Lesson 5: List Processing in Prolog

What's It All About?

1. The basic syntax of the list in Prolog is presented.
2. The pattern matching mechanism frequently used in Prolog to deconstruct/construct lists, known as "head/tail" notation, is introduced.
3. Some list processing programs are written and demoed and discussed.
4. Some list processing exercises are provided.

Representing Lists in Prolog

Lists are represented syntactically in Prolog by placing terms, separated by commas, within square brackets. Some examples ...

1. `[red, yellow, blue]`
2. `[]`
3. `["Desde El Alma", "Poema", "Esta Noche de Luna"]`
4. `[card(jack,heart), card(2,club), card(queen,heart), card(ace,spade), card(10,diamond)]`

Head/Tail Notation

`[Head|Tail]` represents a list with the first element (the **car**) represented by `Head` and the list containing the rest of the elements (the **cdr**) represented by `Tail`.

Some Prolog for thought: What will each variable be bound to as a result of the following Prolog list equations?

1. `[H|T] = [red,yellow,blue]`

2. `[First|Rest] = [one,two]`

3. `[F|R] = [cat]`

4. `[A|[B|[C]]] = [efx(red,rouge), efx(yellow,jaun), efx(blue,bleu,bleue)]`

Head/Tail Referencing Exercises

In the interest of becoming comfortable with head/tail notation, please read through the following redacted interaction, and write down, for each query, how you think that Prolog will respond in terms of variable bindings and result. Then, after you have done your best to reason your way to responses based on your understanding of the head/tail mechanism, perform the demo. Compare your results, and do your best to sort out any misunderstandings that you might have had when mentally doing the exercises.

```
bash-3.2$ swipl
<<redacted>>

?- [H|T] = [red, yellow, blue, green].
<<redacted>>

?- [H, T] = [red, yellow, blue, green].
<<redacted>>

?- [F|_] = [red, yellow, blue, green].
<<redacted>>

?- [_|[S|_]] = [red, yellow, blue, green].
<<redacted>>

?- [F|[S|R]] = [red, yellow, blue, green].
<<redacted>>

?- List = [this|[and, that]].
<<redacted>>

?- List = [this, and, that].
<<redacted>>

?- [a,[b, c]] = [a, b, c].
<<redacted>>

?- [a|[b, c]] = [a, b, c].
<<redacted>>

?- [cell(Row,Column)|Rest] = [cell(1,1), cell(3,2), cell(1,3)].
<<redacted>>

?- [X|Y] = [one(un, uno), two(dos, deux), three(trois, tres)].
<<redacted>>

?-
```

Example List Processors

Demo

```
bash-3.2$ swipl
<<redacted>>

?- consult('list_processors.pro').
% list_processors.pro compiled 0.00 sec, 45 clauses
true.

?- first([apple],First).
First = apple.

?- first([c,d,e,f,g,a,b],P).
P = c.

?- rest([apple],Rest).
Rest = [].

?- rest([c,d,e,f,g,a,b],Rest).
Rest = [d, e, f, g, a, b].

?- last([peach],Last).
Last = peach

?- last([c,d,e,f,g,a,b],P).
P = b

?- nth(0,[zero,one,two,three,four],Element).
Element = zero

?- nth(3,[four,three,two,one,zero],Element).
Element = one

?- writelist([red,yellow,blue,green,purple,orange]).
red
yellow
blue
green
purple
orange
true.

?- sum([],Sum).
Sum = 0.

?- sum([2,3,5,7,11],SumOfPrimes).
SumOfPrimes = 28.
```

```
?- add_first(thing,[],Result).
Result = [thing].

?- add_first(racket,[prolog,haskell,rust],Languages).
Languages = [racket, prolog, haskell, rust].

?- add_last(thing,[],Result).
Result = [thing]

?- add_last(rust,[racket,prolog,haskell],Languages).
Languages = [racket, prolog, haskell, rust]

?- iota(5,Iota5).
Iota5 = [1, 2, 3, 4, 5]

?- iota(9,Iota9).
Iota9 = [1, 2, 3, 4, 5, 6, 7, 8, 9]

?- pick([cherry,peach,apple,blueberry],Pie).
Pie = cherry

?- pick([cherry,peach,apple,blueberry],Pie).
Pie = cherry

?- pick([cherry,peach,apple,blueberry],Pie).
Pie = apple

?- pick([cherry,peach,apple,blueberry],Pie).
Pie = apple

?- pick([cherry,peach,apple,blueberry],Pie).
Pie = blueberry

?- pick([cherry,peach,apple,blueberry],Pie).
Pie = blueberry

?- pick([cherry,peach,apple,blueberry],Pie).
Pie = blueberry

?- pick([cherry,peach,apple,blueberry],Pie).
Pie = cherry

?- make_set([1,1,2,1,2,3,1,2,3,4],Set).
Set = [1, 2, 3, 4]

?- make_set([bit,bot,bet,bot,bot,bit],B).
B = [bet, bot, bit]

?-
```

