Abstractions, functions, and recursion

David Keil, Framingham State University, June 2011

Computer science is concerned with the practical issues of data manipulation by hardware and writing programs to accomplish this data manipulation. But it also has a theoretical aspect that is of critical practical importance. It involves mathematics. Theoretical computer science not only *uses* mathematics, as physics, chemistry and other fields use certain parts of math; computer-science theory *is* a branch of mathematics, also known in part as discrete mathematics.

1. Computers and abstractions

Mathematics is a field of study that works entirely with abstractions, such as numbers and operations. Similarly, computers by their nature work with abstractions, because they are symbol-manipulating machines. Letters, numbers, and words are among the symbols they operate on.

The *process* of abstraction lets us set aside concrete details (e.g., what do we have three of?) and concentrate on the matter at hand (e.g., three of anything plus five of anything equals eight of that thing).

As we will discuss in detail, writing a subprogram module to accomplish part of a program specification, such as output, is procedural abstraction, and designing classes, such as window classes, is a form of data abstraction. In each case we focus initially on deciding how a software component will work and on naming it, but we temporarily leave aside implementation details that are not crucial to the user of the component.

2. Sets and functions

If you studied algebra using a mathematical approach (as opposed to a formula-memorizing approach), then you know about *sets*. Three well-known sets are the set of natural numbers (0, 1, 2 ...), the set of real numbers (numbers that can each be represented as a series of digits with a decimal point somewhere in the series), and the set of truth values, {*True, False*}.

In algebra, geometry, and trigonometry, you encountered *functions*. Perhaps you learned that a function is a set, too. It is a *mapping* from one set to another set, possibly to the same one. Every function consists of ordered pairs of values in such a way that a given value in the first set always maps to a unique determined value in the second one. Thus, given a certain value on the left (the function argument), a function is quite predictable and always returns the same value on the right. Two values on the left, however, may both have the same return value for a certain function.

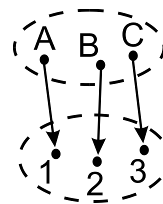
One example of a function is the one that returns the ordinal value, or position, of a letter in a series of letters. The letters ‘A’, ‘B’, and ‘C’, for example, have the positions 1, 2, and 3 in the alphabet.

So our position function might be diagrammed as below. We have the sets {‘A’, ‘B’, ‘C”}, {1, 2, 3}, and {(‘A’,1), (‘B’,2), (‘C’,3)}. The third set is a function, which we could call *index*:

index(‘A’) = 1

index(‘B’) = 2

index(‘C’) = 3



A similar mapping can define what we mean mathematically by oddness ():

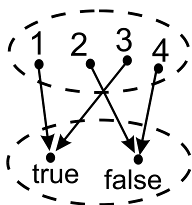
odd(1) = *true*

odd(2) = false

odd(3) = *true*

odd(4) = *false*

...



Notice that while none of our functions have two arrows coming out of the same argument set element, some may have two or more arrows pointing to the same return-value set element.

Finally, we have functions from numbers to numbers, such as the function that returns a value twice as large as its argument (input), illustrated below:

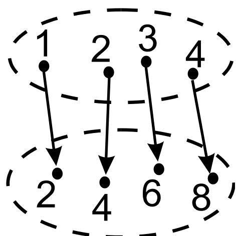
twice(1) = 2

twice(2) = 4

twice(3) = 6

twice(4) = 8

...



Any of the functions described above could be extended to map from or between large or even infinite sets.

3. Computable functions

In mathematics, a function is a passive, static abstraction. However, a very interesting set of functions is the category that can be computed. That is, for each *computable* function, there is one or more algorithm that starts with the argument value and step by step arrives at the return value. The number of steps must be finite for us to call the function computable.

Since you know that computer programs are supposed to be deterministic (predictable), it may make sense to you that every computer program that works correctly produces as its output the return value of a computable function, with the program's input corresponding to the function's argument. It may also make sense that for every computable function, it is possible to write a computer program whose output is the return value of the function. In fact, it is possible to write many such programs. Thus, we have a neat correspondence between the category of possible computer programs and a category of mathematical functions.

