TOWARD AN INTUITIVE AND INTERESTING THEORY COURSE: THE FIRST STEP OF A ROAD MAP

John L. Lowther and Ching-Kuang Shene
Department of Computer Science
Michigan Technological University
Houghton, MI 49931-1295
E-mail: {john,shene}@mtu.edu

Abstract

This paper presents the first step of an attempt in designing intuitive and interesting materials for a theory course. The materials developed cover the **AL5 Basic Computability** unit of the *ACM/IEEE Computing Curricula 2001*, and can be used in a stand-alone theory course. This paper describes a "programming approach" to basic computability. Topics include a proof of the Halting Problem and the use of a simple reduction technique to prove other interesting problems. Details of the chosen computation model, the construction of a universal program, and the Isomorphism Theorem are also discussed. Future topics for this course include advanced computability, computability with real numbers, and the connection between theory and programming languages.

1. INTRODUCTION

In the mind of many students, a theory course is boring, dry, useless, and lacks interesting topics except for, possibly, the Halting Problem. Computation theory was developed at the turn of the 20th century as a direct response to David Hilbert's *Entscheidungsproblem*: find a mechanical procedure to prove *all* mathematical theorems. In today's language, this mechanical procedure is an algorithm. Eventually, three equivalent approaches, λ-definability of Church-Kleene (1932-34), general recursiveness of Gödel-Herbrand (1934) and Turing machines (1936), were developed. In the 50's, λ-calculus became the foundation of type theory and programming language development (*e.g.*, LISP, ISWIM and ML); general recursiveness was developed into the now well-known recursive function theory; and Turing machines became a standard computer science topic. Thus, it is hard to believe a 70-year old well-developed field has no interesting and contemporary topics to teach. To address this problem, we surveyed the literature in classical and modern computation theory to find interesting ways of re-interpreting the materials that are currently covered in a typical course and as specified in unit **AL5 Basic computability** of CC2001 [1].

Since students are used to writing and reasoning about their programs and since programming and reasoning in a high-level language is easier than doing the same with Turing machines, we chose a programming approach. Note that the programming approach is not new [6,8]; however, ours is more approachable and easier to understand. We divide the course materials into four categories: foundations, advanced computability, computability with real numbers, and the connection between theory and programming languages (*e.g.*, design, semantics and type theory). This paper reports our effort to make the foundations of computability more intuitive and interesting, and paves the way to the remaining three categories.

In this paper, Section 2 reviews the proof that the Halting Problem is not computable, Section 3 explores related results and shows that there is no universal anti-virus program, and Section 4 discusses a simple reduction technique that can help prove more non-computability problems. To establish a rigorous foundation, Section 5 presents a simple assembly language-like language and discusses a standard method for encoding a program as a natural number. With this encoding method, Section 6 presents Turing's most important contribution to modern computer design, the concept of universal program. Section 7 discusses the Isomorphism Theorem that shows the computability results obtained in this paper are universal and can be applied to any programming system and programs written in any language as long as the systems satisfy a minor constraint. Finally, Section 8 has our conclusions.

2. THE HALTING PROBLEM

Without loss of generality, all functions are from N to N, where $N = \{ 0, 1, 2, ...,\}$ is the set of natural numbers. A function is *total* if its domain is N. Otherwise, it is *partial*. A function is *partially computable* if it is computed by some program. Note that this program may not halt on some input because it is partial. A function is *computable* if it is computed by some program that halts on every possible input (*i.e.*, total). The *Halting Problem* asks if there is a program that can determine if a given function is computable. Turing showed in his seminal paper that the Halting Problem is not computable [13].

If the Halting Problem is computable, a program **Halt**(p,x) exists that computes the Halting Problem, where p is an arbitrary program to be tested for halting and x its input, such that **Halt**(p,x) returns TRUE if program p running on input x halts. With **Halt**(), we can write a new program **H**() as follows:

```
function   H(x)
begin
     while Halt(x,x) do;
end   H
```

Thus, **H**(x) halts if and only if **Halt**(x,x) returns FALSE, and we have

$$\mathbf{H}(x) \text{ halts} \Leftrightarrow \neg\mathbf{Halt}(x,x)$$

On the other hand, "**H**(x) halts" means program **H** running on input x halts, which is equivalent to "**Halt**(**H**,x) returns TRUE"' Combined with the previous result, we have

$$\mathbf{Halt}(\mathbf{H},x) \Leftrightarrow \neg\mathbf{Halt}(x,x)$$

Since x is an arbitrary input, it can be replaced by **H**:

$$\mathbf{Halt}(\mathbf{H},\mathbf{H}) \Leftrightarrow \neg\mathbf{Halt}(\mathbf{H},\mathbf{H})$$

This is a contradiction and **Halt**(p,x) is not computable.

