Mini Project Report

On

Concepts of Programming Languages (392)

Submitted By:

Ruba Sanad Alhrabi

Khzama mohammed Alsalem

Under the supervision of:

Bayan Almklafee

Search Index

Part 1

Procedural Programming "Pascal"

Introduction

Pascal is an imperative and procedural programming language, which Niklaus Wirth designed in 1968–69 and published in 1970, as a small, efficient language intended to encourage good programming practices using structured programming and data structuring. It is named in honor of the French mathematician, philosopher and physicist Blaise Pascal.

History

Pascal was influenced by the ALGOL W efforts, with the explicit goals of producing a language that would be efficient both in the compiler and at run-time, allow for the development of well-structured programs, and to be useful for teaching students structured programming.[4] A generation of students used Pascal as an introductory language in undergraduate courses.

One of the early successes for language was the introduction of UCSD Pascal, a version that ran on a custom operating system that could be ported to different platforms.

Part 1

Procedural Programming "Pascal"

What is Pascal?

Pascal is a high-level procedural programming language widely used as a as a language to learn general programming concepts. Sometimes Pascal is preferred to other languages, and could be useful to solve technical problems. It is not necessary to learn any other easier language to start learning Pascal, or any of that sort. It is a very easy programming language and helps you understand the basics of the world of programming. Also, it greatly helps you to start learning the C programming language and other language which are procedural. I had various experiences, one of which I learned Pascal, and then migrated to C programming very easily. The Pascal programming language has a programming structure and syntax similar to that of the C programming language. The successor language of Pascal is Delphi which is the object oriented version of Pascal.

General Program Skeleton:

```
1  Program {name of program}
2
3  var {global variable declaration block}
4
5  function {function declaration}
6  {local variables}
7
8  begin
9    ...
10 end;
11
12 procedure {procedure declaration
13 {local variables}
14
15 begin
16   ...
17 end;
18
19 begin {main program block starts}
20   ...
21 end. {the end of main program block}
```

Part

Procedural Programming "Pascal"

Decribe Pascal Grammer and C Grammer:

Pascal	C
Lexical Conventions	
Not case-sensitive; upper and lower ,case letters are equivalent – so somename, SOMENAME, Somename ,Somename, SomeName. **are all the same.**	Case-sensitive; upper and lower case letters are different – so somename, SOMENAME, Somename, SomeName **are all different**
Comments	
(* This is a comment *)	**/* This is a comment */**
Constants	

Identifier1=value;
'This is a string'
size = 100;
pi = 3.14159;
first = 'A';
filename = 'Sara.DAT';
	"This is a string"
	#define size 100
	#define pi 3.14159
	#define first 'A'
	#define filename "XYZ.DAT"

Declarations of variables	
i: integer; r: real; b: boolean; c: char; j, k, l: integer;	int i; float r; char c; int j, k, l;
Enumeration types	
type color=(red, blue); var a:color;	enum color {red, green, blue}; enum color a;

Procedural Programming "Pascal"

Some Example of Pascal language:

1-function:

```
program Project1;

function SayHello:String;
begin
SayHello:='Hello World ';
end;

begin
  writeln(SayHello);

  readln;
end.
```

```
C:\Users\Ruba\AppData\Local\T
Hello World
```

2- procedure

```
program Project1;

procedure SayHello;

begin
writeln('Hello From Procedure ');
end;

begin
 SayHello;

  readln;
end.
```

```
C:\Users\Ruba\AppData\Local\Temp\project1.exe
Hello From Procedure
```

6- Array

```
program Project1;
var x:array[1..10]of integer;

begin
  write('Donner x1:');
  readln(x[1]);
  writeln('x[1]=',x[1]);

  readln;
end.
```

```
C:\Users\Ruba\AppData\Local\Temp\project1.exe
Donner x1:5
x[1]=5
```

Part 1

Procedural Programming "Pascal"

3- parameter passing (by value**)**

```
program Project1;

var
   a, b : integer;
(*procedure definition *)
procedure swap(x, y: integer);

var
   temp: integer;

begin
   temp := x;
   x:= y;
   y := temp;
end;

begin
   a := 100;
   b := 200;
   writeln('Before swap, value of a : ', a );
   writeln('Before swap, value of b : ', b );

   (* calling the procedure swap  by value   *)
   swap(a, b);
   writeln('After swap, value of a : ', a );
   writeln('After swap, value of b : ', b );
end.
```

```
Free Pascal Compiler version 3.0.4 [2018/04/10] for x86_64
Copyright (c) 1993-2017 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling jdoodle.pas
Linking jdoodle
25 lines compiled, 0.2 sec
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

4-parameter passing (by reference)

```
program exCallbyRef;
var
   a, b : integer;
(*procedure definition *)
procedure swap(var x, y: integer);

var
   temp: integer;

begin
   temp := x;
   x:= y;
   y := temp;
end;

begin
   a := 100;
   b := 200;
   writeln('Before swap, value of a : ', a );
   writeln('Before swap, value of b : ', b );

