Primitive Recursive Functions as Description of For-Loops

How to describe a for-loop in precise terms: an example. Let us start with a simple program for computing the power a^b:

```
power = 1;
for(int i = 1; i <= b; i++)
  power = power * a;
```

This is not a precise mathematical notation, because:

- in math, the variable is assumed to have the same value in different parts of the equation, but
- here, e.g., in the statement `power = power * a`, the two occurrences of the variable `power` means different things: the statement means that the variable `power` is assigned a *new* value: the product of the *old* value and a.

To make the description mathematically precise, we must thus explicitly indicate the iteration number, i.e.,

- to describe the original value of this variable by `power(0)`;
- to describe the value of this variable after the first iteration by `power(1)`;
- to describe the value of this variable after the second iteration by `power(2)`, etc.

In these terms, the first statement of the above program

```
power = 1;
```

takes the form

```
power(0) = 1
```

and the statement inside the loop – which says that the next value of the variable `power` is equal to the previous value times a – takes the following form:

```
power(m + 1) = power(m) * a
```

This is not yet a fully correct mathematical description, since the value of the variable `power` depends not only on the number of iteration, it also depends on a. So, to be completely mathematically correct, we should explicitly describe this dependence, i.e., use `power(a, m)` instead of `power(m)`. Now, we get a fully mathematically correct description of the above for-loop-using program:

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

How to describe a for-loop in precise terms: general case. In a general for-loop, we have some parameters; let us denote them by $\overline{n} = (n_1, \ldots, n_k)$. There is also some variable – we will denote it by h – to which:

- first, we assign some initial value – the value which, in general, depends on the parameters $n_1, \ldots, n_k$; we will denote this value by

$$f(\overline{n}) = f(n_1, \ldots, n_k),$$

- and then, on each iteration m of the for-loop, we use the parameters $n_1, \ldots, n_k$, the number m of the iteration, and the previous value of the variable h compute the new value of h; this new value depends on $\overline{n}$, m, and h; we will denote it by $g(\overline{n}, m, h) = g(n_1, \ldots, n_k, m, h)$.

In terms of these notations, the corresponding for-loop takes the following form:

```
h = f(n1, ..., nk);
for(int i = 1, i <= m, i++)
  {h = g(n1, ..., nk, i, h)};
```

The value h depends on the parameters n and on the number of iteration m: $h = h(\overline{n}, m) = h(n_1, \ldots, n_k, m)$. Thus, we get the following general description of a for-loop:

$$h(\overline{n}, 0) = f(\overline{n});$$

$$h(\overline{n}, m + 1) = g(\overline{n}, m, h(\overline{n}, m)),$$

where $\overline{n} \stackrel{\text{def}}{=} (n_1, \ldots, n_k)$.

This leads to the following formal definition of a for-loop – which is called *primitive recursion.*

What is primitive recursion: a definition. *Suppose that we have two functions $f(n_1, \ldots, n_k)$ and $g(n_1, \ldots, n_k, m, h)$. By the result $PR(f, g)$ of applying* primitive recursion *to f and g, we mean the function $h(n_1, \ldots, n_k, m)$ which is defined by the following formulas:*

$$h(n_1, \ldots, n_k, 0) = f(n_1, \ldots, n_k);$$

$$h(n_1, \ldots, n_k, m + 1) = g(n_1, \ldots, n_k, m, h(n_1, \ldots, n_k, m)).$$

What is a primitive recursive function: a definition. *A function is called* primitive recursive *(p.r., for short) if it can be obtained from 0, σ, and π_i^k by using composition $\circ$ and primitive recursion PR.*

Simple examples of p.r. functions.

- A constant function 0 is a p.r. function, according to this definition.
- A constant function 1 is a p.r. function: indeed, $1 = \sigma(0)$, i.e., in terms of the mathematical notation for composition, 1 is $\sigma \circ 0$.
- A constant function 2 is $\sigma(\sigma(0))$, i.e., $\sigma \circ \sigma \circ 0$.

First non-trivial example: addition is primitive recursive. In the previous lecture, we showed that if we have σ (i.e., the ++ operation), then we can use the for-loop to compute addition. In effect, we thus proved that addition is primitive recursive. Let us describe this proof in precise terms.

