number of times instructions of* $\bar{\phi}_i$ *have been executed. Also,* $C \cdot Z \leq$ *number of steps of* ϕ_i *already simulated, for some constant* C.

LEMMA 5.3. *If* $H = 0$, *then at most* $3 \cdot (D_1 + D_2 + \cdots + D_{n_i}) + n_i + D_1$ *steps can be executed before* δ *halts.*

LEMMA 5.4. If $H = 0$, *then the maximum value of* X *is* Z.

Using these lemmas, conditions (a′) and (b′) of the theorem are easily established as follows.

(a) First consider ϕ_i^+. When ϕ_i^+ halts (i.e. $H = 0$), ϕ_i^+ has been executed no more than $C \cdot t\phi_i(x)$ steps ("on the average" ϕ_i^+ is probably executing nearly one for one).[16] Thus, when ϕ_i^+ halts, $H = 0$, and $D_1 \leq X \leq Z \leq C \cdot t\phi_i(x)$, by Lemmas 5.1, 5.2, and 5.4. The total number of steps taken outside ϕ_i^+ is no more than $(4n_i) \cdot Z$, thus no more than $4 \cdot n_i \cdot C \cdot t\phi_i(x)$. When ϕ_i^+ halts, control is in some δ and will not go into another δ. By Lemma 5.3, δ can execute at most $4 \cdot \sum_{j=i}^{n_i} D_j + n_i$ more steps. So that by Lemmas 5.1 and 5.2, at most $4 \cdot Z + n_i \leq 4 \cdot (C \cdot t\phi_i(x) + n_i)$ more steps. Therefore to complete the program we add at most $2 \cdot p_i$ more steps in slipping over unused δ's to complete $\alpha_{p_1(i,n_i,p_i)}$. Hence at least

$$t\alpha_{i_1}(x) \leq (4 \cdot C) \cdot (n_i + 1) \cdot t\phi_i(x),$$

where $i_1 = p_1(i, n_i, p_i)$.

(b) Now consider the case of ϕ_i^{**}. Several steps must be executed for each step of ϕ_i because the switches must be computed (at a maximum cost of $C_1 \cdot p$ statements for SW$(H_1, \cdots, H_p)$), and subtraction must be simulated (at a maximum cost of $C_2 \cdot v$ steps where v is the values of the variable being decremented). In the worst case this added cost is $g(x) = |\phi_i| \cdot C_3 \cdot$ (maximum value of variables in ϕ_i in input x) as a multiplicative factor. Clearly the maximum value of variables in ϕ_i on x is $x + t\phi_i(x)$. Thus $g(x) \leq C_3 \cdot |\phi_i| \cdot (x + t\phi_i(x))$. Recall $t\phi_i(x) \geq x$ for all x, thus $g(x) \leq C_4 \cdot t\phi_i(x)$.

Now by the same reasoning as in case (a) above, we can conclude

$$t\beta_{i_2}(x) \leq (4 \cdot (n_i + 1) \cdot C_4 \cdot t\phi_i(x)) \cdot t\phi_i(x),$$

so $t\beta_{i_2}(x) \leq C \cdot (t\phi_i(x))^2$, where $i_2 = p_2(i, n_i, p_i)$.

[16] If we could count assignments of the type $w \leftarrow n$ as a single step, then the simulation is close to $2 \cdot t\phi_i(x)$.

To complete each of (a) and (b) we need only prove the lemmas.

(3) The proofs of the lemmas are as follows.

PROOF OF LEMMA 5.1.

(1) After the first time through innermost loop, $Z = X$ and no D_i has been increased.

(2) Assume the result true after m steps, to prove that it is true after $m + 1$ steps. At each step only three instruction types can change values. They are

(i) $D_i \leftarrow X$
(ii) $Z \leftarrow Z + 1$
(iii) $D_i \leftarrow D_i - 1$
(iv) $X \leftarrow Z$

Therefore, if $D_i \leq Z$ at m, then (i) can at worst bring some $D_i = X$ which by (iv) is $\leq Z$. The other two instructions can only cause $D_j < Z$. for some j. Q.E.D.

PROOF OF LEMMA 5.2.

(1) Z cannot be increased unless an instruction of $\bar{\phi}_i$ is executed. Therefore $Z <$ number of steps taken in $\bar{\phi}_i$.

(2) Every step of ϕ_i^+ either directly carries out a step of ϕ_i or else carries out the step of ϕ_i after one loop and C extra steps, thus after increasing Z. Thus $C \cdot Z \leq$ number of steps of ϕ_i already simulated.

(3) The argument for ϕ_i^{**} is similar to 2. Q.E.D.

PROOF OF LEMMA 5.3.

(1) If $H = 0$ then by the $*$ statement, the only value that can be assigned to D_i is 0. Also when $D_i = 0$ then the only statements executed in the D_{i+1} loop are "$D_{i+1} \leftarrow D_{i+1} \dot{-} 1$", "go to ____" and "if $D_i \neq 0$ then ____" so that after $3 \cdot D_{i+1}$ steps, $D_{i+1} = 0$.

