Solutions to Test 1

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

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

You can use add(.,.) (sum) and mult(., .) (product) in this expression.

What is the value of this function when $a = b = c = 2$?

Solution. In general, the value x depends on a, b, c, and on the number of the iteration, so we have $x(a, b, c, m)$. For $m = 0$, before the iteration starts, we have

$$x(a, b, c, 0) = a.$$

At each iteration step, we add $b + c$ to the previous value:

$$x(a, b, c, m + 1) = add(x(a, b, c, m), add(b, c)).$$

In general, primitive recursion defines a function of $k + 1$ variables. In our case, $x(a, b, c, m)$ is a function of 4 variables, so $k + 1 = 4$ and $k = 3$. For $k = 3$, the general primitive recursion takes the form:

$$h(n_1, n_2, n_3, 0) = f(n_1, n_2, n_3);$$

$$h(n_1, n_2, n_3, m + 1) = g(n_1, n_2, n_3, m, h(n_1, n_2, n_3, m)).$$

To match, we rename x into h, a into n_1, b into n_2, and c into n_3. As a result, we get:

$$h(n_1, n_2, n_3, 0) = n_1;$$

$$h(n_1, n_2, n_3, m + 1) = add(h(n_1, n_2, m), add(n_2, n_3)).$$

Thus, here, $f = \pi_1^3$, $g = add(\pi_5^5, add(\pi_2^5, \pi_3^5))$ and thus,

$$x = PR(\pi_1^3, add(\pi_5^5, add(\pi_2^5, \pi_3^5))).$$

The desired function F is obtained from x when we take $m = b$, i.e., $F(a, b, c) = x(a, b, c, b)$, thus

$$F = x(\pi_1^3, \pi_2^3, \pi_3^3, \pi_2^3).$$

For $a = b = c = m = 2$:

- first we assign $x = a = 2$;
- then, we assign $i = 1$; since $i \le b = 2$, we go inside the loop and assign, to x, the new value

$$x + b + c = 2 + 2 + 2 = 6;$$

- after that, we increase i by 1, so now $i = 2$; we still have $i \le b = 2$, so we again go into the loop and assign, to x, the new value

$$x + b + c = 6 + 2 + 2 = 10;$$

- now, we again increase i by 1; now $i = 3$; we no longer have $i \le b = 2$, so we get out of the loop.

The resulting value is 10.

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

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

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

Solution. Here, $F = \sigma(\sigma(PR(\ldots)))$, so to the result h of primitive recursion, we should add 2: $F(n_1, \ldots, m) = h(n_1, \ldots, m) + 2$.

In the general expression for primitive recursion $h = PR(f, g)$, g is the function of $k+2$ variables. Here, π_1^4 is a function of four variables, so $add(\sigma(\pi_1^4), \pi_3^4))$ is also a function of four 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,

$$h(n_1, n_2, 0) = \sigma(0) = 1;$$

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

$$add(\sigma(\pi_1^4(n_1, n_2, m, h(n_1, n_2, m))), \pi_3^4(n_1, n_2, m, h(n_1, n_2, m))) =$$

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

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 σ, which means that at the end, we should add 1. So, we get the following code:

```
h = 1;
for(int i = 1; i <= m; i++)
  {h = n1 + 1 + (i - 1);}
F = h + 2;
```

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

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

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 2 and get $F(2, 0, 1) = h(2, 0, 1) + 2 = 3 + 2 = 5$.

Problem 3-4. Prove, from scratch, that the function $f(p) = (p-1)!/p$, where $a!$ is the factorial $a! = 1 \cdot 2 \cdot \ldots \cdot a$, 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. Since, by definition, the composition of p.r. functions is p.r., to prove that the desired function is primitive recursive (p.r.), it is sufficient to prove the following:

- that subtracting 1 is p.r.,
- that factorial is p.r., and
- that integer division is p.r.