The above argument started by noticing that

$$a + b = a + 1 + 1 + \ldots + 1 \ (b \text{ times}).$$

We start with a, and we add one b times. This adding one b times can be described, in terms of the for-loop, as follows:

```
int c = a;
for(int i = 1; i <= b; i++)
  {c = c + 1;}
```

The value c depends on the value a and on the number of iterations m: $c = c(a, m)$. If we explicitly introduce this dependence, we arrive at the following formulas:

$$c(a, 0) = a;$$

$$c(a, m + 1) = c(a, m) + 1.$$

How can we match these formulas with the general description of a primitive recursive function? In general, we define a function $g(n_1, \ldots, n_k, m)$ of $k + 1$ variables. In our case, we have a function $c(a, m)$ of two variables a and m. Thus, to match, we must have $k + 1 = 2$, i.e., $k = 1$. For $k = 1$, the general definition of a primitive recursion takes the following form:

$$h(n_1, 0) = f(n_1);$$

$$h(n_1, m + 1) = g(n_1, m, h(n_1, m)).$$

To match these formulas with what we have in the case of addition, we first need to rename the function and the parameters. For addition, the resulting function is called c, and the parameter is called a, but in the general definition, the function is denoted by h, and the parameter is denoted by n_1. So, to get the match, let us rename c by h, and a by n_1. After this renaming, $c(a, 0) = a$

becomes $h(n_1, 0) = n_1$, and the description of the addition's for-loop takes the form

$$h(n_1, 0) = n_1;$$

$$h(n_1, m+1) = h(n_1, m) + 1.$$

Now, the left-hand sides are exactly the same as in the definition of a p.r. function, so all that remains is to match the right-hand sides. First, we need to match $f(n_1)$ with n_1. This is easy: this function f is simply π_1^1, a function that takes the tuple $\overline{n} = (n_1)$ consisting of only one element n_1 and returns this same element $\pi_1^1(n_1) = n_1$.

Let us match the right-hand sides of the second equality, i.e., let us match $g(n_1, m, h(n_1, m))$ and $h(n_1, m) + 1$. The expression $h(n_1, m) + 1$ means that out of the three possible inputs n_1, m, and $h(n_1, m)$ of the function g, we ignore the first and second ones and only take into account the third one of three. The third one of the tuple of three is π_3^3: $\pi_3^3(n_1, m, h(n_1, m)) = h(n_1, m)$. To this third value $h(n_1, m)$, we add 1, i.e., we apply the operation σ. Thus, here, $g = \sigma \circ \pi_3^3$.

Now that we know both $f = \pi_1^1$ and $g = \sigma \circ \pi_3^3$, we can describe addition $PR(f, g)$ as follows:

$$add = PR(\pi_1^1, \sigma \circ \pi_3^3).$$

This shows that addition is indeed a primitive recursive function.

How to go back from the p.r. expression to the Java code: on the example of addition. The above p.r. expression is nothing else but the description of the original for-loop in precise terms. Based on such an expression, we can always reconstruct the original for-loop. Let us illustrate it on the example of the expression describing addition.

In general, when we write $h = PR(f, g)$, the function f is a function of k variables. In our case, $f = \pi_1^1$ is a function of one variable – since, in general, $\pi_i^k(n_1, \ldots, n_k)$ is a function of k variables. Thus, here, $k = 1$. For $k = 1$, the general formula for primitive recursion takes the form

$$h(n_1, 0) = f(n_1);$$

$$h(n_1, m+1) = g(n_1, m, h(n_1, m)).$$

In our case, $f = \pi_1^1$, so $f(n_1) = \pi_1^1(n_1) = n_1$. Also, $g = \sigma \circ \pi_3^3$, so

$$g(n_1, m, h(n_1, m)) = \sigma(\pi_3^3(n_1, m, h(n_1, m))) = \sigma(h(n_1, m)) = h(n_1, m) + 1.$$

Thus, for our functions f and g, the general formulas of primitive recursion take the following form:

$$h(n_1, 0) = n_1;$$

$$h(n_1, m+1) = h(n_1, m) + 1.$$

To describe this as a program, we need to take into account that the parameters n_1 and m were added to h to make the formulas mathematically correct; there are no such parameters in a program. In the program:

- the first formula – which now becomes $h = n1$ – describes what is done before the loop starts, and
- the second formula – which now becomes $h = h + 1$ – describes what is happening inside the loop, on each of its iterations.

Thus, we arrive at the following program:

```
h = n1;
for(int i = 1; i <= m; i++)
  {h = h + 1;}
```

This is exactly our original program for addition – the only difference is that the variable is now called h and the parameter is now called $n1$.