(2) After $D_1 = 0$, then $3D_2 + 1 + 3D_3 + 1 + \cdots + 3D_n$ steps are executed. D_1 may execute $4D_1$ steps before being set to 0 (the "go to G" is also executed). Q.E.D.

PROOF OF LEMMA 5.4. Trivial by examining $*$. Q.E.D. (Theorem 5.1)

Discussion. The estimate produced in the proof is very crude. There are two basic factors influencing the cost of α_{i_1}, β_{i_2}:

(A) simulation time, the cost of

```
DO $V$
$\bar{\phi}_i$
END
```

and (B) clock time. The clock time, (B), has two subcosts: (i) computation time while the clock is still needed, and (ii) overrun time, the time the clock keeps running after it is no longer needed (after $\bar{\phi}_i$ halts). The cost of (i) is inescapable but is minimized by computing it in parallel. This cost is reflected in the factor $4n_i \cdot Z$, the *time spent outside of* $\bar{\phi}_i$. Notice that the (B) cost depends on n_i, an index reflecting the complexity of the clock. The cost (ii) can be eliminated in the case of SR and it will allow us to reduce the value of the constant C_i in (a). This is done by placing "if $H = 0$ then $+ d$" immediately after "$H \leftarrow 0$" in ϕ_i^+, where $+d$ refers to the output statement which is outside of all loops.

The simulation cost (A) depends on the "structural complexity" of ϕ_i measured in terms of the number and distribution of negative go to's. The value of Z, which determines the nondirect simulation cost as well as the clock cost, actually measures the number of times that negative go to's are executed. Thus if there are few negative go to's, then $t\phi_i^+(\ )$ and $t\phi_i(\ )$ may be very close. The topic of structural complexity and efficiency will be discussed further in Subsection 5.3.

In comparing the structure restrictions on α_{i_1} and β_{i_2} we see that β_{i_2} has a larger nesting complexity than α_{i_1}. Furthermore, if $t\phi_i(x) \leq f_n^{(p)}(x)$ for all x and $n \geq 2$, we know that there is a $\beta_j \in L_n$ for $\phi_i(\ )$. How does the efficiency of β_j compare with that of β_{i_2}? We can say the following.

THEOREM 5.2. *If $\phi_i(\ ) \in \mathcal{R}^1$ and $t\phi_i(x) \leq f_n^{(p)}(x)$ for all x and for $n \geq 2$, then $\exists \beta_j \in L_n$ such that $\beta_j(\ ) = \phi_i(\ )$ and*

$$t\beta_j(x) \leq s(max\{x, t\phi_i(x)\}) \quad \text{for all } x,$$

where $s(x) = 2^x \cdot x$.

PROOF. Replace the two innermost nested loops of the program β_i of Theorem 5.1 with

```
DO X
  *
DO Z
Z ← Z + 1
END
φ_i**
  *
END
```

The resulting program has nesting n as desired, and its run time behaves as claimed for reasons similar to those detailed in Theorem 5.1. Q.E.D.

This theorem illustrates another aspect of the trade-off between structural and computational complexity. In summary, the theorems of this section have determined the cost of putting programs into certain normal forms or restricted forms.

5.2. *Minimum Growth Rates.* The main theorem was proved with the restriction that $t\phi_i(x) \geq x$ for all x. This restriction is necessary because Loop programs constructed as in the proof cannot run in less than x steps. For GR functions, running times below x are possible if the base functions, A_8, are all assigned a cost of one step. However, all languages mentioned, G_3, GR, SR, Algol-R_0^1, and Loop, have a *strong minimum growth rate* in the following sense: there is a recursive monotonic increasing function $\lambda(\ )$ such that if $\lim_{x\to\infty} \inf \phi_i(x) = \infty$, then $t\phi_i(x) \geq \lambda(x)$ for all x except possibly those in a finite set F (write e.f.s. for except on a *f*inite set). That is, if the run times grow, they must grow at least at the rate of $\lambda(\ )$. Given a strong min growth rate $\lambda(\ )$ for the general recursive language GR and the time measure, we know:

COROLLARY 5.2. *There is a recursive function λ^{-1} such that for all $\phi_i(\ ) \in \mathcal{R}^1$ there is an $\alpha_j(\ ) = \phi_i(\ )$ such that $t\alpha_j(x) \leq \lambda^{-1}(max\{x, t\phi_i(x)\})$ for all x.*

THEOREM 5.3. *GR and G_3 have strong minimum growth rates.*

PROOF. We prove first that G_3 has a strong min growth rate $\lambda_3(\ )$ and then show that growth rate in GR can be bounded in terms of λ_3.

(1) The strong minimum growth rate for G_3 is $\lambda_3(x) = x$. To prove this, consider any one argument G_3 program ϕ_i (suppose input is X, output is Y). To determine the minimum growth rate we ask how few steps ϕ_i can take on input x and still have a growing run time. This can be estimated by working backward; given a run time value, $t\phi_i(x) = k$, how large can x be?

If $t\phi_i(x) = k$, then we can write down a finite tree of all possible paths of execution of length k (if there are no conditionals, then the tree has only one branch).

