

CS20: Notes on Recursive Functions

These notes introduce the concept of “primitive recursive” functions which are definable from a small set of trivial functions and operators. We shall also see how to extend this class of functions to obtain all Turing computable functions, which, according to the Church-Turing thesis, coincide with the effectively computable functions. All of the functions that we discuss here are “number-theoretic”; that is, each function returns a natural number and takes zero or more arguments that are each natural numbers. It is also possible to alter all of the definitions below to deal with functions over strings; however, the corresponding definitions are a bit more cumbersome, and they illustrate nothing more than the simple number-theoretic functions.

For more extensive discussions of recursive function theory see Machtey and Young [4], Krishnamurthy [3], Hennie [1], and Kleene [2]. Unfortunately, there is no universally accepted notation for the basic functions and operators of recursive function theory, so the notation differs widely among all of these works. Moreover, the conventions used in these notes (e.g. ZERO, SUCC, PROJ_kⁿ, COMP, and PREC) is highly non-standard, and differs from that found in all of the references cited above.

1 Primitive Recursive Functions

Definition 1 (Basic Functions and Operators) The number-theoretic *primitive recursive functions* are functions from $N \times \dots \times N$ to N that are constructed from three types of *basic functions*:

1. **zero:** defined by $\text{ZERO}(x) \equiv 0$
2. **successor:** defined by $\text{SUCC}(x) \equiv x + 1$
3. **projection:** defined by $\text{PROJ}_k^n(x_1, \dots, x_n) \equiv x_k$

and two *basic operators* for constructing new functions:

1. **composition:** denoted by $h^k = \text{COMP}(f^n, g_1^k, g_2^k, \dots, g_n^k)$
2. **primitive recursion:** denoted by $h^{n+1} = \text{PREC}(f^n, g^{n+2})$

Here all arguments are natural numbers and the superscripts on the functions f , g , and h denote their “arities”; that is, the number of arguments they take. The exact definitions of COMP and PREC are given below. Note that the projection “function” is actually an infinite family of functions, where $n \in N$ and $1 \leq k \leq n$. The function PROJ₀⁰ is defined as a special case, since the definition above breaks down when there are no arguments to return. Thus we define PROJ₀⁰ to be the null-ary function that always returns 0, which is essentially the constant 0.

Definition 2 (COMP) The *composition operator* constructs a new k -ary function h from an n -ary function f and k -ary functions g_1, g_2, \dots, g_n . Specifically, $\text{COMP}(f, g_1, \dots, g_n)$ returns a new k -ary function, h , defined by

$$h(x_1, \dots, x_k) = f(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k)).$$

Note that each argument of f is computed by a function of all k arguments. The g functions needn't be all distinct; in practice, they are often simple projections.

Definition 3 (PREC) The *primitive recursion operator* constructs a new $(n + 1)$ -ary function h from an n -ary function and an $(n + 2)$ -ary function. Specifically, if f is a function of n arguments and g is a function of $n + 2$ arguments, then $\text{PREC}(f, g)$ returns another function, h , of $n + 1$ arguments defined by

$$\begin{aligned} h(0, x_1, \dots, x_n) &= f(x_1, \dots, x_n) \\ h(m + 1, x_1, \dots, x_n) &= g(m, h(m, x_1, \dots, x_n), x_1, \dots, x_n) \end{aligned}$$

Notice that `PREC` produces a function h that is only recursive with respect to its *first* argument. In words, the function f provides the base case for h , when its first argument is zero. For all other values of the first argument h is allowed to depend on the predecessor of the first argument, the value it would return with a decremented first argument, and the rest of the arguments verbatim. This latter dependence is specified by the function g .

Definition 4 (Primitive Recursive Functions) The class of all functions that can be defined using finitely many applications of the basic operators, starting with the basic functions, is called the class of *primitive recursive functions*. This class of functions has three important properties:

1. All primitive recursive functions are total.
2. All primitive recursive functions are effectively computable.
3. Most common total functions are primitive recursive.

The first two properties are fairly obvious: each function is total and trivially computable, and both of the operators result in total functions if their arguments are total. Moreover, the effects of the operators are also trivially computable. The third property is neither precisely stated nor obviously true. We will simply demonstrate how to construct a number of common number-theoretic functions using the three basic functions and two basic operators; it should then become intuitively clear that the process could be continued to yield a vast number of other more complicated functions. In fact, it will take some effort to demonstrate the existence of total functions that are not primitive recursive.

2 Examples of Primitive Recursive Functions

In this section we demonstrate that a number of elementary number-theoretic functions are indeed primitive recursive. For each function, we first give its mathematical definition, then show some intermediate steps toward expressing it in terms of basic functions and operators, and finally we give its formal definition as a primitive recursive function. Once we have demonstrated that a function is primitive recursive, it is permissible to use it in constructing additional primitive recursive functions. For example, we use the `ADD` function in defining the `MULT` function.