   (* calling the procedure swap  by value   *)
   swap(a, b);
   writeln('After swap, value of a : ', a );
   writeln('After swap, value of b : ', b );
end.
```

```
Free Pascal Compiler version 3.0.4 [2018/04/10] for x86_64
Copyright (c) 1993-2017 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling jdoodle.pas
Linking jdoodle
25 lines compiled, 0.1 sec
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

5- recursion

```
1  Program AddNums(output);
2
3  function fact(n: integer): longint;
4  begin
5      if (n = 0) then
6          fact := 1
7      else
8          fact := n * fact(n - 1);
9  end;
10
11 var
12     n: integer;
13
14 begin
15     for n := 0 to 16 do
16         writeln(n, '! = ', fact(n));
17 end.
```

```
Compiling jdoodle.pas
Linking jdoodle
16 lines compiled, 0.2 sec
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 1932053504
14! = 1278945280
15! = 2004310016
16! = 2004189184
```

Part 1

Procedural Programming "Pascal"

Control statements

1- For loop:

```
program Project1;
var i:integer;

begin

  for i:=1 to 10 do
  begin


  writeln('HELLO');
   end;
  readln;
end.
```

```
HELLO
HELLO
HELLO
HELLO
HELLO
HELLO
HELLO
HELLO
HELLO
HELLO
```

" It starts from 1 to 10 and prints the word HELLO ten times if the condition is met"

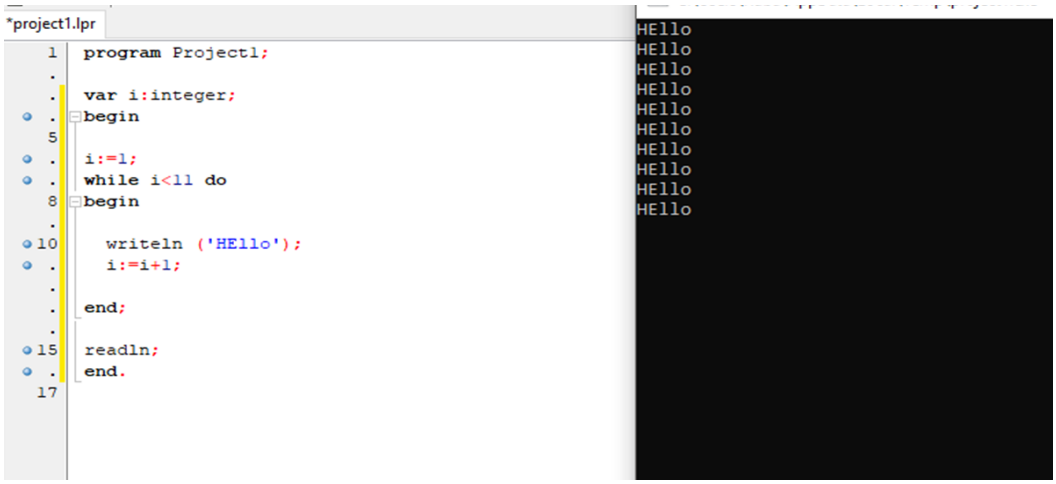
2- IF :

```
program Project1;

begin

if (2<3) then
begin

  writeln ('HEllo');

end;

readln;
end.
```

```
HEllo
```

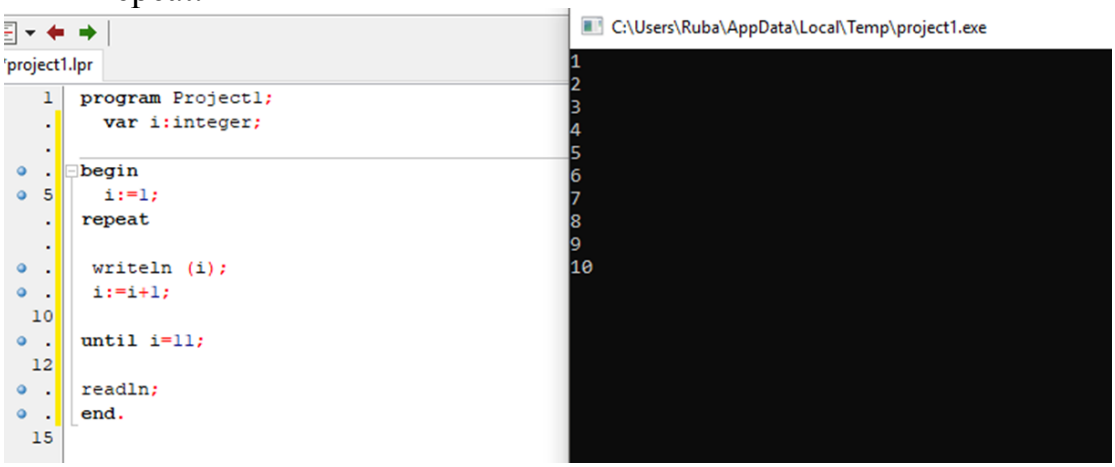
" Check the condition if it is true or not and here check condition 2 is smaller than 3"

