

## Solutions to Test 1

**Problem 1.** Translate, step-by-step, the following for-loop into a primitive recursive expression:

```
int x = p * q;
for (int i = 1; i <= r; i++)
    {x = x + q;}
```

You can use  $\text{add}(\cdot, \cdot)$  (sum) and  $\text{mult}(\cdot, \cdot)$  (product) in this expression.

What is the value of this function when  $p = 1$ ,  $q = 2$ , and  $r = 2$ ?

**Solution.** The value  $x$  depends on  $p$ ,  $q$ , and on the number of the iteration, so we have  $x(p, q, m)$ . For  $m = 0$ , before the iteration starts, we have

$$x(p, q, 0) = \text{mult}(p, q).$$

At each iteration step, we add  $q$  to the previous value:

$$x(p, q, m + 1) = \text{add}(x(p, q, m), q).$$

In general, primitive recursion defines a function of  $k + 1$  variables. In our case, we have a function of 3 variables, so  $k + 1 = 3$  and  $k = 2$ . For  $k = 2$ , the general primitive recursion takes the form:

$$h(n_1, n_2, 0) = f(n_1, n_2);$$

$$h(n_1, n_2, m + 1) = g(n_1, n_2, m, h(n_1, n_2, m)).$$

To match, we rename  $x$  into  $h$ ,  $p$  into  $n_1$ , and  $q$  into  $n_2$ . As a result, we get:

$$h(n_1, n_2, 0) = \text{mult}(n_1, n_2);$$

$$h(n_1, n_2, m + 1) = \text{add}(h(n_1, n_2, m), n_2).$$

Thus, here,  $f = \text{mult}(\pi_1^2, \pi_2^2)$ ,  $g = \text{add}(\pi_4^4, \pi_2^4)$  and thus,

$$f = PR(\text{mult}(\pi_1^2, \pi_2^2), \text{add}(\pi_4^4, \pi_2^4)).$$

For  $p = 1$  and  $q = 2$ :

- first we assign  $x = p \cdot q = 1 \cdot 2 = 2$ ;

- then, we assign  $i = 1$ ; since  $i \leq r = 2$ , we go inside the loop and assign, to  $x$ , the new value  $x + q = 2 + 2 = 4$ ;
- after that, we increase  $i$  by 1, so now  $i = 2$ ; we still have  $i \leq r$ , so we again go into the loop and assign, to  $x$ , the new value  $x + q = 4 + 2 = 6$ ;
- now, we again increase  $i$  by 1; now  $i = 3$ ; we no longer have  $i \leq r$ , so we get out of the loop.

The resulting value is 6.

**Problem 2.** Translate, step-by-step, the following primitive recursive function into a for-loop:

$$F = \sigma(PR(mult(\pi_2^2, \sigma(0)), add(\sigma(\pi_1^4), \pi_3^4))).$$

For this function  $F$ , what is the value  $F(2, 0, 1)$ ?

**Solution.** Here,  $F = \sigma(PR(\dots))$ , so to the result  $h$  of primitive recursion, we should add 1:  $F(n_1, \dots, m) = h(n_1, \dots, m) + 1$ .

In the general expression for primitive recursion  $h = PR(f, g)$ ,  $f$  is the function of  $k$  variables. Here,  $\pi_2^2$  is a function of two variables, so  $f = mult(\pi_2^2, \sigma(0))$  is also a function of two variables, and thus  $k = 2$ . For  $k = 2$ , the general expression for primitive recursion takes the form:

$$h(n_1, n_2, 0) = f(n_1, n_2);$$

$$h(n_1, n_2, m + 1) = g(n_1, n_2, m, h(n_1, n_2, m)).$$

In our case,

$$h(n_1, n_2, 0) = mult(\pi_2^2(n_1, n_2), \sigma(0)) = mult(n_2, 1) = n_2 \cdot 1;$$

$$\begin{aligned} h(n_1, n_2, m + 1) &= \\ add(\sigma(\pi_1^4(n_1, n_2, m, h(n_1, n_2, m))), \pi_3^4(n_1, n_2, m, h(n_1, n_2, m))) &= \\ add(n_1 + 1, m) &= n_1 + 1 + m. \end{aligned}$$

Thus, in the loop, for each iteration  $i = m + 1$ , we add  $m = i - 1$  to the value  $n_1$ .

