Started on	Tuesday, 18 August 2020, 3:28 AM
State	Finished
Completed on	Friday, 21 August 2020, 5:06 PM
Time taken	3 days 13 hours
Marks	6.80/7.00
Grade	**97.14** out of 100.00

Question **1**

Correct

Mark 1.00 out of 1.00

Write a predicate `second(?List,?X)` which succeeds when `X` is the second element of `List`.

Notes

- The definition implies that the predicate should fail if the list has fewer than two elements.
- The notation implies, either of the arguments can be instantiated or unbound (input or output).
- In some test cases, you see the symbol `\+` which means negation. For example, the goal `\+ second([1], X)` will succeed if `second([1], X)` fails.

For example:

Test	Result

`test_answer :-`
`  second([cosc, 2, Var, beethoven], X),`
`  writeln(X).`
	`2`

`test_answer :-`
`  \+ second([1], X),`
`  writeln('OK').`
 	
`test_answer :-`
`  second([_],_),`
`  writeln('The predicate should fail on lists of length one!').`
	`OK`

`test_answer :-`
`  second([a, b, c, d], b),`
`  writeln('OK').`
	`OK`

`test_answer :-`
`  second(L, X),`
`  writeln('OK').`
	`OK`

Answer: (penalty regime: 0, 15, ... %)

```
1 second([_,X|_],Y) :- X = Y.
```

	Test	Expected	Got	

✔	`test_answer :-`
	`  second([cosc, 2, Var, beethoven], X),`
	`  writeln(X).`
		`2`	`2`	✔

✔	`test_answer :-`
	`  \+ second([1], X),`
	`  writeln('OK').`
	
	`test_answer :-`
	`  second([_],_),`
	`  writeln('The predicate should fail on lists of length one!').`
		`OK`	`OK`	✔

✔	`test_answer :-`
	`  second([a, b, c, d], b),`
	`  writeln('OK').`
		`OK`	`OK`	✔

✔	`test_answer :-`
	`  second(L, X),`
	`  writeln('OK').`
		`OK`	`OK`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **2**

Correct

Mark 1.00 out of 1.00

Write a predicate `swap12(?List1, ?List2)` which succeeds when List1 is identical to List2, except that the first two elements are exchanged. The predicate must fail on lists with fewer than two elements. Note that either of the arguments can be bound or unbound (input or output).

For example:

Test	Result

`test_answer :-`
`  swap12([a, b, c, d], L),`
`  writeln(L).`
	`[b,a,c,d]`

`test_answer :-`
`  \+ swap12(L, [1]),`
`  writeln('OK').`
	`OK`

`test_answer :-`
`  swap12(L, [b, a]),`
`  writeln(L).`
	`[a,b]`

`test_answer :-`
`  swap12(L1, L2),`
`  writeln('OK').`
	`OK`

Answer: (penalty regime: 0,15,... %)

```
1 swap12([Q,R|L1], [X,Y|L2]) :- Q = Y, R = X, L1 = L2.
```

	Test	Expected	Got	

✔	`test_answer :-`
	`  swap12([a, b, c, d], L),`
	`  writeln(L).`
		`[b,a,c,d]`	`[b,a,c,d]`	✔

✔	`test_answer :-`
	`  \+ swap12(L, [1]),`
	`  writeln('OK').`
		`OK`	`OK`	✔

✔	`test_answer :-`
	`  swap12(L, [b, a]),`
	`  writeln(L).`
		`[a,b]`	`[a,b]`	✔

✔	`test_answer :-`
	`  swap12(L1, L2),`
	`  writeln('OK').`
		`OK`	`OK`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **3**

Correct

Mark 1.00 out of 1.00

Suppose we are given a knowledge base with the following facts:

```
tran(eins,one).
tran(zwei,two).
tran(drei,three).
tran(vier,four).
tran(fuenf,five).
tran(sechs,six).
tran(sieben,seven).
tran(acht,eight).
tran(neun,nine).
```

Write a predicate `listtran(?G, ?E)` which translates a list of German number words to/from the corresponding list of English number words. For example:

```
listtran([eins,neun,zwei],X).
```

should give:

```
X = [one, nine, two].
```

Your program should also work in the other direction. For example, the query

```
listtran(X, [one, seven, six, two]).
```

should succeed with

```
X = [eins, sieben, sechs, zwei].
```

Hint: to answer this question, first ask yourself “How do I translate the empty list of number words?”. That’s the base case. For non-empty lists, first translate the head of the list, then use recursion to translate the tail.

