Laboratory 7

Variant 4

Class Group 105

Group 24

By Dimitrios Gkolias and Vasileios Ntalas

## Introduction

In Lab 7 we get introduced to Prolog programming language, with which we try to write a program that converts English words of a written number (up to a thousand) into numerical digits N, where N<=1000. The basis of the Prolog language is the definition of the rules, the facts and the queries we are testing each time. The implementation of our approach is explained below.

## Implementation

First of all, we define our facts, for example every English word of a written number is connected to the corresponding number- for each number between 1-9, for every teen (10-19) and for every ten (20,30,...,90).

```
% Units (1-9)
unit(one, 1).
unit(two, 2).
unit(three, 3).
unit(four, 4).
unit(five, 5).
unit(six, 6).
unit(seven, 7).
unit(eight, 8).
unit(nine, 9).

% Teens (10-19)
teen(ten, 10).
teen(eleven, 11).
teen(twelve, 12).
teen(thirteen, 13).
teen(fourteen, 14).
teen(fifteen, 15).
teen(sixteen, 16).
teen(seventeen, 17).
teen(eighteen, 18).
```

```
teen(nineteen, 19).

% Tens (20, 30, ..., 90)
tens(twenty, 20).
tens(thirty, 30).
tens(forty, 40).
tens(fifty, 50).
tens(sixty, 60).
tens(seventy, 70).
tens(eighty, 80).
tens(ninety, 90).
```

After the definition of the facts, in the main function ***to_num(WordString, Number)*** we convert the input to lower case so that the solution can also work with capital letters. Then we split the input string by the spaces and we convert the list of strings to a list of atoms so that we later use atoms as grammar syntax to our rules. Finally, we use our DCG (Definite Clause Grammars) rules to calculate the number with the condition that the number does not surpasses 1000.

```
to_num(WordString, Number) :-
    string_lower(WordString, LowerWordString),           % Convert to
lowercase
    split_string(LowerWordString, " ", "", StringWordsList), % Split by space,
ignore empty parts
    maplist(atom_string, AtomWordsList, StringWordsList), % Convert list of
strings to list of atoms

    % 2. Use DCG to parse the list of atoms and calculate the number
    phrase(number_phrase(Number), AtomWordsList, []),
    Number =< 1000. % Ensure the resulting number is within the constraint
```

Now we will focus on the DCG rules of our program. The fundamental rules that are the building blocks of the next DCG rules are the following:

```
unit_word(Value) --> [Word], { unit(Word, Value) }.
teen_word(Value) --> [Word], { teen(Word, Value) }.
tens_word(Value) --> [Word], { tens(Word, Value) }.
```

These rules are responsible for recognizing and converting individual number words. For instance, tens_word(Value) will attempt to match the current input word against known tens (like 'forty') and unify Value with its numerical equivalent (40). Higher-level rules then use these fundamental recognizers to parse sequences like "forty two" and combine the retrieved numerical values.

Overall, we consider initially three cases, the first is the input string to be 1000, the second is the input string to have hundred in it, and the third is to not have hundred in it. In the second case we use the thirds' case function for the tens and for the tens we consider four different cases.

For case 1 we have a simple assignment:

```
% Case 1: "one thousand"
number_phrase(1000) --> [one, thousand]
```

For case 2, we take the first word as the number of the hundred and find it applying our facts. Then the second word is obviously the "hundred" word, and later we consider three possibilities: the first one is the input to have the word "and" after the hundred in which case we then use another rule for the tens the ***tens_units_val(Value)***, the second one is to not have the word "and" and directly have the tens number, and the third is if nothing follows "hundred" in that case we do not have tens in our string input. This way we also consider the case that the string input does not include the word "and". Finally, we assign the number as { Value is H * 100 + TU }, where H is the number of the first word which we multiply by 100 and the TU are the tens.

```
% Case 2: Numbers with a "hundred" component (e.g., "two hundred and fifty
three", "five hundred")
number_phrase(Value) -->
    unit_word(H),             % e.g., "two"
    [hundred],                % "hundred"
    (      % After "hundred", there can be "and <tens_units>", or just
"<tens_units>", or nothing.
        ( [and], tens_units_val(TU) )     % "and fifty three"
    ;   ( tens_units_val(TU) )           % "fifty three" (without "and" - less
common but possible)
    ;   ( {TU = 0} )                        % Nothing follows "hundred" (e.g.
"two hundred")
    ),
    { Value is H * 100 + TU }.
```

