Recursion Theorems and Self-Replication Via Text Register Machine Programs

Lawrence S. Moss*

Abstract

Register machine programs provide explicit proofs of the s_n^m-Theorem, Kleene's Second Recursion Theorem, and Smullyan's Double Recursion Theorem. Thus these programs provide a pedagogically useful approach. We develop this topic from scratch, hence without appeal to the existence of universal programs, pairing, quotation, or any form of coding device. None of the results are new from the point of view of computability theory apart from the particular formulations themselves. We introduce the notion of a *text register machine*; this is a register machine whose registers contain words from some alphabet and whose instructions are again words from the same alphabet. We work with a particular instruction set whose language of programs we call `1#`. Tools for writing and evaluating `1#` programs have been made freely available: see `www.indiana.edu/~iulg/trm`.

> It is generally recognized that the greatest advances in modern computers came through the notion that programs could be kept in the same memory with 'data,' and that programs could operate on other programs, or on themselves, as though they were data."
>
> Marvin Minsky [5]

1 Introduction

What is the simplest setting in which one can formalize the notion that "programs could operate on other programs, or on themselves"? This paper contains a proposal for such a formalization in the form of a programming

*Mathematics Department, Indiana University, Bloomington, IN 47405 USA. Email: `lsm@cs.indiana.edu`

language `1#`.[1] Using programs of `1#` we obtain explicit programs corresponding to the fixed point theorems from the theory of computation. In addition to re-proving classic results, our goal is to suggest a useful way of teaching them.

To get the simplest formulation of any complicated idea is not always easy, and to do so for programs operating on programs one must make some choices. The goal of this paper is to present a setting in which one could explain the concept to a person who can read mathematical notation but who doesn't necessarily know about programming or computability. Thus the model of computation in this discussion should be as intuitively simple as possible. For this, I have chosen a certain flavor of register machines. Second, the notion of a program should also be as simple as possible: both the syntax and semantics should be explainable in less than fifteen minutes. This rules out high-level programming languages. As they are standardly studied, register machines work on numbers, but their programs are not numbers. So we work with a variant notion, *word register machines*. Such machines process strings over the tiny alphabet $\mathcal{A} = \{\texttt{1}, \texttt{\#}\}$. The key additional feature of our machine is that its instructions and programs are words over this same set $\mathcal{A}$. We call such machines *text register machines*.

Word register machines are a Turing-complete computational formalism. Text register machines also illustrate versions of the main foundational theorems of recursion theory explicitly. So when a result says "there is a program such that ..." then it is often possible to easily exhibit such a program. The results I have in mind are the s_n^m-Theorem and the Second Recursion Theorem. Usually the s_n^m-Theorem is treated by appealing to the Church-Turing thesis (that is, saying that the construction is "obvious but tedious") or else done via the coding of sequences by numbers. Our development is more direct. It also leads to explicit *self-reproducing programs*.

We turn to the main development itself in Section 3. Before that, we have some discussion of how our proposal fits into the history of the subject.

2 Historical and conceptual points

The register machine formalism was introduced in Shepherdson and Sturgis' 1963 paper [6]. Their goal was to provide a formalism for which one could verify that all partial recursive functions are computable by some "finite,

[1] This paper leaves open the pronunciation of the name of the language. The symbol `#` has many names, including *check*, *octothorpe*, *pound sign*, *hash*, and *crosshatch*. It is not quite the sharp sign ♯. The names of languages like $A\#$ and $C\#$ use "sharp" anyways.

discrete, deterministic device supplied with unlimited storage." Their notion of a register machine comes in several variants. Broadly speaking, these variants include machines whose registers contain natural numbers, and also with machines whose registers contain words $w \in \mathcal{A}^*$ over some set $\mathcal{A} = \{a_1, \ldots, a_s\}$ of alphabet symbols. We are concerned in this paper with the latter variant. It has for the most part been forgotten in the literature. For example, Fitting [2] writes, "Register machines come from [Shepherdson and Sturgis 1963]. As originally presented, they manipulated numbers not words; the version presented here is a straightforward modification." And to be sure, I had not known of word register machines before I wrote up this paper. Getting back to the idea itself, it might be worthwhile to recall (in a somewhat updated terminology and notation) the formulation in [6].

In Sections 5 and 6 of [6] we find a formulation of *partial recursive functions on* $\mathcal{A}^*$ and then a definition of *unlimited register machines* over $\mathcal{A}$. These are register machines whose instructions are as follows:

$P^{(i)}(n)$: place a_i on (the right-hand) end of $\langle n \rangle$

$D(n)$: delete the (left-most) letter of $\langle n \rangle$, provided that $\langle n \rangle \neq \epsilon$.

