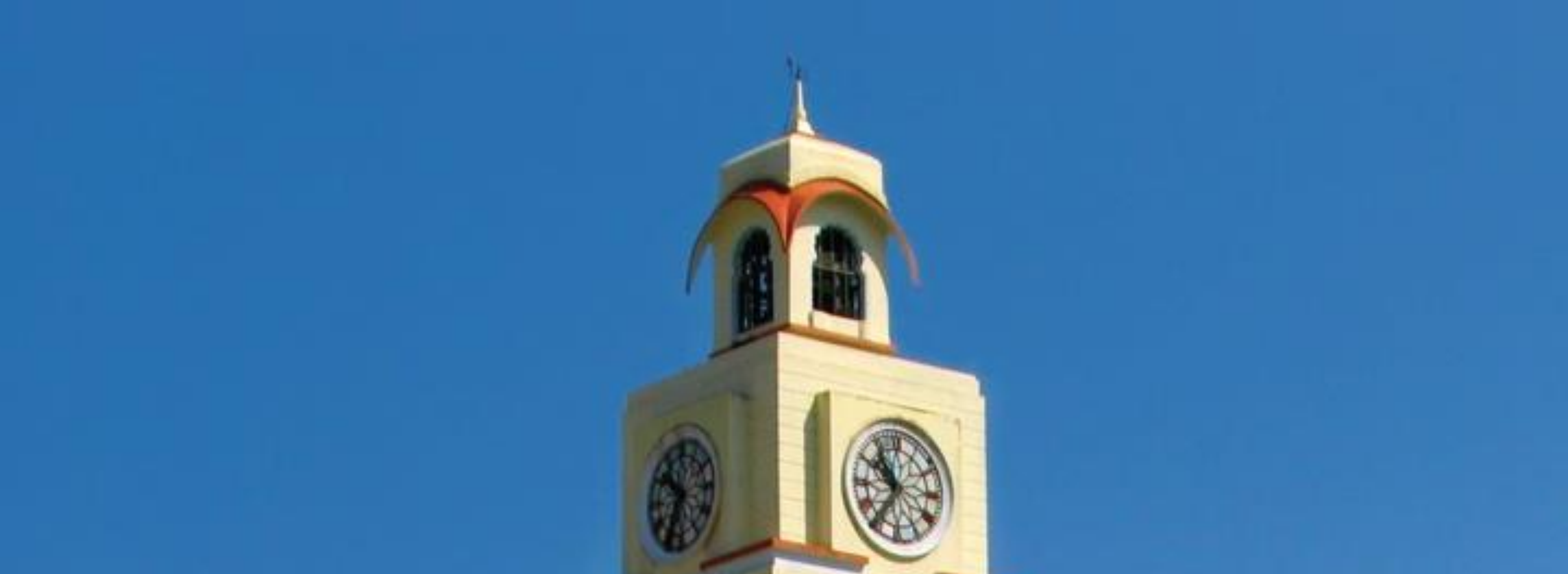


# CS F214 - Logic in CS
# Prolog – Lecture 3

**BITS** Pilani
Pilani Campus

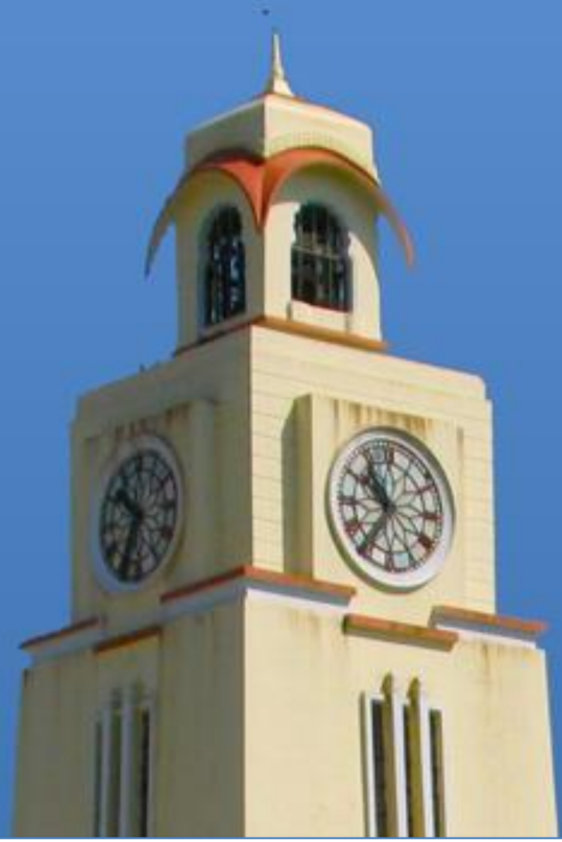
Jagat Sesh Challa

# Today’s Lecture

- Arithmetic and Lists
- Append/2
- Reverse/2 & Reverse/3

innovate achieve lead

BITS Pilani
Pilani Campus



# Arithmetic and Lists

# Arithmetic and Lists

- How long is a list?
  - The empty list has length: zero;
  - A non-empty list has length: one plus length of its tail.

