Theory of Formal Languages (Primitive Recursive Functions)

Lecture 8: Jan. 18, 2019

Prof. K.R. Chowdhary *: Professor of CS*

8.1 Primitive Recursive Functions

The primitive recursive functions are among the number-theoretic functions which are functions from the natural numbers (non negative integers) $0, 1, 2, ...$ to the natural numbers. These functions take n arguments for some natural number n and are called n-ary. These functions demonstrate the computations in the most fundamental form

The primitive recursive (PR) functions are defined using primitive recursion and composition as central operations, and they are proper subsets of recursive functions. The recursive functions are also called *computable functions*. In *computability theory*, the *PR* functions are a class of functions, which form an important building block on the way to fully formalize the *theory of computability*. These functions are also useful in *proof theory* to generate proofs of algorithms.

Most of the functions we study in number theory are primitive recursive. For example, addition, subtraction, multiplication, exponentiation, and *nth* factorial are all primitive recursive. These operations are basis for almost all arithmetic. Thus, it is difficult to think of a functions, which is not primitive recursive.

One method of characterizing a class of functions is to take, as members of the class, all functions obtainable by certain kind of *recursive definition*. A recursive definition for a function is, roughly speaking, a definition wherein values of the function for given arguments are directly related to values of the same function for "simpler" arguments or to values of "simpler" functions. The "simpler" here means, for example, constant functions, as the simplest of all. The recursive definitions can often be made to serve as algorithms.

The recursive definitions are familiar in mathematics. For instance, the function f defined by

$$
\begin{aligned}
&f(0) = 1,\\
&f(1) = 1,\\
&f(x+2) = f(x+1) + f(x),
\end{aligned}
$$

gives the Fibonacci sequence: $1, 1, 2, 3, 5, 8, \ldots$.

8.1.1 Building Blocks for Primitive Recursion

The class of primitive recursive functions is defined in terms of *base functions*: the zero function (which takes no argument and return 0), successor function, and the identity projection functions, which makes use of construction technique of function composition (substitution) and recursion.

The identity functions are typically used to return individual arguments of the function being constructed. Some examples are:

$$u_1^1(x) = x,$$
$$u_1^2(x, y) = x,$$
$$u_2^2(x, y) = y,$$
$$u_1^3(x, y, z) = x.$$

Because writing them every time becomes cumbersome, so for the sake of brevity, we can refer to arguments directly, by writing definitions like $f(x, y) = x + y$, instead of $f(x, y) = u_1^2(x, y) + u_2^2(x, y)$. However, we know that writing arguments directly, means an implicit application of one of the identity functions. Similarly, we should strictly form all constants by composition of the successor and zero functions, which again is cumbersome process. Hence we prefer to write 3 in place of $S(S(S(0)))$, where S is successor function.

Let $\Sigma = \{a_1, ..., a_N\}$ is the alphabet set, the generalized base functions (*axioms*) over the alphabet Σ can be defined as follows:

1. Erase function E, defined such that $E(w) = \epsilon$, for all $w \in \Sigma^*$;
2. For every $j, 1 \leq j \leq N$, the j-successor function S_j, which returns the successor of its argument, defined such that $S_j(w) = wa_j$, for all $w \in \Sigma^*$, and $a_j \in \Sigma$.
3. The projection function P_i^n, defined such that $P_i^n(w_1, ..., w_n) = w_i$, and for all $w_1, ..., w_n \in \Sigma^*$.

Note that P_1^1 is identity function on Σ^*. The projection functions can be used to permute the arguments of another function.

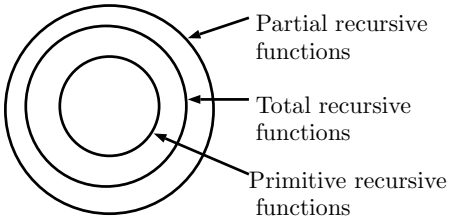
The primitive recursive functions are the basic functions and those which can be obtained from the basic functions by applying the operators of zero[1], successor, composition, and projection, finite number of times. The functions of addition, multiplication, exponential, and superexpontial can be defined by primitive recursion.