$J^{(i)}(n)[E1]$: jump to line $E1$ if $\langle n \rangle$ begins with a_i, otherwise go to the next instruction.

In these, $\langle n \rangle$ denotes the content of register n, and ϵ the empty word. A machine whose instructions are of the above types is called a URM($\mathcal{A}$).

The main result of Appendix B is that all partial recursive functions over $\mathcal{A}^*$ may be computed on some URM($\mathcal{A}$). A variant of the instruction set above comes in Appendix C. It replaces the second and third types of instructions by one "scan and delete" scheme defined as follows:

$Scd(n)[E_1 \ldots, E_s]$: scan the first letter of $\langle n \rangle$; if $\langle n \rangle = \epsilon$, go to the next instruction; if the first letter of $\langle n \rangle$ is a_i, then delete this and go to line E_i.

Then the main result of Appendix C is that one can simulate the delete and jump instructions using the scan and delete operation.

The paper contains numerous other results. However, it does not formulate direct results like the s_n^m-Theorem for URM($\mathcal{A}$) computations. We would like to do this directly. For this, it is essential that $\mathcal{A}$ include whatever symbols are needed to formulate the syntax of the overall language. (In the setting of [6], this is problematic for an interesting reason. If the symbols $P^{(i)}$ are taken to be atomic symbols $P^{(i)}$ – and this seems to be the prima

facie interpretation – then the alphabet $\mathcal{A}$ would have to include those symbols. But this is absurd if $\mathcal{A}$ is to be finite. So we must reformulate the language to get around this.)

Register machines are used in many textbooks due to their intuitive appeal. Needless to say, in writing this paper I looked at many sources to make sure that the development was new. Occasionally register machines are called *counter machines*. Minsky's textbook [5] on automata theory and computability presents a machine model that amounts to arithmetic register machines. He calls them *program machines*; it is not clear from the book why he did not mention [6] in the text even though it appears in the references. He notes that the program is " 'built into' the programming unit of the machine." Hence [program machines] "are *not*, then, 'stored-program' computers." But he then makes the quote reproduced above the introduction of the present paper. Perhaps Minsky did not even present the word register machines because register machines are not a direct model of a stored-program computer. But our point is that register machines operating on words do allow one to come closer to the notions that Minsky cites than machines operating on numbers. His book also does discuss Turing machines whose symbol set goes beyond one or two symbols to be an arbitrary finite set, including the set of symbols used in a "representation" of a Turing machine itself. Problem 7.4–3 of [5] asks a reader to construct a self-reproducing Turing machine, and the solution of course would have to use an expanded symbol set.

There are two papers that present material that seems close to ours. One is Corrado Böhm's paper [1]. He proposes a language P'' with a small instruction set, and proves that it is Turing-complete. P'' is intended to be run on Turing machines. Here are some differences with our work: we feel that our instruction sets would be easier for a complete novice to use than P''. (For example, moving register contents is a simple loop here.) Certainly the programs for the self-replicating program that we end up with is considerably smaller than what one finds for descendants of P'' such as BF. Finally, our languages are regular sets of expressions, whereas languages like P'' are context-free but not regular. (This is a very minor point.) We also know that Neil Jones in [4] has proposed simple languages and studied the s_n^m- and Recursion Theorems. The difference here is that Jones' languages are not based on as simple a machine model as register machines. There are good reasons why Jones works with the languages that he formulated, of course.

There might be classroom situations where one might want a presentation like the one we outline here. I think back to my own first exposure

to Computability Theory, an inspiring course given by Herbert Enderton at UCLA around 1977. The course used register machines as its primary model, and in the first few weeks we had to run programs written on punched cards. Later on, the course turned to μ-recursion, and via the usual coding machinery it presented the s_n^m-Theorem (but not the Recursion Theorem). The development here would allow one to efficiently present all the main results of interest without any of the coding; for some courses this would be a good idea. Instead of punched cards one now has graphical interfaces, and for this the technical overhead in learning the language would be small indeed. The material would work well in any course that wants to discuss self-replicating computer programs, a topic that comes up in various settings where reproduction is studied, including compilers, artificial life, and security. So someone teaching those topics could introduce them using `1#` and the freely available tools for it.

