Programming Project #2
Prolog Programming Assignment

Define and test the Prolog predicates described below. Each of your predicates ***must*** have the same name and signature as the examples below. Your predicates must behave properly on all instances of valid input types.

All predicates should have only *one* definition for a given number of parameters.

Your submission should consist of a single source code text file that includes all facts, predicate definitions, and propositions.

Your file should be named **`<your_net_id>.prolog`**.

You may find additional Prolog language help at the following links:

- SWI-Prolog manual
- SWI-Prolog documentation
- Learn Prolog Now!

1) Odd Multiple of 3 [10 points]

Define a predicate **`oddMultOf3/1`** that determines whether an integer is an odd multiple of 3. A user should be able to enter the predicate with an integer, e.g. **`oddMultOf3(42)`** and evaluate to either **`true`** or **`false`**. If the given parameter is not an integer, your predicate should display the message "**`ERROR: The given parameter is not an integer`**".

Examples:

```
?- oddMultOf3(171).
true.

?- oddMultOf3(100).
false.

?- oddMultOf3(12).
false.

?- oddMultOf3(4.2).
ERROR: The given parameter is not an integer

?- oddMultOf3(-9).
true.
```

2) List Product

Define a predicate **`list_prod/2`** that takes a list of numbers as a first parameter and determines the product of all of the list elements in the second parameter. Your predicate should have the signature **`list_prod(+List, +Number)`**. The product of an empty list should be zero.

Examples:

```
?- list_prod([4,3], Product).
Product = 12.

?- list_prod([7,8,0,13], Product).
Product = 0.

?- list_prod([6,2,5,10], Product).
Product = 600.

?- list_prod([], Product).
Product = 0.
```

3) Palindrome

Define a predicate **`palindrome/1`** that takes a list of numbers as a single parameter and evaluates whether the list is the same both backward and forward, i.e. a “palindrome” list. Your predicate should have the signature **`palindrome(List)`**. Note that the list to be tested may be heterogenous data types.

Examples:

```
?- palindrome([4,3,4]).
true.

?- palindrome([7,2,5,7]).
false.

?- palindrome([d,4,4,d]).
true.

?- palindrome([]).
true.

?- palindrome([a]).
true.
```

4) Second Minimum [10 points]

Define a predicate **`secondMin/2`** with the signature **`secondMin(List, Min2)`** where **`Min2`** is the second lowest *unique* valued element in some list of numbers, **`List`**. If the list has fewer than two unique elements, then your predicate should display the following, "**`ERROR: List has fewer than two unique elements.`**" If one more elements of **`List`** is not a number, then your predicate should display the following for the first encounter of a non-number element, "**`ERROR: "`*`element`*`" is not a number.`**", where ***`element`*** is the value of the non-number element. You defintion may *not* use the built-in **`sort/2`** predicate as a helper predicate. However, you may define your own **`mySort/2`**.

Examples:

```
?- secondMin([17,29,11,62,37,53], M2).
M2 = 17

?- secondMin([512], M2).
ERROR: List has fewer than two unique elements.

?- secondMin([7,5.2,3,6,-3.6,9,-2], M2).
M2 = -2

?- secondMin([12,2,b,7], M2).
ERROR: "b" is not a number.

?- secondMin([3,3,3], M2).
ERROR: List has fewer than two unique elements.
```

5) Classify

Define a predicate **`classify/3`** that takes a list of integers as an parameter and generates two lists, the first containing containing the even numbers from the original list and the second sublist containing the odd numbers from the original list. Your predicate should have the signature **`classify(List, Even, Odd)`**.

Examples:

```
?- classify([8,7,6,5,4,3], Even, Odd).
Even = [8,6,4]
Odd = [7,5,3]

?- classify([7,2,3,5,8], Even, Odd).
Even = [2,8]
Odd = [7,3,5]

?- classify([-4,11,-7,9,0], Even, Odd).
Even = [-4,0]
Odd = [11,-7,9]

?- classify([5,13,29], Even, Odd).
Even = []
Odd = [5,13,29]

?- classify([], Even, Odd).
Even = []
Odd = []
```

6) Bookends

Design a predicate **`bookends/3`** whose three parameters are all lists. the predicate tests if the first list parameter is a prefix of the third and if the second list parameter is a suffix of the third. Note that the lists in the first and second parameters may overlap.

Examples:

```
?- bookends([1],[3,4,5],[1,2,3,4,5]).
true.

?- bookends([],[4],[1,2,3,4]).
true.

?- bookends([8,7,3],[3,4],[8,7,3,4]).
true.

?- bookends([6],[9,3],[6,9,3,7]).
false.

?- bookends([],[],[2,4,6]).
true.

?- bookends([23],[23],[23]).
true.
```

7) Subslice

Design a predicate **`subslice/2`** that tests if the first list parameter is a contiguous series of elements anywhere within in the second list parameter.

Examples:

```
?- subslice([2,3,4],[1,2,3,4]).
true.

?- subslice([8,13],[3,4,8,13,7]).
true.

?- subslice([3],[1,2,4]).
false.

?- subslice([],[1,2,4]).
true.

?- subslice([1,2,4],[]).
false.
```

8) Shift

Design a predicate **`shift/3`** that "shifts" or "rotates" a list *N* places to the left. *N* may be a negative number, i.e. rotate to the right. Your predicate should have the signature **`shift(+List, +Integer, +List)`**. Note that the rotated list should be the same length as the original list.

Examples:

```
?- shift([a,b,c,d,e,f,g,h],3,Shifted).
Shifted = [d,e,f,g,h,a,b,c]

?- shift([1,2,3,4,5],1,Shifted).
Shifted = [2,3,4,5,1]

?- shift([a,b,c,d,e,f,g,h],-2,Shifted).
Shifted = [g,h,a,b,c,d,e,f]
```

9) Luhn Algorithm

Design a predicate **`luhn/1`** that is an implementation of the Luhn Algorithm and returns **`true`** if the parameter is an integer that passes the Luhn test and **`false`** otherwise.

Examples:

```
?- luhn(799273987104).
true.

?- luhn(49927398717).
false.

?- luhn(49927398716).
true.
```

10) Graph

Design *two* predicates **`path/2`** and **`cycle/1`** that determine structures within a graph whose directed edges are encoded with given instances of **`edge/2`**. For example, **`path(x,y)`** should evaluate to **`true`** if a path exists from vertex **`x`** to vertex **`y`**, and **`false`** otherwise. And **`cycle(x)`** should evaluate to **`true`** if a cycle exists which includes vertex **`x`**.

Note: All edges are directional.

Note: Your solution should avoid infinite recursion.

Note: The Knowledge Base of edges below is for example only. You are just responsible for the definitions of **`path/2`** and **`cycle/1`**. The Knowledge Base edges used for grading will be different.

Examples:

```
% Knowledge Base
edge(a,b).
edge(b,c).
edge(c,d).
edge(d,a).
edge(d,e).
edge(b,a).

?- path(b,d)
true.

?- path(e,b).
false.

?- path(c,a).
true.

?- cycle(b).
true.

?- cycle(e).
false.
```