3- While :



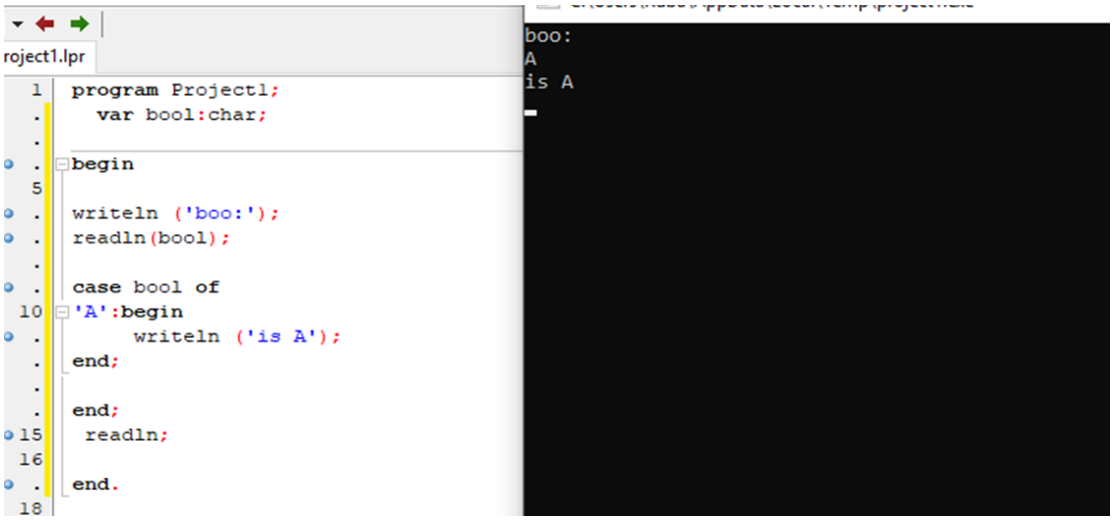
" We put primitive values of the type of correct numbers and compare whether they are larger or not and all are increasing"

4- Repeat:



" Print the numbers from 1 to 10"

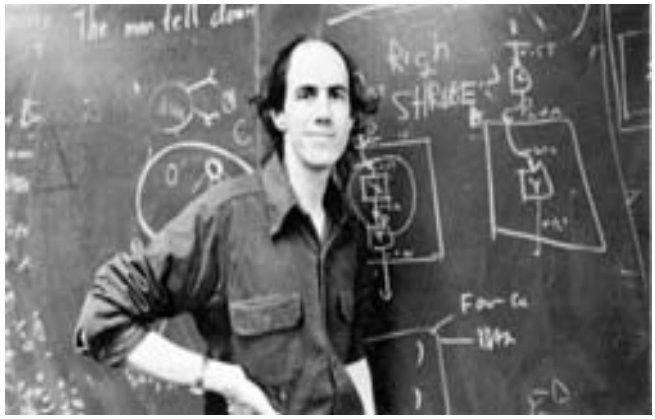
5-case:



" Enter the user's value of type characters and check if the condition is approved or not"

Part 2

Procedural Programming "Scheme"



Introduction

Scheme is a general-purpose computer programming language. It is a high-level language, supporting operations on structured data such as strings, lists, and vectors, as well as operations on more traditional data such as numbers and characters. While Scheme is often identified with symbolic applications, its rich set of data types and flexible control structures make it a truly versatile language. Scheme has been employed to write text editors, optimizing compilers, operating systems, graphics packages, expert systems, numerical applications, financial analysis packages, virtual reality systems, and practically every other type of application imaginable. Scheme is a fairly simple language to learn, since it is based on a handful of syntactic forms and semantic concepts and since the interactive nature of most implementations encourages experimentation. Scheme is a challenging language to understand fully, however; developing the ability to use its full potential requires careful study and practice.

History

Scheme started in the 1970s as an attempt to understand Carl Hewitt's Actor model, for which purpose Steele and Susana wrote a "tiny Lisp interpreter" using MacLeish and then "added mechanisms for creating actors and sending messages". Scheme was originally called "Schemer", in the tradition of other Lisp-derived languages such as Planner or

Conniver. The current name resulted from the authors' use of the ITS operating system, which limited filenames to two components of at most six characters each. Currently, "Schemer" is commonly used to refer to a Scheme programmer.

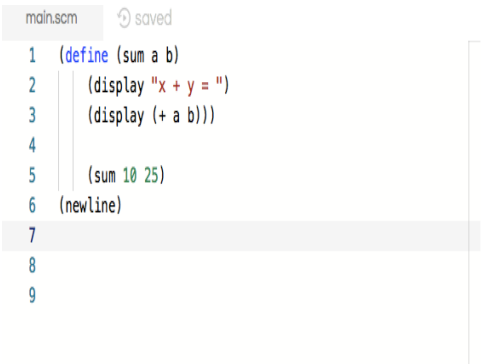
What is Schemer?