For example:

Test	Result

`tran(eins,one).`
`tran(zwei,two).`
`tran(drei,three).`
`tran(vier,four).`
`tran(fuenf,five).`
`tran(sechs,six).`
`tran(sieben,seven).`
`tran(acht,eight).`
`tran(neun,nine).`
 	
`test_answer :-`
`  listtran([eins, neun, zwei], X),`
`  writeln(X).`
	`[one,nine,two]`

`test_answer :-`
`  listtran([], []),`
`  writeln('OK').`
	`OK`

`tran(eins,one).`
`tran(zwei,two).`
`tran(drei,three).`
`tran(vier,four).`
`tran(fuenf,five).`
`tran(sechs,six).`
`tran(sieben,seven).`
`tran(acht,eight).`
`tran(neun,nine).`
 	
`test_answer :-`
`  listtran(X, [one, seven, six, two]),`
`  writeln(X).`
	`[eins,sieben,sechs,zwei]`

`test_answer :-`
`  listtran(L1, L2),`
`  writeln('OK').`
	`OK`

`tran(1, one).`
`tran(2, two).`
`tran(3, three).`
 	
`test_answer :-`
`  listtran([1, 2, 3], X),`
`  writeln(X).`
	`[one,two,three]`

Answer: (penalty regime: 0,15,... %)

```
listtran([], []).
listtran([X1|Y1], [X2|Y2]) :- tran(X1, X2), listtran(Y1, Y2).
```

	Test	Expected	Got	

✔	`tran(eins,one).`
	`tran(zwei,two).`
	`tran(drei,three).`
	`tran(vier,four).`
	`tran(fuenf,five).`
	`tran(sechs,six).`
	`tran(sieben,seven).`
	`tran(acht,eight).`
	`tran(neun,nine).`
	
	`test_answer :-`
	`  listtran([eins, neun, zwei], X),`
	`  writeln(X).`
		`[one,nine,two]`	`[one,nine,two]`	✔

✔	`test_answer :-`
	`  listtran([], []),`
	`  writeln('OK').`
		`OK`	`OK`	✔

✔	`tran(eins,one).`
	`tran(zwei,two).`
	`tran(drei,three).`
	`tran(vier,four).`
	`tran(fuenf,five).`
	`tran(sechs,six).`
	`tran(sieben,seven).`
	`tran(acht,eight).`
	`tran(neun,nine).`
	
	`test_answer :-`
	`  listtran(X, [one, seven, six, two]),`
	`  writeln(X).`
		`[eins,sieben,sechs,zwei]`	`[eins,sieben,sechs,zwei]`	✔

✔	`test_answer :-`
	`  listtran(L1, L2),`
	`  writeln('OK').`
		`OK`	`OK`	✔

✔	`tran(1, one).`
	`tran(2, two).`
	`tran(3, three).`
	
	`test_answer :-`
	`  listtran([1, 2, 3], X),`
	`  writeln(X).`
		`[one,two,three]`	`[one,two,three]`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **4**

Correct

Mark 1.00 out of 1.00

Write a predicate `twice(?In, ?Out)` whose left argument is a list, and whose right argument is a list consisting of every element in the left list repeated twice. For example, the query

```
twice([a,4,buggle],X).
```

gives

```
X = [a,a,4,4,buggle,buggle].
```

and the query

```
twice(X, [1, 1, 2, 2]).
```

gives

```
X = [1,2].
```

and the query

```
twice(X, [a, a, b, b, c]).
```

fails.

Hint: to answer this question, first ask yourself “What should happen when the first argument is the empty list?”. That’s the base case. For non-empty lists, think about what you should do with the head, and use recursion to handle the tail.

For example:

Test	Result

`test_answer :-`
`  twice([a, b, c, d], L),`
`  writeln(L).`
	`[a,a,b,b,c,c,d,d]`

`test_answer :-`
`  twice(L, [1, 1, 2, 2, 3, 3]),`
`  writeln(L).`
	`[1,2,3]`

`test_answer :-`
`  twice([], []),`
`  writeln('OK').`
	`OK`

`test_answer :-`
`  twice(L1, L2),`
`  writeln('OK').`
	`OK`

`test_answer :-`
`  \+ twice(L, [a, a, b]),`
`  writeln('OK').`
	`OK`

Answer: (penalty regime: 0,15,... %)

```
1 twice([], []).
2 twice([X|L1], [Y,Z|L2]) :-
3 Y = X, Z = X, twice(L1, L2).
4
```

	Test	Expected	Got	

	Test	Expected	Got	

✔	`test_answer :-`
	`  twice([a, b, c, d], L),`
	`  writeln(L).`
		`[a,a,b,b,c,c,d,d]`	`[a,a,b,b,c,c,d,d]`	✔

✔	`test_answer :-`
	`  twice(L, [1, 1, 2, 2, 3, 3]),`
	`  writeln(L).`
		`[1,2,3]`	`[1,2,3]`	✔

✔	`test_answer :-`
	`  twice([], []),`
	`  writeln('OK').`
		`OK`	`OK`	✔

✔	`test_answer :-`
	`  twice(L1, L2),`
	`  writeln('OK').`
		`OK`	`OK`	✔

✔	`test_answer :-`
	`  \+ twice(L, [a, a, b]),`
	`  writeln('OK').`
		`OK`	`OK`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **5**

Correct

Mark 1.00 out of 1.00

Write a predicate `remove(+X, +ListIn, ?ListOut)` that succeeds if *ListOut* can be obtained by removing all instances of *X* from *ListIn*. Note that the first two arguments will always be bound (input argument).

