Wan Hussain Wan Ishak

School of Computing
UUM College of Arts and Sciences
Universiti Utara Malaysia

(P) 04-9285150
(E) hussain@uum.edu.my
(U) http://wanhussain.com
Lecture notes

List, Operators and Arithmetic

- Arithmetic
- Constructing Expressions
- List Processing
- Character Strings and Structures
List

- List is one of the most important Prolog data structures.

- A list is an ordered sequence of zero or more terms written between square brackets and separated by commas.

  \[a,b,c,d]
The elements of a list can be any kind of Prolog terms, including other lists.

\[ [a, b, c, d] \]

Elements

- [1,2,3,4]
- [a1,a2,a3]
- [sawi, kangkung]
- [kucing(comel), kucing(hitam)]
- [[satu,dua],[tiga,empat]]
List

- The empty list is written:

  \[
  \[
  \]

  \[
  \]

- Please note one element list \([a]\) is not equivalent to the atom \(a\).

  \(? \leftarrow [a] = a.\)  
  no
List

- List can be constructed or decomposed through unification.

<table>
<thead>
<tr>
<th>Unify with</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>([a, b, c] = X)</td>
<td>(X = [a, b, c])</td>
</tr>
<tr>
<td>([X, b, Z] = [a, Y, c])</td>
<td>(X = a, \ Y = b, \ Z = c)</td>
</tr>
<tr>
<td>([[a, b], c] = [X, Y])</td>
<td>(X = [a, b], \ Y = c)</td>
</tr>
<tr>
<td>([a(b), c(X)] = [Z, c(a)])</td>
<td>(X = a, \ Z = a(b))</td>
</tr>
</tbody>
</table>
List

- More examples

<table>
<thead>
<tr>
<th>Unify with</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[X</td>
<td>Y] = [a,b,c,d]</td>
</tr>
<tr>
<td>[X</td>
<td>Y] = [a]</td>
</tr>
<tr>
<td>[X,Y</td>
<td>Z] = [a,b,c]</td>
</tr>
<tr>
<td>[X,Y,Z</td>
<td>A] = [a,b,c]</td>
</tr>
<tr>
<td>[X,Y</td>
<td>Z] = [a</td>
</tr>
</tbody>
</table>
List

- The list can be divided into head and tail by the symbol ‘|’.

\[ H \mid T \]

- The first element is the head and the rest are the tail.
- Example:

\[ a \mid [b,c,d,e] \]
List

\[ a \mid [b, c, d, e] \]

- The tail of a list is always a list, the head of a list is an element.

- Every nonempty list has a head and a tail.

\[
[a, b, c, d] = [a \mid [b, c, d]] \\
[a] = [a \mid []]
\]
# List Manipulation

## Built-in predicates

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>append/3</code></td>
<td>join or split lists</td>
</tr>
<tr>
<td><code>length/2</code></td>
<td>get the length of a Prolog list</td>
</tr>
<tr>
<td><code>member/2</code></td>
<td>get or check a member of a list</td>
</tr>
<tr>
<td><code>member/3</code></td>
<td>get or check a member of a list and its position</td>
</tr>
<tr>
<td><code>remove/3</code></td>
<td>remove an element from a list</td>
</tr>
<tr>
<td><code>removeall/3</code></td>
<td>remove all occurrences of an item from a list</td>
</tr>
<tr>
<td><code>reverse/2</code></td>
<td>check or get the reverse of a list</td>
</tr>
</tbody>
</table>
List Manipulation

- append/3

Syntax:

\[
\text{append(First, Second, Whole)}
\]

Example:

Join list

\[
?- \text{append([a,b], [c,d], Whole).}
\]
\[
\text{Whole} = [a,b,c,d]
\]

Splitting list

\[
?- \text{append([a,b], Second, [a,b,c,d]).}
\]
\[
\text{Second} = [c,d]
\]

\[
?- \text{append(First, [c,d], [a,b,c,d]).}
\]
\[
\text{First} = [a,b]
\]
List Manipulation