# Length of a list in Prolog

```
len([],0).
len([_|L],N):-
    len(L,X),
    N is X + 1.
```

```
?- len([a,b,c,d,e,[a,x],t],X).
X=7
yes
?-
```

# Accumulators

- This is quite a good program
  - Easy to understand
  - Relatively efficient
- But there is another method of finding the length of a list
  - Accumulators
  - Accumulators are variables that hold intermediate results

# Defining acclen/3

- The predicate **acclen/3** has three arguments
  - The list whose length we want to find
  - The length of the list, an integer
  - An accumulator, keeping track of the intermediate values for the length

# Defining acclen/3

- The accumulator of **acclen/3**
  - Initial value of the accumulator is 0
  - Add 1 to accumulator each time we can recursively take the head of a list
  - When we reach the empty list, the accumulator contains the length of the list

# Length of a list in Prolog

```
acclen([],Acc,Length):-
    Length = Acc.

acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

```
?-
```

# Length of a list in Prolog

```
acclen([],Acc,Length):-
    Length = Acc.

acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

add 1 to the accumulator each time we take off a head from the list

```
?-
```

# Length of a list in Prolog

```
acclen([],Acc,Length):-
    Length = Acc.

acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

When we reach the empty list, the accumulator contains the length of the list

```
?-
```

# Length of a list in Prolog

```
acclen([],Acc,Acc).

acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

```
?-
```

# Length of a list in Prolog

```
acclen([],Acc,Acc).

acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

```
?-acclen([a,b,c],0,Len).
Len=3
yes

?-
```

# Search Tree for acclen/3

```
acclen([ ],Acc,Acc).

acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

```
?- acclen([a,b,c],0,Len).
     /            \
   no        ?- acclen([b,c],1,Len).
                 /              \
               no         ?- acclen([c],2,Len).
                              /               \
                            no          ?- acclen([],3,Len).
                                            /            \
                                          Len=3           no
```

# Adding a Wrapper Predicate

innovate achieve lead

```
acclen([ ],Acc,Acc).

acclen([ _|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).

length(List,Length):-
     acclen(List,0,Length).
```

```
?-length([a,b,c], X).
X=3
yes
```

# Tail recursion

- Why is acclen/3 better than len/2 ?
  - acclen/3 is tail-recursive, and len/2 is not
- Difference:
  - In tail recursive predicates the results is fully calculated once we reach the base clause
  - In recursive predicates that are not tail recursive, there are still goals on the stack when we reach the base clause

# Comparision

*Not tail-recursive*

```
len([],0).
len([_|L],NewLength):-
    len(L,Length),
    NewLength is Length + 1.
```

*Tail-recursive*

```
acclen([],Acc,Acc).
acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

# Search Tree for len/2

```
?- len([a,b,c], Len).
    /              \
  no    ?- len([b,c],Len1),
           Len is Len1 + 1.
           /                \
        no       ?- len([c], Len2),
                    Len1 is Len2+1,
                    Len is Len1+1.
                  /               \
               no         ?- len([], Len3),
                             Len2 is Len3+1,
                             Len1 is Len2+1,
                             Len is Len1 + 1.
                             /                      \
          Len3=0, Len2=1,                        no
           Len1=2, Len=3
```

```
len([],0).
len([_|L],NewLength):-
    len(L,Length),
    NewLength is Length + 1.
```

# Search Tree for acclen/3

```
?- acclen([a,b,c],0,Len).
    /              \
  no         ?- acclen([b,c],1,Len).
                 /               \
               no         ?- acclen([c],2,Len).
                              /                \
                            no          ?- acclen([],3,Len).
                                            /              \
                                         Len=3              no
```

```
acclen([ ],Acc,Acc).

acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

# Comparing Integers

- Some Prolog arithmetic predicates actually do carry out arithmetic by themselves
- These are the operators that compare integers

# Comparing Integers

| Arithmetic | Prolog |
|---|---|
| $x < y$ | X < Y |
| $x \leq y$ | X =< Y |
| $x = y$ | X =:= Y |
| $x \neq y$ | X =\= Y |
| $x \geq y$ | X >= Y |
| $x > y$ | X > Y |

# Comparison Operators

- Have the obvious meaning
- Force both left and right hand argument to be evaluated

```
?- 2 < 4+1.
yes

?- 4+3 > 5+5.
no
```