Students in general do not have much difficulty in understanding this line of reasoning; however, they ask a lot of questions. Here is a list of some important ones.

First, can we use a program as an argument, especially replacing x by **H** in the last step, because the "type" of the arguments are different? **Second**, is this line of reasoning, language and system dependent? In other words, does this argument hold if the "program" is written in a different language and runs on a different operating system or machine? **Third**, Hilbert expected a "mechanical procedure" and we use a"`program." Are mechanical procedures, algorithms, and programs the same?

The first question is difficult to answer completely because we need to convince students that first-order programming languages (*e.g.*, LISP) treat programs and data indifferently and a program can be modified and then executed. However, there is an easy and initially acceptable explanation. We encourage students to think each byte of the executable file of a program as a base 256 digit. Thus, a program in its binary executable form is simply an integer, and can be handled and processed in the usual way. A rigorous answer to this question will be discussed later (Section 5). The second question relates to the *Isomorphism Theorem* which states, intuitively, that a result derived from one programming system also holds in other programming systems as long as some fundamental assumptions are fulfilled (Section 7). Therefore, we can use any programming system for our reasoning. The answer to the third question is the Church-Turing Thesis.

3. EXPLORING FURTHER

Next, we explore possible applications of the proof technique described in the previous section. There are two key concepts. The first is the **while** statement that negates what we expect. More precisely, **H**() halts only if **Hal**}(x,x) returns FALSE. The second is *self-reference*: running x as a program on x as an input. In fact, self-reference is even more important than negation, because it is used in the foundation of mathematics, set theory, and type theory in programming languages.

A universal anti-virus program determines if a program running on an input is a virus. We shall show that no such program exists [5]. Assume **Anti-Virus**(p,x) is an anti-virus program that returns TRUE if program p running on input x is a virus. Consider program **V**() below:

```
function  V(x)
begin
    if  ¬Anti-Virus(x,x)  then
        inject a virus and destroy the OS
 end  V
```

Program **V**(x) does the opposite of what **Anti-Virus**() says. Is **V**() is a virus? Since **V**(x) is a virus only if **Anti-Virus**(x,x) returns FALSE, we have

$$\mathbf{V}(x) \text{ is a virus } \Leftrightarrow \neg\textbf{Anti-Virus}(x,x)$$

Since "**V**(x) is a virus" means that **V** running on input x is a virus, **Anti-Virus**(**V**,x) should return TRUE. Hence, we have

$$\mathbf{V}(x) \text{ is a virus } \Leftrightarrow \neg\textbf{Anti-Virus}(\mathbf{V},x)$$

Combining these two results we have

$$\textbf{Anti-Virus}(\mathbf{V},x) \Leftrightarrow \neg\textbf{Anti-Virus}(x,x)$$

Finally, replacing x with **V** for self-reference, we have

$$\textbf{Anti-Virus}(\mathbf{V},\mathbf{V}) \Leftrightarrow \neg\textbf{Anti-Virus}(\mathbf{V},\mathbf{V})$$

This is a contradiction and **Anti-Virus**() is not computable. Hence, there is no universal anti-virus program!

This proof and that of the Halting Problem follow exactly the same line of reasoning: (1) assume the problem is computable so that a program is available to solve the problem, (2) use this program to write a related program that negates its original purpose, (3) ask the question if the new program satisfies the assumption of the problem, and (4) use self-reference to derive a contradiction that the given problem is not computable. While the Halting Problem may appear too theoretical, the non-existence of universal anti-virus program easily inspires students' curiosity. With some effort, students can create many interesting problems that can be proved to be non-computable. For example, there is no universal debugger that can scan a given program and its input to determine if it is bug-free. We believe this approach can significantly increase the understanding of the basic proving technique for similar problems.