- length/2

Syntax:

```
length(Term, Length)
```

Example:

```
?- length([a,b,c,d], Length).
Length = 4
```
List Manipulation

- **member/2**

**Syntax:**

```
member(Element, List)
```

**Example:**

?- member(a, [a,b,c,d]).
  yes

?- member(Element, [a,b,c,d]).
  Element = a ;
  Element = b ;
  Element = c ;
  Element = d ;
List Manipulation

- `member/3`

**Syntax:**

\[\text{member}(\text{Element}, \text{List}, \text{Position})\]

**Example:**

\[
\begin{align*}
\text{?- member}(c, [a,b,c,d], \text{Position}). \\
\text{Position} = 3 ; \\
\text{- member}(\text{Element}, [a,b,c,d], 2). \\
\text{Element} = b
\end{align*}
\]
List Manipulation

- member/3 (more example):

```prolog
?- member(Element, [a,b,c,d], Position).
Element = a,
Position = 1;

Element = b,
Position = 2;

Element = c,
Position = 3;

Element = d,
Position = 4;
```

Get the element and its position
List Manipulation

- remove/3

Syntax:

\[
\text{remove(Element, List, Remainder)}
\]

Example:

Takeout an element

\[
?\text{- remove(b, [a,b,c,d], Remainder).}
\]

\[
\text{Remainder} = [a,c,d] ;
\]

Element missing????

\[
?\text{- remove(Element, [a,b,c,d], [a,b,d]).}
\]

\[
\text{Element} = c ;
\]
List Manipulation

- remove/3 (more example):

```prolog
?- remove(Element, [a,b,c,d], Remainder).
Element = a ,
Remainder = [b,c,d] ;

Element = b ,
Remainder = [a,c,d] ;

Element = c ,
Remainder = [a,b,d] ;

Element = d ,
Remainder = [a,b,c] ;
```

What element can be removed
List Manipulation

- removeall/3

Syntax:

removeall(Item, List, Remainder)
List Manipulation

- removeall/3 (Example):

**Remove all repeated elements**

?- removeall(a, [a,b,a,b,a], Remainder).
Remainder = [b,b]

**Remove all elements that match with the first element**

?- removeall(Item, [a,b,a,b,a], Remainder).
Item = a,
Remainder = [b,b]

?- removeall(Item, [b,a,b,a], Remainder).
Item = b,
Remainder = [a,a]
List Manipulation

- Reverse/2

Syntax:

reverse(List, Revlist)

Example:

Reverse the list

?- reverse([a,b,c,d], Revlist).
Revlist = [d,c,b,a]

Causes error

?- reverse(List, [d,c,b,a]).
List = [a,b,c,d];

Error 4, Heap Space Full, Trying ewrite/1
List Manipulation - Exercise

What is the output?

?- append([ab],[b,c,d], X).

?- reverse([b,c,d], R), append([ab], R, X).

?- reverse([b,c,d], R), member(F, R, 1), remove(F, R, B), append([ab], B, X).
List – Constructing Predicates

- `append/3`
- `length/2`
- `member/2`
- `member/3`
- `remove/3`
- `removeall/3`
- `reverse/2`

- ✓ How to define or construct the predicates?
- ✓ How they are working/How they manipulate the list?
List – Constructing Predicates

- `append/3` → `addon/3`

Syntax:

```
addon(First, Second, Whole)
```

Definition:

```
addon([], X, X). ← stop the recursion

addon([H|T1], X, [H|T2]) :-
   addon(T1, X, T2). ← add element into the list
```
List – Constructing Predicates

append/3 → addon/3

?- addon([a,b],[c],W).

Ep. 1

Call (1) → addon([],X,X).

Call (2) → addon([H|T1],X,[H|T2]).

Call (2.1) → addon(T1,X,T2).

false  [a,b] \= []

true  [a,b]=[H\|T1]  H=a

[a,b]=[H\|T1]  T1=[b]

[c]=X  X=[c]

W=[H\|T2]  W=[a\|T2]

T2=_

addon([b],[c],T2).  → go to Ep. 2
List – Constructing Predicates

- `append/3` $\Rightarrow$ `addon/3`

From Ep.1, (2.1) $\Rightarrow$ `addon([b],[c],T2).