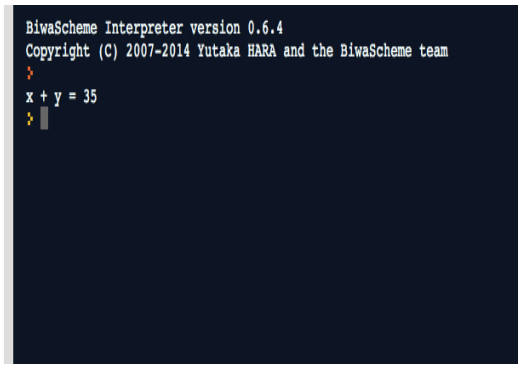
is a programming language that supports multiple paradigms, including functional and imperative programming. It is one of the three main dialects of Lisp, Unlike Common Lisp, Scheme follows a minimalist design philosophy, specifying a small standard core with powerful tools for language extension.

Some Example of Schemer language:

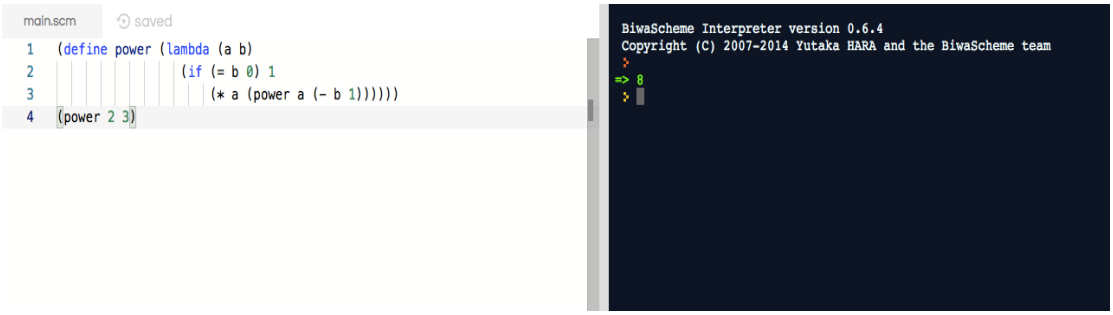
1-(define (sum a b)
(display "x + y = ")
(display (+ a b)))

(sum 10 25)
(newline)





2-(define power (lambda (a b)
(if (= b 0) 1
(* a (power a (- b 1))))))
(power 2 3)



3- (define readmax (lambda (L max)
(cond((null? L) max)
((> (car L) max) (readmax(cdr L) (car L)))

(else (readmax (cdr L) max)))))
(readmax '(89 45 67 88 99 22 44 55 65 32)10)

4- (define factorial (lambda (n result)
(cond ((= n 1) result)
(else(factorial (- n 1) (* n result))))))

(factorial 5 2)

for n >= 0, 0 <= k <= n (! is factorial and 0! = 1):
B(n, k) = n! / ((n – k)! k!)

main.scm saved

```
(define factorial (lambda (n result)
                  (cond ((= n 1) result)
                  (else(factorial (- n 1) (* n result)
                  )))))


                  (factorial 5 2)
```

```
BiwaScheme Interpreter version 0.6.4
Copyright (C) 2007-2014 Yutaka HARA and the BiwaScheme team
 >
=> 240
 >
```

5- (define min1 (lambda (L minnum)
(cond ((null? L) minnum)
((< (car L) minnum) (min1 (cdr L)(car L)))
(else(min1 (cdr L) minnum)))))

(define min (lambda (L) (min1 L (car L))))

(define max1 (lambda (L maxnum)
(cond ((null? L) maxnum)
((> (car L) maxnum) (max1 (cdr L)(car L)))
(else(max1 (cdr L) maxnum)))))

(define max (lambda (L) (max1 L (car L))))

(define minmax(lambda (L) (list (min L) (max L))))
(define examplelist (list 1 34 66 77 12 33 55 99 100))
(min examplelist)
(max examplelist)
(minmax examplelist)

main.scm saved

```
                  (car L)))
                  (else(min1 (cdr L) minnum)))))

(define min (lambda (L) (min1 L (car L))))

(define max1 (lambda (L maxnum)
                  (cond ((null? L) maxnum)
                  ((> (car L) maxnum) (max1 (cdr L)
                  (car L)))
                  (else(max1 (cdr L) maxnum)))))

(define max (lambda (L) (max1 L (car L))))

(define minmax(lambda (L) (list (min L) (max L))))
 (define examplelist (list 1 34 66 77 12 33 55 99 100))
 (min examplelist)
 (max examplelist)
( minmax examplelist)
```

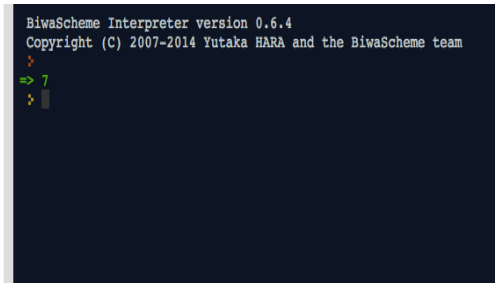
```
BiwaScheme Interpreter version 0.6.4
Copyright (C) 2007-2014 Yutaka HARA and the BiwaScheme team
 >
=> (1 100)
 >
```