This is true whether we take a narrow view and consider only programs with numeric inputs and outputs, or a broad view, considering inputs and outputs that include text, graphics, mouse clicks, and so forth. Since all computer data is stored as bit patterns, and we have shown that any bit pattern corresponds to a number, therefore we could consider only that set of functions from numbers to numbers and still defend our claim that the class of computable mathematical functions has a correspondence to the set of all possible computer programs.

Now, when we ourselves compute functions by hand we use operators and other notation to specify the steps we take to get from the function's argument to its return value. For example, any two numbers have a sum, so we could define a mathematical function *sum*(*a,b*) whose arguments could be any natural numbers *a* and *b*. This function would return the sum of the two arguments. We may use the operator, +, to denote our use of the function, in this way: *a*+*b*.

Computing the sum, or +, function, may consist of looking up one or more values in an addition table and possibly taking other steps.

The expression, $(2 + 5) \times 4$, with the operators + and ×, corresponds to the application of a product function to two arguments, one of which is the sum of 2 and 5:

$(2 + 5) \times 4 = product(sum(2,5), 4)$

Finally, the language of mathematics provides us with ways to express functions of different numbers of arguments if the argument values progress in a natural way from one to the other. Consider *sum*(1, 2, 3, 4) or *sum*(1, 2, ... *n*) for some natural number *n*; in other words, consider the sum of a series of consecutive numbers starting with 1. Mathematics has a way to express such functions.:

$$sum(1, 2, 3, 4) = \sum_{k=1}^{4} k$$

$$sum(1, 2, ..., n) = \sum_{k=1}^{n} k$$

Here the symbol Σ (a Greek letter) is pronounced “sigma” or “summation.” It is used more or less as a super-powered plus sign.

Thus, mathematics gives us two ways to express the values of certain functions: the functional notation, with function names followed by arguments in parentheses, and operator notation. The operator notation often points to a step-by-step method for computing a function that is computable.

A *computation* is a finite sequence of concrete steps that begins with a computable function’s arguments and ends with its result or return value. For example,

$$\sum_{k=1}^{4} k = sum(1, 2, 3, 4)$$
$$= 1 + 2 + 3 + 4 = 3 + 3 + 4 = 6 + 4 = 10$$

The computation of this function took three steps, equal to the number of plus signs in the first expression with operators.

A computation is a particular series of operations on particular data values, whereas an algorithm is an abstract plan for computations on any of a wide variety of data values.

Whereas a function is a passive set of ordered pairs, a computation entails activity. A computer program or a subprogram computes a mathematical function. Later you will learn about Java “functions”. These are subprograms, not mathematical functions.

4. Recursively definable functions

Some functions in mathematics are not computable. For example, we can imagine a function whose argument is the executable file for a computer program, and whose return value is *true* or *false*. This function take the value *true* if the argument program ever goes into an infinite loop (“hangs” or “freezes”, in your experience). It returns *false* if the argument program always terminates. Such a function would be highly useful implemented as a computer program, because with such a program we could certify whether the software we are buying, selling, or using is reliable in a crucial way.

Unfortunately, this function is uncomputable, regardless of what processor the executable file is designed to run on. The program we would like to write, to compute this function and earn perhaps billions of dollars, cannot be written.

How can we recognize computable functions, so as to avoid taking on impossible software-development tasks and focus on work with fruitful prospects? It turns out that every computable function can be expressed according to a certain form, the *recurrence*.

Here is a simple recurrence that assumes the operation “+” is defined:

$$sum(a,b) = a + b$$

It is too trivial to discuss.

Here is a more interesting one, which assumes that division is defined:

$$quotient(a,b) = \begin{cases} a \div b & \text{if } b \neq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Notice that the definition on the right side of the recurrence is conditional; it depends on the value of *b*. This recurrence says that *quotient* is a function that has a return value of any arguments *a* and *b*, except where *b* is 0. To be undefined for certain argument values does not disqualify *quotient* from being a function.