- Call (1) $\Rightarrow$ `addon([],X,X).
- Call (2) $\Rightarrow$ `addon([H|T1],X,[H|T2]).
- Call (2.1) $\Rightarrow$ `addon(T1,X,T2).

\[ false \quad [b]\|=[] \]
\[ true \quad [b]=[H|T1] \quad H=b \]
\[ [c]=X \quad T1=[] \]
\[ W=[H|T2] \quad X=[c] \]
\[ W=[b|T2] \quad W=[] \]
\[ T2=_ \]

addon([],[],T2). $\Rightarrow$ go to Ep. 3
List – Constructing Predicates

append/3 ⊢ addon/3

From Ep.2, (2.1) ⊢ addon([],X,X).

Call (1) ⊢ addon([],X,X).

Ep. 3

Stop at Ep. 3 rule (1). Rule (2) will not execute.
List – Constructing Predicates

?- addon([a,b],[c],W).  W = [a,b,c]

Ep. 1 (2)  
(2.1)  
addon([a|[b]], [c], [a|T2]):- [a|[b,c]],  T2 = [b,c]

Ep. 2 (2)  
(2.1)  
addon([b|[]], [c], [b|T2]):- [b|[c]],  T2 = [c]

Ep. 3 (1)  
addon([], [c], [c]).

(1)  
addon([], X, X).

(2)  
addon([H|T1],X,[H|T2]):-  
(2.1)  
addon(T1, X, T2).
List – Constructing Predicates

- length/2 \(\rightarrow\) noOfTerms/2

Syntax:

\[
\text{noOfTerms(Term, Length)}
\]

Definition:

\[
\text{noOfTerms([],0).}
\]

\[
\text{noOfTerms([H|Tail],K):-}\ 
\text{noOfTerms(Tail,J),}
\text{K is J + 1.}
\]

\(\leftarrow\) stop the recursion

\(\leftarrow\) calculate the length
List – Constructing Predicates

- length/2 \rightarrow noOfTerms/2

\[
\text{noOfTerms}([a,b,c],T).
\]

Ep. 1

Call (1) \rightarrow \text{noOfTerms}([],0).

Call (2) \rightarrow \text{noOfTerms}([H \mid \text{Tail}],K).

\quad \rightarrow \text{Call (2.1) \rightarrow noOfTerms}\text{(Tail,J)}.

\quad \rightarrow \text{noOfTerms}([b,c],J). \rightarrow \text{go to Ep. 2}

\text{Call (2.2) \rightarrow pending. (At the moment Sub goal 2.2 will not be called.)}

\text{Ep. 1}

(1) \text{noOfTerms}([],0).

(2) \text{noOfTerms}([H \mid \text{Tail}],K):- 
\quad (2.1) \text{noOfTerms}\text{(Tail,J)}, 
\quad (2.2) \text{K is J + 1.}

false \quad [a,b,c]\not=\[]

true \quad [a,b,c]=[H \mid \text{Tail}] 
\quad H=a 
\quad T=K 
\quad \text{Tail}=[b,c] 
\quad T=K=_
List – Constructing Predicates

**length/2 → noOfTerms/2**

From Ep.1, (2.1) → noOfTerms([b,c],J).

Call (1) → noOfTerms([],0).

Call (2) → noOfTerms([H|Tail],K).

- Call (2.1) → noOfTerms(Tail,J).
- Call (2.2) → pending. *(At the moment Sub goal 2.2 will not be called.)*

Ep.2

false [b,c]![equals]![empty]

true [b,c]![equals]![H|Tail]

H=b
Tail=[c]
J=K=_

noOfTerms([c],J). → go to Ep. 3
List – Constructing Predicates

**length/2** $\rightarrow$ **noOfTerms/2**

From Ep.2, (2.1) $\rightarrow$ noOfTerms([c],J).