On the edges after each decision node, the condition on the variable being tested is written down. Since we are estimating x, we need to record only the condition on x. These are always of the form $X - n = 0$ or $X - n > 0$ since the conditional is, if $X \neq 0$ then ____. We now consider two possible types of terminating branches in the execution tree (the branch is terminating if it causes OUT Y as the last node).

(A) The last decision on this path was $X - n = 0$. In this case $n \leq k$ because at most k operations can be performed on X. Therefore, the maximum value of X is k and we conclude that $x \leq k$; so the growth rate is $t\phi_i(k) \geq k$, and the growth function is $\lambda_3(x) = x$.

(B) The last decision on the path was $X - n > 0$. In this case, for all $x > n$ the program terminates in k steps. Therefore lim inf $t\phi_i(x) < \infty$ which violates the hypothesis. So no such path exists.

Since only case A can hold, the growth rate is $\lambda_3(x) = x$.

(2) To establish a growth rate for GR, notice that since GR can be translated uniformly into G_3, there is for each GR arithmetic function (say $x \dot{-} y$) a cost $s_i(x, y)$ in terms of G_3. If $t\phi_i(x) = y$, then the simulation cost using G_3 can be determined. Let $S(x, y) = \sum_{i=1}^{p} s_i(x, y)$ for p the number of arithmetic instructions of GR. Then $S(\ )$ bounds the cost of simulating any GR arithmetic operation. Thus since $S(x, y)$ is monotone in x, y, the simulation cost will be at most $S(v_1, v_1) + S(v_2, v_2) + \cdots + S(v_y, v_y)$, where v_i is the maximum value in any variable at step i.

This maximum value v_i can be determined as a function of v_0, the maximum initial value, and y the number of steps. The time measure $\{t\phi_i(\ )\}$ has a *speed limit*, *sl*, that is, in y steps a program with maximum initial value v_0 cannot produce a value larger than $sl(v_0, y)$. Thus after y steps, $v_y \leq sl(v_0, y)$. Since $S(\ )$ is monotone the value $y \cdot S(sl(v_0, y), sl(v_0, y)) = t(v_0, y)$ will be the maximum number of simulation steps required. The function $t(\ )$ is increasing in v and y, and because of the e.f.s. conditions on min growth rate we need only consider $T(y) = t(y, y)$. Since T is increasing, T^{-1} is defined.

The minimum growth rate in GR, say λ, must satisfy $\lambda(x) > T^{-1}(\lambda_3(x))$ e.f.s.

Q.E.D.

The idea of a speed limit which appears in this proof will be of interest to us in Section 6, on abstract subrecursive complexity measures.

To finish this section we note that Theorem 5.1 is not constructive in the sense that given ϕ_i we cannot determine n_i and p_i effectively.

THEOREM 5.4. (*a*) *There is no algorithm to determine for any GR program* ϕ_i *whether* $\phi_i(\ ) \in \mathcal{R}^1$. *If* $\phi_i(\ ) \in \mathcal{R}^1$, *then there is an* n *such that* $\exists p$ *and* $*$ $t\phi_i(x) \leq f_n^{(p)}(x)$ *for all* x. (*b*) *However, given the information that* $\phi_i(\ ) \in \mathcal{R}^1$, *there is no algorithm to determine an* n *satisfying* $*$. (*c*) *Moreover, given the information that* $\phi_i(\ ) \in \mathcal{L}_n$, *there is no algorithm to find the least* p *satisfying* $*$.

PROOF. Case (a): This is a well-known fact. It is proved by embedding the halting problem in the decision. Namely design $\phi_{\sigma(i,n)}$ such that on input x it runs $\phi_n(n)$ for x steps. If this halts, it then computes a nonprimitive recursive function. If it does not halt, it computes the successor function $X \leftarrow X + 1$. Knowing whether $\phi_{\sigma(i_0,n)} \in \mathcal{R}^1$ is equivalent to knowing whether $\phi_n(n) \downarrow$.

Cases (b) and (c) are similar.

5.3. *Speed-Up Theorem for* $\mathcal{R}^1$. One of the most interesting theorems in the theory of computational complexity is Blum's "speed-up" theorem.

THEOREM 5.5. *For all $r(\ )$, $a(\ )$ in $\mathcal{R}$ there is an $f(\ ) \in \mathcal{R}$ such that*

(i) $f(x) \geq a(x)$ e.f.s.,

(ii) for all $\phi_{i_j}(\ ) = f(\ )$, there is a $\phi_{i_{j+1}}(\ ) = f(\ )$ such that $r(x, t\phi_{i_{j+1}}(x)) > t\phi_{i_j}(x)$ e.f.s.

This theorem is proved in Blum [1] and Hartmanis and Hopcroft [13].

This says that there are peculiar functions whose computation time can be "sped up" by an arbitrary amount $r(\ )$ almost everywhere. However, Blum has shown that the speed-up cannot be effective in the following sense.