Multiplication is primitive recursive. The product $a \cdot b$ is nothing else but a added to itself b times:

$$a \cdot b = a + a + \ldots + a \ (b \text{ times}).$$

To get a correct result the first time we add a, we need to start with 0. This b times adding the value a can be described, in terms of the for-loop, as follows:

```
int c = 0;
for(int i = 1; i <= b; i++)
  {c = c + a;}
```

The value c depends on the value a and on the number of iterations m: $c = c(a, m)$. If we explicitly introduce this dependence, we arrive at the following formulas:

$$c(a, 0) = 0;$$

$$c(a, m + 1) = c(a, m) + a.$$

How can we match it with the general description of a primitive recursive function? In general, we define a function $h(n_1, \ldots, n_k, m)$ of $k+1$ variables. In our case, we have a function $c(a, m)$ of two variables a and m. Thus, to match, we must have $k + 1 = 2$, i.e., $k = 1$. For $k = 1$, the general definition of a primitive recursive function takes the form

$$h(n_1, 0) = f(n_1);$$

$$h(n_1, m + 1) = g(n_1, m, h(n_1, m)).$$

To match with what we have, we first need to rename the function and the parameters. In our case, the function is named c, and the parameter is named a, but in the general definition, the function is denoted h, and the parameter is denoted n_1. So, to get the match, let us rename c by h, and a by n_1. After this renaming, $c(a, 0) = 0$ becomes $h(n_1, 0) = 0$, and the description of the multiplication's for-loop takes the form

$$h(n_1, 0) = 0;$$

$$h(n_1, m+1) = h(n_1, m) + n_1.$$

Now, the left-hand sides are exactly the same as in the definition of a p.r. function, so all that remains is to match the right-hand sides. First, we need to match $f(n_1)$ with 0. This is easy: this function f is simply 0, a function that always returns 0.

Let us match the right-hand sides of the second equality, i.e., let us match $g(n_1, m, h(n_1, m))$ and $h(n_1, m) + n_1$. The expression $h(n_1, m) + n_1$ means that out of the three possible inputs n_1, m, and $h(n_1, m)$ of the function g, we ignore the second one and add the first and the third ones, i.e., the values

$$\pi_3^3(n_1, m, h(n_1, m)) = h(n_1, m) \text{ and } \pi_1^3(n_1, m, h(n_1, m)) = n_1.$$

Thus, here, $g = add(\pi_3^3, \pi_1^3)$.

Now that we know both $f = 0$ and $g = add(\pi_3^3, \pi_1^3)$, we can describe multiplication $PR(f, g)$ as follows:

$$mult = PR(0, add(\pi_3^3, \pi_1^3)).$$

In particular, substituting the known p.r. expression for addition

$$add = PR(\pi_1^1, \sigma \circ \pi_3^3)$$

into this formula, we conclude that

$$mult = PR(0, (PR(\pi_1^1, \sigma \circ \pi_3^3))(\pi_3^3, \pi_1^3)).$$

This shows that multiplication is indeed a primitive recursive function.

How to go back from the p.r. expression to the Java code: on the example of multiplication. In general, when we write $h = PR(f, g)$, the function f is a function of k variables, and the function g is a function of $k + 2$ variables. In our case, $add(\pi_3^3, \pi_1^3)$ is a function of 3 variables, so we need to take k for which $k + 2 = 3$. Thus, here, $k = 1$.

For $k = 1$, the general formula for primitive recursion take the form

$$h(n_1, 0) = f(n_1);$$

$$h(n_1, m+1) = g(n_1, m, h(n_1, m)).$$

In our case, $f = 0$ and $g = add(\pi_3^3, \pi_1^3)$, so

$$g(n_1, m, h(n_1, m)) = add(\pi_3^3(n_1, m, h(n_1, m)), \pi_1^3(n_1, m, h(n_1, m))) =$$

$$add(h(n_1, m), n_1).$$

Thus, for our functions f and g, the general formulas of primitive recursion take the form

$$h(n_1, 0) = 0;$$

$$h(n_1, m+1) = add(h(n_1, m), n_1).$$

To describe this as a program, we need to take into account that the parameters n_1 and m were added to h to make the formulas mathematically correct; there are no such parameters in a program. In the program:

- the first formula – which now becomes $h = n1$ – describes what is done before the loop starts, and
- the second formula – which now becomes $h = add(h, n1)$ – describes what is happening inside the loop, on each of its iterations.

Thus, we arrive at the following program:

```
h = 0;
for(int i = 1; i <= m; i++)
  {h = add(h, n1);}
```

This is exactly our original program for addition – the only difference is that the variable is now called h and the parameter is now called $n1$.

We can substitute, into this program, the above program for computing add; then, we get the following program with two embedded for-loops:

```
h = 0;
for(int i = 1; i <= m; i++)
  {h = h;
   for(int j = 1; j <= n1; j++)
     {h = h + 1;}}
```

Comment. Notice that the statement `h = h;` was added just because we were following the general algorithm. Of course, it is not needed, so if we care about efficiency, it has to be eliminated. But, honestly, computing $a + b$ as a for-loop with b iterations is clearly not a very efficient algorithm anyway. At this point, we are not (yet) taking efficiency into account, we are simply trying to analyze what is computable and what is not computable.