Ep. 3

- Call (1) $\rightarrow$ noOfTerms([],0).
- Call (2) $\rightarrow$ noOfTerms([H | Tail],K).
  - Call (2.1) $\rightarrow$ noOfTerms(Tail,J).
  - Call (2.2) $\rightarrow$ pending. *(At the moment Sub goal 2.2 will not be called.)*

- false [c]\=[]
  - true [c]=[H | Tail]
    - H=c
    - Tail=[]
    - J=K=\_

- noOfTerms([],J). $\rightarrow$ go to Ep. 4
List – Constructing Predicates

\[ \text{length/2} \rightarrow \text{noOfTerms/2} \]

From Ep.3, (2.1) \( \rightarrow \) noOfTerms([],J).

Call (1) \( \rightarrow \) noOfTerms([],0).

Ep. 4

Prolog will now return to Ep. 3 to execute 2.2. Followed by Ep. 2 and Ep. 1.
List – Constructing Predicates

? - noOfTerms([a, b, c], T).

(1) noOfTerms([], 0).

(2) noOfTerms([H | Tail], K):-
(2.1) noOfTerms(Tail, J),
(2.2) K is J + 1.

(2) noOfTerms([a | [b, c]], K):
(2.1) noOfTerms([b, c], J),
(2.2) K is J + 1.

(2) noOfTerms([a | [b, c]], K):
(2.1) noOfTerms([b, c], J),
(2.2) K is J + 1.

(2) noOfTerms([b | [c]], K):
(2.1) noOfTerms([c], J),
(2.2) K is J + 1.

(2) noOfTerms([c | []], K):
(2.1) noOfTerms([], J),
(2.2) K is J + 1.

K = 0 + 1
K is 0 + 1
K = 1 + 1
K is 1 + 1
K = 2 + 1
K is 2 + 1

In console

T = 3
yes

noOfTerms/2
List – Constructing Predicates

\[ \text{noOfTerms([a|[b,c]],K).} \]

- \( K \text{ is 2 + 1} \)
  - \( J = K = 2 \)

- \( K \text{ is 1 + 1} \)
  - \( J = K = 1 \)
  - \( \text{noOfTerms([b|[c]],K).} \)
    - \( K \text{ is 0 + 1} \)
      - \( J = K = 0 \)
      - \( \text{noOfTerms([],0).} \)

- \( \text{noOfTerms([c|[]],K).} \)
  - \( K \text{ is 0 + 1} \)
    - \( J = K = 0 \)

List – Constructing Predicates

- member/2 $\Rightarrow$ is_in/2

Syntax:

\[ \text{is}_\text{in} (\text{Element}, \text{List}) \]

Definition:

\[ \text{is}_\text{in} (X, [X|T]). \quad \leftarrow \text{check if the Element is Head of the list} \]

\[ \text{is}_\text{in} (X, [H|T]):- \]
\[ \quad \text{is}_\text{in}(X, T). \quad \leftarrow \text{if not, traverse the rest of the list (Tail)} \]
List – Constructing Predicates

\[
is\_in(X, [a, b, c]).
\]

\[
is\_in(X, [b, c]).
\]

\[
is\_in(X, [c]).
\]

\[
is\_in(X, []).\]

\[
X = a
\]

Return \(X = a\)

\[
X = b
\]

Return \(X = b\)

\[
X = c
\]

Return \(X = c\)
List – Constructing Predicates

\[
is_{\text{in}}(c, [a, b, c]).
\]

\[
c \neq a \quad \text{is}_{\text{in}}(c, [b, c]).
\]

\[
c \neq b \quad \text{is}_{\text{in}}(c, [c]).
\]

\[
c = c \quad \text{Return yes}
\]

\[
is_{\text{in}}(c, []).$
\]
List – Constructing Predicates

- remove/3 $\rightarrow$ take_out/3

Syntax:
\[
take\_out\ (\text{Element}, \text{List}, \text{Remainder})
\]

Definition:
\[
take\_out\ (X, [X|R], R). \quad \leftarrow \text{element found}
\]
\[
take\_out\ (X, [F|R], [F|S]):-\quad \leftarrow \text{traverse the list, at the same time create new list}
\]
\[
take\_out\ (X, R, S).
\]
List – Constructing Predicates

- reverse/2 → overturn/2

Syntax:
> overturn (List, RevList)

Definition:
> overturn (List, RevList):-
  overturn2(List, [], RevList).