THEOREM 5.6. *Let $r(\ ) \in \mathcal{R}$ be any sufficiently large function. Let $f(\ ) \in \mathcal{R}$; then there does not exist a program π such that if $\phi_i(\ ) = f(\ )$, then $\pi(i)$ halts and $r(x, t\phi_{\pi(i)}(x)) < t\phi_i(x)$ e.f.s. and $\phi_{\pi(i)}(\ ) = f(\ )$.*

In the case of GR programs and the time measure, $\{t\phi_i(\ )\}$, "sufficiently large r" means $r(x, y) > y^2$ e.f.s. Thus there is no way to go effectively from ϕ_{i_j} to $\phi_{i_{j+n}}$ for all n.

The noneffectiveness of the speed-up means that it is impossible to exhibit examples of square speed-up in GR. For the purpose of illustrating the speed-up theorem, this is disappointing. (In fact from a constructive point of view, the result is a "non-speed-up" theorem.) One might thus ask whether square speed-ups could be illustrated in the Loop language or some subrecursive language where the structure is simple. This question has occurred to several people. The first step in answering it is to prove an $\mathcal{R}^1$-speed-up theorem using a simple language like Loop. One would aim to prove:

THEOREM 5.7. *For all $r(\ )$, $a(\ ) \in \mathcal{R}^1$ there is an $f(\ ) \in \mathcal{R}^1$ such that $f(\ ) \geq a(\ )$ e.f.s., and for all $\alpha_{i_j}(\ ) = f(\ )$ there is an $\alpha_{i_{j+1}}(\ ) = f(\ )$ such that $r(x, t\alpha_{i_{j+1}}(x)) < t\alpha_{i_j}(x)$ e.f.s.*

This theorem cannot be proved by carrying out the Blum [1] proof directly to $\mathcal{R}^1$. It can, however, be proved using different methods, for example, those in [12] and [17]. However, it has not been shown that this $\mathcal{R}^1$-speed-up is noneffective.

From Theorem 5.1 it is possible to easily prove the above Theorem 5.7 and to prove directly that for sufficiently large r the speed-up cannot be effective. Namely, the proof is to apply Blum's proof for a given $r(\ ) \in \mathcal{R}^1$ to yield an $f(\ ) \in \mathcal{R}^1$ function with $r(\ )$ speed-up in GR. Then by Theorem 5.1 the SR programs also have r speed-up for $r(x, y) \geq y^2$ e.f.s. Finally the speed-up cannot be effective in SR because it would lead to an effective GR speed-up by the following argument.

In more detail, suppose π speeds up SR programs in the sense that if $\alpha_i(\ ) = f(\ )$; then $\pi(i)$ halts and $r(x, t\alpha_{\pi(i)}(x)) < t\alpha_i(x)$ e.f.s. Then define a program $\bar{\pi}$ in GR which uses a fixed SR way to compute $f(\ )$, say α_f. Given ϕ_i, $\bar{\pi}$ assumes that $\phi_i(\ ) = f(\ )$ and that $t\phi_i(x) < t\alpha_f(x)$ e.f.s. Therefore, using a bound $f_n^{(p)}$ such that $t\alpha_f(x) < f_n^{(p)}(x)$ for all x, it produces the image program $\bar{\phi}_i$ and the simulation program $\alpha_{p_1(i,n,p)}$ according to the method of Theorem 5.1. Now if $\phi_i(\ ) = f(\ )$, and ϕ_i is reasonably fast, i.e. $t\phi_i(x) < f_n^{(p)}(x)$, then $r(x, t\alpha_{\pi(p_1(i,n,p))}(x)) \leq t\alpha_{p_1(i,n,p)}(x)$

e.f.s. To handle the case when $\phi_i(\ ) = f(\ )$ but $\phi_i(\ )$ is slow (large), we modify $\alpha_{p_1(i,n,p)}$ so that if time $f_n^{(p)}$ is exceeded, then $\alpha_{\pi(f)}(x)$ is computed. Call the new image $\alpha_{p_1'(i,n,p)}$. Now the program α_π is an r speed-up of any $\phi_i(\ ) = f(\ )$.

The same arguments will work for pure Loop, but now the "sufficiently large r" must be increased to compensate for the simulation of $x \dot{-} 1$.

$$\begin{tikzcd} \overset{\text{SR}}{0} \arrow[d, "\pi"'] & \overset{\text{GR}}{0} \arrow[l, "k"'] \arrow[d, dashed, "\bar{\pi}"] \\ 0 \arrow[r, "id"'] & 0 \end{tikzcd}$$

diagram commutes

The situation is summarized by the above diagram. The map k is the translation into SR given by Theorem 5.1. The downward maps, π, $\bar{\pi}$ represent the hypothetical "acceleration functions" (i.e. functions which produce speed-up).

5.4. Relevance to the "Go To" Controversy. We have studied certain facets of program structure found in high-level languages like Fortran, Algol, and PL/I. The use of the more sophisticated languages like Algol and PL/I has caused a certain controversy over the need for "go to's." The motivation for the controversial discussions is the fact that the use of "go to's" in Algol destroys the logical simplicity of programs and makes description of the computation difficult. Therefore, it is desirable to minimize their use. The question arises of whether they can be eliminated entirely without unbearable sacrifice [15].