Code: First

```
first([H|_], H).
```

Code: Rest

```
rest([_|T], T).
```

Code: Last

```
last([H|[]], H).
last([_|T], Result) :- last(T, Result).
```

Code: Nth

```
nth(0,[H|_],H).
nth(N,[_|T],E) :- K is N - 1, nth(K,T,E).
```

Code: Writelist

```
writelist([]).
writelist([H|T]) :- write(H), nl, writelist(T).
```

Code: sum

```
sum([],0).
sum([Head|Tail],Sum) :-
  sum(Tail,SumOfTail),
  Sum is Head + SumOfTail.
```

Code: Add_first

```
add_first(X,L,[X|L]).
```

Code: Add_last

```
add_last(X,[],[X]).
add_last(X,[H|T],[H|TX]) :- add_last(X,T,TX).
```

Code: Iota

```
iota(0,[]).
iota(N,IotaN) :-
  K is N - 1,
  iota(K,IotaK),
  add_last(N,IotaK,IotaN).
```

Code: Pick

```
pick(L,Item) :-
  length(L,Length),
  random(0,Length,RN),
  nth(RN,L,Item).
```

Code: Make_set

```
make_set([],[]).
make_set([H|T],TS) :-
  member(H,T),
  make_set(T,TS).
make_set([H|T],[H|TS]) :-
  make_set(T,TS).
```

List Processing Exercises

The list processing exercises posed in this section are suggested by a reading of the following demo. By completing the exercises, you will be in a position to run a demo quite like that which appears below. Big picture, defining the predicates, and generating a demo quite like the following demo, is what you are being asked to do in this section of the lesson.

In short, please do the following:

1. Read the demo, doing your best to make sticky in your mind what the various predicates are intended to do.
2. Write the predicates according to the specifications provided. (Please read the specifications, even if you think that you have a good idea of what the predicates are intended to do.)
3. Recreate the demo, with the understanding that the parts in which nondeterminism plays a role will be rendered differently from just what you see in the given demo.