> overturn2([H | T], Z, W):-
  overturn2(T, [H | Z], W). ← the element position

> overturn2([], X, X).
Character Strings

Three ways to represent a string of characters in Prolog:

- As an atom – atoms are compact but hard to take apart or manipulate.

- As a list of ASCII codes – can use standard list processing techniques on them.

- As a list of one-character atoms - can use standard list processing techniques on them.
Character Strings

- Use write/1 with double quotes.

Example:

?- write("abc").

[97,98,99]
Character Strings

- Use put/1 or putb/1.

Example:

```
write_str([H|T]):-
  put(H), write_str(T).
write_str([]).
```
Character Strings

Convert between an atom or number and a byte list - name/2.

Syntax:

name(Atomic, List)

Example:

?- name(makan, List).
List = [109,97,107,97,110]

?- name(Atomic, [109,97,107,97,110]).
Atomic = makan
Character Strings

- Example (Prolog codes):

```prolog
splitNprint(Ayat):-  
  name(Ayat,AyatList),  
  write_str(AyatList).

write_str([H|T]):-  
  put(H), write_str(T).

write_str([]).
```

Output:

?- splitNprint(ahmad).

ahmad
Character Strings (Exercise)

Write a code/rule to print out as follows:

?- print_splits("prolog").
prolog
pro log
prolo g
prolog

print_splits(Word):-
    print_split(Word,Word).
print_split([],Word).
print_split([_|T],Word):-
    append(H,T, Word),
    write_str(H),
    write(‘ ‘),
    write_str(T),nl,
    print_split(T, Word).

write_str([]).

write_str([H|T]):-
    put(H),
    write_str(T).
Character Strings (Exercise)

Write a code/rule to check whether the word end with “s” or not:

?- end_with_s("flowers").
yes

?- end_with_s("car").
no

?- end_with_s("cars").
yes
Operators Notation

Data structure in Prolog:

functor(arg1, arg2, …, argN).

Functor can become operator

example:

eat(ali, rice).

Can be written as:

ali eat rice.
### Operators Notation

#### Types of operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infix</td>
<td>3 + 4</td>
</tr>
<tr>
<td>Prefix</td>
<td>+(3,4)</td>
</tr>
<tr>
<td>Postfix</td>
<td>8 factorial</td>
</tr>
</tbody>
</table>
Linguistic Operator

- Operator can be defined using op/3.

- op/3 - Declares an operator with a given type and precedence.

- Syntax:

  \[ \text{op(Precedence, Type, Name)} \]

  - Integer 0-1199
  - Atom or list of atoms

  Name of the operator
Linguistic Operator

- Precedence value for operator

- The value between 0 – 1199.

- The precedence value can influence the structure meaning:
  - low precedence meaning high priority.
  - Example: operator with 33 is given higher priority compared to 35.
  - operator with 33 will be first executed.
Linguistic Operator – Precedence Value

- The value between 0 – 1199.

- The precedence value can influence the structure meaning:
  - low precedence meaning high priority.
  - Example: operator with 33 is given higher priority compare to 35.
  - operator with 33 will be first executed.
Linguistic Operator – Precedence Value

Example

Defining operator ‘is_in’ and ‘room’

| ?- op(35,xfx,is_in).  yes
| ?- op(33,fx,room). yes
| ?- display(apple is_in room kitchen).
    is_in(apple,room(kitchen))

Error 42: Syntax Error
| Goal : ered(_11848,_11850)

| ?- display(room kitchen is_in house).
    room(is_in(kitchen,house))
Linguistic Operator – Precedence Value

Example

?- op(35,xfx,is_in).  yes
?- op(33,fx,room).   yes
?- op(36,fx,room).   yes

?- display(apple is_in room kitchen).  is_in(apple, room(kitchen))

?- display(room kitchen is_in house).  room(is_in(kitchen,house))
Linguistic Operator – Defining Type

- If the precedence value is the same, Prolog will read either from left to right or right to left based on the type.