3 The language `1#`

A register is a time-varying word indexed by a positive integer. We work with a machine whose registers are R1, R2, R3, ... Each program uses a fixed finite number of registers. We may either take the ideal machine which we now describe to have infinitely many registers, or else to have a number which includes all registers in whatever program we are running on it.

Syntax The basic alphabet of symbols is $\mathcal{A} = \{\texttt{1},\texttt{\#}\}$.

There are five types of instructions, and the full syntax is listed in Figure 1.

The set of programs is just the set of all nonempty concatenations of sequences of instructions. The set of instructions is a regular set, and hence so is the set of programs. The programs are uniquely (and efficiently) parsed into sequences of instructions.

We sometimes employ abbreviations in writing programs, and in particular we use | for the concatenation operation on programs. We also add explanations in English. None of this is part of the official language.

We experimented with variations on the syntax in order to get an instruction set which minimized the lengths of the programs of interest. Nothing beat Figure 1. Having extra characters allows for binary numbers, but this does not quite compensate for the extra branches in the case statements.

instr	meaning	instr	meaning
$\texttt{1}^n\texttt{\#}$	add `1` at the end of Rn	$\texttt{1}^n\texttt{\#\#\#\#}$	go backward n steps
$\texttt{1}^n\texttt{\#\#}$	add `#` at the end of Rn	$\texttt{1}^n\texttt{\#\#\#\#\#}$	cases on Rn
$\texttt{1}^n\texttt{\#\#\#}$	go forward n steps		

Figure 1: The instruction set of `1#`. Here $n \geq 1$, and $\texttt{1}^n$ is n consecutive `1`s.

Semantics The easiest way to present the semantics is to run an evaluator in connection with our tutorial on this topic. For those reading this on its own, here are the details.

The registers store words $w \in \mathcal{A}^*$. Running a program, say p, means executing a sequence of steps, and at each step one or another of the instructions which comprise p is *active*. It is convenient to number those instructions. We begin with instruction 1 of p. When p starts, some registers might contain words; these are the *inputs*. Actually, we do not distinguish between a register being empty and its containing the empty string. The various instructions in our set involve writing to the end of a register, popping an element from the front of a register and then branching according to what was found, and outright transfer (=goto) statements. Here is more detail on all of these.

All writing is to be done at the end (the right side) of words. So if R6 contains the string `11##` and we execute the instruction `111111#`, then R6 will then have `11##1`. After executing a write instruction, the next instruction of p (if there is one) is the active one.

Executing a case statement $\texttt{1}^n\texttt{\#}^5$ removes the leftmost symbol of the word in register n, so that item is no longer there. After this, there are three *continuation branches*. In order, those branches are first for the case when register n is empty, then when the popped element is `1`, and finally for `#`. Suppose R3 contains the string `1##1` and instruction 17 of our program p contains the instruction `111#####`. In executing this, we drop the initial 1, the tail `##1` then remains in R3, and finally we proceed to instruction $17 + 2 = 19$ of p.

The transfer instructions are all *relative*; i.e., they specify a forward or backward transfer of some positive number of instructions.

Consider the execution of a program p. If at some point, instruction k is active and it asks to transfer forward l steps, and if p has $k + l - 1$ instructions, then after executing instruction k, we say that p *halts*. There are similar ways for p to halt following the add instructions and even case statements. Informally, and a bit incorrectly, we say that p halts if the active

line is "one below the last instruction of p."

1#-computable partial functions Let $n \geq 0$, and let $p \in \mathcal{A}^*$. We define the partial function $\varphi_p^{(n)} : (\mathcal{A}^*)^n \rightharpoonup \mathcal{A}^*$ by

$$\varphi_p^{(n)}(x_1, \ldots, x_n) \quad = \quad y$$

if p is a program, and if p is run with x_1 in R1, x_2 in R2, ..., x_n in Rn, and all other registers empty, then eventually the machine comes to a halt with y in R1 and all other registers empty. These partial functions $\varphi_p^{(n)}$ are called 1#-*computable*. (We allow $n = 0$, and in this case we would write $\varphi_p(\ )$. And in all cases we usually drop the superscript from the notation when it is clear.)

This notion of 1#-computability is not the only one worth studying. For many purposes, one would want 1#-*computability using the first k registers*. Equally well, one often wants definitions that keep the original input intact; the notion studied here loses the input.

Notation like $\varphi_p(x_1, \ldots, x_n) = \varphi_q(y_1, \ldots, y_m)$ has the usual meaning: either both sides are undefined, or both are defined and equal.

The empty string ϵ is not a program, and so $\varphi_\epsilon^{(n)}$ is the empty function for all n.