Power function is primitive recursive. Let us now go back to the power function a^b with which we started this lecture. The power a^b is nothing else but a multiplied by itself b times:

$$a^b = a \cdot a \cdot \ldots \cdot a \ (b \text{ times}).$$

To get a correct result the first time we multiply by a, we need to start with 1. This b times multiplying by the value a can be described, in terms of the for-loop, as follows:

```
int c = 1;
for(int i = 1; i <= b; i++)
  {c = c * a;}
```

The value c depends on the value a and on the number of iterations m: $c = c(a, m)$. If we explicitly introduce this dependence, we arrive at the following formulas:

$$c(a, 0) = 1;$$

$$c(a, m + 1) = c(a, m) \cdot a.$$

How can we match it with the general description of a primitive recursive function? In general, we define a function $g(n_1, \ldots, n_k, m)$ of $k+1$ variables. In our case, we have a function $c(a, m)$ of two variables a and m. Thus, to match, we must have $k+1=2$, i.e., $k=1$. For $k=1$, the general definition of a primitive recursive function takes the form

$$h(n_1, 0) = f(n_1);$$

$$h(n_1, m+1) = g(n_1, m, h(n_1, m)).$$

To match with what we have, we first need to rename the function and the parameters. In our case, the function is named c, and the parameter is named a, but in the general definition, the function is denoted h, and the parameter is denoted n_1. So, to get the match, let us rename c by h, and a by n_1. After this renaming, $c(a, 0) = 1$ becomes $h(n_1, 0) = 1$, and the description of the power's for-loop takes the form

$$h(n_1, 0) = 1;$$

$$h(n_1, m+1) = h(n_1, m) \cdot n_1.$$

Now, the left-hand sides are exactly the same as in the definition of a p.r. function, so all that remains is to match the right-hand sides. First, we need to match $f(n_1)$ with 1. This is easy: this function f is simply $1 = \sigma(0)$, a function that always returns 1.

Let us match the right-hand sides of the second equality, i.e., let us match $g(n_1, m, h(n_1, m))$ and $h(n_1, m) \cdot n_1$. The expression $h(n_1, m) \cdot n_1$ means that out of the three possible inputs n_1, m, and $h(n_1, m)$ of the function g, we ignore the second one and multiply the first and the third ones, i.e., the values

$$\pi^3_3(n_1, m, h(n_1, m)) = h(n_1, m) \text{ and } \pi^3_1(n_1, m, h(n_1, m)) = n_1.$$

Thus, here, $g = mult(\pi^3_3, \pi^3_1)$.

Now that we know both $f = \sigma \circ 0$ and $g = mult(\pi^3_3, pi^3_1)$, we can describe $PR(f, g)$ as follows:

$$power = PR(\sigma \circ 0, mult(\pi^3_3, \pi^3_1)).$$

This shows that the power function is indeed a primitive recursive function.

Comment. In principle, we can plug in here the above expression for *mult* and get a complex expression in terms of 0, σ, π^k_i, composition, and primitive recursion.

Factorial is primitive recursive. The factorial $fact(m) = m!$ is defined as the product of all the numbers from 1 to m, i.e., as

$$m! = 1 \cdot 2 \cdot \ldots \cdot m.$$

To get the correct result, we need to start with 1. Computation of this formula can be implemented by the following for-loop:

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

The value c depends only on the number of the iteration m, so we get $c = c(m)$. The first formula becomes $c(0) = 1$. With the second formula, we need to be careful: the value $c(m+1)$ is, by definition, the value after the $(m+1)$-st iteration. This value is obtained from the previous value $c(m)$ by multiplying by $i = m + 1$. So, we get:

$$c(0) = 1;$$

$$c(m+1) = c(m) \cdot (m+1).$$

Here, we are defining a function of 1 variable, and the general primitive recursion defined a function of $k+1$ variables. Thus, here, $k+1 = 1$, so $k = 0$, and the general expression for primitive recursion takes the form:

$$h(0) = f;$$

$$h(m+1) = g(m, h(m)).$$

To match, we rename the above two equations and get

$$h(0) = 1;$$

$$h(m+1) = h(m) \cdot (m+1).$$

Thus here, $f = 1 = \sigma \circ 0$ and $g = mult(\pi_2^2, \sigma \circ \pi_1^2)$, so

$$fact = PR(\sigma \circ 0, mult(\pi_2^2, \sigma \circ \pi_1^2)).$$

Function $F(n) = 2n + 3$ is primitive recursive. Indeed, we know that multiplication and addition are p.r., and 2 and 3 are p.r. So, the above function is simply a composition of p.r. functions – and thus it is itself primitive recursive.

Comment. We could describe this function as a for-loop, but in this case, it is easier to just take into account that it is a composition of functions for which we have already proved that they are primitive recursive.