- Type for operator

<table>
<thead>
<tr>
<th>Types</th>
<th>Meaning</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fx</td>
<td>non-associative prefix operator</td>
<td>prefix</td>
</tr>
<tr>
<td>fy</td>
<td>right associative prefix operator</td>
<td>prefix</td>
</tr>
<tr>
<td>xf</td>
<td>non-associative postfix operator</td>
<td>postfix</td>
</tr>
<tr>
<td>yf</td>
<td>left associative postfix operator</td>
<td>postfix</td>
</tr>
<tr>
<td>xfx</td>
<td>non-associative infix operator</td>
<td>infix</td>
</tr>
<tr>
<td>xfy</td>
<td>right associative infix operator</td>
<td>infix</td>
</tr>
<tr>
<td>yfx</td>
<td>left associative infix operator</td>
<td>infix</td>
</tr>
</tbody>
</table>
Using Linguistic Operator

Example cth.pl

% Linguistic operators
:- op(800, fx, if),
    op(700, xfx, then),
    op(300, xfy, or),
    op(200, xfy, and).

% Facts
binatang(comel).
binatang(tompok).
makan(comel, ikan).
Makan(tompok, ikan).
mengiau(comel).
mengiau(tompok).

% Rules
if binatang(X) and makan(X, ikan) then kucing(X).
Using Linguistic Operator

Examine

% Rules
if binatang(X) and makan(X, ikan) then kucing(X).

Standard Syntax
if(then(and(binatang(_13214), makan(_13214, ikan)),
kucing(_13214)))
2 + 1 = ?

Arithmetic – how we do calculation in Prolog.

Use predicate “is/2” not “=/2“.
Arithmetic Operation

- `is/2` takes an arithmetic expression on its right, evaluates it, and unifies the result with its argument on the left.

- Format:

  `<argument> is <arithmetic expression>`.

Example:

- X is 2 + 1.
- X is 9.
- A = 1, X is A + 5.
- 2 is 1 + 1.
Arithmetic Operation

Wrong expression:

2 + 1 is 3.  \(\leftarrow\) Type error - not in the right format.

2 is 1 + X  \(\leftarrow\) Instantiation error – value of X is unknown.
Arithmetic Operation

- Different with “is/2” and “=/2“:

- **is/2** evaluate its right argument and unifies it with its argument on the left – do calculation.

- **=/2** is to test if both arguments on the left and right are unify – do unification.
# Arithmetic Operators

### Common arithmetic operators (infix operators)

<table>
<thead>
<tr>
<th>Operator &amp; Symbol</th>
<th>Example &amp; Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>X is 3 + 2.</td>
</tr>
<tr>
<td>-</td>
<td>X is 3 - 2.</td>
</tr>
<tr>
<td>*</td>
<td>X is 3 * 2.</td>
</tr>
<tr>
<td>/</td>
<td>X is 3 / 2.</td>
</tr>
<tr>
<td>//</td>
<td>X is 3 // 2.</td>
</tr>
</tbody>
</table>
Arithmetic Operation