The answer to the simple question of whether they can be eliminated at all is a trivial yes. Using the Kleene normal form we can express every number theoretic computation ϕ_i as $\phi_{k(i)}$ =

```
IN $I, X$
DO WHILE $S = 0$
$Y \leftarrow Y + 1$
$S \leftarrow T(I, X, Y)$
END
$Z \leftarrow U(Y)$
OUT $Z$
```

where $T(\ )$ is the computation predicate. The T-predicate can be computed in Loop, and we know that Loop does not need any conditionals. But this answer is uninteresting.

We can offer more enlightening comments on the situation. Consider the following "go-to-free" languages

General recursive:

(1) $[A_8, P_4, (\diamondsuit), \overset{\text{DO}}{\text{END}}, \overset{\text{DOWH}}{\text{END}}, \quad E, R]$ Algol-gf

(2) $[+1, -1, \neq 0, (\diamondsuit), \overset{\text{DOWH}}{\text{END}}]$ G$_3$-gf

Subrecursive:

(3) $[A_8, P_4, (\diamondsuit), \overset{\text{DO}}{\text{END}}, E]$ SR-gf

(4) $[+1,\ -1,\ \neq 0,\ (\overset{\text{DO}}{\Diamond}),\ \text{END}]$ Loop$_3$-gf

(5) $[+1,\ 0,\ (\ ),\ \neq 0,\ (\overset{\text{DO}}{\Diamond}),\ \text{END}]$ Loop-gf

To study the effects of the go to on efficiency, one might investigate the relative efficiency of these languages and their counterparts with go to's.

For example, we can immediately see that using the Kleene normal form, a program in G_3-gf can mimic a G_3 program within a fixed cost, C, in size and h ( ) in efficiency.

An interesting question is whether a more reasonable simulation works in this context to give a small efficiency factor $h(\ )$, like $h(x) = 2x$ or $\log(x) + x$.

To prevent simple answers to the simulation problem, such as we have given here in Section 5 (by using only the Loop part of the language and appealing to Theorem 5.1), one could investigate the efficiency of the program which *uses a minimal number of DO-loops*. This will force use of the nested conditional as much as possible.

For the subrecursive languages, the comparison between go to and go-to-free versions is decisive. Using the methods of Section 5 we can simulate forward go to's with nested conditionals without decreasing efficiency by more than a constant factor. The cost in terms of size between SR and SR-gf is at most $(l^2 + 5 \cdot l)/2$ where $l = |\alpha_i|$. The method of translation, in brief informal terms, is as follows. Move all go to's from inside loops by using a statement like $*$ in Theorem 5.1 to get control outside the loop. Then put the conditional after the END of the loop. Now given the conditional, "if $v \neq 0$ then $+C$", followed by statement s, where $+C$ refers to statement $\bar{s}$, replace the conditional by, "if $v \neq 0$ then $[\bar{s},\text{end}]$ else $[s,\text{end}]$", where $[t,\text{end}]$ refers to the segment from statement t to the end of the program. Such a translation does not increase the number of loops, only the length of the program and the number of conditionals (their number at most doubles).

This result is not at all practical, but it allows us to roughly quantify the value of go to statements in a subrecursive language.

6. *Conclusion*

Although many of the results here are interesting or difficult only because of the special nature of the languages involved, e.g. Theorem 5.1 for Loop, the general principles (relative efficiency, simulation, parallelism, clock mechanisms, etc.) apply to a wide class of computing systems (machines and/or languages).

A more abstract theory of subrecursive computing systems would not only clarify the extent of this generality, it would render the whole approach to subrecursive phenomena more palatable. It would also help isolate the critical features of the proofs and constructions.

For these and numerous other reasons, we would propose an abstract treatment of certain aspects of concrete subrecursive complexity theory. This is a difficult matter to handle, and we hope eventually to contribute to an adequate treatment. For the moment we speculate on one approach to the area and suggest some problems. These comments should also shed a more general light on Sections 2–5.

Let $\mathcal{L}$ be the subrecursive class of functions which we intend to characterize, say $\mathcal{L}$ is an r.e. subset of $\mathcal{R}$. We might begin with an indexing

$$\alpha\colon N \to \mathcal{L} \subset \mathcal{R} \quad \text{obtained from} \quad \phi\colon N \to \mathcal{PR},$$

obtained by selecting a subset of $\{\phi_i(\ )\}$ by a function of τ. Thus $\alpha_n = \alpha(n) = \phi(\tau(n)) = \phi_{\tau(n)}$, so $\alpha\colon N \to \mathcal{L}$. Let $m\alpha_i(x) = m\phi_{\tau(i)}(x)$ for all x. This defines a measure $A = \{m\alpha_i(\ )\}$.

There are certain obvious restrictions that must be placed on $\{m\alpha_i\}$ for it to qualify as a subrecursive measure. Among the desirable attributes would be

(a) $\exists\, m(\ ) \in \mathcal{L}$ such that $m\alpha_i = \alpha_{m(i)}$. Thus the measure is *syntactically definable* within the class $\mathcal{L}$.