4. SIMPLE REDUCTIONS

This section describes a simple reduction technique. Intuitively, problem *A* “reduces” to problem *B* if with the help of a program **P** we can show that *A* holds if and only if *B* holds. Normally, program **P** is an instance of *A* and contains an instance of *B*. In other words, an instance of *B* is embedded in an instance of *A*. Thus, the computability of *B* dictates the computability of *A*. Or, the computability of *A* depends on the computability of *B*. We choose to use an intuitive approach without a formal definition because the many-one reduction concept requires more preparation. Since the Halting Problem is not computable, it is usually problem *B*.

A *constant* program always returns the same value for every input. Is checking whether or not a program is constant, computable? We shall reduce this problem to the Halting Problem. Let *p* be an arbitrary program and *c* a constant. Consider program **C**():

```
function  C(x)     // instance A
begin
     run program p using p as its input;     // instance B
     return  c;
end  C
```

Thus, **C**() is a constant program iff program *p* running on itself as input halts. Since “program ***p*** running on itself as input halts” is **Halt**(*p*,*p*), we have

$$\mathbf{C}() \text{ is a constant function } \Leftrightarrow \mathbf{Halt}(p,p)$$

Hence, testing to see if **C**() is a constant function reduces to the Halting Problem via the above program. Since *p* is an arbitrary program and **Halt**(*p*,*p*) is not computable, checking if **C**() is a constant function is not computable! The non-existence of universal anti-virus program can also be proved similarly.

This reduction is a starting point for showing the non-computability of many other problems. For example, the Equality problem is not computable. Given two programs **f**(x) and **g**(x), the Equality problem asks if $\mathbf{f}(x)=\mathbf{g}(x)$ for all $x \in N$. This is not computable, since it is equivalent to checking if program $(\mathbf{f} - \mathbf{g})(x)$ is a zero (constant) program. It happens very frequently in real world that an old system written in an old language (*e.g.*, Fortran 66) must be reengineered or cloned using a modern language (*e.g.*, C++). The question is: Do both systems perform exactly the same way? This is the Equality problem, which is not computable. Therefore, making sure the old and the new systems perform the same way is an art rather than a science!

The same reduction technique can also help prove other interesting results. Consider the following program **I**(x):

```
function  I(x)     // instance A
begin
    run program p using p as its input;     // instance B
    return  x;
end  I
```

It reduces identity checking to the Halting Problem since

$$\mathbf{I}(x) \text{ is an identity program } \Leftrightarrow \mathbf{Halt}(p,p)$$

Therefore, checking if a program is an identity program is not computable. In fact, this program provides more than we expect. Program **I**(x) is also increasing, one-to-one and onto. Thus, checking if a program satisfies one of these properties is not computable. Once we reach this point, our experience shows that some students ask a simple and very important question: Why can this simple scheme be used to prove the non-computability of so many problems? Is there a general way to do this? Even if students do not raise this question, we should point out that Rice's Theorem is the answer. Rice's Theorem states that virtually all interesting properties that involve checking the input-output behavior of a program are not computable.

5. BUILDING A SOLID FOUNDATION

In this section, we shall make our computation model rigorous and answer the first question posted at the end of Section 2. There are many such computation models in existence, some of which use assembly instructions, while some others use very simple high-level instruction-like statements. In general, these computation models are similar to each other with minor variations. The computation model we use can be found in many well-known computation theory textbooks [2,3].

5.1 Computation Model

Our computation model consists of memory and instructions. The memory has three sections, *input*, *working* and *output*. Each of the *input* and *working* memory has a finite but unbounded number of variables. *Input* variables are named as $X1$, $X2$, ..., and *working* variables are $W1$, $W2$, ... The *output* has only one variable Y. Each of these variables can hold an arbitrary precision natural number. The *input* and *working* variables are used for passing arguments to a program and for saving intermediate results, respectively. The *output* variable is used for returning a single value. With a standard encoding technique (*e.g.*, Gödel numbering), one can encode multiple output values into a single natural number. All variables are initialized to zero before a program runs, and can be used in a program for computation.