A final comment: it is clear that restricting our notion from sequences from $\mathcal{A} = \{1, \#\}$ to sequences of 1's alone, the 1#–partial computable functions are exactly the partial computable functions in the classical sense. There is also a formulation of this result which uses $\mathcal{A}$ to represent numbers in binary.

4 Programs

4.1 Programs to move register contents

Here is a program $\mathsf{move}_{2,1}$ which writes the contents of R2 onto the end of R1, emptying R2 in the process:

11#####	cases on R2	1111####	go backward 4
111111###	go forward 6	1#	1 branch: add 1 to R1
111###	1 branch: go forward 3	111111####	go backward 6
1##	# branch: add # to R1		

Note that in our displayed examples we often write the instructions going down the two columns. Similarly, we can build $\mathsf{move}_{m,n}$ for all distinct

`1#####`	cases on R1	`111111####`	go backward 6
`111111111###`	empty branch	`11#`	1: add 1 to R2
`11111###`	to `1` branch	`11##`	add `#` to R2
`11#`	`#` branch	`111111111####`	go backward 9
`11##`	add `#` to R2	$\mathsf{move}_{2,1}$	
`11##`	add `#` to R2		

`1#####11111111###11111###11#11##11##111111####11#11##`
`111111111####11#####111111###111###1##1111####1#111111####`

Figure 2: The program write.

numbers m and n. The official program $\mathsf{move}_{m,n}$ is

$$\texttt{1}^m\texttt{\#\#\#\#\#11111\#\#\#111\#\#\#1}^n\texttt{\#\#1111\#\#\#\#1}^n\texttt{\#111111\#\#\#\#}.$$

4.2 Comparison and reversal

It is a good exercise to write some programs dealing with string manipulations. One would be to write a program compare with the following property. When run with x in R1 and y in R2, compare halts with 1 in R1 (and nothing in any other register) if $x = y$, and with R1 (and all other registers) empty if $x \neq y$.

A better exercise is to write a program that reverses words.

4.3 A program to write a program to write a word

Figure 2 contains a program write with the following property: when write is started with x in R1 and R2 empty, we eventually halt with a word $y = \varphi_{\mathsf{write}}(x)$ in R1 and all other registers empty. Moreover, y is a concatenation of instructions `1#` and `1##`. So running y writes the original x after whatever happens to be in R1, and does not touch the other registers. This implies that for all x,

$$\varphi_{\varphi_{\mathsf{write}}(x)}(\,) \quad = \quad x.$$

The official program is shown at the bottom of Figure 2; the parse is on top. We use the horizontal line to indicate that $\mathsf{move}_{2,1}$ is concatenated at that point.

4.4 s_1^1

We construct a program s_1^1 which, when when started with a program p in R1 and a word q in R2, and R3 and R4 empty, yields a program $\varphi_{\mathsf{s}_1^1}(p,q)$ which, when started with r in R1, yields the same word as would be obtained when p is started with q in R1 and r in R2. In particular, when p is a program,

$$\varphi_{\varphi_{\mathsf{s}_1^1}(p,q)}(r) \quad = \quad \varphi_p(q,r). \tag{1}$$

We take s_1^1 to be

$$\mathsf{move}_{1,3} \,|\, \mathsf{move}_{2,1} \,|\, \mathsf{write} \,|\, \mathsf{move}_{1,2} \,|\, \varphi_{\mathsf{write}}(\mathsf{move}_{1,2}) \,|\, \mathsf{move}_{2,1} \,|\, \mathsf{move}_{3,1}$$

We use | as a symbol for concatenation of programs. Note also that the third of the seven segments is write, the program that we saw in Section 4.3 just above; the fifth is the result of applying that program to $\mathsf{move}_{1,2}$. Then the following equations show the desired result (1):

$$\begin{array}{lcl} \varphi_{\mathsf{s}_1^1}(p,q) & = & \mathsf{move}_{1,2} \,|\, \varphi_{\mathsf{write}}(q) \,|\, p \\ \varphi_{\mathsf{move}_{1,2} \,|\, \varphi_{\mathsf{write}}(q) \,|\, p}(r) & = & \varphi_p(q,r) \end{array}$$

Incidentally, it turns to be easier to directly program some of the textbook applications of the s_n^m-Theorem than to use the theorem itself. So we shall not use s_1^1 in the sequel, in particular not in the next section and not in Section 6 on the Second Recursion Theorem.