1. The function $prev(n) = n - 1$ is p.r., since it can be represented as:

$$prev(0) = 0;$$

$$prev(a+1) = a.$$

2. Factorial $fact(n) = 1 \cdot 2 \cdot \ldots \cdot n$ can be represented in the following form:

$$fact(0) = 1;$$

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

Thus, to prove that factorial is p.r., it is sufficient to prove that multiplication is p.r.

2.1. Multiplication can be represented as

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

thus

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

$$mult(a, b+a) = mult(a, b) + a.$$

So, to prove that multiplication is p.r., it is sufficient to prove that addition is p.r.

2.2. Addition is p.r. since the function

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

can be represented as

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

$$add(a, b+1) = add(a, b) + 1.$$

3. Integer division b/a – which we will denote by $div(a, b)$ – can be represented as:

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

$$div(a, m + 1) = if\ (rem(a, m + 1) = 0)\ then\ div(a, m) + 1\ elsediv(a, m).$$

Thus, to prove that division is p.r., it is sufficient to prove that remainder is p.r. and that if-then-else construction is p.r.

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

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

$$rem(a, b + 1) = if\ (rem(a, b) + 1 < b)\ then\ (rem(a, b) + 1)\ else\ 0.$$

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.

3.2. 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.

3.3. Subtraction can be represented as

$$sub(b, 0) = b;$$

$$sub(b, r + 1) = sub(b, r) \dot{-} 1.$$

We already proved that the function $prev(n) = n \dot{-} 1$ is p.r., so $sub(a, b)$ is also p.r.

3.4. 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:

$$pos(0) = 0;$$

$$pos(m + 1) = 1.$$

3.5. Let us no show that the if-the-else construction is p.r. Indeed, the if-then-else construction can be represented as

$$if\ (P(\overline{n}))\ then\ f(\overline{n})\ else\ g(\overline{n}) = P(\overline{n}) \cdot f(\overline{n}) + (1 \dot{-} P(\overline{n})) \cdot g(\overline{n}).$$

We already proved that multiplication is primitive recursive, so remainder is also p.r., and thus, division is p.r.

4. Thus, the desired function – which is a composition of p.r. functions – is also p.r.

Problem 5. Prove that the following function $f(p)$ is μ-recursive: $f(p) = p!$ when p is either 1 or 2, and $f(p)$ is undefined for all other p.

Solution. Here, $1! = 1$ and $2! = 1 \cdot 2$, so

$$f = \mu m.((p = 1 \,\&\, m = 1) \vee (p = 2 \,\&\, m = 2)).$$

When p is equal to 1 or 2, then the smallest m for which the corresponding property

$$(p = 1 \,\&\, m = 1) \vee (p = 2 \,\&\, m = 2)$$

is satisfied is exactly $m = p!$. For all other p, this property is never satisfied, so the function is undefined.

Problem 6. Translate the following μ-recursive expression into a while-loop:

$$f(a) = \mu m.(m \cdot a > a).$$

For this function f, what is the value of $f(0)$? $f(2)$?

Solution.

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

For $a = 0$, for all m, we have $m \cdot a = a = 0$, so the inequality $m \cdot a > a$ is never satisfied. Thus, for $m = 0$, this function is undefined.

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

Problem 7-8. Suppose that someone comes up with a new proof that not every computable function is primitive recursive, by providing two new examples of function $N(n)$ and $N'(n)$ which are computable but not primitive recursive. What if, in addition to 0, π_i^k, and σ, we also allow both new function in our constructions? Let us call functions that can be obtained from 0, π_i^k, σ, $N(n)$, and $N'(n)$ by using composition and primitive recursion *2-primitive recursive* functions. Will then every computable function be 2-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 2-p.r. function, a natural number that we will call this function's *2-pr-code*. This will be done in several steps.