We only need the following three instructions:

L:	$X \leftarrow X + 1$	Add 1 to variable X
L:	$X \leftarrow X - 1$	Subtract 1 from variable X
L:	`IF` $X \neq 0$ `GOTO` $\underline{L}$	Goto label $\underline{L}$ if $X \neq 0$

Each instruction has an optional label L chosen from $L1$, $L2$, ... If the target label $\underline{L}$ in the `IF-GOTO` instruction does not exist, the program halts. Otherwise, the execution transfers to the *first* occurrence of $\underline{L}$ from the beginning of the program. If the value of a variable is zero, the subtraction instruction has no effect (*i.e.*, cut-off subtraction).

Other instructions can be implemented using macros. Let A, B and C be three variables. Macro $B \leftarrow A$ can be implemented by subtracting 1 from A and adding 1 to B until A is zero; $C \leftarrow A+B$ can be done by copying A to C and adding 1 to C B times; $C \leftarrow A*B$ can be done by adding A to C B times; integer division $\lfloor A/B \rfloor$ and remainder can be done with subtraction; and unconditional goto can be implemented with a non-zero variable. Note that the value of one of the involved variables may be destroyed in a computation. However, one can always restore its original value with an extra copy. Thus, all commonly used instructions can be implemented with macros, and a program can always be "compiled" to this simple language.

5.2 Encoding a Program

We shall encode each instruction into a natural number. The variables are ordered into a linear sequence: Y, $X1$, $W1$, $X2$, $W2$, $X3$, $W3$, ... The number (or address) of variable V, $\#(V)$, is its position in the sequence: $\#(Y)=0$, $\#(Xi)=2i-1$ and $\#(Wi)=2i$. Each label is also numbered with its index (*i.e.*, $\#(Li)=I$).

The Gödel pairing function maps an order pair (x,y) to a natural number $\langle x,y\rangle = 2^x(2y+1) - 1$. This is a one-to-one and onto function from $N \times N$ to N. For example, $\langle 2,4\rangle = 2^2(2\times 4+1) - 1 = 35$. Conversely, given a natural number w, we can compute $w+1$, extract the power of 2 (*i.e.*, 2^x), and compute y from the remaining odd number. For example, given $w=103$, we have $w+1=104=2^3\times 13=2^3(2\times 6+1)$ and $103=\langle 3,6\rangle$. In the following, if $w=2^x(2y+1) - 1$, we shall write $x=\pi_1(w)$ and $y=\pi_2(w)$. Thus, we have $\pi_1(103)=3$ and $\pi_2(103)=6$.

Each of the three instructions has three fields: a label, a variable, and an operation code for the addition and subtraction and a target label for the `IF-GOTO`. Therefore, they can be encoded in the following way. If an instruction has no label, the label field is zero. Since there are only three instructions, we can assign operation codes 0 and 1 to addition and subtraction instructions, respectively. Since the `IF-GOTO` has a target label whose number starts with 1, it can have an operation code of $\#(\underline{L})+1 \geq 2$.

L:	$V \leftarrow V + 1$	$\langle \#(L), \langle 0, \#(V)\rangle\rangle$
L:	$V \leftarrow V - 1$	$\langle \#(L), \langle 1, \#(V)\rangle\rangle$
L:	`IF` $V \neq 0$ `GOTO` $\underline{L}$	$\langle \#(L), \langle \#(\underline{L})+1, \#(V)\rangle\rangle$

For example, "$L1$: $X1 \leftarrow X1 - 1$" is encoded as $\langle 1,\langle 1,1\rangle\rangle = \langle 1,5\rangle = 21$, and "$Y \leftarrow Y+1$" is encoded as $\langle 0, \langle 0,0\rangle\rangle = 0$. On the other hand, since $22=\langle 0,11\rangle$ and $11=\langle 2,1\rangle$, we have $22=\langle 0,\langle 2,1\rangle\rangle$, and the corresponding instruction is "`IF` $X1 \neq 0$ `GOTO` $L1$." In this way, any instruction can be encoded into a natural number and any natural number can be decoded back to a valid instruction.