For example:

Test	Result

`test_answer :-`
`  remove(a, [a, b, a, c, d, a, b], L),`
`  writeln(L).`
	`[b,c,d,b]`

`test_answer :-`
`  remove(2, [2], L),`
`  writeln(L).`
	`[]`

`test_answer :-`
`  remove(d, [a, b, c], L),`
`  write(L).`
	`[a,b,c]`

`test_answer :-`
`  remove(a, [], L),`
`  write(L).`
	`[]`

`test_answer :-`
`  remove(term2, [term1, term2, term3], [term1, term3]),`
`  write('OK').`
	`OK`

Answer: (penalty regime: 0,15,... %)

```
remove(_, [], []).
remove(X, [H1|T1], L) :- H1 = X, remove(X, T1, L), !.
remove(X, [H1|T1], [H2|T2]):- H1 = H2, remove(X, T1, T2 ).
```

	Test	Expected	Got	

✔	`test_answer :-`
	`  remove(a, [a, b, a, c, d, a, b], L),`
	`  writeln(L).`
		`[b,c,d,b]`	`[b,c,d,b]`	✔

✔	`test_answer :-`
	`  remove(2, [2], L),`
	`  writeln(L).`
		`[]`	`[]`	✔

✔	`test_answer :-`
	`  remove(d, [a, b, c], L),`
	`  write(L).`
		`[a,b,c]`	`[a,b,c]`	✔

✔	`test_answer :-`
	`  remove(a, [], L),`
	`  write(L).`
		`[]`	`[]`	✔

✔	`test_answer :-`
	`  remove(term2, [term1, term2, term3], [term1, term3]),`
	`  write('OK').`
		`OK`	`OK`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **6**

Correct

Mark 1.00 out of 1.00

Write a predicate `split_odd_even(+ListIn, ?ListA, ?ListB)` whose first argument is a list, and whose second and third arguments are the odd and even indexed elements in that list respectively. Assume the first element of a list is indexed 1.

For example:

Test	Result

`test_answer :-`
`  split_odd_even([a,b,c,d,e,f,g], A, B),`
`  write(A),`
`  writeln(B).`
	`[a,c,e,g][b,d,f]`

`test_answer :-`
`  split_odd_even([1,2,3,5], A, B),`
`  write(A),`
`  writeln(B).`
	`[1,3][2,5]`

Answer: (penalty regime: 0,15,... %)

```
1 split_odd_even([], [], []).
2 split_odd_even([H], [H], []).
3 split_odd_even([H1, H2|T], [H1|T1], [H2|T2]) :- split_odd_even(T, T1, T2).
```

	Test	Expected	Got	

✔	`test_answer :-`
	`  split_odd_even([a,b,c,d,e,f,g], A, B),`
	`  write(A),`
	`  writeln(B).`
		`[a,c,e,g][b,d,f]`	`[a,c,e,g][b,d,f]`	✔

✔	`test_answer :-`
	`  split_odd_even([1,2,3,5], A, B),`
	`  write(A),`
	`  writeln(B).`
		`[1,3][2,5]`	`[1,3][2,5]`	✔

✔	`test_answer :-`
	`  split_odd_even([d], A, B),`
	`  write(A),`
	`  writeln(B).`
		`[d][]`	`[d][]`	✔

✔	`test_answer :-`
	`  split_odd_even([], A, B),`
	`  write(A),`
	`  writeln(B).`
		`[][]`	`[][]`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Question **7**

Correct

Mark 0.80 out of 1.00

Write a predicate `preorder(+Tree, Traversal)` that determines the preorder traversal of a given binary tree. Each tree/subtree is either a `leaf` or of the form `tree(root, left_subtree, right_subtree)`. A preorder traversal records the current node, then the left branch, then the right branch.

For example:

Test	Result
`test_answer :- preorder(leaf(a), L),` `writeln(L).`	`[a]`
`test_answer :- preorder(tree(a, tree(b, leaf(c), leaf(d)), leaf(e)), T),` `writeln(T).`	`[a,b,c,d,e]`

Answer: (penalty regime: 10, 20, ... %)

```
1 preorder(tree(A, B, C), [A|X]) :- preorder(B, B1), preorder(C, C1), append(B1,C1,X).
2 preorder(leaf(D), [D]).
```

	Test	Expected	Got	
✔	`test_answer :- preorder(leaf(a), L),` `writeln(L).`	`[a]`	`[a]`	✔
✔	`test_answer :- preorder(tree(a, tree(b, leaf(c), leaf(d)), leaf(e)), T),` `writeln(T).`	`[a,b,c,d,e]`	`[a,b,c,d,e]`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.80/1.00**.

◄ 4. Declarative programming with Prolog (i)

Jump to...

6. Constraint satisfaction problems ►