

# Laboratory 7

Variant #3

Mariya Zacharneva and Ruslan Melnyk

## The task

The task is to write a program in Prolog to convert an input integer  $N$  to an English word,  $N < 1000$ .

## Prolog

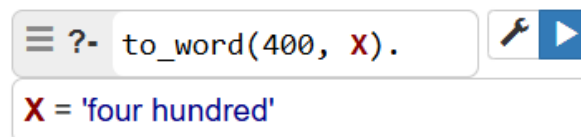
Prolog is a declarative, logic programming language. The main parts of the program are facts and rules:

- Facts represent some statement, which can be true or false. For example, you can state a fact as `animal(dog)`, then query `?- animal(dog)` will return true, and `?- animal(frog)` will return false.
- Rules allow us to state some conclusions from existing facts by defining relationships between them. For example, if rule `alive(X) :- animal(X)` and fact `animal(dog)` are defined, `?- alive(dog)` will return true.

As mentioned before, the database created from rules and facts can be queried by the user, to check if some new facts are true.

Also, one can query the database as `?- alive(dog)`, which will return `X = dog` – this mechanism allows us to find objects, for which some fact is true. We will be using this in our task.

## Usage



## Program

First of all, we have to define the database of words, which are used as building bricks for numbers from 1 to 999.

1	<code>word(0, "zero").</code>	12	<code>word(11, "eleven").</code>	22	<code>word(10, "ten").</code>
2	<code>word(1, "one").</code>	13	<code>word(12, "twelve").</code>	23	<code>word(20, "twenty").</code>
3	<code>word(2, "two").</code>	14	<code>word(13, "thirteen").</code>	24	<code>word(30, "thirty").</code>
4	<code>word(3, "three").</code>	15	<code>word(14, "fourteen").</code>	25	<code>word(40, "forty").</code>
5	<code>word(4, "four").</code>	16	<code>word(15, "fifteen").</code>	26	<code>word(50, "fifty").</code>
6	<code>word(5, "five").</code>	17	<code>word(16, "sixteen").</code>	27	<code>word(60, "sixty").</code>
7	<code>word(6, "six").</code>	18	<code>word(17, "seventeen").</code>	28	<code>word(70, "seventy").</code>
8	<code>word(7, "seven").</code>	19	<code>word(18, "eighteen").</code>	29	<code>word(80, "eighty").</code>
9	<code>word(8, "eight").</code>	20	<code>word(19, "nineteen").</code>	30	<code>word(90, "ninety").</code>
10	<code>word(9, "nine").</code>				

These are all the facts used in our program: these will be used later on when applying rules.

Now we can start creating rules, which will describe all possible cases in our program, starting from the simplest ones:

- The number is already in the database (for example, 12 or 80) → just search the database.

```
32 to_word(N, W) :- word(N, W), !.
```

The part at the end of the line, "!", means that the algorithm should not search for any other rules that might suit in this case. By default, the algorithm returns the first findings, but it can provide other findings as well if prompted. We don't need this for our task: if the number is already defined in basic facts, nothing else should be done.

- The number is between 20 and 99, meaning that it has a decimal part:

```
42 to_word(N, W) :-
43     N > 19,
44     N < 100,
45     D is N // 10 * 10,
46     word(D, W1),
47     R is N mod 10,
48     R > 0,
49     word(R, W2),
50     atomic_list_concat([W1, W2], " ", W), !.
```

The idea is the following: firstly we check if the number is within the boundaries; then we assign D to be the decimal number without the remainder (for example, for N = 23, D = 20) and we know that it will be present in the database; then we take a remainder separately (for N = 23, R = 3) and search it separately as well; in the end, we concatenate the result.

Again, at the end we marked that the algorithm should not search for any more suitable rules; it is not strictly necessary, but our algorithm distinguishes every possible case, so it makes sense to explicitly indicate that.

- Last case, the number is between 100 and 999:

```

60 to_word(N, W) :-
61     N > 99,
62     H is N // 100,
63     word(H, W1),
64     S is N mod 100,
65     S > 0,
66     to_word(S, W2),
67     atomic_list_concat([W1, "hundred and", W2], " ", W), !.

```






The idea is the same as in the previous case: we divide the number into two parts – hundreds and the remaining part – and process them separately, concatenating the result in the end.

## Zero problem

If we will use the code from above without any twickement (removing  $R > 0$  and  $S > 0$  checks), we will face the problem: after providing the algorithm with numbers like 400, it will output “four hundred and zero”, which is formally correct, but not what we want. To solve this, we creating two more rules, which will handle this case:

<pre> 34 to_word(N, W) :- 35     N &gt; 19, 36     N &lt; 100, 37     D is N // 10 * 10, 38     0 is N mod 10, 39     word(D, W), !. </pre>	<pre> 51 to_word(N, W) :- 52     N &gt; 99, 53     H is N // 100, 54     word(H, W1), 55     0 is N mod 100, 56     atomic_list_concat([W1, "hundred"], " ", W), !. </pre>
---	--

## Usage examples

<div> <div>≡ ?- to_word(4, X).</div> <div>  </div> </div> <div>X = "four"</div>	<div> <div>≡ ?- to_word(37, X).</div> <div>  </div> </div> <div>X = 'thirty seven'</div>
<div> <div>≡ ?- to_word(211, X).</div> <div>  </div> </div> <div>X = 'two hundred and eleven'</div>	<div> <div>≡ ?- to_word(365, X).</div> <div>  </div> </div> <div>X = 'three hundred and sixty five'</div>
<div> <div>≡ ?- to_word(700, X).</div> <div>  </div> </div> <div>X = 'seven hundred'</div>	

## **Conclusion**

In this lab we familiarize ourselves with the Prolog programming language, which finds its applications in AI due to its declarative nature and logical structure. We practiced defining facts, rules, and handling edge cases. This task improved our understanding of logic-based programming and its relevance to AI.