After encoding each instruction into a natural number, encoding the whole program is easy. Let the instructions of a program of k instructions be encoded in natural numbers $n1$, $n2$, ..., nk. The program is encoded as

$$p_1^{n1} p_2^{n2} \cdots p_k^{nk} - 1$$

where $p_1=2$, $p_2=3$, $p_3=5$, ... (*i.e.*, p_i is the i-th prime number). Hence, a program can be "assembled" into a natural number and a natural number can be "disassembled" back to a program. Consequently, programs and natural numbers have a one-to-one and onto correspondence, and we can assign each program with a nickname, its corresponding number. The following program copies the non-zero input $X1$ to the output Y:

```
L1:   X1 ← X1 − 1
      Y ← Y + 1
      IF X1 ≠ 0 GOTO L1
```

Since the three instructions are encoded as 21, 0 and 22, the program has its number $2^{21}3^{0}5^{22}-1$. On the other hand, since $2799=2^4 3^0 5^2 7^1 - 1$, the program has four encoded instructions: $4=\langle 0,\langle 0,1\rangle\rangle$, $0\ \langle 0,\langle 0,0\rangle\rangle$, $2=\langle 0,\langle 1,0\rangle\rangle$ and $1=\langle 1,\langle 0,0\rangle\rangle$, and the program is

```
        $X1 \leftarrow X1 + 1$
        $Y \leftarrow Y + 1$
        $Y \leftarrow Y - 1$
$L1$:   $Y \leftarrow Y + 1$
```

5.3 Discussion

While we can "disassemble" a natural number back to a set of encoded instructions, there is no guarantee that the relation between higher-level source programs and assembly instruction programs is also one-to-one. For example, we are not sure if two consecutive $X1 \leftarrow X1+1$ instructions is the result of compiling $X1 := X1 + 2$, two $X1 := X1+1$, or a part of $X1 := X1 + 3$. Hence, we will only use programs written in these simple statements in our reasoning. The major advantage of using these primitive statements is that the "disassembly" process is unique.

This "compile" and "assemble" process converts all possible programs to natural numbers, including those may not halt (*i.e.*, partially computable), and establishes a one-to-one and onto relation. Hence, we shall name the programs as $\varphi_0(X1,X2,\ldots)$, $\varphi_1(X1,X2,\ldots)$, $\varphi_2(X1,X2,\ldots)$, ..., where the index i of φ_i is the number of the corresponding program. When the arguments of a program is unimportant, we will only write φ_0, φ_1, φ_2, ...

6. A UNIVERSAL PROGRAM

We shall write a program $\Phi^n(X1,X2,\ldots,Xn,k)$ that takes a program k and its input (*i.e.*, $X1$, $X2$, ..., Xn), executes k, and delivers the result. Hence, Φ^n can be considered as an interpreter that executes the instructions of program k. Because it can execute every program that has n input arguments, Φ^n is called a *universal program* of order n.

Program for Φ^n is easy to write. The values of all variables can be encoded into a natural number

$$M = 2^{Y}3^{X1}5^{W1}7^{X2}11^{W2}13^{X3}17^{W3}\cdots$$

where Y, Xi and Wi represent the values of the variables. Thus, Y is the power of the first prime number $p_1=2$, Xi and Wi are the powers of prime numbers p_{2i} and p_{2i+1}, respectively. Since only the first n input arguments contain values and all other variables are zero initially, the initial value of M is $M = p_2^{X1}p_4^{X2}\cdots p_{2n}^{Xn}$. With the encoded program in k, the encoded input in M, and a program counter c, Φ^n simply emulates what a CPU does:

```
**function** $\Phi^n(X1, X2, \cdots, Xn, k)$
**begin**
```

```
    $M := p_2^{X1} p_4^{X2} \cdots p_{2n}^{Xn}$;
    $L$ := program length (*i.e.*, no. of instructions);
    $c := 1$;
    **while** $1 \le c \le L$ **do**
        fetch the $c$-th instruction;
        decode the instruction;
        execute the instruction and update $c$;
    **end while**
**end** $\Phi^n$
```

Because k is the encoded program, p_k is larger than the prime number used for encoding k's last instruction. Therefore, the length of k can be determined as follows: (1) start with p_k and go downward; (2) for each p_j find its power n_j in k; and (3) if n_j is non-zero, j is the length of k. Note that computing the power of p_j can be done by counting the number of times that p_j can evenly divide k.