The functions that we arrive at are in general grossly inefficient; for example, adding two natural numbers x and y is accomplished by adding 1 to x a total of y times. The situation is even worse for multiplication, and still worse for exponentiation. However, the objective is to understand what is ultimately computable using these functions without regard to efficiency. In other words, our immediate goal is to study *computability*, not *complexity*.

2.1 Predecessor

The “predecessor” function, denoted `PRED`, is almost the inverse of the successor function; the only exception is zero, whose predecessor is defined to be zero. Thus,

$$\text{PRED}(x) = \begin{cases} x - 1 & \text{when } x > 0 \\ 0 & \text{when } x = 0 \end{cases} .$$

The predecessor function is easily defined in terms of primitive recursion. To see this, we start by observing the two cases

$$\begin{aligned} \text{PRED}(0) &= 0 \\ \text{PRED}(m + 1) &= m, \end{aligned}$$

which completely define the function. We then express each of these cases as a function of the appropriate arity; since PRED is a unary function, we need functions f and g which are null-ary and binary, respectively. In particular, we require f and g such that

$$\begin{aligned}\text{PRED}(0) &= f() \\ \text{PRED}(m+1) &= g(m, \text{PRED}(m)),\end{aligned}$$

according to the definition of primitive recursion. In this case, both functions correspond to simple projections; $f = \text{PROJ}_0^0$, and $g = \text{PROJ}_1^2$. Thus, we have

$$\text{PRED} = \text{PREC}(\text{PROJ}_0^0, \text{PROJ}_1^2).$$

Note that there are no function arguments appearing in this final expression. Rather, the function PRED is expressed entirely in terms of combinations of other functions.

2.2 Addition

We can similarly define addition using a binary function. Mathematically, we wish to define the function ADD so that

$$\text{ADD}(x, y) = x + y.$$

Again, we first observe that the two cases

$$\begin{aligned}\text{ADD}(0, y) &= y \\ \text{ADD}(m+1, y) &= \text{ADD}(m, y) + 1,\end{aligned}$$

completely define the ADD function. Note that the second case uses a “recursive” reference to itself with the first argument decremented, which is something we get for free from primitive recursion. Since the successor function can increment by one, we can easily find primitive recursive functions f and g , which are unary and ternary, respectively, such that

$$\begin{aligned}\text{ADD}(0, y) &= f(y) \\ \text{ADD}(m+1, y) &= g(m, \text{ADD}(m, y), y).\end{aligned}$$

In particular, $f = \text{PROJ}_1^1$ and $g = \text{COMP}(\text{SUCC}, \text{PROJ}_2^3)$. That is, the function g simply applies the successor function to its second argument; the other two arguments are ignored. We conclude that

$$\text{ADD} = \text{PREC}(\text{PROJ}_1^1, \text{COMP}(\text{SUCC}, \text{PROJ}_2^3)).$$

2.3 Multiplication

The definition of multiplication is very similar to addition. We define the function MULT so that

$$\text{MULT}(x, y) = x * y.$$

Observing that

$$\begin{aligned}\text{MULT}(0, y) &= 0 \\ \text{MULT}(m+1, y) &= \text{MULT}(m, y) + y,\end{aligned}$$

we now define a unary function, f , and a ternary function, g , such that

$$\begin{aligned}\text{MULT}(0, y) &= f(y) \\ \text{MULT}(m+1, y) &= g(m, \text{MULT}(m, y), y).\end{aligned}$$

It follows that $f = \text{ZERO}$ and $g = \text{COMP}(\text{ADD}, \text{PROJ}_2^3, \text{PROJ}_3^3)$. Finally, we have

$$\text{MULT} = \text{PREC}(\text{ZERO}, \text{COMP}(\text{ADD}, \text{PROJ}_2^3, \text{PROJ}_3^3)).$$

2.4 Factorial

To define the factorial function $\text{FACT}(x) = x!$, observe that

$$\begin{aligned}\text{FACT}(0) &= 1 \\ \text{FACT}(m+1) &= (m+1) * \text{FACT}(m).\end{aligned}$$

we now define a null-ary function, f , and a binary function, g , such that

$$\begin{aligned}\text{FACT}(0) &= f() \\ \text{FACT}(m+1) &= g(m, \text{FACT}(m)).\end{aligned}$$

