TOWARDS MAKING THEORY OF COMPUTATION COURSE MORE UNDERSTANDABLE AND RELEVANT: RECURSIVE FUNCTIONS, FOR-LOOPS, AND WHILE-LOOPS

Kreinovich, Vladik, PhD, Professor
Kosheleva, Olga, PhD, Associate Professor
vladik@utep.edu , olgak@utep.edu

Abstract: In this paper, we show how we can make a theory of computation course more understandable and more relevant: namely, we show that a seemingly abstract notion of primitive recursion is a direct counterpart to for-loops, while the mu-recursion is an analog of while-loops.

Keywords: theory of computation, teaching computing, primitive recursion, mu-recursion, for-loops, while-loops

Theory of Computation is useful. Theory of computation is very important for computing practice; see, e.g., [1]. Many of its general abstract concepts are actively used in practice: e.g., finite automata are a useful technique in designing computer hardware and in designing compilers.

Even negative (non-computability) results are practically useful. Indeed, a natural tendency is to write each code so that it can reused later for similar problems -- i.e., to make it as general as possible. However, when a problem is generalized too much, it sometimes becomes algorithmically unsolvable. It is thus desirable to know when a feasible algorithm is not possible -- so as not to waste time on trying to design an impossible general algorithm.

From this viewpoint, it is desirable that the students not only learn the *results* of theory of computations, but they should also learn the *proofs* -- so that in the future, when facing similar computational problems, they will be able to prove algorithmic impossibility of too broad generalizations by appropriately modifying these proofs.

Pedagogical problem. And here lies a pedagogical problem: while the current textbooks explain the relevance of the *results*, the *proofs* use abstract notions whose relevance to computing is unclear.

For the students to be able to understand and modify the proofs, they should be able to understand the *motivations* behind the steps and the *relation* between the abstract notions and computer practice.

What we do in this paper. In this paper, we explain the relation between the abstract notions and computer practice on the example of the first notion with which many graduate theory of computation courses start: the notion of a recursive function.

Notion of a recursive function: brief history. This notion -- as well as an auxiliary notion of a primitive recursive function -- comes from the 1930s pioneering work of the US mathematician Alonzo Church.

The possibility to clarify these notions stems from the fact that Church's formalism provided a way to describe computations -- what programming languages do now, once computers and compilers have been invented -- and many of its features strongly influenced the actual programming languages (starting with LISP and Scheme).

In this sense, programming languages and theoretical concepts have a joint origin. Yes, the programming languages and the theory formalisms evolved in somewhat different directions. However, as we will show, it turns out that it is possible to relate recursive functions to programming practice.

Specifically, we will show that primitive recursive functions can be interpreted as formalizing for-loops, while the notion ofmu-recursion (used to define general recursive functions) can be viewed as a natural formalization of while-loops.

How to describe a for-loop in precise terms? Let us start with a simple program for computing the power a^m:

```
power = 1;
for(int i = 1; i <= m; i++)
        {power = power * a;}
```

This is not a precise mathematical notation, because in math, a variable is assumed to have the same value in different parts of the equation, but here,

```
power = power * a;
```

means that to the variable "power", we assign a *new* value: the product of the *old* value and the value a.

To make the description mathematically precise, we must thus explicitly indicate the iteration number at which we consider the value of the variable "power": initially, before iterations start, we have the value power(0), then after the first iteration, we have the value power(1), etc. In these terms, the above code takes the following form:

```
power(0) = 1
power(i + 1) = power(i) * a
```

The value of the variable "power" also depends on the value of the variable a. If we explicitly describe this dependence, we end up with the following description:

```
power(a, 0)  = 1
power(a, i + 1) = power(i, a) * a
```

In a general for-loop with parameters $a_1, \ldots, a_k$, in which a variable h changes:

- we first assign some initial value to the variable h:

  $$h(a_1, \ldots, a_k, 0) = f(a_1, \ldots, a_k)$$

  for some expression $f(a_1, \ldots, a_k)$, and then

- on each iteration, we use the previous value of h, the values of the parameters a*j*, and the number of the iteration I to produce a new value:

  $$h(a_1, \ldots, a_k, i + 1) = g(a_1, \ldots, a_k, i, h(a_1, \ldots, a_k, i))$$

for some expression g.

Resulting description of a for-loop. Thus, we arrive at the following formal description of a for-loop. Let $a_1, \ldots, a_k$ be a list of parameters. To describe a for-loop in which a variable h changes, we need to know:

- an algorithm $f(a_1, \ldots, a_k)$ assigning an initial value to h; this value may depend on the parameters $a_1, \ldots, a_k$; and

- an algorithm $g(a_1, \ldots, a_k, i, h(a_1, \ldots, a_k, i))$ that describes what is happening inside the loop, on each iteration.

Once we have these two algorithms f and g, we can describe a function h computed by the for-loop:

$$h(a_1, \ldots, a_k, 0) = f(a_1, \ldots, a_k); \quad h(a_1, \ldots, a_k, i + 1) = g(a_1, \ldots, a_k, i, h(a_1, \ldots, a_k, i));$$

This is exactly Church's *primitive recursion*! So, *primitive recursion* can be described as *a natural formalization of a for-loop.*

Beyond for-loops: a while-loop. In the traditional *for-loop*, we know beforehand how many iterations we make.

In some algorithms, we do not know beforehand how many iterations we will need. Instead, we run iterations x_k until the process converges. For example, to compute the square root of a given number a, we can use the following iterative algorithm:

$$x_0 = 1; \quad x_{k+1} = (1/2) * (x_k + a / x_k).$$

We continue these iterations until the difference between the values of the two consequent iterations becomes smaller than a pre-defined threshold t:

$$| x_{k+1} -- x_k | < t.$$

In general, we run a *while-loo*p, which runs until a certain stopping condition P is satisfied.

So, to describe while-loops, we need to describe the smallest m for which the condition

$$P(n_1, \ldots, n_k, m)$$

holds, where $n_1, \ldots, n_k$ denote auxiliary parameters.

This value smallest value

$$\text{mu } m \,.\, P(n_1, \ldots, n_k, m)$$

is exactly Church's mu-recursion!

Thus, the basic ideas behind recursive functions are exactly natural formalizations of for- and while-loops.

Resulting meaning. We have shown that:

- primitive recursion corresponds to for-loops, and
- mu-recursion corresponds to while-loops.

From this viewpoint, many theoretical results acquire natural practical meaning.

As an example, let us consider the result that not every computable function is primitive recursive. This result is important because it is the first, simplest example of the diagonal construction that is later used in many other proofs.

From our viewpoint, the meaning of this result is as follows: while-loops are needed, because not all computable functions can be computed by using only for-loops.

Acknowledgments. This work was supported in part by the US National Science Foundation grants HRD-0734825, HRD-1242122, and DUE-0926721.

References

1. Sipser, M. Introduction of the Theory of Computation. Cengage Learning, Technology, Boston, Massachusetts, USA, 2013.