# Comparison Operators



- Have the obvious meaning
- Force both left and right hand argument to be evaluated

```
?- 4 = 4.
yes

?- 2+2 = 4.
no

?- 2+2 =:= 4.
yes
```

# Comparing Numbers

- We are going to define a predicate that takes two arguments, and is true when:
  - The first argument is a list of integers
  - The second argument is the highest integer in the list
- Basic idea
  - We will use an accumulator
  - The accumulator keeps track of the highest value encountered so far
  - If we find a higher value, the accumulator will be updated

# Definition of accMax/3

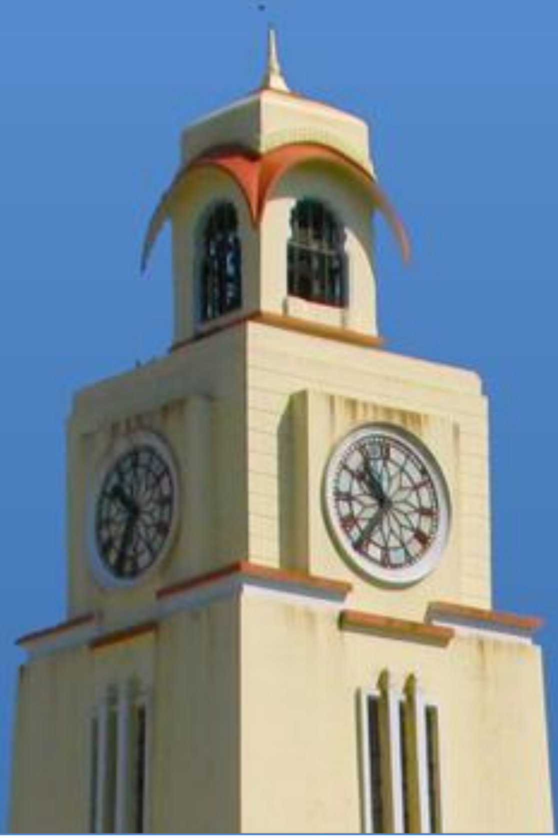
```
accMax([H|T],A,Max):-
    H > A,
    accMax(T,H,Max).

accMax([H|T],A,Max):-
    H =< A,
    accMax(T,A,Max).

accMax([],A,A).
```

```
?- accMax([1,0,5,4],0,Max).
Max=5
yes
```

# Adding a wrapper max/2

```
accMax([H|T],A,Max):-
    H > A,
    accMax(T,H,Max).

accMax([H|T],A,Max):-
    H =< A,
    accMax(T,A,Max).

accMax([],A,A).

max([H|T],Max):-
    accMax(T,H,Max).
```

```
?- max([1,0,5,4], Max).
Max=5
yes

?- max([-3, -1, -5, -4], Max).
Max= -1
yes

?-
```



innovate achieve lead

BITS Pilani
Pilani Campus

# Append

# Append

- We will define an important predicate **append/3** whose arguments are all lists
- Declaratively, append(L1,L2,L3) is true if list L3 is the result of concatenating the lists L1 and L2 together

```
?- append([a,b,c,d],[3,4,5],[a,b,c,d,3,4,5]).
yes

?- append([a,b,c],[3,4,5],[a,b,c,d,3,4,5]).
no
```

# Append, viewed procedurally

- From a procedural perspective, the most obvious use of append/3 is to concatenate two lists together
- We can do this simply by using a variable as third argument

```
?- append([a,b,c,d],[1,2,3,4,5], X).
X=[a,b,c,d,1,2,3,4,5]
yes

?-
```

# Definition of append/3

```
append([ ], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

- Recursive definition
  - Base clause: appending the empty list to any list produces that same list
  - The recursive step says that when concatenating a non-empty list [H|T] with a list L, the result is a list with head H and the result of concatenating T and L

# How append/3 works

- Two ways to find out:
  - Use trace/0 on some examples
  - Draw a search tree!

    Let's consider a simple example

```
?- append([a,b,c],[1,2,3], R).
```

# Search Tree Example

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

```
?- append([a,b,c],[1,2,3], R).
  /                    \
†                R = [a|R0]
               ?- append([b,c],[1,2,3],R0)
                  /                \
                 †              R0=[b|R1]
                              ?- append([c],[1,2,3],R1)
                                 /                \
                                †             R1=[c|R2]
                                            ?- append([],[1,2,3],R2)
                                                 /               \
                                          R2=[1,2,3]              †
```

**R2=[1,2,3]**
**R1=[c|R2]=[c,1,2,3]**
**R0=[b|R1]=[b,c,1,2,3]**
**R=[a|R0]=[a,b,c,1,2,3]**

# Using append/3

- Now that we understand how append/3 works,  let's look at some applications
- Splitting up a list:

```
?- append(X,Y, [a,b,c,d]).