In the above, "fetch the c-th instruction" is actually computing the power of p_c in k. If the power is n_c, the label, operation code, and involved variable are $l = \pi_1(n_c)$, $o = \pi_1(\pi_2(n_c))$ and $v = \pi_2(\pi_2(n_c))$. Since the value of a variable is the power of a prime number p_v in M, adding and subtracting 1 to and from a variable are equivalent to multiplying and dividing M by p_v, respectively. With this information, the following executes the instruction:

```
$c := c + 1$;  // advance program counter
$e$ := the power of $p_v$ in $M$;
**if** $o = 0$ **then**  // addition
    $M := M * p_v$
**else if** $o = 1$ **and** $e \ne 0$ **then** // subtraction
    $M := \lfloor M/p_v \rfloor$
**else if** $o \ge 2$ **and** $e \ne 0$ **then** // IF-GOTO
    $c := 0$;
    **for** $i$:=1 **to** $L$ **do**
        **if** $\pi_1$(the power of $p_i$ in $k$)  $= o - 1$ **then**
            $c := i$; // next instruction
            **break**;
        **end if**
    **end for**
**end if**
```

We have shown that the programming system based on our computation model has a universal program that can execute any program of n arguments. Moreover, if n is also an input argument, Φ^n can be modified easily to execute every program. Note that if the input program is partially computable, Φ^n may not halt and is also partially computable. Why is the universal program concept important? In Turing's words, "we only need one machine" rather than building one machine for each program. This is exactly the principle of building modern computers: we need general purpose computers (*i.e.*, universal programs) that can execute other programs. Therefore, Turing's universal program concept is perhaps the *most* significant contribution to modern computer design [4]! Some students may ask a simple question: would different CPUs yield different computability conclusions? This is answered in the next section.

7. THE ISOMORPHISM THEOREM

A *programming system* is simply a list of *all* partially computable and computable programs written in a particular language. A programming system is *universal* if it has a universal program. A universal programming system is *acceptable* if the composition construction operator of programs is computable. Note that non-universal programming systems are not very interesting because each program requires a special computer. Non-acceptable programming systems are not interesting either because there is no "computable" way to join two programs together to become a single program. Given two programs φ_i and φ_j in the list of programs, the composition $\varphi_k = \varphi_i \circ \varphi_j$ is also a program and should be a member of the programs list. In an acceptable programming system, the construction of φ_k from φ_i and φ_j is computable. This means we can write a program, which always halts, that takes i and j and outputs k. In our simple system, the construction of k from i and j can easily be done in the following way: **(1)** preserve i; **(2)** add copying macros to move the output of i from variables Xi's, Wi's and Y to $X1$, $X2$, ... with variable renaming if necessary; **(3)** append j to the copy macros with variable renaming if necessary; and **(4)** encode the resulting program. It is obvious that the program that performs these four step always halts for any two input programs i and j. Therefore, our simple system is an acceptable programming system.

Since a Turing machine is equivalent to a RAM model which, in turn, is obviously equivalent to the simple language in this paper, Turing machine programs also form an acceptable programming system. Under the Unix system, the Bourne shell language is powerful enough to write an interpreter for simulating itself (*i.e.*, universal program) and the pipe command provides the needed composition operator. Hence, conceptually Unix can be considered as an acceptable programming system. With the concept of acceptable programming systems, we have the following:

The Isomorphism Theorem Given any two acceptable programming systems φ_1, φ_2, φ_3, ... and ψ_1, ψ_2, ψ_3 , ..., there is a computable, one-to-one and onto function σ such that $\varphi_x = \psi_{\sigma(x)}$ for every x.

Therefore, the computability of one programming system can be mapped by an isomorphism σ, which acts very similar to a cross-compiler/translator, to another system as long as both are acceptable. Consequently, the computability results discussed earlier are not specific to our simple language system, and are general results that can be applied to all modern computer systems and languages. Hence, our approach that employs an intuitive and easy to understand way of re-interpreting the computability theory is equivalent and as powerful as other approaches. The Isomorphism Theorem signifies the end of this paper's basic computability module. Since its proof requires other sophisticated results, we do not present a proof in class. Instead, we discuss its meaning and the impact on computability. We will offer a proof in the advanced computability module.

8. CONCLUSIONS

For a number of years, we taught a beginning graduate level computability course based on recursive function theory [3, 9, 11, 12]. We faced a major challenge because of the huge differences in the theoretical background of our students. In fact, some were only about the level of beginning undergraduate students in terms of theoretical training. As a result, the rigorous development of the course topics was frequently interrupted by simple but important questions. This inspired us to create intuitive explanations to be used in classroom or office-hour discussions. This paper reports the result of this effort.