- By definition, an 2-p.r. function is obtained from 0, σ, π_i^k, $N(n)$, 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 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 2-pr-code of the original 2-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 a 2-pr-code of some 2-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 2-pr-code was obtained from the original expression as follows

$$\text{expression} \stackrel{\text{LaTeX}}{\longrightarrow} \text{ASCII expression} \stackrel{\text{ASCII}}{\longrightarrow} \text{binary sequence} \stackrel{\text{append 1}}{\longrightarrow} \text{2-pr-code}.$$

Thus, to get back from the 2-pr-code to the original 2-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 2-pr-code of any 2-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 2-pr-code of a 2-p.r. function, we return $f(c) = f_c(c) + 1$;
- otherwise, if c is *not* a 2-pr-code of a 2-p.r. function, we return

$$f(c) = 0.$$

Let is prove that this function is computable but not 2-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 a 2-pr-code – in which case we return $f(c) = 0$ – or generates the file f_c.
- If c is an 2-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:

$$\xrightarrow{c} \text{is } c \text{ a 2-pr-code? } \xrightarrow{\text{yes}} \text{apply } f_c \text{ to } c \xrightarrow{f_c(c)} \text{add 1} \xrightarrow{f_c(c)+1}$$

$$\downarrow \text{no}$$

$$0$$

Proving that the function $f(c)$ is not 2-primitive recursive. We will prove this by contradiction. Let us assume that the function $f(c)$ *is* 2-primitive recursive. Let c_0 denote its 2-pr-code. Then, by definition of f_c as a code that computes the original 2-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 2-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 2-primitive recursive. Thus, this assumption is wrong, and the above defined function $f(c)$ is not 2-primitive recursive.

Conclusion. We have come up with a function $f(c)$ which is computable but not 2-primitive recursive. Thus, the result is proven.

Problem 9. Design a Turing machine for computing $n+4$ in unary code. Trace it for $n = 1$.

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, etc.,
- after we add four 1s, we go back.

The resulting Turing machine is as follows:

- start, – → R, working
- working, 1 → R
- working, – → 1, R, added1
- added1, – → 1, R, added2
- added2, – → 1, R, added3
- added3, – → 1, L, back
- back, 1 → L
- back, – → halt.

Tracing:

$\underline{-}$	1	–	–	–	–	–	...	start
–	$\underline{1}$	–	–	–	–	–	...	working
–	1	$\underline{-}$	–	–	–	–	...	working
–	1	1	$\underline{-}$	–	–	–	...	added1
–	1	1	1	$\underline{-}$	–	–	...	added2
–	1	1	1	1	$\underline{-}$	–	...	added3
–	1	1	1	$\underline{1}$	1	–	...	back 1
–	1	1	$\underline{1}$	1	1	–	...	back
–	1	$\underline{1}$	1	1	1	–	...	back
–	$\underline{1}$	1	1	1	1	–	...	back
$\underline{-}$	1	1	1	1	1	–	...	back
$\underline{-}$	1	1	1	1	1	–	...	halt

Problem 10. Design a Turing machine for computing $n + 4$ in binary code. Trace it for $n = 1$.

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

- we skip the last two bits; i we see blank, we replace it with 0;
- 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, – → R, skip1st
- skip1st, 1 → R, skip2nd
- skip1st, 0 → R, skip2nd
- skip2nd, 1 → R, moving
- skip2nd, 0 → R, moving
- skip2nd, – → 0, R, moving
- moving, 1 → 0, R
- moving, 0 → 1, L, back
- moving, – → 1, L, back
- back, 0 → L
- back, 1 → L
- back, – → halt

Here is the tracing:

$\underline{-}$	1	–	–	–	–	...	start
–	$\underline{1}$	–	–	–	–	...	skip1st
–	1	$\underline{-}$	–	–	–	...	skip2nd
–	1	0	$\underline{-}$	–	–	...	moving
–	1	$\underline{0}$	1	–	–	...	back
–	$\underline{1}$	0	1	–	–	...	back
$\underline{-}$	1	0	1	–	–	...	back
$\underline{-}$	1	0	1	–	–	...	halt