X=[ ]         Y=[a,b,c,d];
X=[a]         Y=[b,c,d];
X=[a,b]       Y=[c,d];
X=[a,b,c]     Y=[d];
X=[a,b,c,d]   Y=[ ];
no
```

# Prefix and suffix

- We can also use append/3 to define other useful predicates
- A nice example is finding prefixes and suffixes of a list

# Definition of prefix/2

```
prefix(P,L):-
    append(P,_,L).
```

- A list P is a prefix of some list L when there is some list such that L is the result of concatenating P with that list.
- We use the anonymous variable because we don't care what that list is.

# Use of prefix/2

```
prefix(P,L):-
    append(P,_,L).
```

```
?- prefix(X, [a,b,c,d]).
X=[ ];
X=[a];
X=[a,b];
X=[a,b,c];
X=[a,b,c,d];
no
```

# Definition of suffix/2

```
suffix(S,L):-
    append(_,S,L).
```

- A list S is a suffix of some list L when there is some list such that L is the result of concatenating that list with S.
- Once again, we use the anonymous variable because we couldn’t care less what that list is.

# Use of suffix/2

```
suffix(S,L):-
    append(_,S,L).
```

```
?- suffix(X, [a,b,c,d]).
X=[a,b,c,d];
X=[b,c,d];
X=[c,d];
X=[d];
X=[];
no
```

# Definition of sublist/2

- Now it is very easy to write a predicate that finds sub-lists of lists
- The sub-lists of a list L are simply the prefixes of suffixes of L

```
sublist(Sub,List):-
    suffix(Suffix,List),
    prefix(Sub,Suffix).
```

# append/3 and efficiency

- The **append/3** predicate is useful, and it is important to know how to use it
- It is of equal importance to know that **append/3** can be source of inefficiency
- Why?
  - Concatenating a list is not done in one simple action
  - But by traversing down one of the lists

# Question

- Using **append/3** we would like to concatenate two lists:
  - List 1: [a,b,c,d,e,f,g,h,i]
  - List 2: [j,k,l]
- The result should be a list with all the elements of list 1 and 2, the order of the elements is not important
- Which of the following goals is the most efficient way to concatenate the lists?
  - ?- append([a,b,c,d,e,f,g,h,i],[j,k,l],R).
  - ?- append([j,k,l],[a,b,c,d,e,f,g,h,i],R).

# Answer

- Look at the way **append/3** is defined
- It recurses on the first argument, not really touching the second argument
- That means it is best to call it with the shortest list as first argument
- Of course you don't always know what the shortest list is, and you can only do this when you don't care about the order of the elements in the concatenated list
- But if you do, it can help make your Prolog code more efficient

# Reversing a List

- We will illustrate the problem with append/3 by using it to reverse the elements of a list
- That is, we will define a predicate that changes a list [a,b,c,d,e] into a list [e,d,c,b,a]
- This would be a useful tool to have, as Prolog only gives easy access to the front of the list

# Naïve reverse

Recursive definition

1. If we reverse the empty list, we obtain the empty list
2. If we reverse the list [H|T], we end up with the list obtained by reversing T and concatenating it with [H]

To see that this definition is correct, consider the list [a,b,c,d].

- If we reverse the tail of this list we get [d,c,b].
- Concatenating this with [a] yields [d,c,b,a]

# Naïve reverse in Prolog

```
naiveReverse([],[]).
naiveReverse([H|T],R):-
    naiveReverse(T,RT),
    append(RT,[H],R).
```

- This definition is correct, but it does an awful lot of work
- It spends a lot of time carrying out appends
- But there is a better way…

# Reverse using an accumulator

- The better way is using an accumulator
- The accumulator will be a list, and when we start reversing it will be empty
- We simply take the head of the list that we want to reverse and add it to the head of the accumulator list
- We continue this until we reach the empty list
- At this point the accumulator will contain the reversed list!

# Reverse using an accumulator

```
accReverse([ ],L,L).
accReverse([H|T],Acc,Rev):-
    accReverse(T,[H|Acc],Rev).
```

# Adding a Wrapper Predicate

```
accReverse([ ],L,L).
accReverse([H|T],Acc,Rev):-
    accReverse(T,[H|Acc],Rev).
```

```
reverse(L1,L2):-
    accReverse(L1,[ ],L2).
```

# Illustrating the Accumulator

| | |
|---|---|
| List: [a,b,c,d] | Accumulator: [] |
| List: [b,c,d] | Accumulator: [a] |
| List: [c,d] | Accumulator: [b,a] |
| List: [d] | Accumulator: [c,b,a] |
| List: [] | Accumulator: [d,c,b,a] |

# Next

**Assignment coming up on Nalanda.**

*THANK YOU*