For the third case we look at the numbers with no hundreds in them. For the function ***tens_units_val(Value)*** we have multiple clauses. We first try to see if the number with no hundreds is a number with tens and ones, so in that case we perform for the tens the fact ***tens_word(T)***, for the ones ***unit_word(U)***, and finally we assign the value to be T + U: { Value is T + U }. If this first rule fails we try a second clause that handles the case that the number is only made up of tens, for example "twenty", "forty" (as a whole number, or part of "four hundred and twenty" used in the above hundreds function). In this case we assign with

the fact ***tens_word(Value)***. The other two clauses are similar handling the cases of teens and ones, ***teen_word(Value)*** and ***unit_word(Value)*** respectively.

```
% Case 3: Numbers without a "hundred" component (0-99) (e.g., "forty two",
"seven")
number_phrase(Value) -->
    tens_units_val(Value).



% e.g., "forty five", "twenty one"
tens_units_val(Value) -->
    tens_word(T),
    unit_word(U),
    { Value is T + U }.

% e.g., "twenty", "forty" (as a whole number, or part of "four hundred and
twenty")
tens_units_val(Value) -->
    tens_word(Value).

% e.g., "thirteen", "nineteen"
tens_units_val(Value) -->
    teen_word(Value).

% e.g., "five", "nine"
tens_units_val(Value) -->
    unit_word(Value).
```

This process sums up our program! We then test some queries, for example:

```
?- to_num("ninety nine", N).

?- to_num("three hundred and ninety four", N).
```

The numbers that fail are of course numbers above 1000 and the number 0.

## Conclusion

This laboratory exercise provided a practical introduction to the Prolog programming language, focusing on its strengths in symbolic computation and rule-based pattern matching through the development of a number-to-word converter. The objective was to create a program capable of

translating English number words (up to "one thousand") into their corresponding numerical digits, adhering to the constraint $N \leq 1000$.

The implemented solution leveraged Prolog's core features: facts, rules, and Definite Clause Grammars (DCGs). A foundational set of facts meticulously defined the numerical values for unit words (e.g., unit(one, 1)), teen words (e.g., teen(eleven, 11)), and tens words (e.g., tens(twenty, 20)). The primary predicate, to_num/2, handled initial input processing, including case normalization and tokenization of the word string into a list of atoms suitable for DCG parsing. The heart of the conversion logic resided within the DCG rules. Fundamental rules like unit_word/1, teen_word/1, and tens_word/1 served to identify and extract values from individual word tokens. These were then composed into more complex rules: tens_units_val/1 elegantly handled numbers from 0-99 by defining multiple clauses to parse structures like "forty five," "twenty," or "seven." The main number_phrase/1 rule managed the overall number structure, successfully distinguishing and processing "one thousand," numbers involving "hundred" (with or without "and"), and numbers directly handled by tens_units_val/1. The disjunctive capabilities of DCGs were particularly useful in managing variations like the presence or absence of "and" after "hundred."

Testing confirmed the program's ability to correctly convert a range of valid inputs, such as "ninety nine" to 99 and "three hundred and ninety four" to 394, while correctly failing for inputs outside the defined scope, like numbers greater than one thousand or the unhandled "zero."

This project successfully demonstrated the power and conciseness of Prolog and DCGs for parsing natural language-like structures. The declarative nature of defining grammatical rules allowed for a clear and relatively straightforward implementation of a non-trivial conversion task. The exercise was a valuable experience in understanding logical programming paradigms and their application to symbolic processing problems.