Not All Computable Functions Are Primitive Recursive

What we had so far. So far, we have shown that many computable functions are primitive recursive – i.e., that they can be computed by using only the for-loops. So, you may be under the impression that all computable functions are primitive recursive. However, it has been proven that this impressions is wrong.

For-loops are not always sufficient. It turns out that there are computable functions which cannot be computed by using only the for-loops – i.e., which are not primitive recursive.

What we will do. We will provide two proofs of this statement.

- The first proof will be described in detail, and it will be more understandable, but the resulting example will be very artificial.
- The second example of a computable function which is not primitive recursive is about a function which makes sense – but the proof itself will be more complicated, so much more complicated that we will only describe the main ideas of this second proof.

Theorem. *There exists a computable function which is not primitive recursive.*

Proof.

First part of the proof. Let us first describe how we can assign, to each p.r. function, a natural number that we will call this function's *pr-code*. This will be done in several steps.

- By definition, a p.r. function is obtained from 0, σ, and π_i^k by using composition $\circ$ and primitive recursion PR. Thus, each such function can be described by an expression containing these symbols and parentheses (and). For example, addition is described as $PR(0, \sigma \circ \pi_3^3)$.
- Symbols 0 and PR are ASCII symbols, which means that they can be directly typed on a usual computer keyboard. However, symbols σ, π_i^k, and $\circ$ are not. To describe them in ASCII, we can use, e.g., LaTeX, a language specifically designed by renowned computer scientist Donald Knuth to translate mathematical symbols into ASCII. In this language, σ, π_i^k, and $\circ$ are described as follows:

```
\sigma,  \pi^k_i, \circ
```

- After we use this translation, we get a sequence of ASCII symbols. For example, the expression corresponding to addition takes the form

  ```
  PR(0,\sigma\circ\pi^3_3)
  ```

  According to ASCII, each ASCII symbol is represented in a computer as a sequence of 0s and 1s. For example, P is represented as $50_{16} = 0101\ 0000$, R is represented as $52_{16} = 0101\ 0010$, etc., so the expression for addition takes the form

  $$0101\ 0000\ 0101\ 0010\ \ldots$$

- Finally, we append 1 in front of the resulting sequence of 0s and 1s, and interpret the resulting binary sequence as a binary number. For example, for addition, we will get

  $$1\ 0101\ 0000\ 0101\ 0010\ \ldots$$

  This number is what we will call a pr-code of the original p.r. function.

Comment. We append 1 in front to make sure that we will be able to uniquely reconstruct the original binary sequence from this number. If we did not add 1, this would not always be possible. For example, different sequences 0, 00, 000 would correspond to the same natural number 0. So, based on the value 0, we would not be able to tell which of the three binary sequences was used.

On the other hand, if we append 1 to these three binary sequences, we get three different binary numbers: $10_2 = 2_{10}$, $100_2 = 4_{10}$, and $1000_2 = 8_{10}$.

Second part of the proof: an important lemma. To prove our result, we will need the following lemma (= auxiliary theorem):

Lemma. *There exists an algorithm that, given a natural number c:*

- *checks whether c is a pr-code of some p.r. function, and*
- *if it is, produced an executable file for computing this function; we will denote this file by f_c.*

How we can prove this lemma. A pr-code was obtained from the original expression as follows

$$\text{expression} \stackrel{\text{LaTeX}}{\longrightarrow} \text{ASCII expression} \stackrel{\text{ASCII}}{\longrightarrow} \text{binary sequence} \stackrel{\text{append 1}}{\longrightarrow} \text{pr-code}.$$

Thus, to get back from the pr-code to the original p.r. function, we need to follow these steps in reverse order:

- First, we strip off the first 1 from the *natural* binary description of the given natural number n, i.e., from a description in which we skip all leading 0s; for example, $n = 5$ is represented as 101, not as 0101. As a result, we get a binary sequence. This step is not possible only in one case: when $n = 0$; in this case, we stop this algorithm and return the answer that $n = 0$ is not a pr-code of any p.r. function.

- Second, we check whether the resulting binary sequence is indeed a sequence of valid ASCII symbols. This is what computers do all the time.

- Third, we use the LaTeX compiler to check that the corresponding ASCII sequence is a valid LaTeX expression. This is what LaTeX compilers do all the time. If it is a valid LaTeX expression, LaTeX translates in into a sequence of mathematical symbols.

- Finally, we check whether the resulting sequence of mathematical symbols is syntactically correct; e.g.,

  $$PR(0,$$

  is not syntactically correct: we have an opening parenthesis but not a closing one, and there is nothing after the comma. This is what compilers do all the time. If it is syntactically correct, then we can use the same ideas that we used before to translate this expression into the Java code: PR corresponds to the for-loop. etc.

Final step of the proof. Let us now consider the function $f(c)$ which is defined as follows:

- if c is a pr-code of a p.r. function, we return $f(c) = f_c(c) + 1$;

- otherwise, if c is *not* a pr-code of a p.r. function, we return $f(c) = 0$.