An important property of the primitive recursive (PR) functions is that they are *recursively enumerable* subset of all *total recursive* functions. This means that there is single computable function $f(e, m)$ such that:

- for every primitive recursive g, there is an e such that $g(n) = f(e, n)$ for all n,
- for every e, the function $h(n)$ defined as $\lambda_n f(e, n)$ is primitive recursive.

[1] *Erase function* is for symbol elements, the zero function is used for integer arguments.

A *partial recursive function* is one that can be computed by a *Turing Machine*. A *total recursive* function is partial recursive function, which is defined for every input. Every primitive function is total recursive, but not all total recursive functions are primitive recursive. An Ackermann's function $A(m, n)$ is well known example of a total recursive function that is not primitive recursive.



.

Figure 8.1: Functions hierarchy

8.1.2 Composition

The composition is used in the normal way to supply the result of one function as an argument to another, as in $f(x) = g(h(x))$. Let there be an alphabet $\Sigma = \{a_1, ..., a_N\}$. The composition of any function g of m arguments

$$g : \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \longrightarrow \Sigma^*, \tag{8.1}$$

and any m functions $h_1 \dots h_m$ each having n arguments,

$$h_i : \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \longrightarrow \Sigma^*, \tag{8.2}$$

is a function, say $f = g \circ h$, expressed as

$$f : \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \longrightarrow \Sigma^*. \tag{8.3}$$

The mapping shows that each function returns a single value. The function f is denoted as $f = g \circ (h_1, \dots, h_m)$, such that

$$f(w_1, \dots, w_n) = g(h_1(w_1, \dots, w_n), \dots, h_m(w_1, \dots, w_n)), \tag{8.4}$$

for all $w_1, \dots, w_n \in \Sigma^*$. This is because the g will behave as a projection function. As an example, $f = g \circ (P_2^2, P_1^2)$ such that $f(w_1, w_2) = g(w_2, w_1)$.

The projection functions can be used to avoid the apparent rigidity in terms of the *arity* of the PR functions. Using the composition, with various projection functions, it is possible to pass a subset of the arguments of one function to another function. For example, consider that there are two functions g and h which are PR and each has *arity-2*. Then,

$f(a, b, c) = g(h(c, a), h(a, b))$

is primitive recursive. A formal definition of above using projections function can be given to compute f using g and h.

$$f(a, b, c) = g(h(P_3^3(a, b, c), P_1^3(a, b, c)), h(P_1^3(a, b, c), P_2^3(a, b, c))).$$

8.1.3 Primitive Recursion

An m-argument function f may be recursively defined by specifying two other functions:

1. function g of $m-1$ arguments which calculates f's value when its (f's) first argument is null,
2. function h of $m+1$ arguments, which calculates f's new value when its (f's) first argument is increased by one from previous.

Considering that alphabet is $\Sigma = \{a_1, ..., a_N\}$, a primitive recursive can be defined as follows.

$$g : \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m\text{-}1} \longrightarrow \Sigma^*, \tag{8.5}$$

where $m \geq 2$, and any n functions $h_1 \ldots h_n$ each having $m+1$ arguments,

$$h_i : \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \longrightarrow \Sigma^*, \tag{8.6}$$

then the composition of g and $h_1 \ldots h_n$ using primitive recursion gives the function f as

$$f : \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \longrightarrow \Sigma^*. \tag{8.7}$$

Further, the following recursions are satisfied:

$$f(\epsilon, w_2, \ldots, w_m) = g(w_2, \ldots, w_m),$$
$$f(ua_1, w_2, \ldots, w_m) = h_1(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m),$$
$$\cdots = \ldots,$$

$$f(ua_N, w_2, \ldots, w_m) = h_n(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m), \tag{8.8}$$

for all $u, w_1, \ldots, w_n \in \Sigma^*$.

When $m = 1$, for some fixed $w \in \Sigma^*$, we have

$$f(\epsilon) = w,$$

$$f(ua_1) = h_1(u, f(u)),$$

$$\cdots = \ldots,$$

$$f(ua_N) = h_n(u, f(u)),$$

for all $u \in \Sigma^*$.