Other arithmetic functions supported in LPA Prolog

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X^Y$</td>
<td>$X$ to the power of $Y$.</td>
</tr>
<tr>
<td>abs($X$)</td>
<td>the absolute value of $X$. e.g. abs(-3.5) returns 3.5.</td>
</tr>
<tr>
<td>acos($X$)</td>
<td>the arccosine of $X$ in degrees.</td>
</tr>
<tr>
<td>aln($X$)</td>
<td>$e$ to the power of $X$.</td>
</tr>
<tr>
<td>alog($X$)</td>
<td>10 to the power of $X$.</td>
</tr>
<tr>
<td>asin($X$)</td>
<td>the arcsine of $X$ in degrees.</td>
</tr>
<tr>
<td>atan($X$)</td>
<td>the arctangent of $X$ in degrees.</td>
</tr>
<tr>
<td>cos($X$)</td>
<td>the cosine of $X$ degrees.</td>
</tr>
<tr>
<td>fp($X$)</td>
<td>the fractional part of $X$. e.g. fp(-3.5) returns -0.5.</td>
</tr>
<tr>
<td>int($X$)</td>
<td>the first integer equal to or less than $X$. e.g. int(-3.5) returns -4.</td>
</tr>
<tr>
<td>ip($X$)</td>
<td>the integer equal part of $X$. e.g. ip(-3.5) returns -3.</td>
</tr>
<tr>
<td>ln($X$)</td>
<td>the natural logarithm of $X$.</td>
</tr>
<tr>
<td>log($X$)</td>
<td>the base 10 logarithm of $X$.</td>
</tr>
<tr>
<td>max($X$, $Y$)</td>
<td>the maximum value of $X$ and $Y$. e.g. max(-3.5, 4) returns 4.</td>
</tr>
<tr>
<td>min($X$, $Y$)</td>
<td>the minimum value of $X$ and $Y$. e.g. min(-3.5, 4) returns -3.5.</td>
</tr>
<tr>
<td>rand($X$)</td>
<td>a random floating point number between zero and $X$.</td>
</tr>
<tr>
<td>sign($X$)</td>
<td>-1 if $X$ is negative, 0 if $X$ is 0, or 1 if $X$ is positive. e.g. sign(-3.5) returns -1.</td>
</tr>
<tr>
<td>sin($X$)</td>
<td>the sine of $X$ degrees</td>
</tr>
<tr>
<td>sqrt($X$)</td>
<td>the square root of $X$.</td>
</tr>
<tr>
<td>tan($X$)</td>
<td>the tangent of $X$ degrees.</td>
</tr>
</tbody>
</table>
Arithmetic Operators

The precedence of operators:

Example:

\[ X \text{ is } 1 - 2 \times 5^2 / 8 + 6 \]

Equivalent to

\[ X \text{ is } (1 - (2 \times (5^2)) / 8) + 6 \]

Where ^ is performed first, then \* and /, and finally + and -. 
Using Arithmetic Operators

Example:

?- papar(5).
1
21
321
4321

1. printOut(N):-
2. printOut(1, N).
3.
4. printOut(N, N).
5.
6. printOut(X, N):-
7. printOut_num(X),
8. nl,
9. NewX is X + 1,
10. printOut(NewX, N).
11.
12. printOut_num(0).
13.
14. printOut_num(X):-
15. write(X),
16. NewX is X - 1,
17. printOut_num(NewX).
Constructing Expression

Different between Prolog an other programming languages is:

- Other programming language evaluate arithmetic expression\s whenever they occur

- Prolog evaluates them only in specific places.
Constructing Expression

List of built-in Predicates that evaluate expressions.

<table>
<thead>
<tr>
<th>R is Ex</th>
<th>Evaluates Ex and unifies result with R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex1 =:= Ex2</td>
<td>test if the results of two expressions are equal</td>
</tr>
<tr>
<td>Ex1 == Ex2</td>
<td>test if the results of two expressions are not equal</td>
</tr>
<tr>
<td>Ex1 &gt; Ex2</td>
<td>test if the result of one expression is greater than another</td>
</tr>
<tr>
<td>Ex1 &lt; Ex2</td>
<td>test if the result of one expression is less than another</td>
</tr>
<tr>
<td>Ex1 &gt;= Ex2</td>
<td>test if one expression is greater than or equal to another</td>
</tr>
<tr>
<td>Ex1 =&lt; Ex2</td>
<td>test if one expression is less than or equal to another</td>
</tr>
</tbody>
</table>
Exercise

Explain which of the following succeed, fail, or raise error conditions, and why:

1. ?- 5 is 2 + 3.
2. ?- 5 =:= 2+3.
3. ?- 5 = 2 + 3.
4. ?- 5 is 2 + What.
5. ?- 4+1 = 2+3.
6. ?- 4+1 is 2+3.
7. ?- 4+1 =:= 2+3.
8. ?- What is 2+3.
9. ?- What =:= 2+3.
10. ?- What is 5.
11. ?- What = 5.
12. ?- What =\= 2+3.
Practical Calculation

- When using expressions in Prolog, we are mixing the styles of encoding knowledge.

- Normal syntax:

  ?- X is 2 + 3.

- Can be written as:

  ?- sum(2, 3, X).
Practical Calculation

Example:

doublevalue(X, Z):-
    Z is X * X.

areaOfSquare(P, L, Area):-
    Area is P * L.