Declarative Programming: Handout 6

Yılmaz Kılıçaslan
November 08, 2023

Key Concepts: *Arithmetic, Natural Numbers, Computability, Complexity, Recursive Function, Nested Recursion, Ackermann's Function.*

RECURSION IN ARITHMETIC

1 INTRODUCTION

Last week we particularly emphasized that rules should preferably be defined in terms of existing facts and rules. We also had a look at a very interesting extension of this approach where relationships are defined in terms of themselves, which is referred to as *recursive programming*.

Today we will try to sharpen our way of thinking and programming recursively by writing recursive programs that define arithmetic concepts and operations. It is worth noting that Prolog does not define arithmetic as we do here, but rests on the underlying capabilities of the computer directly. Our aim is just pedagogical.

2 ARITHMETIC

2.1 Basic Issues

Arithmetic is defined as the branch of mathematics that is concerned with the study of numbers in terms of various operations on them. Therefore, if we want to re-define (a fragment of) arithmetic in a recursive way, we first need to define the *base* cases for numbers and operations.

We will take *0 (zero)* as our base number and the *successor/predecessor* operation (i.e., increment/decrement by 1 (one)) as our base operation. We will define everything else recursively on the basis of these (and the greater-than relation, >).

2.2 Natural Numbers

We will confine the domain of numbers to operate to natural numbers. Here is the recursive procedure defining natural numbers:

(1) *Procedure defining the natural numbers:*

```
% Base Clause
natural_number(0).

% Recursive Clause
natural_number(N):-
    N > 0,
    M is N - 1,
    natural_number(M).
```

Below is the simplest query to this program:

(2) *Query to ask if 0 is a natural number:*

```
?- natural_number(0).
```

This query is resolved without having recourse to the recursive clause. However, queries issued for larger numbers will require the recursive part to be involved in the resolution, as shown by the following example (where the goal `N > 0` is assumed not to be used in the program):

(3) *Query to ask if 7 is a natural number:*

```
?- trace, natural_number(7).
   Call: (11) natural_number(7) ? creep
   Call: (12) _33588 is 7+ -1 ? creep
   Exit: (12) 6 is 7+ -1 ? creep
   Call: (12) natural_number(6) ? creep
   Call: (13) _33726 is 6+ -1 ? creep
   Exit: (13) 5 is 6+ -1 ? creep
   Call: (13) natural_number(5) ? creep
   Call: (14) _33864 is 5+ -1 ? creep
   Exit: (14) 4 is 5+ -1 ? creep
   Call: (14) natural_number(4) ? creep
   Call: (15) _34002 is 4+ -1 ? creep
   Exit: (15) 3 is 4+ -1 ? creep
   Call: (15) natural_number(3) ? creep
   Call: (16) _34140 is 3+ -1 ? creep
   Exit: (16) 2 is 3+ -1 ? creep
   Call: (16) natural_number(2) ? creep
   Call: (17) _34278 is 2+ -1 ? creep
   Exit: (17) 1 is 2+ -1 ? creep
   Call: (17) natural_number(1) ? creep
   Call: (18) _34416 is 1+ -1 ? creep
   Exit: (18) 0 is 1+ -1 ? creep
   Call: (18) natural_number(0) ? creep
   Exit: (18) natural_number(0) ? creep
   Exit: (17) natural_number(1) ? creep
   Exit: (16) natural_number(2) ? creep
   Exit: (15) natural_number(3) ? creep
   Exit: (14) natural_number(4) ? creep
   Exit: (13) natural_number(5) ? creep
   Exit: (12) natural_number(6) ? creep
   Exit: (11) natural_number(7) ?
```

We observe that the recursive clause has been repeatedly chosen (by the interpreter) to resolve a goal until the input number has become the base number, which is 0.

Note that the input number is checked to be larger than zero (by the goal N > 0) in the body of the recursive clause. This is necessary to prevent the interpreter from going into an infinite loop.

2.3 Addition

Addition can be defined in terms of itself as follows:

(4) *Procedure defining the addition operation:*

```
% Base Clause
add(0, R, R).

% Recursive Clause
add(N, M, R):-
    natural_number(N),
    natural_number(M),
    N1 is N - 1,
    add(N1, M, R1),
    R is R1 + 1.
```

Notice that addition can be thought of as a repeated incrementation.

Below are two example queries (together with the responses) issued for the program above:

(5) *Addition with the identity element:*

```
?- add(0, 5, R).
R = 5
```

(6) *Addition of two numbers:*

```
?- add(7, 5, R).
R = 12
```

2.4 Multiplication

Below is a recursive definition of multiplication in prolog:

(7) *Procedure defining the multiplication operation:*

```
% Base Clause
multiply(0, _, 0).

% Recursive Clause
multiply(N, M, R):-
    natural_number(N),
    natural_number(M),
    N1 is N - 1,
    multiply(N1, M, R1),
    add(R1, M, R).
```

Note that this time multiplication can be thought of as a repeated addition.