Clearly both *sum* and *quotient* are computable, since we compute them all the time by hand or with calculators.

You may wonder what is recurring in the above recurrences. Nothing—but they follow a format in which something *may* recur, i.e., occur over and over.

Consider this:

base cases

$$sum(a,b) = \begin{cases} \text{undefined} & \text{if } a < 0 \\ b & \text{if } a = 0 \\ sum(a-1, b+1) & \text{otherwise} \end{cases}$$

recursive

Here we define the *sum* function conditionally on the value of a. In the case where a is negative, we choose to leave the return value undefined. (If we wrote a program to compute the function, it would show an error message.) If a is 0, our function expression $sum(a,b)$ takes the value b. Thus, for example, $sum(0,5) = 5$ by our definition. OK? So much for the easy part.

In the case where a is positive, our function definition reuses itself, or *recurses*: $sum(a-1,b+1)$ is returned. Let's step through the definition as it recurses with the argument values 2 and 1, to see how $sum(2,1)$ comes out:

$sum(2,1) = sum(2-1,1+1)$ by the recursive case

$= sum(1,2)$ applying the − and + operators

$= sum(1-1,2+1)$ by the recursive case

$= sum(0,3)$ applying the operators

$= 3$ by the base case

Our new definition of *sum* yields $2 + 1 = 3$.

You might notice that as we apply and re-apply our definition of the *sum* function, the arguments a and b take on successive values where a decreases with each step of recursion and b increases. Thus whatever initial values a and b take, on some eventual recursive invocation of the definition, a will take the value 0 and the second line of the recurrence (b, if $a = 0$), the base case, will apply.

5. Recursive functions are computable

Why would we ever want to define a function *sum* in this way, when "$sum(a,b) = a + b$" seems simpler? Because in practice the addition operator is more complex to apply than would appear on the surface. For us, addition requires at least an addition table and possibly repeated one-digit additions and carries. It is similar for a computer. But adding 1 to (incrementing) a value or subtracting 1 from (decrementing) it is simpler; quite simple, at the hardware level. Mathematically, we could call finding the successor or predecessor of a natural number a *primitive* operation, defined simply by the very nature of natural numbers. The natural numbers are defined as 0 and the successors of natural numbers, where each natural number has exactly one immediate successor.

What is significant is that any function that we can define by a recurrence is computable if every component on the right side of the recurrence is computable. In other words, even if we don't choose to use recursion in every algorithm we use in a computer program, nevertheless if it is possible to express as a recurrence the function our program is to compute (the problem it is to solve), we can be sure that it is also possible to write our desired program using some algorithm that computes the function. When we are solving a problem, it is useful to have a way of knowing that it can be solved.

Any loop can be specified using recursive pseudocode. Let's consider a very simple looping problem. What do you do to walk a distance of n steps? We could say, "Repeat n times: take one step," but another solution offers itself. Consider this algorithm:

```
Walk (num-steps)
If num-steps > 0
   Take a step
   Walk (num-steps – 1)
```

Here there are two possibilities. If the argument, or operand, or parameter, *num-steps*, is zero, then the algorithm will do nothing. But if it is one or higher, the algorithm reinvokes itself with a slightly smaller argument. That slightly smaller argument might be zero, or the reinvoked version of the algorithm might call itself with an argument of zero. Eventually, *num-steps* will get down to zero, and the recursion will stop.

Mathematics and computer science are in some ways two paradises, two playgrounds, for skeptical people. Every claim about functions, computable functions, and recursion made above can be proven, though we don't do so here. Though not every correct program can be proven correct mathematically, every such program can be rewritten in such a way that the result can be proven correct.

Every computer with the proper software-development environment is an inexpensive laboratory for testing problem solutions empirically. If you have a bright software idea, you can build and demonstrate a working prototype yourself.

As we shall see, a mathematical approach to software development has some practical advantages over the empirical testing approach. Software engineers make use of both.