Demo

```
bash-3.2$ swipl
<<redacted>>

?- consult('list_processors.pro').
% list_processors.pro compiled 0.00 sec, 45 clauses
true.

?- product([],P).
P = 1.

?- product([1,3,5,7,9],Product).
Product = 945.

?- iota(9,Iota),product(Iota,Product).
Iota = [1, 2, 3, 4, 5, 6, 7, 8, 9],
Product = 362880

?- make_list(7,seven,Seven).
Seven = [seven, seven, seven, seven, seven, seven, seven]

?- make_list(8,2,List).
List = [2, 2, 2, 2, 2, 2, 2, 2]

?- ?- but_first([a,b,c],X).
X = [b, c].

?- but_last([a,b,c,d,e],X).
X = [a, b, c, d]
```

```
?- is_palindrome([x]).
true

?- is_palindrome([a,b,c]).
false.

?- is_palindrome([a,b,b,a]).
true

?- is_palindrome([1,2,3,4,5,4,2,3,1]).
false

?- is_palindrome([c,o,f,f,e,e,e,e,f,f,o,c]).
true

?- noun_phrase(NP).
NP = [the, blue, piano] ;
false.

?- noun_phrase(NP).
NP = [the, electric, flag]

?- noun_phrase(NP).
NP = [the, silver, flag]

?- noun_phrase(NP).
NP = [the, tiny, light]

?- noun_phrase(NP).
NP = [the, unlucky, moon]

?- sentence(S).
S = [the, postmodern, banana, skimmed, the, blue, car]

?- sentence(S).
S = [the, electric, piano, played, the, silver, flag]

?- sentence(S).
S = [the, silver, piano, ate, the, blue, hat]

?- sentence(S).
S = [the, blue, piano, dimmed, the, tiny, hat]

?- sentence(S).
S = [the, blue, hat, saw, the, tiny, robot]

?- sentence(S).
S = [the, silver, flag, played, the, blue, flag]

?- sentence(S).
S = [the, electric, car, tipped, the, tiny, book]

?- sentence(S).
```

```
S = [the, postmodern, banana, tipped, the, silver, moon]

?- sentence(S).
S = [the, unlucky, piano, ate, the, silver, car]

?- sentence(S).
S = [the, silver, moon, skimmed, the, unlucky, dress]

?- sentence(S).
S = [the, tiny, moon, dimmed, the, unlucky, robot]

?- sentence(S).
S = [the, electric, banana, dimmed, the, unlucky, flag]

?- sentence(S).
S = [the, postmodern, robot, admired, the, tiny, piano]

?- sentence(S).
S = [the, unlucky, book, dimmed, the, electric, hat]

?- sentence(S).
S = [the, postmodern, flag, tipped, the, unlucky, moon]

?- sentence(S).
S = [the, blue, hat, skimmed, the, tiny, banana]

?-
```

Specifications

1. Define the order 2 predicate called `product` which takes a list of numbers as an input parameter and produces the product of the numbers in the list as an output parameter. Hint: Let the `sum` predicate presented in the "Examples" section of this lesson be your guide.
2. Define the order 2 predicate called `factorial` which takes a positive integer as an input parameter and produces the factorial of the given number as an output parameter. Constraint: Make use of the `iota` predicate defined in the "Examples" section of this lesson along with the `product` predicate.
3. Define the order 3 predicate called `make_list` which takes a nonnegative integer as its first input parameter, a data item as its second input parameter, and which produces a list consisting of the specified number of occurrences of the specified piece of data as its output parameter. Constraint: Write this definition recursively.
4. Define the order 2 predicate called `but_first` which takes a non-empty list as input parameter and produces the "cdr" of the list as its output parameter. Hint: First consisder the case in which the input list is a singleton, regardless of just what the sole element happens to be. Next, consider a longer list.
5. Define the order 2 predicate called `but_last` which takes a non-empty list as input parameter and produces the "rdc" of the list as its output parameter. Constraint: Make use of the primitive predicate `reverse` two times in defining this predicate. (If you can't guess how `reverse` works, just Google it.)
6. Define the order 1 predicate called `is_palindrome` which takes a list as its sole parameter and succeeds only if the list is a palindrome. Constraints: (1) Use `first` and `last` and `but_first` and `but_last` in writing this function. (2) Use recursion.

7. Define an order 1 predicate called `noun_phrase` which produces a noun phrase of length three consisting of the word "the" followed by an adjective selected at random from a list of six adjectives followed by a noun selected at random from a list of eight nouns. Constraints: (1) Come up with your own list of six adjectives and eight nouns, rather than use mine. (2) Use the `pick` predicate from the "Exercises" section of this lesson.
8. Define an order 1 predicate called `sentence` which produces a sentence consisting of a random noun phrase followed by a past tense verb followed by a noun phrase. Constraints: (1) Use the tokennoun_phrase predicate as well as the `pick` predicate. (2) Come up with 7 past tense verbs on which to draw.