4.5 The programs diag and self

This section illustrates self-replicating programs in `1#`. We begin with the following program which we'll call diag for the rest of this paper. When diag is run with x in R1, the result is $\varphi_{\mathsf{write}}(x) \,|\, x$ in R1. It follows from this that

$$\varphi_{\varphi_{\mathsf{diag}}(x)}(\,) \quad = \quad \varphi_{\varphi_{\mathsf{write}}(x) \,|\, x}(\,) \quad = \quad \varphi_x(x). \tag{2}$$

Here is the informal description of diag:

Move x from R1 into R2, and at the same time put $\varphi_{\mathsf{write}}(x)$ in R3. Move R3 back to the now-empty R1. Finally, move R2 onto the end of R1.

The way to formalize the "at the same time" is to write one combined

loop:

`1#####`	cases on R1	`1111111####`	go back 7
`11111111111###`	empty branch: go 11	`11#`	add 1 to R2
`111111###`	`1` branch: go 6	`111#`	add 1 to R3
`11##`	`#`: add `#` to R2	`111##`	add `#` to R3
`111#`	add 1 to R3	`11111111111####`	go back 11
`111##`	add `#` to R3	$\mathsf{move}_{3,1}$	
`111##`	add `#` to R3	$\mathsf{move}_{2,1}$	

One way to get a self-writing program self is to apply this program diag to diag itself. When we run diag on itself, we get $\varphi_{\mathsf{write(diag)}}|\mathsf{diag}$ in R1. So when we run self on nothing, we first write diag into R1; and second we run diag. This gives us self, as desired. For a more formal proof, we use (2) with $x = \mathsf{diag}$:

$$\varphi_{\mathsf{self}}(\,) \quad = \quad \varphi_{\varphi_{\mathsf{diag}}(\mathsf{diag})}(\,) \quad = \quad \varphi_{\mathsf{diag}}(\mathsf{diag}) \quad = \quad \mathsf{self}.$$

One can find the full programs on `www.indiana.edu/~iulg/trm`.

Programs like self are often called *quines* following Douglas Hofstadter in [3]; there are several web pages devoted to collections of them, for example. A standard example is to take λ-terms as the programs, β-conversion as the notion of execution, and then to consider $(\lambda x.xx)(\lambda x.xx)$.

It might be useful to expose the device behind our self-replicating program self by rendering it into English. We are interested in "programs" (sequences of instructions) of a simple form, including instructions to print various characters, and instructions which accept one or more sequences of words as arguments, and so on. We'll allow quotation, and we won't attempt to formulate a minimal language, or a formal semantics. We're aware of the semantic problems that are being pushed under the rug, but the point is only to hint at a rendering of diag and self. Perhaps the most direct example of a self-replicating program would be `write me`, but this not immediately translated to `1#`. Instead, we want a version of diag, and we take

```
write the instructions to write what you see before it    (3)
```

Here "what you see" and "it" refer to the input. So applying this informal rendering of diag to `write me` would give

```
print "w" print "r" print "i" print "t"
print "e" print " " print "m" print "e" write me
```

Applying this version of diag to itself gives a long program which would look like

```
print "w" print "r" print "i" print "t" print "e"     ···
print "r" print "e" print " " print "i" print "t"          (4)
write the instructions to write what you see before it
```

Executing (4) prints instructions to print (3) followed by (3) itself. This is (4).

5 Exercises

I have used most of the exercises below as classroom exercises before turning to Kleene's Recursion Theorem. As one might expect, some of the problems can be solved by appealing to the Recursion Theorem. However, the direct solutions are usually shorter.

1. Write a program which when started on all empty registers writes itself to R1 and `#` to R2.

2. Write an infinite sequence of pairwise different programs

   $$p_1, p_2, \ldots, p_n, \ldots,$$

   such that for all n, running p_n with all registers empty gives p_{n+1} in R1.

3. Write a self-replicating program that begins with the program to transfer ahead one instruction, `1###`.

4. Which programs p have the property that there is a self-replicating program which begins with p?

5. Write a program s which when run with R2, R3, ..., initially containing the empty string, writes s itself into R1 after whatever happens to be there to begin with.

6. Write a program p with the property that when run on a string q in R1, p runs and halts with 1 in R1 if $q = p$, and runs and halts with R1 empty if $q \neq p$. (So p "knows itself".)

7. Write a program p with the property that when run on a program q in R1, the result is the same as would be the case when q is run with p in R1. (So *program* and *data* have changed roles!) [You will want a circular interpreter for this.]