The function f is trivial to construct by composing the successor function and a projection, via $\text{COMP}(\text{SUCC}, \text{PROJ}_0^0)$. Unfortunately, the g function requires $m+1$, not m which is passed to it. This requires an additional composition with the successor function. Thus, g must compute the product of $\text{COMP}(\text{SUCC}, \text{PROJ}_1^2)$ and PROJ_2^2 . We conclude that

$$\text{FACT} = \text{PREC}(\text{COMP}(\text{SUCC}, \text{PROJ}_0^0), \text{COMP}(\text{MULT}, \text{COMP}(\text{SUCC}, \text{PROJ}_1^2), \text{PROJ}_2^2)).$$

2.5 Proper Subtraction

Since subtraction is not properly defined from $N \times N$ to N , we introduce a restricted form of subtraction called *proper subtraction*:

$$\text{PSUB}(x, y) = \begin{cases} x - y & \text{when } x \geq y \\ 0 & \text{when } x < y. \end{cases}$$

To define PSUB we really wish to apply recursion to the second argument, not the first. In particular, we wish to use the facts that

$$\begin{aligned}\text{PSUB}(x, 0) &= x, \\ \text{PSUB}(x, y+1) &= \text{PRED}(\text{PSUB}(x, y)),\end{aligned}$$

where the recursive call decrements the second argument rather than the first. This is the first example in which the definition of primitive recursion appears to be limiting. However, we can easily swap the arguments by composing the function with projection operators. In this way we can apply recursion to any argument we choose, simply by moving it to the first position. Applying this trick to PSUB, we have

$$\text{PSUB} = \text{COMP}(\text{PREC}(\text{PROJ}_1^1, \text{COMP}(\text{PRED}, \text{PROJ}_2^3)), \text{PROJ}_2^2, \text{PROJ}_1^2).$$

2.6 Other Examples

Here we list a few more examples of primitive recursive functions with “hints” as to how they are defined. That is, rather than completely formal constructions we employ several obvious conventions. For example, the COMP operator is eliminated by simply substituting one function into another. It should be clear how to convert these descriptions into their formal equivalents.

$$\begin{aligned}\text{ABSDIFF}(x, y) &= \text{ADD}(\text{PSUB}(x, y), \text{PSUB}(y, x)) \\ \text{ZERO?}(x) &= \text{PREC}(\text{SUCC}(\text{PROJ}_0^0), \text{ZERO}(\text{PROJ}_1^2)) \\ \text{EQUAL?}(x, y) &= \text{ZERO?}(\text{ABSDIFF}(x, y)) \\ \text{NOT}(x) &= \text{PREC}(\text{PROJ}_0^0, \text{SUCC}(\text{ZERO}(\text{PROJ}_1^2))) \\ \text{BRANCH}(p, x, y) &= \text{PREC}(\text{PROJ}_2^2, \text{PROJ}_3^4)\end{aligned}$$

Here we have used the convention of ending predicate names with “?”; in this context a predicate is a function that returns 1 or 0, indicating “true” or “false”. The function ABSDIFF returns the absolute difference of its arguments, $|x - y|$. The BRANCH function returns the second argument if the first is true (non-zero), and the third otherwise. Note that ZERO?, NOT, and BRANCH all use primitive recursion in a trivial way. That is, they simply use the fact that primitive recursion distinguishes between the first argument being zero or non-zero, and ignore the recursion altogether.

3 Total Functions that are Not Primitive Recursive

Thus far we have managed to show that many simple number-theoretic functions are primitive recursive. However, by a counting argument it is clear that there must exist total number-theoretic functions that are *not* primitive recursive. This follows from the fact that there are only countably many primitive recursive functions, whereas there are uncountably many functions from N to N . Yet this still leaves open the question as to whether there are any *effectively computable* total number-theoretic functions that are not primitive recursive. The answer is yes, although it is somewhat challenging to construct an example.

The first example of a total function that is effectively computable but not primitive recursive is Ackermann’s function. Clearly, Ackermann’s function is effectively computable, since it is an easy matter to write a program that computes it. However, demonstrating that it is not primitive recursive is rather tricky. The result hinges on the fact that primitive recursive functions exhibit growth rates that are limited by the number of applications of COMP and PREC that were used to define them. It can be shown that Ackermann’s function exceeds this rate of growth for any finite number of basic operators; thus it cannot be primitive recursive. See Hennie [1, pages 231–234] for the details of this proof.

Another way to construct such a function is through diagonalization. First, observe that it is possible to *effectively enumerate* all unary primitive recursive functions; that is, we can compute a sequence of primitive recursive functions f_0, f_1, f_2, \dots such that every conceivable unary primitive recursive function can be found in the list; moreover, given any n , there is an effective procedure for finding f_n . Given such an enumeration, we can then construct the function

$$g(x) = f_x(x) + 1,$$

Then g is total and effectively computable, yet not primitive recursive. The fact that it is not primitive recursive follows from diagonalization; that is... (*to be added after homework 3 is collected*).