6- (define hi (lambda (n)
(lambda(x) (+ n x))))

((hi 5)2)

main.scm saved

```
(define hi (lambda (n)
                (lambda(x) (+ n x))))

                ((hi 5 )2)
```

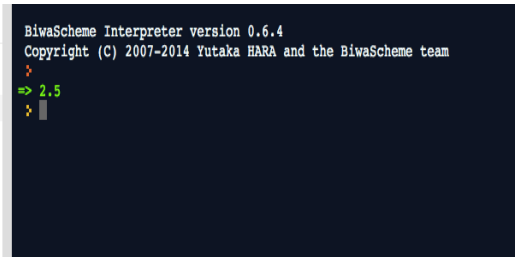


7-(define (avg lst)
(/ (apply + lst) (length lst)))

(avg '(1 2 3 4))

main.scm saved

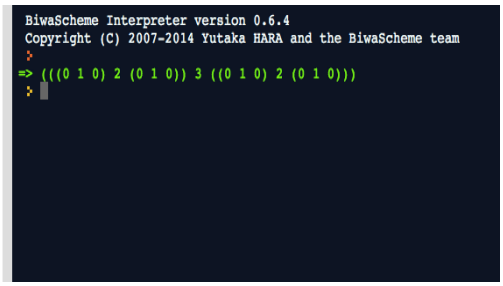
```
(define (avg lst)
  (/ (apply + lst) (length lst)))

(avg '(1 2 3 4))
```



8-(let ([tree 0])
(set! tree (list tree 1 tree))
(set! tree (list tree 2 tree))
(set! tree (list tree 3 tree))
tree)

main.scm saved

```
(let ([tree 0])
     (set! tree (list tree 1 tree))
     (set! tree (list tree 2 tree))
     (set! tree (list tree 3 tree))
     tree)
```



Part 3

Procedural Programming " Prolog"

Introduction

Prolog is a general-purpose logic programming language associated with artificial intelligence and computational linguistics.

Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is declarative: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations.
Prolog was one of the first logic programming languages, and remains the most popular among such languages today.
Prolog is well-suited for specific tasks that benefit from rule-based logical queries such as searching databases, voice control systems, and filling templates.

History

The name Prolog was chosen by Philippe Roussel as an abbreviation for programmation en logique (French for programming in logic). It was created around 1972 by Alain Colmerauer with Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses. It was

motivated in part by the desire to reconcile the use of logic as a declarative knowledge representation language with the procedural representation of knowledge that was popular in North America in the late 1960s and early 1970s.

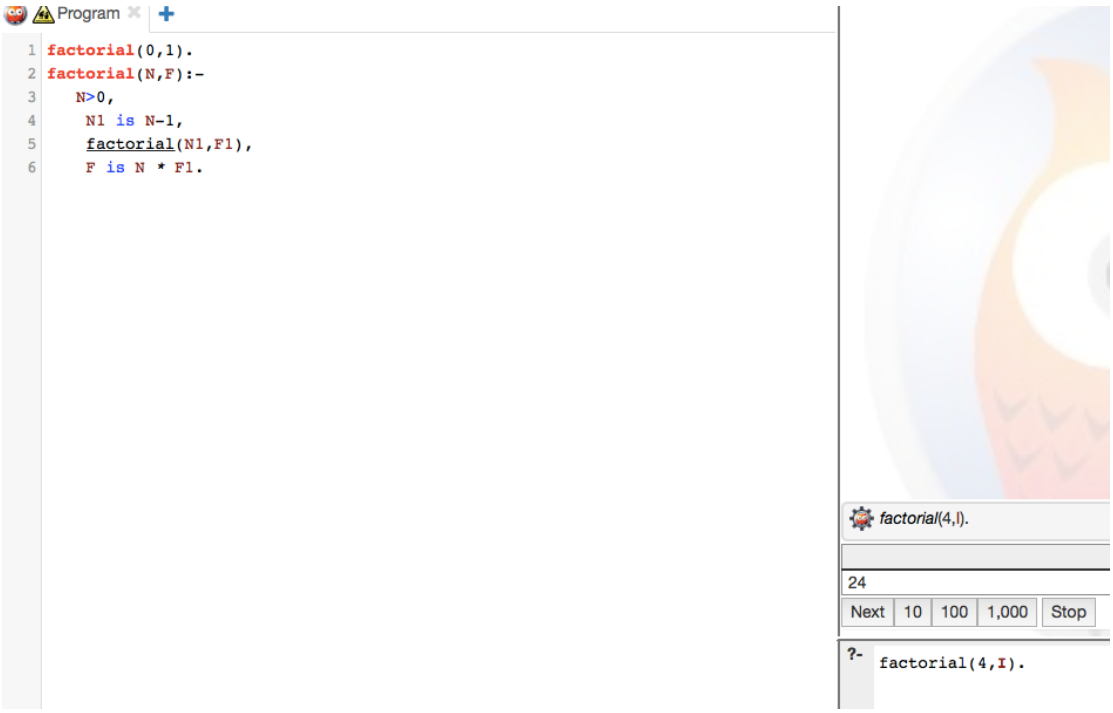
What is Prolog?