For numerical functions, (i.e., $\Sigma = \{a_1\}$), the scheme of primitive recursive functions is simpler, as follows:

$$\begin{aligned} f(0, x_2, \ldots, x_m) &= g(x_2, \ldots, x_m), \\ f(x+1, x_2, \ldots, x_m) &= h_1(x, f(x, x_2, \ldots, x_m), x_2, \ldots, x_m) \end{aligned} \tag{8.9}$$

for all $x, x_2, \ldots, x_m \in \mathbb{N}$, where $\mathbb{N}$ is natural numbers set. Note that in the second argument, the x ultimately becomes 0, the $f(x, x_2, ..., x_m)$ finally becomes $f(0, x_2, ..., x_m)$, which is calculated as $g(x_2, ..., x_m)$. By this, the first, and second arguments of h are computed.

Consider that, it is required to compute the sum of integers 1 to n, i.e.,

$$sumint(x) = \sum_{n=0}^{x} n, \tag{8.10}$$

which adds all integers from 0, up to x.

The sum of all integers up to and including x may be specified recursively with two functions: $g() = 0$ and $h(x, y) = x + y + 1$, which allows us to calculate that

$sumint(0) = g() = 0$,
$sumint(1) = h(0, sumint(0)) = h(0, 0) = 1$,
$sumint(2) = h(1, sumint(1)) = h(1, 1) = 3$,
$sumint(3) = h(2, sumint(2)) = h(2, 3) = 6$,
$sumint(4) = h(3, sumint(3)) = h(3, 6) = 10$,
and so on. Note that, the function *sumint* is "add" function here.

8.2 PR functions are computable

We want to show that every PR function is mechanically computable. Given the general strategy just described, it is enough to show that in three statements:

1. The initial functions are computable.
2. If f is defined by composition from computable functions g and h, then f is also computable.
3. If f is defined by primitive recursion from the computable functions g and h, then f is also computable.

The statement '1.' above is trivial: the initial functions S, Z, and P_i^k are effectively computable by a simple algorithm; and statement '2.' – the composition of two computable functions g and h is computable (we just feed the output from whatever algorithmic routine evaluates g as input, into the routine that evaluates h).

To illustrate statement '3.' above, return to factorial function, defined as:

$$0! = 1$$
$$(Sy)! = y! \times Sy$$

The first clause gives the value of the function for the argument 0; then we repeatedly use the second recursion clause to calculate the function's value for $S0$, then for $SS0$, $SSS0$, etc. So the definition encapsulates an algorithm for calculating the function's value for any number, and corresponds exactly to a certain simple kind of computer routine. And obviously the argument generalizes.

8.3 PR functions by 'for'loop

Compare the PR definition of the factorial with the following schematic program:

1. $fact = 1$; {initialize $fact$ by 0!}
2. *For* $y = 0$ *to* $n - 1$
3. $\quad fact = (fact \times Sy)$
4. *Loop*

Here *fact* is a memory register that we initialize with 0!. Then the program enters a loop. The crucial thing about executing a 'for'loop is that the total number of iterations to be run through is fixed in advance: we number the loops from 0, and in executing the loop, we increment the counter by one on each cycle. So in this case, on loop number y the program replaces the value in the register with Sy times the previous value (we will assume the computer already knows how to find the successor of y and do that multiplication). When

the program exits the loop after a total of n iterations, the value in the register fact will be $n!$ [petsmth10].

More generally, for any one-place function f defined by recursion in terms of g and the computable function h, the same program structure always does the trick for calculating $f(n)$. Now compare,

$$\begin{aligned} f(0) &= g \\ f(Sy) &= h(y, f(y)) \end{aligned} \tag{8.11}$$

with the corresponding program:

1. $func = g$
2. $For\ y = 0\ to\ n - 1$
3. $\quad func = h(y, func)$
4. *Loop*

Note that, so long as h is (already) computable, the value of $f(n)$ will be computable using this 'for' loop that terminates with the required value in the register $func$. Similarly, of course, for many-place functions. For example, the value of the two-place function defined by,

$$\begin{aligned} f(x, 0) &= g(x) \\ f(x, Sy) &= h(x, y, f(x, y)) \end{aligned} \tag{8.12}$$

is calculated by the algorithmic program

1. $func = g(m)$
2. $For\ y = 0\ to\ n - 1$
3. $\quad func = h(m, y, func)$
4. *Loop*

which gives the value for $f(m, n)$ so long as g and h are computable.

