THE UNIVERSITY OF MICHIGAN
COLLEGE OF LITERATURE, SCIENCE, AND THE ARTS
Computer and Communication Sciences Department

Technical Report

A Simple Programming Approach To Basic Computability Theory

Dennis P. Geller
Bernard P. Zeigler

with assistance from:

Department of Health, Education, and Welfare
National Institutes of Health
Grant No. GM-12236
Bethesda, Maryland

and

National Science Foundation
Grant No. GJ-519
Washington, D.C.

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR
March 1971

UMR
1555

1. Introduction

While the classical Turing Machine concepts are quite valuable, for computer science students the equivalence between computable functions and partial recursive functions can be demonstrated in a clearer manner by dealing with a simple compiler-like language capable of defining the partial recursive functions. Such an approach is taken, by Nelson [4], Minsky [3] and Engeler [1].

In presenting this material in a graduate level introductory course we found ourselves going well beyond the contributions of these authors. We decided to implement a version of such a language on an IBM 1800 computer so that students could write, debug and run programs in the usual manner. This necessitated a more rigorous treatment of the underlying constructs than is presented in the above texts. In the course of this work, we realized that the ideas developed could, quite naturally, be used to write a universal program in the language, i.e., a program which given a description of any program and its input data, simulates the computation of the latter program. We feel this approach to basic computability theory is sufficiently novel and useful to warrant exposure to other instructors in the area.

2. The Language

The language we present is essentially that given in [4, p.82], the major differences being the way we handle the macros (called subroutines in [4]) and that we permit "scratch" registers.[1]

A *machine* is specified by a set R of *registers*, a subset $I \subseteq R$ of *input registers*, a distinguished register p not in R called the *print register*, and a *program*. A register can store any non-negative integer.

[1] It is interesting to note that without these modifications the purported proofs given by Nelson can not be carried out.

The program is a list $1.I_{n_1};2.I_{n_2};\ldots;N.I_{n_N};$ of statement numbers (i) and associated instructions (I_{n_i}). Each instruction is one of the following:

SET(A,B)	load the contents of register A into register B.
ADD(A)	add 1 to register A
SUB(A)	subtract 1 from register A unless the contents of A is zero, in which case this is treated as a STOP() instruction.
STOP()	halt and print the contents of the print register
P(A)	load the contents of A into the print register
TRANS(A,n,m)	if the contents of register A is zero branch to instruction n; otherwise branch to instruction m.

It is assumed that the machine executes the program starting with instruction 1 and proceeds sequentially, except when the order of execution is modified by a TRANS instruction. Such a machine M computes a partial function Φ_M whose arguments are the initial contents of the input registers; the registers in $R-I$, and p, are assumed to initially contain zero. It is of course convenient to introduce certain conventions, such as interpreting a transfer out of the bounds of the program as a STOP(), but these details are relatively straightforward. In what follows, if X is a register then c(X) denotes the contents of X.

The formal language lends itself readily to the introduction of macros. Suppose F is a program in the language which operates on a pair of input registers. We augment the basic instruction list with a new *macro* instruction

F(A,B,C) load register C with $\Phi_F(c(A),c(B))$.

Of course to make sense of such an instruction we need a macro expander. This is a program which accepts a program composed of macros and basic

instructions and produces an equivalent program (i.e., computing the same partial function) consisting only of basic instructions. There are some details to take care of. For example, to be acceptable as a macro definition F must be rewritten so that the computation it directs is stopped only at the physically last instruction. In this way when programs with macros are expanded as basic instruction lists, the execution will proceed smoothly from one macro-derived list to the next. Such details are relatively straightforward and we shall not dwell on them.

3. The Universal Machine

To give students facility with using the formal language, we wrote a simple *interpreter* in FORTRAN. For the purposes of the interpreter, the instruction types and registers were numbered. As an example, the program in Figure 1a is reproduced in the interpreter language in Figure 1b [the program sets p = A-B if A>B and 0 if B≥A without destroying A or B]. The assignment of numbers to registers and instructions in the example are as follows:

SET	1	A	8
ADD	2	B	9
SUB	3		
P	4	D	10
TRANS	5	E	11
STOP	6		

Thus each instruction in the program was expressed as a sequence of integers.