Here is an example of multiplication with the absorbing element, 0:

(8) *Multiplication with the absorbing element:*

```
?- multiply(0, 5, R).
R = 0
```

Here is a multiplication of two numbers:

(9) *Multiplication of two numbers:*

```
?- multiply(7, 5, R).
R = 35
```

2.5 Exponentiation

Exponentiation, too, can be recursively defined, as the following Prolog procedure illustrates:

(10) *Procedure defining the exponentiation operation:*

```
% Base Clauses
exponent(0, M, 0):- M > 0.
exponent(N, 0, 1):- N > 0.

% Recursive Clause
exponent(N, M, R):-
    M > 0,
    M1 is N - 1,
    exponent(N, M1, R1),
    multiply(R1, N, R).
```

Exponentiation is a repeated multiplication.

Below are some examples concerning exponentiation:

(11) *Raising 0 to the power of 5:*

```
?- exponent(0, 5, R).
R = 0
```

(12) *Raising 5 to the power of 0:*

```
?- exponent(5, 0, R).
R = 1
```

(13) *Raising 5 to the power of 3:*

```
?- exponent(5, 3, R).
R = 125
```

(14) *Failure to raise 0 to the power of 0:*

```
?- exponent(0, 0, R).
false
```

3 A QUICK LOOK AT COMPUTATION

3.1 Ackermann's Function

This course has two major objectives: to introduce the paradigm of declarative programming by teaching a logic programming language (which is Prolog) and to provide insight to computer science through the perspective of that paradigm. The *theory of computer science* is split into two sub-theories: the *theory of computation* and the *theory of information*. The theory of computation is further split into two sub-theories: the *theory of computability* and the *theory of complexity*.

The theory of computability was originally known as *recursive function theory*. Recursive functions are those the value of which is defined by the application of the same function to smaller arguments. They are used to check whether a particular function is computable.

A recursive function can be either *primitive* or *non-primitive*. All primitive recursive functions are computable but not every non-primitive recursive function is uncomputable. All the arithmetic functions that we have translated into Prolog up to this point (i.e., the successor, addition, multiplication, and exponentiation functions) are all primitive recursive. An interesting example of non-primitive recursive functions is Ackermann's function. It is interesting because it is computable. It is a function which takes two input arguments and defined by three cases:

(15) *Definition of Ackermann's function:*

```
% Base Clause
Ack(0, j)  = j+1                    for j ≥0

% Recursive Clauses
Ack(i, 0)  = Ack(i-1, 1)            for i > 0
Ack(i, j)  = Ack(i-1, Ack(i, j-1))  for i, j > 0
```

Here is the C code to implement Ackermann's function:

(16) *Code to implement Ackermann's function:*

```
int ack(int i, int j)
{
  if (i == 0 && j >= 0)
    return j + 1;
  if (i >= 0 && j == 0)
    return ack(i-1, 1);
  if (i > 0 && j > 0)
    return ack(i-1, ackermann(i, j-1));
}
```

Note that one of the arguments in the third case calls Ackermann's function once again. This is called *nested recursion* and is what makes Ackermann's function a non-primitive recursive function.

Translating Ackermann's function into Prolog is left to you as an exercise.

Below is a definition of Ackermann's function in prolog:

(17) *Procedure defining Ackermann's function:*

```
% Base Clause
ackermann(0, M, R):-
    M >= 0,
    R is M + 1.

% Recursive Clauses
ackermann(N, 0, R):-
    N > 0,
    N1 is N - 1,
    ackermann(N1, 1, R).

ackermann(N, M, R):-
    N > 0,
    M > 0,
    M1 is M - 1,
    ackermann(N, M1, R1),
    N1 is N - 1,
    ackermann(N1, R1, R).
```

3.2 A few words about complexity

If we examine the primitive recursive functions we saw in Section 2 in terms of the number of inferences needed for their computation, we will observe that they become more and more 'complex' in accordance with the order they were presented.

Ackermann's function is even more complex than the others, even though it is computable. In fact, it grows faster than any primitive recursive function. This is a result of its non-primitive nature.

Computing the Ackermann function can be restated in terms of an infinite table. Below is a small upper-left fragment of that table (from https://helloacm.com/ackermann-function/):

(18) *Table of Values for Ackermann's Function with Small Values:*

Values of A(*m*, *n*)

m\n	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=2+(n+3)-3$
2	3	5	7	9	11	$2n+3=2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^2}-3$	65533 $=2^{2^{2^2}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^2}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^2}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^2}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3}-3$

Note how fast the function grows at the fourth row, which corresponds to the so-called super-exponentiation operation.

4 Conclusion

Recursion is very prevalent. It pervades almost everywhere. We have seen that all arithmetic can be built up recursively starting with natural numbers. Of course, each artihmetic operation becomes more complex than the operations on the basis of which it is defined.

5 Exercise

A. Translate the rules in the Prolog programs above into FOL.

B. Examine all the functions we saw in terms of the number of inferences needed for their computation.

Reference Texts

- Sterling, L. and Shapiro E. (1986) The Art of prolog. Cambridge, Massachusetts: MIT Press.
- https://helloacm.com/ackermann-function/