Let is prove that this function is computable but not primitive recursive.

Proving that the function $f(c)$ is computable. This proof is straightforward: we just show how this function can be computed. Suppose that we are given a natural number c. Then, to compute $f(c)$, we do the following:

- First, we apply the algorithm A whose existence is proven by the lemma. This algorithm either tells us that c is not a pr-code – in which case we return $f(c) = 0$ – or generates the file f_c.

- If c is a pr-code, we apply the executable file f_c to the number c, resulting in the value $f_c(c)$, and then add 1 to the result.

This can be described as follows:

$$\xrightarrow{c} \text{is } c \text{ a pr-code? } \xrightarrow{\text{yes}} \text{apply } f_c \text{ to } c \xrightarrow{f_c(c)} \text{add 1} \xrightarrow{f_c(c)+1}$$
$$\downarrow \text{no}$$
$$0$$

Proving that the function $f(c)$ is not primitive recursive. We will prove this by contradiction. Let us assume that the function $f(c)$ *is* primitive recursive. Let c_0 denote its pr-code. Then, by definition of f_c as a code that computes the original p.r. function, for every possible input n, we have

$$f_{c_0}(n) = f(n).$$

In particular, for $n = c_0$, we have

$$f_{c_0}(c_0) = f(c_0).$$

On the other hand, by definition of the function $f(c)$, since c_0 is a pr-code, we get

$$f(c_0) = f_{c_0}(c_0) + 1.$$

By comparing these two equalities, we conclude that $f_{c_0}(c_0) = f_{c_0}(c_0)+1$, i.e., if we subtract $f_{c_0}(c_0)$ from both sides, that $0 = 1$. This is clearly a contradiction.

The only assumption that we made to get this contradiction is that the function $f(c)$ is primitive recursive. Thus, this assumption is wrong, and the above defined function $f(c)$ is not primitive recursive.

Conclusion. We have come up with a function $f(c)$ which is computable but not primitive recursive. Thus, the theorem is proven.

Comment. Instead of adding 1 to $f_c(c)$, we could add 2, 3, or any other non-zero number or any computable non-zero expression, for the resulting function, the same proof will show that it is not primitive recursive.

A graphical representation of the final part of the proof. The final part of the proof is known as a *diagonal construction*. To understand why, let us describe it graphically. To make this description clear, instead of the above definition of the pr-code, let us use a different definition, in which $f_0(n) \equiv 0$, 1 is not a pr-code, $f_2(n) = n + 1$, $f_3(n) = n^2$, $f_4(n) = 2n$, etc. Let us place the values $f_c(n)$ of each function f_c in row number c; for values which are not pr-codes, we will place a minus sign instead of a value:

$f_c \backslash n$	0	1	2	3	4	...
$f_0(n)$	0	0	0	0	0	...
$f_1(n)$	–	–	–	–	–	...
$f_2(n)$	1	2	3	4	5	...
$f_3(n)$	0	1	4	9	16	...
$f_4(n)$	0	2	4	6	8	...
...	...	...	...	...	...	...

To describe the value of $f(c)$, we need to take into account only the elements $f_c(c)$, i.e., the elements – underlined and in bold in the following table – which lie on the diagonal of this matrix:

$f_c \backslash n$	0	1	2	3	4	...
$f_0(n)$	**0**	0	0	0	0	...
$f_1(n)$	–	–	–	–	–	...
$f_2(n)$	1	2	**3**	4	5	...
$f_3(n)$	0	1	4	**9**	16	...
$f_4(n)$	0	2	4	6	**8**	...
...	...	...	...	...	...	...
$f(c)$	1	0	4	10	9	...

Towards the second proof. When we described which functions are primitive recursive, we started with the function $f(n) = n + 1$. We iterated this adding-1 operation b times, and we got addition:

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

Then, we iterated addition b times, and we got multiplication:

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

After that, we iterated multiplication b times, and got the power function:

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

A natural idea is to continue in the same way, and iterate the power function:

$$a^{\left(a^{\left(\ldots^{(a^a)}\right)}\right)} \ (b \text{ times}).$$

The resulting function – denoted by ${}^b a$ – was introduced by Archimedes when he tried to count how many grains of sand are there in the world – this number was a symbol of something very large in ancient times; for example, the Bible promises to Abraham that his descendants will be as numerous as grains of sand. (By the way, the Bible also promises that they will be as numerous as stars in the sky, which makes less sense, since there are only a few thousand visible stars, and the ancient folks knew no others.)

And why stop there? We can iterate the new operation ${}^b a$ several (b) times, etc.