(b) $\lambda i, x, y\, M(\tau(i), x, y) \in \mathcal{L}$. The measure function restricted to α_i belongs to $\mathcal{L}$.

(c) $\exists\, s(\ ) \in \mathcal{L}$ such that $\alpha_i(x) \le s\,(m\alpha_i(x))$ for all x. Thus the measure has a *speed limit* in $\mathcal{L}$.

(d) $\exists\, h(\ ) \in \mathcal{L}$ such that $m\alpha_{m(i)}(x) \le h(m\alpha_i(x), x)$ for all x. Thus the complexity functions are $h(\ )$ *honest* for some $h(\ )$ in $\mathcal{L}$.

These properties are analogues of the Blum axioms. Blum's Axiom 1 forces the measure to have arbitrarily large complexity functions, e.g. it prevents $\Phi_i(x) = 0$ for all x from being a measure. This is accomplished here by (c).

Among the consequences desired for the subrecursive measures are those theorems of the general theory which hold in the class $\mathcal{L}$. For example, when $\mathcal{L} = \mathcal{R}^1$ we want

(1) speed-up theorem,

(2) compression theorem (upward diagonalization theorem or jump theorem when stated in terms of classes),

(3) gap theorem,

(4) honesty theorem,

(5) union theorem.

Many important abstract properties can be established using the recursive relationship [1, Th. 2] in the following manner. Prove the result for T a specific measure like time, then show that the result is measure independent, and finally use the recursive relationship to carry over the result to any other measure, i.e. speed-up theorem [2 and 13].

Using the same technique with abstract subrecursive measures requires an $\mathcal{L}$-recursive relationship. $\mathcal{L}$-recursive relationships are defined as follows: If $A = \{m\alpha_i\}$ and $B = \{m\beta_i\}$ are $\mathcal{L}$-measures, then there is an r in $\mathcal{L}$ such that

(i) $m\alpha_i(x) \le r(m\beta_i(x), x)$ e.f.s.,

(ii) $m\beta_i(x) \le r(m\alpha_i(x), x)$ e.f.s.

This attribute does not follow from (a)–(c) because it involves two formalisms while the others are all "internal" or "coordinate-free" properties. In the Blum case, recursive relationship holds because the indexings are acceptable. The satisfying fact is that acceptable indexings are given an intrinsic or coordinate-free definition. A satisfactory definition of $\mathcal{L}$-acceptable indexing would presumably lead to the $\mathcal{L}$-relationship among measures.

Some interesting observations can already be made about (a), (b), and (c) as possible axioms. First, they are independent but insufficient to guarantee either the compression theorem or a recursive relationship between measures. Even (a), (b), and (c) plus compression do not guarantee a recursive relationship. However, if $\mathcal{L}$ is closed under $\mu\le$ and iteration, then (a) implies the gap theorem. In [16] Lewis shows that (a), (b), and (c) allow non-r.e. complexity classes.

Results like Theorem 5.1 would follow from the existence of the function $T_n(i, x, y)$ of Section 4 in $\mathcal{L}$ and from a parallel cost axiom of the form

(d) $\exists p \in \mathcal{L}\ \forall i\ \forall j,\ m\alpha_i(\alpha_j) \leq p(m\alpha_i(\ ), m\alpha_j(\ ))$.

The function $p(\ )$ represents the cost of parallelism in the formalism. For general recursive formalisms and measures such as $\{t\beta_i(\ )\}$, $p(\ )$ always exists because of a recursive relationship with models like multitape Turing machines. However, there are subrecursive formalisms without that property (at first sight Loop might appear to be one).

Appendix

Here we prove the bounding lemma for Loop. The bounding lemma for SR follows the same plan, so it is omitted.

Bounding Lemma for Loop. *If $\beta_i(\ ) : N^n \to N^p$, $\beta_i \in L_n$, then $\beta_i(x_1, \cdots, x_n)_j \leq f_n^{(|\beta_i|)}(\max\{x_1, \cdots, x_n\} + 2)$.*

Proof. We prove the theorem by double induction, on depth, n, and within depth on length, $|\beta_i|$.

(1) Assume that depth is 0, $\beta_i \in L_0$.

(a) Let $|\beta_i| = 1$. Then β_i is either $X \leftarrow X + 1$, $Y \leftarrow X$, or $X \leftarrow 0$. Clearly the maximum value is $\beta_i(x) = x + 1$ and $\beta_i(x) \leq f_0(x) = x + 1$ for all x.