Now, our mini-program for the factorial calls the multiplication function which can itself be computed by a similar 'for' loop (invoking addition). And addition can in turn be computed by another 'for' loop (invoking the successor). So reflecting the downward chain of recursive definitions:

factorial $\Rightarrow$ *multiplication* $\Rightarrow$ *addition* $\Rightarrow$ *successor*

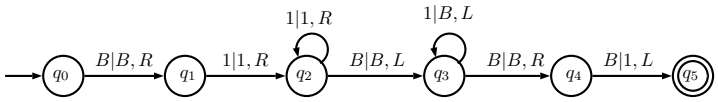


Figure 8.2: Turing machine for zero function

There is a program for the factorial containing nested 'for' loops, which ultimately calls the primitive operation of incrementing the contents of a register by one (or other operations like setting a register to zero, corresponding to the zero function, or copying the contents of a register, corresponding to an identity function).

The above point obviously generalizes, giving us: Primitive recursive functions are effectively computable by a series of (possibly nested) 'for'loops.

The converse is also true. Take a 'for' loop which computes the value of a function f for given arguments, a loop which calls on two prior routines, one which computes a function g (used to set the value of f with some key argument set to zero), the other which computes a function h (which is used on each loop to fix the next value of f as that argument is incremented). This plainly corresponds to a definition by recursion of f in terms of g and h. And generalizing: if a function can be computed by a program using just 'for' loops as its main programming structure – with the program's built in' functions all being PR – then the newly defined function will also be primitive recursive. This gives us a way of convincing ourselves that a new function is PR: sketch out a routine for computing it and check that it can all be done with a succession of (possibly nested) 'for' loops which only invoke already known PR. functions: then the new function will be primitive recursive.

8.4 Some Primitive Recursive functions

Following are some examples of primitive recursive functions. Most number-theoretic functions, which can be defined using recursion on a single variable are primitive recursive.

8.4.1 Zero function

The zero function Z is the function,

$$Z(x) = 0, \tag{8.13}$$

that makes contents of Turing machine as zero. The machine that computes zero function, converts all 1s into Bs, except the first 1', i.e., $Z(1.1^n) = 1$.

Note that, in the Fig. 8.2, the Turing machine scans the entire input, and while returning back to original position it converts all 1s to 0s, and then first digit is made 1 to indicate that value is zero. This could also be done by leaving 1st 1 in left most position unchanged, and remaining all the 1s are converted into Bs in forward move of tape-head.

The two different mechanisms computing the same function indicates the difference between function and algorithm – the two machines are two algorithms to computes the same function zero.

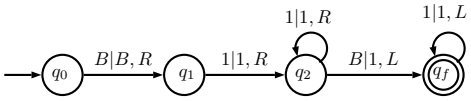


Figure 8.3: Turing machine for successor function

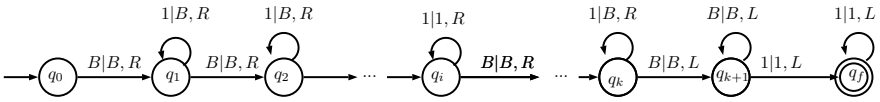


Figure 8.4: Turing machine for Projection function

8.4.2 Successor Function

The successor function S is the function,

$$S(x) = x + 1, \tag{8.14}$$

that converts converts the contents of Turing machine from 1^k to 1^{k+1}. Hence, the machine that computes the successor function, add a 1 immediately after the input to its right.

8.4.3 Projection Function

A k-variable projection function $P_i^{(k)}$ returns ith argument out of total k arguments, i.e.,

$$P_i^{(k)}(n_1, n_2, ..., n_i, ..., n_k) = n_i. \tag{8.15}$$

8.4.4 Predecessor Function

The predecessor function is defined as,

$$pred(n) = \begin{cases} 0 & if \quad n = 0 \\ n - 1 & otherwise. \end{cases} \tag{8.16}$$

It is computing D machine (i.e., decrement), which removes the right most 1 for the input greater than zero, else it keeps input unchanged.