What is important about this otherwise trivial exercise is that it provides a convenient Gödel encoding of the instructions for the purposes of a universal machine. In the interpreter language each instruction consists of an instruction number plus at most 4 additional positive integers, the first being the instruction type and the others (if any) referring to registers and instruction numbers; for convenience, let us say that each instruction is represented by an instruction number plus exactly 4 integers, some of which may be zero.

To code a program for the universal machine U, we give the machine U four input registers, R_1, R_2, R_3, R_4 which will hold the program instructions. If the program instructions are $N_i, M_{i1}, M_{i2}, M_{i3}, M_{i4}$ and there are n instructions then the content of register R_j is $\prod_{i=1}^{n} P_i^{M_{ij}}$ where P_k is the k-th prime. For example in Figure 1(b), $c(R_1) = 2^1 3^1 5^5 7^5 11^3 13^3 17^5 19^4 23^6$.

Two other registers are necessary. An input register, R_r, is used to code the contents of the registers $S_1, \ldots, S_t$ of the simulated machine. At any time during the simulation, $c(R_r) = \prod_{i=1}^{t} P_i^{c(S_i)}$. Finally, we include a register R_c to hold the number of the next instruction to be executed. Of course, the machine U also has a number of other scratch registers for its internal computations.

With these conventions the program for the universal machine U can be described clearly and succinctly in terms of macros. The program is given in Figure 2, and the definitions for the macros used are given in Figure 3. Understanding the operation of the universal machine is straightforward once the basic operations carried out by the macros are understood. These are essentially decoding and encoding operations: for example, the macro $MLOG(P_i, n)$ is used to extract the greatest power of P_i dividing n

(which would be the contents of the i'th register if $n = c(R_T)$). The length of the expanded program is 1750 statements. Note: Since most students in our department are familiar with SNOBOL4, the final macroexpander and interpreter were written in this language, which proved to be a rather expensive choice. However, although the program could have been written more efficiently in 360 Assembler, we still feel that SNOBOL4 was a good choice for pedagogic reasons. Similarly, even in SNOBOL4 the macro expansion process could have been circumvented by creating programmer-defined functions [2] from the macros so that they would be treated as subroutines (but then we would not have had a true emulation of the formal theory.)

4. Conclusions

There are some definite advantages to the approach to recursive function theory and universal machines which we have outlined. First, the fundamental concepts of computation are related directly to a computer language very similar to those with which most students are very familiar. This is done before, rather than after, the presentation of Turing machines. In this way, an already existing source of intuition is used as a base to develop an understanding of the automata-theoretic abstractions which usually are grasped only with much effort.

Second, one can usefully exploit parallels in recursive function theory and familiar computer concepts. Composition of functions, primitive recursion, and minimalization, for example, are realized by simple programs employing macros for already defined functions. The distinctions between syntax and semantics arise naturally, since the concepts of legal programs and the associated interpretive machines must be formalized in order to carry out the proofs leading up to the presentation of Church's thesis.

Third, the format permits students to explore, through homework problems, the concepts of partial recursive functions more deeply. Whereas more than one Turing machine program as a homework assignment is a chore, a more "realistic" language enables the student to easily write programs which perform bounded minimization, course of values recursion, and the like. The students' sense of dealing with "real" computation is further enhanced by the process of punching his program cards, obtaining a computer output listing, etc. Also, as a plus for the grader, these programs can be tested by actual computer interpretation.

Finally, the operation of the universal machine as a model of a general purpose computer becomes much clearer when stripped of the machinery necessary to manipulate a Turing machine.

REFERENCES

1. Engeler, E. Formal Languages, Markham, Chicago, 1968.

2. Griswold, R.E.; Poage, J.F.; and Polonsky, I.P. The SNOBOL4 Programming Language, Prentice-Hall, Englewood Cliffs, N.J., 1968.