(b) Assume the result for $|\beta_i| = n$, to prove it for $|\beta_i| = n + 1$. Since $\beta_1 \in L_0$, it has the form $s_1 ; s_2 ; \cdots ; s_n ; s_{n+1}$, where each s_i is an assignment. By induction, the maximum value in any register in $s_1 ; \cdots ; s_n$ is $f_0^{(n)}(\max\{x_1, \cdots, x_n\} + 2)$. If s_{n+1} does not increase any output variables, say y_i, then clearly the result holds. If not, then s_{n+1} is $Y_i \leftarrow Y_i + 1$. So Y_i is bounded by $f_0(f_0^{(n)}(\max\{x_1, \cdots, x_n\} + 2))$. Q.E.D. (step (1))

(2) Assume the result for $\beta_i \in L_n$. To show it for $\beta_i \in L_{n+1}$:

(a) Suppose $\beta_i \in L_{n+1}$ and $|\beta_i| = 2(n + 1) + 1$ (this is the minimum possible length for depth $n + 1$). Then β_i has the form

```
DO $X_1$
$B$
END
```

where B is a program in L_n. Hence by assumption,

$$B \leq f_n^{(|B|)}(\max\{x_1, \cdots, x_n\} + 2).$$

The program B computes a vector valued function, say

$$\langle B(Z_1, \cdots, Z_k)_1, B(Z_1, \cdots, Z_k)_2, \cdots, B(Z_1, \cdots, Z_k)_q\rangle = \langle Y_1, \cdots, Y_q\rangle$$

Only the outputs among $Y_1, \cdots, Y_q$ which are also inputs (i.e. occur among $Z_1, \cdots, Z_k$, call them *feedback variables*) can effect the output of β_i as a function of X_1. If there are no such outputs, then

$$\boxed{\begin{array}{l}\text{DO } X\\ B\\ \text{END}\end{array}}$$

is equivalent to B and the result follows immediately.

Thus assume some Y_i are also Z_j's. By including all input variables among the outputs, it is easy to describe the type of iteration specified by DO X_i ; B; END. To that end, form a new vector valued function $\bar{B}$ having the same inputs as B but including all Z_i and Y_j among its outputs. Say $\bar{B}$ is

$$\langle \bar{B}(Z_1, \cdots, Z_k)_1, \cdots, \bar{B}(Z_1, \cdots, Z_k)_p \rangle = \langle \bar{Y}_1, \cdots, \bar{Y}_p \rangle$$

where $p \geq k$, $p \geq q$. For simplicity, assume that $Z_i = \bar{Y}_i$ for $i = 1, \cdots, k$. Then B is obtained from $\bar{B}$ by selecting a subset of outputs and permuting them. Clearly DO X_i ; $\bar{B}$; END is no smaller than DO X_i ; B; END in the sense that for every Y_i there is a $\bar{Y}_j$ such that $Y_i = \bar{Y}_j$.

We now present a succinct notation for DO X_i ; $\bar{B}$; END. This is notation for vector iteration of a simple type.

(i) $\bar{B}^{(0)}(Z_1, \cdots, Z_p)_i = Z_i, \qquad i = 1, \cdots, p.$

(ii)

$$\langle \bar{B}^{(n+1)}(Z_1, \cdots, Z_p) \rangle = \langle \bar{B}(\bar{B}^{(n)}(Z_1, \cdots, Z_p)_1, \cdots, \bar{B}^{(n)}(Z_1, \cdots, Z_p)_p)_1, \cdots, \bar{B}(\bar{B}^{(n)}(Z_1, \cdots, Z_p)_1, \cdots, \bar{B}^{(n)}(Z_1, \cdots, Z_p)_p) \rangle.$$

Now we know by definition of the iterative that

DO X_1
$\bar{B}$
END

is $\langle \bar{B}^{(X_1)}(Z_1, \cdots, Z_p) \rangle$.

By the induction hypothesis,

$$\bar{B}(Z_1, \cdots, Z_p) \leq f_n^{(|\bar{B}|)}(\max\{Z_1, \cdots, Z_p\} + 2) \overset{\text{def}}{=} h(y),$$

where $h(y) = f_n^{(|\bar{B}|)}(y)$ and $y = \max\{Z_1, \cdots, Z_p\} + 2$. So

$$\bar{B}^{(X_1)}(Z_1, \cdots, Z_p) \leq h^{(X_1)}(y).$$

But

$$h^{(X_1)}(y) = f_n^{(X_1 \cdot |\bar{B}|)}(y)$$

and notice

$$f_n^{(X_1 \cdot |\bar{B}|)}(y) \leq f_{n+1}(y) \quad \text{if} \quad y \geq X_1 \cdot |\bar{B}|.$$

Also

$$f_n^{(X_1 \cdot |\bar{B}|)}(y) \leq f_{n+1}^{(|\bar{B}|)}(y) \quad \text{if} \quad y \geq X_1,$$

but indeed $y = \max\{x_1, \cdots, x_n\} + 2 \geq x_i$ for any x_i and the loop variable X_1 has the value x_i for some $i = 1, \cdots, p$. Hence

$$\bar{B}^{(X_1)}(Z_1, \cdots, Z_p) \leq f_{n+1}^{(|\bar{B}|)}(\max\{x_1, \cdots, x_n\} + 2)$$

because $y \leq \max\{x_1, \cdots, x_n\} + 2$. So the result for $|\beta_i| = 2(n + 1) + 1$ holds with room to spare ($|\bar{B}|$ is much larger than necessary).