Prolog stands for programming in logic.
Prolog is the most widely used language to have been inspired by logic Programming research.
Prolog is the only successful example of the family of logic programming languages.

Some Example of Prolog language:

1- (Fac n) a function that returns the factorial of n passed in parameter:

```
factorial(0,1).
factorial(N,F):-
  N>0,
   N1 is N-1,
   factorial(N1,F1),
   F is N * F1.
```



2- fac(N,X) is provable iff X is the factorial of N:

```
factorial(0,1).
factorial(N,F):-
  N>0,
   N1 is N-1,
   factorial(N1,F1),
   F is N * F1.
```

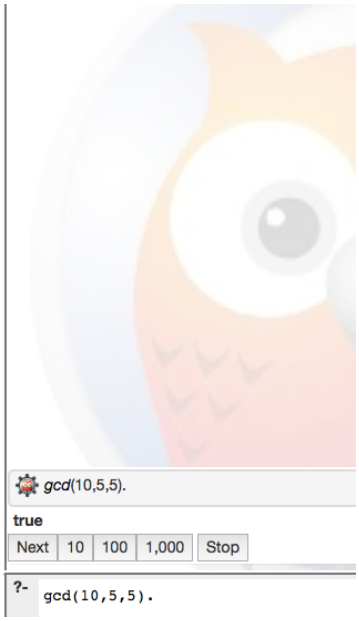
3- (GCD) return the gcd of x and y passed in parameters.

gcd(0, X, X):- X > 0,!.
gcd(X, Y, Z):- X >= Y, XI is X-Y,gcd(XI,Y,Z).
gcd(X, Y, Z):- X < Y , XI is Y-X,gcd(XI,X,Z).

```
1 gcd(0, X, X):- X > 0,!.
2 gcd(X, Y, Z):- X >= Y, XI is X-Y,gcd(XI,Y,Z).
3 gcd(X, Y, Z):- X < Y , XI is Y-X,gcd(XI,X,Z).
```

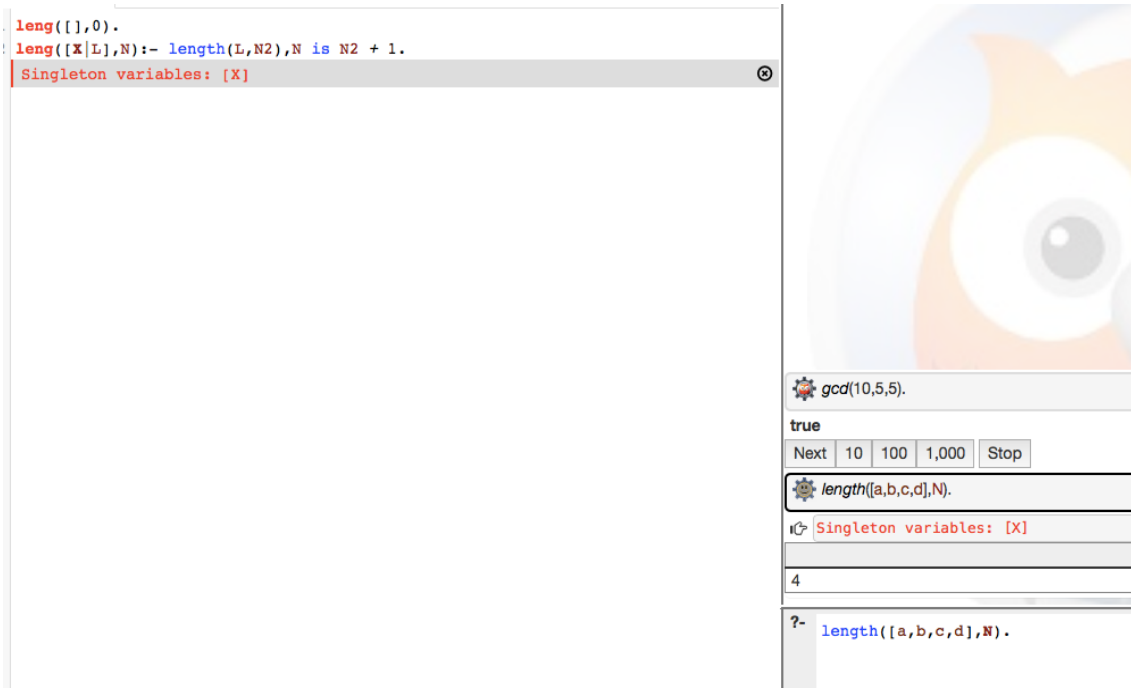
4- gcd(X,Y,Z) is provable iff Z is the gcd of X and Y:

```
gcd(0, X, X):- X > 0,!.
gcd(X, Y, Z):- X >= Y, XI is X-Y,gcd(XI,Y,Z).
gcd(X, Y, Z):- X < Y , XI is Y-X,gcd(XI,X,Z).
```