To the result  $h(n_1, n_2, m)$  of primitive recursion, we apply  $\sigma$ , which means that at the end, we should add 1. So, we get the following code:

```
h = n2 * 1;
for(int i = 1; i <= m; i++)
    {h = n1 + 1 + (i - 1);}
F = h + 1;
```

From the above formulas, for  $n_1 = 2$  and  $n_2 = 0$ , we first get:

$$h(2, 0, 0) = 0 \cdot 1 = 0.$$

To get the value  $h(2, 0, 1)$ , we need to take  $m + 1 = 1$ , i.e.,  $m = 0$ , then we get:

$$h(2, 0, 1) = 2 + 1 = 3.$$

Finally, we add 1 and get  $F(2, 0, 1) = h(2, 0, 1) + 1 = 3 + 1 = 4$ .

**Problem 3-4.** Prove, from scratch, that the function  $f(p, q) = p \% (q + 2)$  is primitive recursive. Start with the definitions of a primitive recursive function, and use only this definition in your proof – do not simply mention results that we proved in class, prove them.

**Solution.** To prove that the desired function is primitive recursive (p.r.), we will show:

- that addition is p.r., and
- that remainder is p.r.

1. Addition is p.r. since the function

$$add(a, b) = a + b = a + 1 + \dots + 1 \text{ (} b \text{ times)}$$

can be represented as

$$\begin{aligned} add(a, 0) &= a; \\ add(a, b + 1) &= add(a, b) + 1. \end{aligned}$$

2. The remainder function  $rem(a, b) = b \% a$  can be represented as follows:

$$\begin{aligned} rem(a, 0) &= 0; \\ rem(a, b + 1) &= \text{if } (rem(a, b) + 1 < b) \text{ then } (rem(a, b) + 1) \text{ else } 0. \end{aligned}$$

To show that this expression is p.r., we need to show:

- that  $<$  is p.r., and
- that the if-then-else construction is p.r.

2.1. Let us first show that the relation  $r < b$  is p.r. Indeed, the condition  $r < b$  is equivalent to  $b \dot{-} r > 0$ , where:

- $b \dot{-} r = b - r$  if  $b > r$  and
- $b \dot{-} r = 0$  otherwise.

So, to prove that  $r < b$  is p.r., it is sufficient to prove that the subtraction  $sub(b, r) = b \dot{-} r$  is p.r., and that the relation  $a > 0$  is p.r.

2.1.1. Subtraction can be represented as

$$\begin{aligned} sub(b, 0) &= b; \\ sub(b, r + 1) &= sub(b, r) \dot{-} 1. \end{aligned}$$

So, to show that subtraction is p.r., it is sufficient to prove the function  $prev(n) = n \dot{-} 1$  is p.r.

2.1.2. Indeed, the function  $prev(n)$  can be represented as follows:

$$\begin{aligned}prev(0) &= 0; \\prev(n + 1) &= n.\end{aligned}$$

2.1.3. The relation  $a > 0$  describing that  $a$  is positive – we will denote it  $pos(a)$  – is p.r.: indeed, 0 is not positive, but every number of the type  $m + 1$  is:

$$\begin{aligned}pos(0) &= 0; \\pos(m + 1) &= 1.\end{aligned}$$

2.2. Let us now show that the if-the-else construction is p.r. Indeed, the if-then-else construction can be represented as

$$\textit{if } (P(\bar{n})) \textit{ then } f(\bar{n}) \textit{ else } g(\bar{n}) = P(\bar{n}) \cdot f(\bar{n}) + (1 \div P(\bar{n})) \cdot g(\bar{n}).$$

So, to complete the proof that remainder is p.r., it is sufficient to prove that multiplication is primitive recursive.