3. Minsky, M. Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, N.J., 1967.

4. Nelson, R.J. Introduction to Automata, Wiley, New York, 1968.

	(a)		(b)
1.	SET(A,D);	1.	1,8,10
2.	SET(B,E);	2.	1,9,11
3.	TRANS(E,8,4);	3.	5,2,8,4
4.	TRANS(D,9,5);	4.	5,10,9,5
5.	SUB(D);	5.	3,10
6.	SUB(E);	6.	3,11
7.	TRANS(A,3,3);	7.	5,8,3,3
8.	P(D);	8.	4,10
9.	STOP();	9.	6

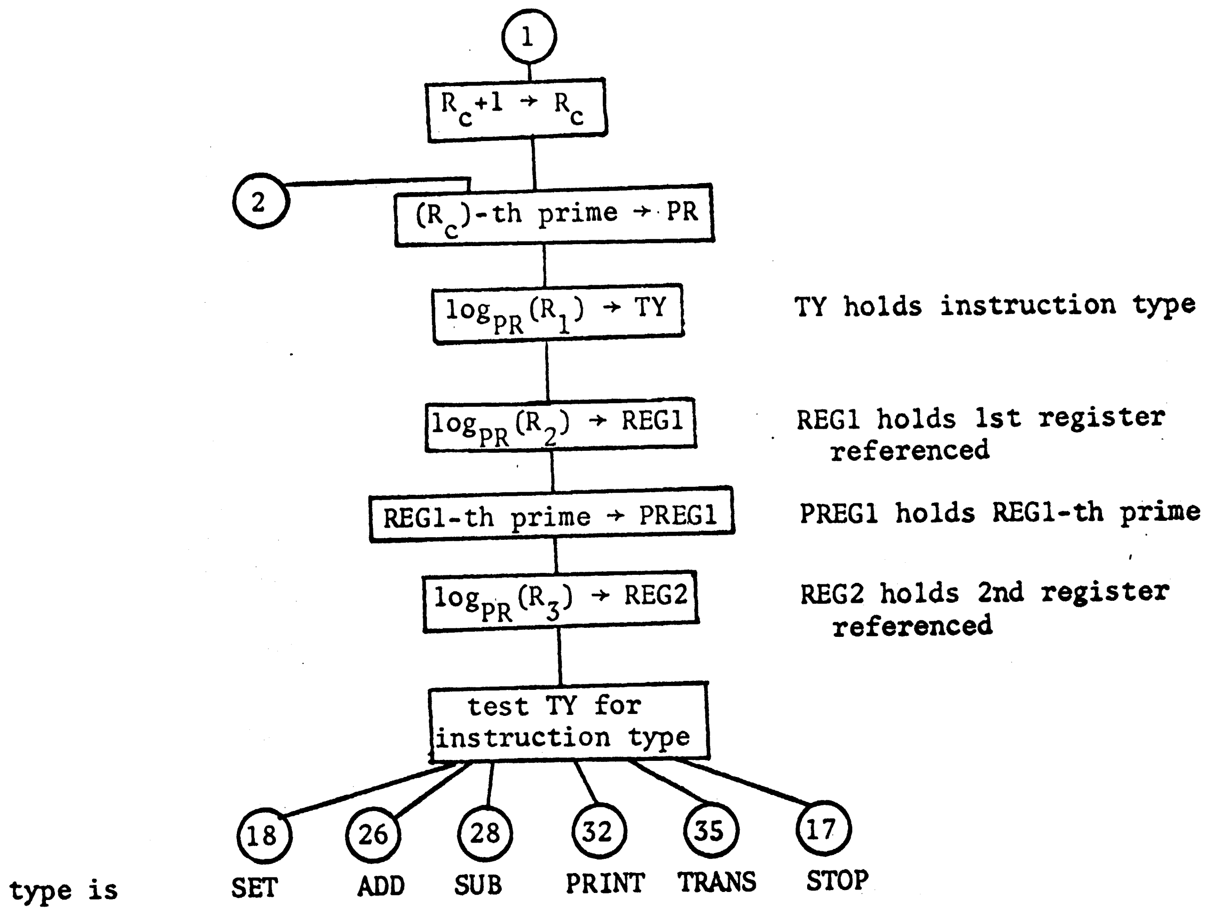
Figure 1

	Instruction	Comment
1.	ADD(R_c);	increment R_c
2.	MPRIME(R_c,PR);	
3.	MLOG(PR,R_1,TY);	gets instruction type
4.	MLOG(PR,R_2,REG1);	gets number of first register referenced
5.	MPRIME(REG1,PREG1);	prime whose exponent is c(REG1) in R_r
6.	MLOG(PR,R_3,REG2);	gets second element of instruction
7.	SUB(TY);	in 7-16, evaluating instruction type.
8.	TRANS(TY,18,9);	type SET if c(TY)=0
9.	SUB(TY);	
10.	TRANS(TY,26,11);	type ADD if c(TY)=0
11.	SUB(TY);	
12.	TRANS(TY,28,13);	type SUB if c(TY)=0
13.	SUB(TY);	
14.	TRANS(TY,32,15);	type P if c(TY)=0
15.	SUB(TY);	
16.	TRANS(TY,35,17);	type TRANS if c(TY)=0