5- (length L) returns the length of the simple list L:

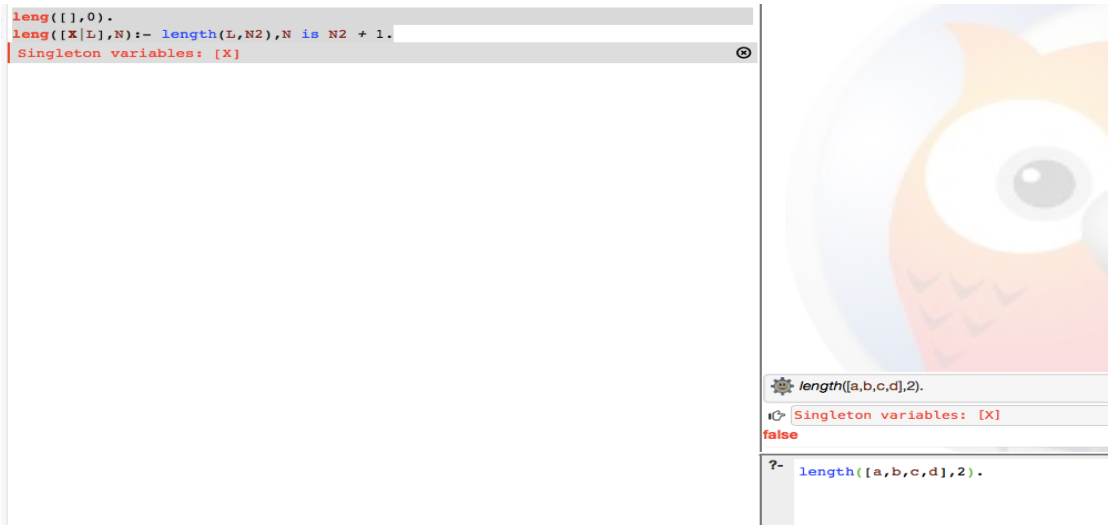
leng([],0).
leng([X|L],N):- length(L,N2),N is N2 + 1.



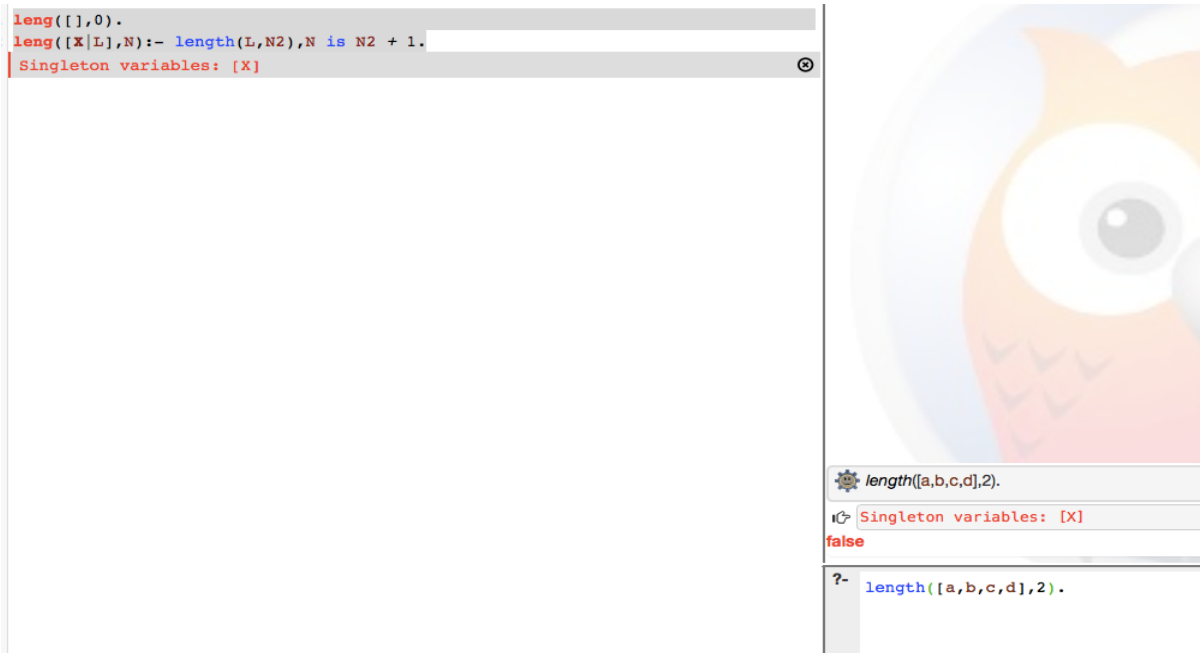
Here's a predicate length(L,N) that computes length N of a list L:
leng([],0).
leng([X|L],N) :- length(L,N2), N is N2 + 1.