4 Bounded Minimization

It is often convenient to use a third type of operator, known as *bounded minimization*, in constructing primitive recursive functions.

Definition 5 (BMIN) The *bounded minimization operator* constructs a new n -ary function from an n -ary function and an $(n + 1)$ -ary function. Specifically, if f is a function of n arguments and g is a function of $n + 1$ arguments, then $\text{BMIN}(f, g)$ returns another function, h , of n arguments defined by

$$h(x_1, \dots, x_n) = \min_{y=0}^{f(x_1, \dots, x_n)} [g(y, x_1, \dots, x_n) \neq 0]$$

If $g(y, x_1, \dots, x_n)$ is zero for all $0 \leq y \leq f(x_1, \dots, x_n)$, then h returns the value $f(x_1, \dots, x_n) + 1$. In words, the function h returns the smallest value of y (up to and including the computed upper limit) such that $g(y, x_1, \dots, x_n)$ is nonzero.

When the bounded minimization operator BMIN is applied to functions f and g that are both primitive recursive, the result is another primitive recursive function. That is, the resulting function could have been defined using only the basic functions and operators. This follows from the fact that if

$$h^n = \text{BMIN}(f^n, g^{n+1}),$$

then the function h could equally well be defined as

$$h(x_1, \dots, x_n) = \sum_{y=0}^{f(x_1, \dots, x_n)} \left(\prod_{i=0}^y \text{ZERO}?(g(i, x_1, \dots, x_n)) \right),$$

which is easily verified to be primitive recursive whenever both f and g are. Thus, the bounded minimization operator is inessential; it does not permit us to define any function that we would not otherwise be able to define.

5 Unbounded Minimization

A slight variation of bounded minimization, known as *unbounded minimization*, results in a new type of operator that *cannot* always be reduced to the basic functions and operators.

Definition 6 (UMIN) The *unbounded minimization operator* constructs a new n -ary function from an $(n+1)$ -ary function. Specifically, if g is a function of $n+1$ arguments, then $\text{UMIN}(g)$ returns another function, h , of n arguments defined by

$$h(x_1, \dots, x_n) = \min_{y=0} [g(y, x_1, \dots, x_n) \neq 0]$$

In words, the function h returns the smallest value of y such that $g(y, x_1, \dots, x_n)$ is nonzero. If g is zero for all $y \in N$, then this function *diverges*; that is, it is undefined. Consequently, unbounded minimization can create functions that are non-total, even if the function it is applied to is total. The set of primitive recursive functions is therefore not closed under the UMIN operator.

The important thing to understand about unbounded minimization is that it permits us to define non-total functions; that is, functions that are undefined, or *divergent*, at some values in the domain. A trivial example of a non-total function that can be defined using unbounded minimization is

$$\text{UMIN}(\text{ZERO}),$$

which is a unary function that diverges on all input values.

6 Partial Recursive Functions

The introduction of unbounded minimization leads to a new and very important class of functions, which we call the *partial recursive* functions.

Definition 7 (Partial Recursive Functions) The class of all functions that can be defined using finitely many applications of the basic operators *plus unbounded minimization*, starting with the basic functions, is called the class of *partial recursive functions*. This class of functions has three important properties:

1. Some partial recursive functions are non-total (i.e. they diverge on some input).
2. All partial recursive functions are effectively computable.
3. All known effectively computable functions are partial recursive.

According to the Church-Turing thesis, the set of effectively computable functions is precisely the set of partial recursive functions. In fact, partial recursive functions are just one of many different abstractions that, according to the Church-Turing thesis, exactly capture the notion of effective computability.

Finally, we define one more important class of functions. We define the class of *recursive functions* to be those partial recursive functions that happen to be total. Thus

$$\text{recursive functions} = \text{total functions} \cap \text{partial recursive functions}$$

An example of a function that is “recursive”, but not “primitive recursive” is Ackermann’s function. Although Ackermann’s function requires unbounded minimization to define it, it is nevertheless a total function. We will explore this class of functions further when we study Turing’s notion of computability.

References

- [1] Fred Hennie. *Introduction to Computability*. Addison-Wesley, Reading, Massachusetts, 1977.
- [2] Stephen C. Kleene. Origins of recursive function theory. In *20th Annual Symposium on the Foundations of Computer Science*, pages 371–382, San Juan, Puerto Rico, October 1979.
- [3] E. V. Krishnamurthy. *Introductory Theory of Computer Science*. Springer-Verlag, New York, 1983.
- [4] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*. Theory of Computation Series. North Holland, New York, 1978.