17.	STOP();	c(TY) was initally 6
18.	MPRIME(REG2,PREG2);	prime whose exponent is c(REG2) in R_r
19.	MLOG(PREG2,R_r,EX);	gets c(REG2)
20.	MEXP(PREG2,EX,EXP);	
21.	MDIV(R_r,EXP,R_r);	sets c(REG2)=0 in R_r
22.	MLOG(PREG1,R_r,EX);	finds c(REG1)
23.	MEXP(PREG2,EX,EXP);	
24.	MMULT(R_r,EXP,R_r);	sets c(REG2)=c(REG1)
25.	TRANS(R_1,1,1);	unconditional branch
26.	MMULT(R_r,PREG1,R_r);	adds 1 to c(REG1)
27.	TRANS(R_1,1,1);	
28.	MLOG(PREG1,R_r,EX);	finds c(REG1)
29.	TRANS(EX,17,30);	if c(REG1)=0, stop
30.	MDIV(R_r,PREG1,R_r);	subtracts 1 from c(REG1)
31.	TRANS(R_1,1,1);	
32.	MLOG(PREG1,R_r,EXP);	
33.	P(EXP);	sets p=c(REG1)
34.	TRANS(R_1,1,1);	
35.	SET(REG2,INS1);	c(REG2) is really an instruction number
36.	MLOG(PR,R_4,INS2);	gets second instruction number
37.	MLOG(PREG1,R_r,EXP);	finds c(REG1)
38.	TRANS(EXP,39,41);	
39.	SET(INS1,R_c);	if c(REG1)=0
40.	TRANS(R_1,2,2);	
41.	SET(INS2,R_c);	if c(REG1)>0
42.	TRANS(R_1,2,2);	

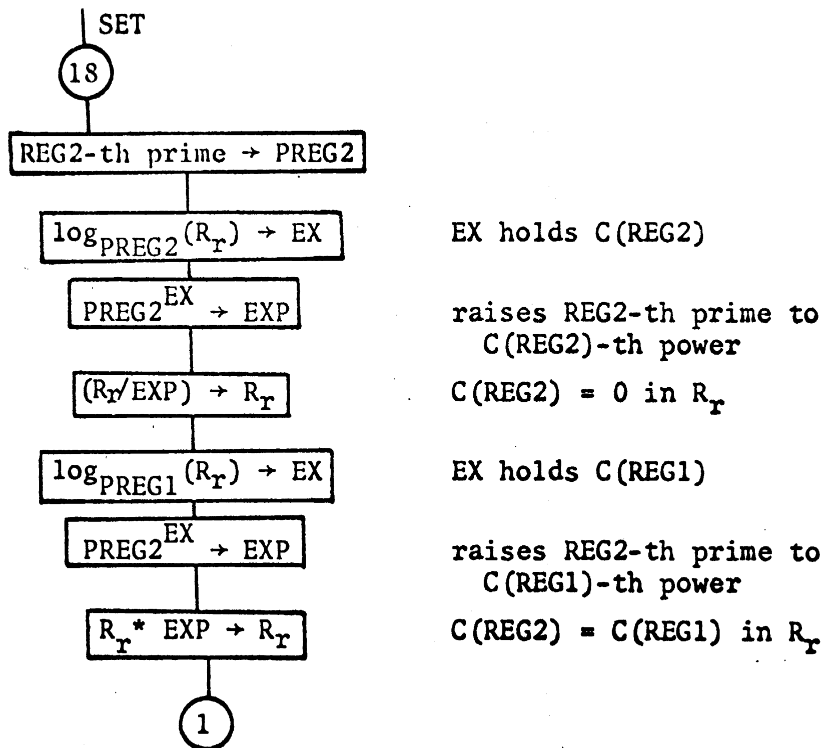
Program for a universal machine
Figure 2