Remember, [] represents the empty list, the list with no members. The first line says the empty list has length zero. The second line says that the length of any other list is just one more than the length of the list created by removing the first item. For instance:

?- length([a,b,c,d],N).
N=4

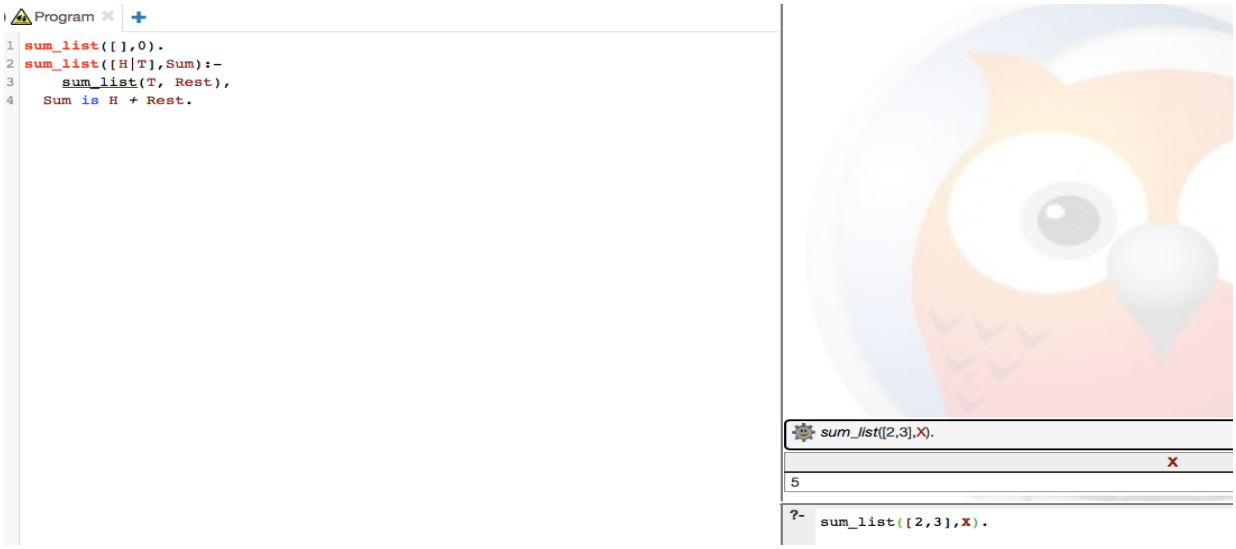


6 - length(L,X) is provable if X is the length of the list L:



7- (sum L) return the summation of the elements of the simple list L:

```
sum_list([],0).
sum_list([H|T],Sum):-
    sum_list(T, Rest),
  Sum is H + Rest.
```



8-(occurrences):

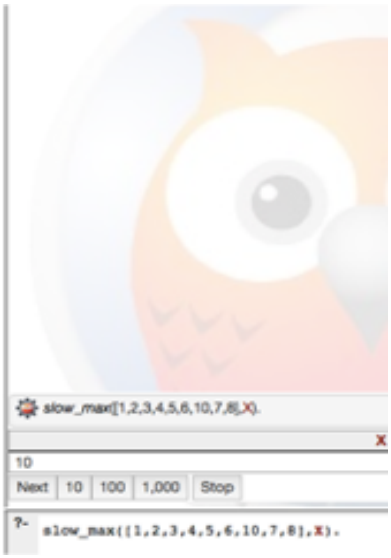
```
count_occurrences(List, Occ):-
    findall([X,L], (bagof(true,member(X,List),Xs), length(Xs,L)), Occ).
```

bagof(true,member(X,List),Xs) is satisfied for each distinct element of the list X with Xs being a list with its length equal to the number of occurrences of X in List

9- function scheme maximum number

slow_max(L, Max) :-
  select(Max, L, Rest), \+ (member(E, Rest), E > Max).

```
slow_max(L, Max) :-
   select(Max, L, Rest), \+ (member(E, Rest), E > Max).
```



10- function power

```
pow(_,0,1).
pow(X,Y,Z) :-
     pow(X,Y-1,X*Z).
```

```
pow(_,0,1).

pow(X,Y,Z) :- Y1 is Y - 1,
              pow(X,Y1,Z1), Z is Z1*X.
```

11- Function avrage
average(1,1).
average(N,X) :- K is N-1 ,K>1 , average(K,S1) , X is /(+(S1,N),N).
average(N,X) :- K is N-1 ,K=:=1 , average(K,S1) , X is +(S1,N).

```
Program
average(1,1).
        average(N,X) :- K is N-1 ,K>1 , average(K,S1) , X is /(+(S1,N),N).
        average(N,X) :- K is N-1 ,K=:=1 , average(K,S1) , X is +(S1,N).
```

12- permutation Function
is_permutation(Xs, Ys) :-
msort(Xs, Sorted),
msort(Ys, Sorted).

```
is_permutation(Xs, Ys) :-
  msort(Xs, Sorted),
  msort(Ys, Sorted).
```