Since the complete set of knowledge units is still under development, there is no rigorous classroom evaluation. Based on our experience in many years, this programming approach, interesting examples, and discussions connecting programming, assembling and disassembling, and cross compiling have helped students who had little theoretical background understand the merit and appreciate the beauty of computability. Additionally, the same set of material was also used in a CS orientation course. A brief outline was published as a poster [10]. Reactions from students and poster visitors were very positive and encouraging. Some instructors indicated that they will use some of our examples and way of reasoning. The extended abstract of our poster was used in several universities as a reading material [7]. This suggests that our approach is sound, interesting and teachable.

In the near future, we will finalize the development of additional categories: advanced computability, computability with real numbers, and the connection between theory and programming languages. The **Advanced Computability** module will include some of the most important theorems, their meaning, and their importance to programming systems. Possible topics will include, but are not limited to, the *s-m-n* (or parameter) theorem, recursion and fixed-point theorems, and Rice's Theorem and its variations. The **Computability with Real Number** module will discuss the meaning of “computable real numbers” based on finite precision arithmetic, a number of different and equivalent computation models, the meaning of computable real functions and real operators, and some well-known problems with real numbers. For example, there exists a polynomial-time computable function f on [0,1] such that its derivative exists but is not computable. Moreover, if the second derivative of f exists and is continuous on [0,1], then the first derivative of f is polynomial-time computable. This module will also include some interesting modern results (*e.g.,* the Julia set and Mandelbrot set are not computable). The **Connection Between Theory and Programming Language** module will present the equivalence among Turing machines, recursive function theory and λ-calculus, and the impact of the latter two, especially the λ-calculus, on functional programming and programming language development. We shall introduce the basic concepts of λ-calculus and tie it to our programming approach. Then, we shall show that the *self-reference* concept is actually another facet of Russell's paradox that will cause a correctly written program not to halt. Then, type theory is introduced to make a programming language weaker so that a compiler can perform "checks" to prevent self-reference. We also will include some contemporary and interesting topics (*e.g.*, the quantum and DNA computing models). In this way, most major developments in computation theory are covered. Once these materials are in place, we will teach an experimental theory course with our new materials and conduct a rigorous and controlled classroom evaluation and assessment study.

ACKNOWLEDGMENT

The authors are supported by the National Science Foundation under grant DUE-0127401. The second author is also supported by an IBM Eclipse Innovation Award 2003.

REFERENCES

1 ACM/IEEE, *Computing Curricula 2001*, available at www.acm.org/sig/sigcse/ccs001.

2 Walter S. Brainerd and Lawrence H. Landweber, *Theory of Computation*, John Wiley & Sons, 1974.

3 M. D. Davis, R. Sigal and E. J. Weyuker, *Computability, Complexity, and Languages*, second edition, Academic Press, 1994.

4 M. D. Davis, *The Universal Computer*, W. W. Norton & Company, 2000.

5 W. J. Dowling, There Are No Safe Virus Tests, *American Mathematical Monthly*, Vol. 96 (1989), No. 9, pp. 835-836.

6 A. J. Kfoury, R. N. Moll and M. A. Arbib, *A Programming Approach to Computability*, Springer-Verlag, 1982.

7 Katherine St. John, *Computer Science 75010: Theoretical Computer Science*, Graduate Center, City University of New York, 2002. (http://comet.lehman.cuny.edu/stjohn/teaching/tcs)

8 John S. Mallozzi and Nicholas J. de Lillo, *Computability with Pascal*, Prentice Hall, 1984.

9 Hartley Rogers, Jr., *Theory of Recursive Functions and Effective Computability*, paperback edition, MIT Press, 1987.

10 Ching-Kuang Shene, Can Computation Theory Be Taught in an Interesting Way? a poster in *ACM 33rd SIGCSE Technical Symposium*, 2002, pp. 426.

11 Carl H, Smith, *A Recursive Introduction to the Theory of Computation*, Springer-Verlag, 1994.

12 Robert I. Soare, *Recursively Enumerable Sets and Degrees*, Springer-Verlag, 1987.

13 A. Turing, On Computable Numbers, with an Application to the \`Entscheidungsproblem', Proceedings of the London Mathematical Society, second series, Vol. 42 (1936), pp. 230-265. Correction, Vol. 43 (1937), pp. 544-546.