NAME	FUNCTIONAL NAME	DESCRIPTION
MZERO(A)		set c(A) = 0
MMULT(A,B,C)	$A \cdot B$	c(C) = c(A)c(B)
MSGB(A,B)	$\overline{sg}(A)$	c(B) = 1 if c(A) = 0, 0 otherwise
MSG(A,B)	sg(A)	c(B) = 0 if c(A) = 0, 1 otherwise
MSUB(A,B,C)	$A \dot{-} B$	c(C) = A-B if A≥B, 0 otherwise
MLT(A,B,C)		c(C) = $sg(B \dot{-} A)$; 1 if c(A)<c(B), 0 otherwise
MLE(A,B,C)		c(C) = 1 if c(A)≤(B), 0 otherwise
MDIV(A,B,C)	$[\frac{A}{B}]$	c(C) = $[\frac{c(A)}{c(B)}]$; the greatest integer less than $\frac{c(A)}{c(B)}$
MRES(A,B,C)	res(A,B)	c(C) = $A \dot{-} ([\frac{A}{B}] \cdot B)$
MADD(A,B,C)	A+B	c(C) = c(A) + c(B)
MNDIV(A,B)		c(B) = $\sum_{i=1}^{c(A)} \overline{sg}(res(A,i))$; the number of divisors of c(A)
MABS(A,B,C)	$\|A-B\|$	c(C) = $(A \dot{-} B) + (B \dot{-} A)$
MPI(A,B)	π_A	c(B) = the number of primes less than or equal to c(A) [π_A does not count 1 as a prime]
MEXP(A,B,C)	A^B	c(C) = $c(A)^{c(B)}$
MEQ(A,B,C)		c(C) = $\overline{sg}(\|A-B\|)$; if c(A)=c(B), 0 otherwise
MPRIME(A,B)	P_A	c(B) = the c(A)-th prime [starting with P_1=2]
MNTDIV(A,B,C)		c(C) = 0 if c(A) divides c(B), 1 otherwise
MLOG(A,B,C)	$\log_A(B)$	c(C) = the largest power of c(A) which divides c(B)

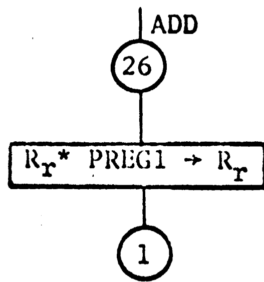
Macros used in the universal machine
Figure 3



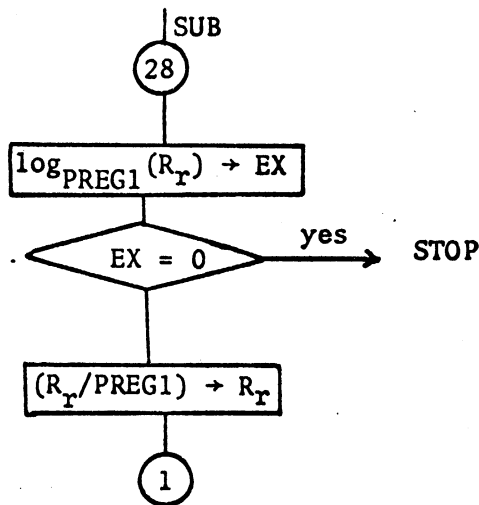
Flow chart for the universal machine
Figure 4



Flow chart for the universal machine
Figure 4 (cont.)