The *predecessor* function acts as the opposite of the successor function and is recursively defined by the following rules:

$$\begin{aligned} pred(0) &= 0, \\ pred(y + 1) &= y. \end{aligned} \tag{8.17}$$

These rules can be converted into a more formal definition by primitive recursion:

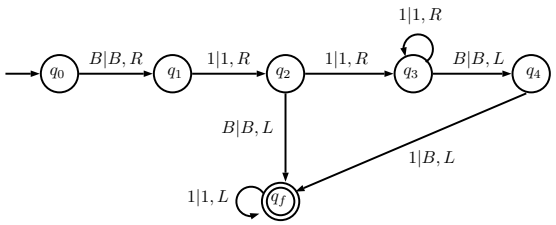


Figure 8.5: Turing machine for Predecessor function

$$pred(0) = 0,$$
$$pred(S(y)) = P_2^2(pred(y), y). \tag{8.18}$$

8.4.5 Addition Function

The addition function can be recursively defined with following rules.

$$add(x, 0) = g(x) = x$$

$$\begin{aligned} add(x, y+1) &= h(x, y, add(x, y)) \\ &= add(x, y) + 1. \end{aligned}$$

The $add(x, y) + 1$ is nothing but $S(add(x, y))$.

The function *add* computes the sum of two natural numbers. The definition of $add(x, 0)$ indicates that the sum of any number with zero is number itself. The recursion step defines the sum of x and $y+1$ as the sum of x and y, which is the result of the addition of previous value of the recursive variable, incremented by 1.

An application of above is:

$$\begin{aligned} add(3, 2) &= add(3+1)+1 \\ &= S(add(3+1)) \\ &= S(S(add(3+0)+1)) \\ &= S(S(S((3+0))) \\ &= S(S(S(3))) \\ &= 5. \end{aligned}$$

In order to fit in strict primitive recursive definition, we define the *Add* function as follows.

$$add(x, 0) = P_1^{(1)}(x)$$
$$add(S(y), x) = S(P_1^{(3)}(add((y, x), y, x)))$$

Here $P_1^{(3)}$ is the projection function that takes three arguments and returns first. $P_1^{(1)}$ is simply identity function. Its inclusion is required by the definition of the primitive recursive function operator above, and it plays the role of f in equation 8.9. The composition of S and $P_1^{(3)}$, which is primitive recursive, plays the role of g.

8.4.6 Subtraction function

Because primitive recursive functions use natural numbers rather than integers, and the natural numbers are not closed under subtraction, a limited subtraction function is studied in this context. This limited subtraction function $sub(a, b)$ returns $b - a$ if it is non negative and returns 0 otherwise.

The limited subtraction function is definable from the predecessor function in a manner analogous to the way addition is defined from successor:

$$sub(0, x) = P_1^{(1)}(x),$$
$$sub(S(y), x) = pred(P_1^{(3)}(sub(y, x), y, x)). \tag{8.19}$$

Here $sub(a, b)$ corresponds to $b - a$; for the sake of simplicity, the order of the arguments has been switched from the standard definition to fit the requirements of primitive recursion. This could easily be rectified using composition with suitable projections.

8.4.7 Multiplication Function

The multiplication is implemented using primitive recursive functions of addition, subtraction, and predecessor.

$$1 * x = x$$
$$x * (y + 1) = x * pred(y + 1) + x \tag{8.20}$$

Alternatively, it can be also specified as,

$$mult(0, x) = 0,$$
$$mult(y + 1, x) = add(mult(y, x), x). \tag{8.21}$$

8.4.8 Exponentiation function

The exponentiation function can be implemented by multiplication.

$$x^0 = 1,$$
$$x^{y+1} = x^y * x \tag{8.22}$$

Alternatively, we can have

$rexpt(0, x) = 1,$
$rexp(y + 1, x) = mult(rexp(y, x), x),$
$rexp(y, x) = rexp \circ (P_2^2, P_1^2).$
$supexp(0, x) = 1,$
$supexp(y + 1, x) = exp(x, supexp(y, x)).$