2.2.1 Indeed, multiplication can be represented as

$$mult(a, b) = a \cdot b = a + \dots + a \text{ (} b \text{ times),}$$

thus

$$\begin{aligned}mult(a, 0) &= 0; \\mult(a, b + a) &= mult(a, b) + a.\end{aligned}$$

So, remainder is p.r.

3. Since addition and remainder are p.r., the desired function – which is their composition – is also p.r.

**Problem 5.** Prove that the following function  $f(p, q)$  is  $\mu$ -recursive:  $f(p, q) = p/q$  when each of the values  $p$  and  $q$  is either 1 or 3, and  $f(p, q)$  is undefined for other pairs  $(p, q)$ .

**Solution.**  $f = \mu m.((p = 1 \vee p = 3) \& (q = 1 \vee q = 3) \& (m = p/q))$ .

When each of the values  $p$  and  $q$  is equal to 1 or 3, then the smallest  $m$  for which the corresponding property

$$(p = 1 \vee p = 3) \& (q = 1 \vee q = 3) \& (m = p/q)$$

is satisfied is exactly  $m = p/q$ . For all other pairs  $(p, q)$ , this property is never satisfied, so the function is undefined.

**Problem 6.** Translate the following  $\mu$ -recursive expression into a while-loop:

$$f(a, b) = \mu m. (m \cdot a \geq b).$$

For this function  $f$ , what is the value of  $f(2, 4)$ ?  $f(2, 5)$ ?

**Solution.**

```
int m = 0;
while(!(m * a <= b))
    {m++;}
```

For  $a = 2$  and  $b = 4$ , the smallest natural number  $m$  for which  $m \cdot a \geq 4$ , i.e., for which  $m \cdot 2 \geq 4$ , is  $m = 2$ , so  $f(2, 4) = 2$ .

For  $a = 2$  and  $b = 5$ , the smallest natural number  $m$  for which  $m \cdot a \geq 5$ , i.e., for which  $m \cdot 2 \geq 5$ , is  $m = 3$ , so  $f(2, 5) = 3$ .

**Problem 7-8.** Suppose that someone comes up with a new proof that not every computable function is primitive recursive, by providing a new example of a function  $N(n)$  which is computable but not primitive recursive. What if, in addition to  $0$ ,  $\pi_i^k$ , and  $\sigma$ , we also allow this new function  $N(n)$  in our constructions? Let us call functions that can be obtained from  $0$ ,  $\pi_i^k$ ,  $\sigma$ , and  $N(n)$  by using composition and primitive recursion *N-primitive recursive* functions. Will then every computable function be *N-primitive recursive*? Prove that your answer is correct.

**Detailed solution.** On the test, a shorter form is OK.

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

- By definition, an *N-p.r.* function is obtained from  $0$ ,  $\sigma$ ,  $\pi_i^k$ , and  $N(n)$  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  $\sigma$ ,  $\pi_i^k$ , 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,  $\sigma$ ,  $\pi_i^k$ , 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 ...

- 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 ...

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



**Second part of the proof: an important lemma.** To prove our result, we will need the following lemma:

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

- *checks whether  $c$  is an  $N$ -pr-code of some  $N$ -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.** An  $N$ -pr-code was obtained from the original expression as follows

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

Thus, to get back from the  $N$ -pr-code to the original  $N$ -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  $N$ -pr-code of any  $N$ -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  $\text{\LaTeX}$  compiler to check that the corresponding ASCII sequence is a valid  $\text{\LaTeX}$  expression. This is what  $\text{\LaTeX}$  compilers do all the time. If it is a valid  $\text{\LaTeX}$  expression,  $\text{\LaTeX}$  translates it 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 an  $N$ -pr-code of an  $N$ -p.r. function, we return  $f(c) = f_c(c) + 1$ ;
- otherwise, if  $c$  is *not* an  $N$ -pr-code of an  $N$ -p.r. function, we return