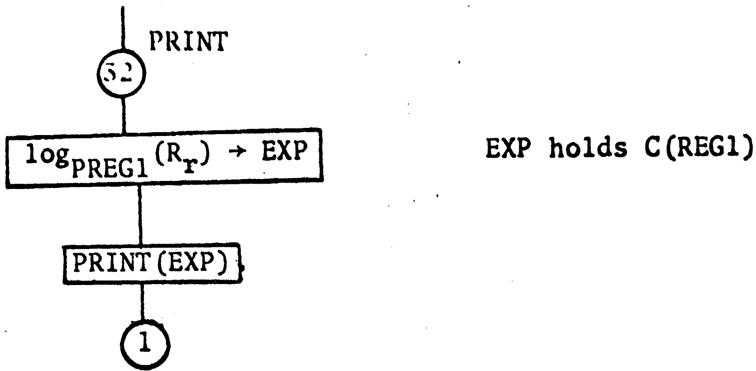
add 1 to C(REG1) in R_r

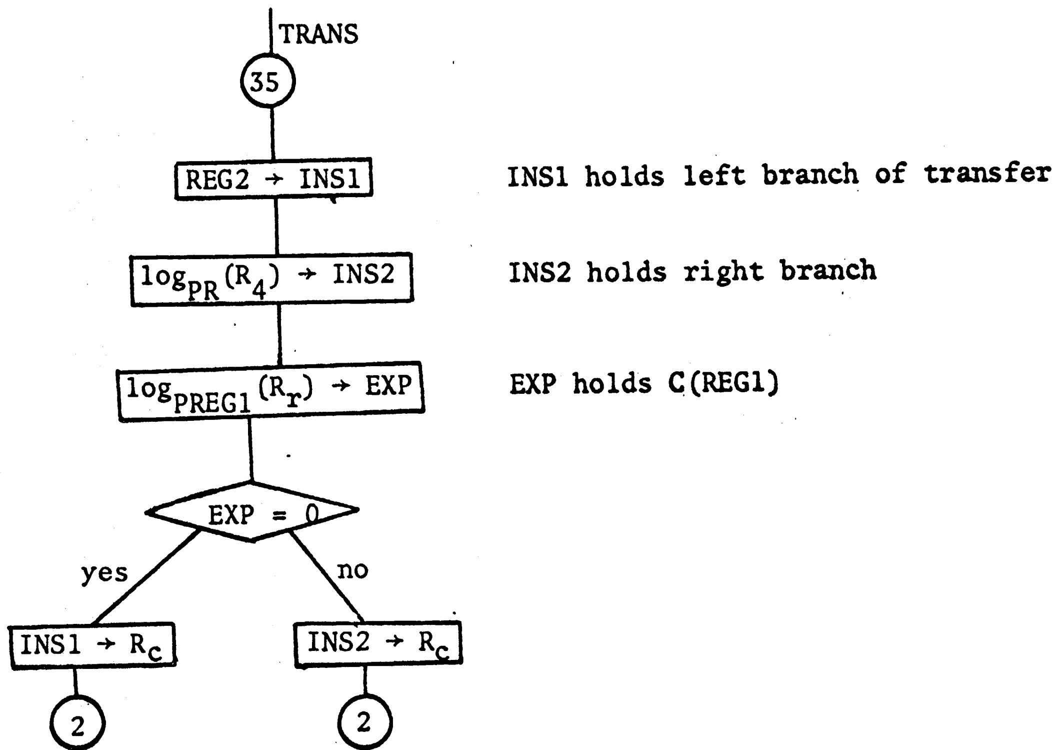


EX holds C(REG1)

subtracts 1 from C(REG1) in R_r

Flow chart for the universal machine
Figure 4 (cont.)





Flow chart for the universal machine
Figure 4 (concluded)

UNIVERSITY OF MICHIGAN
3 9015 02826 0829