(b) Assume the result for $|\beta_i| \leq m$. To show it for $|\beta_i| = m + 1$: Either β_i has the form A; B for $|A| > 0$, $|B| > 0$, $A, B \in \mathrm{L}_{n+1}$, or it has the form

DO X_i ; B; END for $B \in L_n$. The latter case proceeds exactly as case (a). So we assume β_i has the form A; B. Then by the induction hypothesis on length we know

$$A \leq f_{n+1}^{(|A|)}(y_1), \qquad B \leq f_{n+1}^{(|B|)}(y_2),$$

where y_1, y_2 are the maxima of the inputs plus 2; so $y_2 \leq \max\{x_1, \cdots, x_n\} + 2$, $i = 1, 2$. Notice, A; B is bounded by

$$f_{n+1}^{(|B|)}(f_{n+1}^{(|A|)}(y_1)) \leq f_{n+1}^{(|A|+|B|)}(y_1)$$

So the result holds. Q.E.D.

REFERENCES

(*Note.* References [5, 10, 18] are not cited in the text.)

1. Blum, M. A machine-independent theory of the complexity of recursive functions. *J. ACM 14*, 2 (April 1967), 322–336.
2. Blum, M. On effective procedures for speeding up algorithms. Conf. Record of ACM Symp. on Theory of Computing, Marina del Rey, Calif., 1969, pp. 43–53.
3. Blum, M. On the size of machines. *Inform. Contr. 11* (1967), 257–265.
4. Borodin, A. Complexity classes of recursive functions and the existence of complexity gaps. Conf. Record of ACM Symp. on Theory of Computing, Marina del Rey, Calif., 1969, pp. 67–78.
5. Cleave, John P. A hierarchy of primitive recursive functions. *Z. Math. Logik Grundlagen Math. 9* (1963), 331–345.
6. Cobham, A. The intrinsic computational difficulty of functions. Proc. 1964 Internat. Congress for Logic, Methodology, and the Philosophy of Science. North-Holland, Amsterdam, 1965, pp. 24–30.
7. Constable, R. L. Upward and downward diagonalization over axiomatic complexity classes. Comput. Sci. Rep. 69-32, Cornell U., Ithaca, N. Y., 1969.
8. Constable, R. L. On the size of programs in subrecursive formalisms. Conf. Record of Second Annual ACM Symp. on Theory of Computing, Northampton, Mass., 1970, pp. 1–9; part of this also appears as: Subrecursive programming languages II, On program size, *J. Comput. Syst. Sci. 5* (1971), 315–334.
9. Elgot, C. C., and Robinson, A. Random-access stored-program machines, an approach to programming languages. *J. ACM 11*, 4 (Oct. 1964), 365–399.
10. Engler, Erwin. *Formal Languages; Automata and Structures.* Markham Co., Chicago, 1968.
11. Grzegorczyk, A. Some classes of recursive functions. Rozprawy Mathematcyzne, No. 4, Instytut Matematyczny Polskiej Akademie Nauk, Warsaw, Poland, 1953, pp. 1–45.
12. Hartmanis, J., and Stearns, R. E. On the computational complexity of algorithms. *Trans. AMS 117*, 5 (1965), 285–306.
13. Hartmanis, J., and Hopcroft, J. E. An overview of the theory of computational complexity. *J. ACM 18*, 3 (July 1971), 444–475.
14. Kleene, S. C. *Introduction to Metamathematics.* Van Nostrand, Princeton, N. J., 1952.
15. Knuth, D., and Floyd, R. Notes on avoiding "go to" statements. *Inform. Proc. Letters 1* (1971), 23–31.
16. Lewis, F. D. The enumerability and invariance of complexity classes. *J. Comput. Syst. Sci. 5* (1971), 286–303.
17. McCreight, E. M., and Meyer, A. R. Classes of computable functions defined by bounds on computation: Preliminary report. Conf. Record of ACM Symp. on Theory of Computing, Marina del Rey, Calif., 1969, pp. 79–88.
18. Meyer, A. R., and Fischer, P. C. On computational speed-up. IEEE Conf. Record, 9th Annual SWAT, 1968, pp. 351–355.
19. Meyer, A. R., and Ritchie, D. M. The complexity of loop programs. Proc. 22nd National Conf. ACM, ACM Pub. P-67, 1967, Thompson Book Co., Washington, D.C., pp. 465–469.

20. Meyer, A. R., and Ritchie, D. M. A classification of functions by computational complexity. Proc. Hawaii Intern. Conf. on System Sciences, U. of Hawaii Press, 1968, pp. 17–19.
21. Minsky, M. *Computation, Finite and Infinite*. Prentice-Hall, Englewood Cliffs, N. J., 1967.
22. Ritchie, R. W. Classes of predictably computable functions. *Trans. AMS 106* (1963), 139–173.
23. Robinson, R. M. Primitive recursive functions. *Bull. AMS 53* (1947), 915–942.
24. Rogers, H., Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
25. Scott, Dana. Some definitional suggestions for automata theory. *J. Comput. Syst. Sci. 1* (1967), 187–212.
26. Shepherdson, J. C., and Sturgis, H. E. Computability of recursive functions. *J. ACM 10*, 2 (April 1963), 217–255.
27. Wegner, P. *Programming Languages, Information Structures and Machine Organization*. McGraw-Hill, New York, 1968.

RECEIVED MAY 1970; REVISED AUGUST 1971