Let us describe this general procedure in precise terms. We start with the function $F_0(a, b) = a + 1$. We will call this a function of the 0-th order. Once we have a function $F_k(a, b)$ of k-th order, we can describe the function $F_{k+1}(a, b)$ of the next order as the application of F_k to a b times, i.e., as:

$$F_{k+1}(a, b + 1) = F_k(a, F_{k+1}(a, b)).$$

This way:

- $F_1(a,b) = a + b$ is addition,
- $F_2(a,b) = a \cdot b$ is multiplication,
- $F_3(a,b) = a^b$ is the power function,
- $F_4(a,b) = {}^b a$, etc.

Now, we finally get to construct a naturally defined function which is computable but not primitive recursive.

Ackermann function. Let us apply a diagonal construction and consider a new function $A(n) \stackrel{\text{def}}{=} F_n(n,n)$. This function was introduced by a German mathematician Wilhelm Ackermann in the 1920s; it is thus known as the Ackermann function.

This function grows very quickly. Indeed, let us compute the first few values of the Ackermann function.

- For $n = 0$, we have $A(0) = F_0(0,0) = 0 + 1 = 1$.
- For $n = 1$, we have $A(1) = F_1(1,1) = 1 + 1 = 2$.
- For $n = 2$, we have $A(2) = F_2(2,2) = 2 \cdot 2 = 4$.
- For $n = 3$, we have $A(3) = F_3(3,3) = 3^3 = 27$. So far, the values are reasonable, but wait until we get to $n = 4$.
- For $n = 4$, we have

  $$A(4) = {}^4 4 = 4^{\left(4^{\left(4^4\right)}\right)}.$$

  Here, $4^4 = 4 \cdot 4 \cdot 4 \cdot 4 = 256$, then

  $$4^{\left(4^4\right)} = 4^{256} = \left(2^2\right)^{256} = 2^{2 \cdot 256} = 2^{512}.$$

  Since $2^{10} = 1024 \approx 1000 = 10^3$, we have

  $$4^{\left(4^4\right)} = 2^{512} = \left(2^{10}\right)^{51.2} \approx \left(10^3\right)^{51.2} = 10^{153.6},$$

  i.e., a decimal number with 154 digits. This is already an astronomical number, but this is not all! To compute $A(4)$, we need to raise 4 to this humongous power:

  $$A(4) = {}^4 4 = 4^{\left(4^{\left(4^4\right)}\right)} \approx 4^{\left(10^{153.6}\right)}.$$

  This number is really huge!

Theorem. *The Ackermann function $A(n)$ is computable but not primitive recursive.*

Proof. That the Ackermann function is computable is clear: to compute each value $A(n) = F_n(n, n)$, we need n embedded for-loops. Let us prove that $A(n)$ is not primitive recursive.

Indeed, primitive recursive means using for-loops. For each such function, we can count how many for-loops it uses – i.e., since for-loops are represented by primitive recursion, how many PR symbols it uses.

Let us analyze how fast functions using k for-loops can grow. Let us first start with the case $k = 0$, when a function does not use any for-loops at all. This means that this function is obtained from 0, σ, and π_i^k by composition. Projection π_i^k does not increase the values. The only function that increases the values is the function $\sigma(n) = n+1$. In each composition, there are finitely many such functions. Each of them increase the original value by 1, so at most, the value will be increased by a constant a – the number of σ's in the description of a given function. In other words, for any such function $f(n)$, we have

$$f(n) \le n + a \text{ for some constant } a.$$

Let us now consider the case $k = 1$, when we have a single for-loop. Inside the loop, we have a function of order $k = 0$, so in each iteration, at most, we add a constant a. The number of such iterations is also bounded by $n + a$. If we add $n + a$ times the same constant a, we get

$$f(n) \le n + (n + a) \cdot a = n \cdot (a + 1) + c,$$

for some constant c. So, for large n, for which $n \ge c$, we have $f(n) \le n \cdot a'$, where $a' = a + 2$. Thus, for all such functions, for sufficiently large n, we have

$$f(n) \le n \cdot a \text{ for some constant } a.$$

Similarly, for $k = 2$, we have $\le n \cdot a$ iterations, in each of which the original value n is multiplied by no more than constant, so the result is bounded by multiplying by this constant n times – i.e., by the power:

$$f(n) \le a^n.$$

In general, by induction, we can prove that for any p.r. function that uses k for-loops, for sufficiently large n, we get

$$f(n) \le F_{k+1}(a, n) \text{ for some constant } a.$$

If the Ackermann function was primitive recursive, then it would have some finite number k of PR expressions and thus, it will be limited by $F_{k+1}(a, n)$:

$$A(n) \le F_{k+1}(a, n).$$

However, as one can check, each function $F_{k+1}(a, b)$ grows faster than $F_k(a, b)$: multiplication grows faster than addition, power function grows faster than multiplication, etc. Thus, for $n > k + 1$, the value $A(n) = F_n(n, n)$ grows much faster than $F_{k+1}(a, n)$, so the above inequality is not possible.

This proves that the Ackermann function is not primitive recursive.