$f(c) = 0.$

Let us prove that this function is computable but not  $N$ -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  $\mathcal{A}$  whose existence is proven by the lemma. This algorithm either tells us that  $c$  is not an  $N$ -pr-code – in which case we return  $f(c) = 0$  – or generates the file  $f_c$ .
- If  $c$  is an  $N$ -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:

$$\begin{array}{l} \xrightarrow{c} \text{is } c \text{ an } N\text{-pr-code?} \xrightarrow{\text{yes}} \text{apply } f_c \text{ to } c \xrightarrow{f_c(c)} \text{add } 1 \xrightarrow{f_c(c)+1} \\ \downarrow \text{no} \\ 0 \end{array}$$

**Proving that the function  $f(c)$  is not  $N$ -primitive recursive.** We will prove this by contradiction. Let us assume that the function  $f(c)$  is  $N$ -primitive recursive. Let  $c_0$  denote its  $N$ -pr-code. Then, by definition of  $f_c$  as a code that computes the original  $N$ -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 an  $N$ -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  $N$ -primitive recursive. Thus, this assumption is wrong, and the above defined function  $f(c)$  is not  $N$ -primitive recursive.

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

**Problem 9.** Design a Turing machine for computing  $n+2$  in unary code. Trace it for  $n = 3$ .

**Solution.** Main idea is similar to computing  $n + 1$  in unary code:

- we go step-by-step until we find the first blank space,
- then, we replace this blank space with 1, and inform the headquarters that we have added the first 1;
- then, we replace the next blank space with 1, and go back.

The resulting Turing machine is as follows:

- start,  $- \rightarrow R$ , working
- working,  $1 \rightarrow R$
- working,  $- \rightarrow 1$ , R, added1
- added1,  $- \rightarrow 1$ , L, back
- back,  $1 \rightarrow L$
- back,  $- \rightarrow$  halt.

Tracing:

<u>-</u>	1	1	1	-	-	-	...	start
-	<u>1</u>	1	1	-	-	-	...	working
-	1	<u>1</u>	1	-	-	-	...	working
-	1	1	<u>1</u>	-	-	-	...	working
-	1	1	1	<u>-</u>	-	-	...	working
-	1	1	1	1	<u>-</u>	-	...	added1
-	1	1	1	<u>1</u>	1	-	...	back
-	1	1	<u>1</u>	1	1	-	...	back
-	1	<u>1</u>	1	1	1	-	...	back
-	<u>1</u>	1	1	1	1	-	...	back
<u>-</u>	1	1	1	1	1	-	...	back
<u>-</u>	1	1	1	1	1	-	...	halt

**Problem 10.** Design a Turing machine for computing  $n + 2$  in binary code. Trace it for  $n = 3$ .

**Solution.** The corresponding Turing machine was described in the lecture. Its main idea is as follows: Here is the resulting algorithm:

- we skip the last bit,
- after that, if we see 1, we replace 1 with 0;
- if we see 0 or blank, we replace them with 1 and start going back.

The corresponding Turing machine rules are:

- start,  $- \rightarrow R$ , skip
- skip,  $1 \rightarrow R$ , moving
- skip,  $0 \rightarrow R$ , moving
- moving,  $1 \rightarrow 0$ , R
- moving,  $0 \rightarrow 1$ , L, back
- moving,  $- \rightarrow 1$ , L, back
- back,  $0 \rightarrow L$
- back,  $1 \rightarrow L$
- back,  $- \rightarrow \text{halt}$

Here is the tracing:

-	1	1	-	-	-	...	start
-	<u>1</u>	1	-	-	-	...	skip
-	1	<u>1</u>	-	-	-	...	moving
-	1	0	-	-	-	...	moving
-	1	<u>0</u>	1	-	-	...	back
-	<u>1</u>	0	1	-	-	...	back
-	1	0	1	-	-	...	back
-	1	0	1	-	-	...	halt