8. Write two "twin" programs s_1 and s_2 with the properties that (a) $s_1 \neq s_2$; (b) running s_1 with all registers empty gives s_2 in R1; (c) running s_2 with all registers empty gives s_1 in R1.

9. Work out a version of the Busy Beaver problem in this setting.

6 Kleene's Second Recursion Theorem

Here is our formulation of this fundamental result: Let p be a program, and consider $\varphi_p^{(2)}$. Then there is a program q^* so that for all r,

$$\varphi_{q^*}(r) \quad = \quad \varphi_p(q^*, r)$$

For the proof, let $\hat{q}$ be as follows (later we set $q^* = \varphi_{\hat{q}}(\hat{q})$):

$$\mathsf{diag} \,|\, \mathsf{move}_{1,2} \,|\, \varphi_{\mathsf{write}}(\mathsf{move}_{1,4}) \,|\, \mathsf{move}_{2,1} \,|\, \varphi_{\mathsf{write}}(\mathsf{move}_{4,2} \,|\, p)$$

We thus have that for all x,

$$\varphi_{\hat{q}}(x) \quad = \quad \mathsf{move}_{1,4} \,|\, \varphi_{\mathsf{diag}}(x) \,|\, \mathsf{move}_{4,2} \,|\, p$$

so that $\varphi_{\varphi_{\hat{q}}(x)}(r) = \varphi_p(\varphi_x(x), r)$. (This last point is worth checking in detail; it uses the fact that diag only uses the first three registers.) Let $q^* = \varphi_{\hat{q}}(\hat{q})$. So

$$\varphi_{q^*}(r) \quad = \quad \varphi_{\varphi_{\hat{q}}(\hat{q})}(r) \quad = \quad \varphi_p(\varphi_{\hat{q}}(\hat{q}), r) \quad = \quad \varphi_p(q^*, r).$$

There are also versions of this result for $\varphi_p^{(k)}$ with $k \geq 3$, and the details are similar.

7 Smullyan's Double Recursion Theorem

Our final result is Smullyan's Double Recursion Theorem, a result used by Smullyan in work on recursion theory connected to the Gödel Incompleteness Theorems. Let p and q be programs. Consider the functions of three arguments $\varphi_p^{(3)}(a, b, x)$ and $\varphi_q^{(3)}(a, b, x)$. There are programs a^* and b^* so that for all x, $\varphi_{a^*}(x) = \varphi_p(a^*, b^*, x)$, and $\varphi_{b^*}(x) = \varphi_q(a^*, b^*, x)$.

We lack the space to exhibit a^* and b^* explicitly in terms of p and q (but Exercise 8 in Section 5 above could be a good first step). One example application of the Double Recursion Theorem would be to find p and q so that $\varphi_p(p) = q$, $\varphi_p(q) = p$, $\varphi_q(p) = q$, and $\varphi_q(q) = p|q$.

8 Using 1#

Several students in my class enthusiastically implemented `1#`. Some of their work has been polished, and it is available at `www.indiana.edu/~iulg/trm` along with other material. The most widely usable is a Java program that comes with a pleasant interface that allows one to watch registers change as programs run. Other interpreters come with tools which make it easier to write programs; so in effect one can write in a slightly-higher-level language that is translated back to a bona fide `1#` program. There also is a universal program (a circular interpreter) for `1#`, that is a `1#` program u such that for all p and q, $\varphi_u(p,q) = \varphi_p(q)$.

Acknowledgements

My thanks to Will Byrd and Jiho Kim for numerous comments, corrections, references, and suggestions.

References

[1] Böhm, Corrado."On a family of Turing machines and the related programming language", ICC Bull. 3, 187-194, July 1964.

[2] Fitting, Melvin. *Computability Theory, Semantics, and Logic Programming*. Oxford Logic Guides, 13. The Clarendon Press, Oxford University Press, New York, 1987.

[3] Hofstadter, Douglas R. *Gödel, Escher, and Bach: an Eternal Golden Braid*. Basic Books, Inc. New York, New York.

[4] Jones, Neil. Computer implementation and applications of Kleene's *s-m-n* and recursion theorems. in Y. N. Moschovakis (ed) *Logic From Computer Science*, 243–263, Math. Sci. Res. Inst. Publ., 21, 1992.

[5] Minsky, Marvin L. *Computation: Finite and Infinite Machines.* Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.

[6] Shepherdson, J. C. and Sturgis, H. E. Computability of recursive functions. *J. Assoc. Comput. Mach.* 10 217–255, 1963.