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.

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

List

- The elements of a list can be any kind of Prolog terms, including other lists.

\[ [1, 2, 3, 4] \quad [a_1, a_2, a_3] \]
\[ [\text{sawi, kangkung}] \quad [\text{kucing(comel)}, \text{kucing(hitam)}] \]
\[ [[\text{satu, dua}], [\text{tiga, empat}]] \]
List

- The empty list is written:

\[
\[
\]

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

\[? - [a] = a.\]

\[\text{no}\]

List

- The list can be divided into head and tail by the symbol \(\mid\).

\[
\text{[H|T]}\]

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

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

List

- List can be constructed or decomposed through unification.

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

List

- 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>append/3</td>
<td>join or split lists</td>
</tr>
<tr>
<td>length/2</td>
<td>get the length of a Prolog list</td>
</tr>
<tr>
<td>member2</td>
<td>get or check a member of a list</td>
</tr>
<tr>
<td>member3</td>
<td>get or check a member of a list and its position</td>
</tr>
<tr>
<td>remove/3</td>
<td>remove an element from a list</td>
</tr>
<tr>
<td>removeall/3</td>
<td>remove all occurrences of an item from a list</td>
</tr>
<tr>
<td>reverse/2</td>
<td>check or get the reverse of a list</td>
</tr>
</tbody>
</table>

List

- More examples

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

- **append/3**
  
  Syntax: 
  \[ \text{append}(	ext{First}, \text{Second}, \text{Whole}) \]
  
  Example:
  
  - `append([a,b], [c,d], Whole).` 
    - Whole = [a,b,c,d]
  
  - `append([a,b], Second, [a,b,c,d]).` 
    - Second = [c,d]
  
  - `append(First, [c,d], [a,b,c,d]).` 
    - First = [a,b]

Join list

Splitting list

- **length/2**
  
  Syntax: 
  \[ \text{length}(\text{Term}, \text{Length}) \]
  
  Example:
  
  - `length([a,b,c,d], Length).` 
    - Length = 4

- **member/3**
  
  Syntax: 
  \[ \text{member}(\text{Element}, \text{List}, \text{Position}) \]
  
  Example:
  
  - `member(c, [a,b,c,d], Position).` 
    - Position = 3
  
  - `member(Element, [a,b,c,d], 2).` 
    - Element = b

Get Element
at given position

- **member/3 (more example):**
  
  Example:
  
  - `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

- **remove/3**
  
  Syntax: 
  \[ \text{remove}(\text{Element}, \text{List}, \text{Remainder}) \]
  
  Example:
  
  - `remove(b, [a,b,c,d], Remainder).` 
    - Remainder = [a,c,d]
  
  - `remove(Element, [a,b,c,d], [a,b,d]).` 
    - Element = c

Takeout an
element

Element
missing????
List Manipulation

- remove/3 (more example):

  ```prolog
  | 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

- removeall/3

  Syntax:
  ```prolog
  removeall(Item, List, Remainder)
  ```

  Example:
  ```prolog
  | removeall(a, [a,b,a,b,a], Remainder).
  | Remove = [b,b]
  | removeall([a,b,a,b,a], [b,a,b,a], [a,a])
  | Remove = [b,b]
  | removeall([b,a,b,a], [a,b,a], [a,a])
  ```

List Manipulation - Exercise

- What is the output?

  ```prolog
  | ?- 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]):-
```

- add element into the list
```
addon(T1,X,T2).
```

<table>
<thead>
<tr>
<th>Ep. 1</th>
<th>Ep. 2</th>
<th>Ep. 3</th>
</tr>
</thead>
</table>
| Call (1) | addon([],X,X).
| Call (2) | addon([H|T1],X,[H|T2]).
| Call (2.1) | addon(T1,X,T2).
| false | [a,b]=[c]
| false |
| true | [a,b]=[H|T2]
| false |
| true | [a,b]=[H|T2]
| false |
| true | [a,b]=[H|T2]
| false |
| true | [a,b]=[H|T2]
| false |

List – Constructing Predicates

- `length/2` → `noOfTerms/2`

**Syntax:**
```
oOfTerms(Term, Length)
```

**Definition:**
```
oOfTerms([],0).
```

- stop the recursion
```
oOfTerms([H|Tail],K):-
```

- calculate the length
```
noOfTerms(Tail,J),
```

```
K is J + 1.
```

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

length/2 → noOfTerms/2

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

Ep. 1

Call (1) → noOfTerms([],0).

noOfTerms([a,b,c],1).

Ep. 2

Call (1) → noOfTerms([a,b,c],1).

false [a,b,c]=[c]

true [a,b]= [a,b,c]

Ep. 3

Call (2) → noOfTerms([b,c],J).

Ep. 4

Call (1) → noOfTerms([b,c],J).

true J=0

false J>0

noOfTerms([b,c],J) → go to Ep. 1

noOfTerms(Tail,J) → go to Ep. 2

noOfTerms([b,c],J) → go to Ep. 3

noOfTerms(Tail,J) → go to Ep. 4

noOfTerms([b,c],J). → go to Ep. 2

noOfTerms(Tail,J). → go to Ep. 3

noOfTerms([b,c],J). → go to Ep. 4

noOfTerms(Tail,J) → go to Ep. 1

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

**member/2 → is_in/2**

Syntax:

\[
\text{is_in} \ (\text{Element}, \text{List})
\]

Definition:

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

\[
is_in \ (X,[H|T]) \leftarrow \text{if not, traverse the rest of the list (Tail)}
\]

\[
is_in(X,T)
\]

List – Constructing Predicates

**remove/3 → take_out/3**

Syntax:

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

Definition:

\[
take_out \ (X,[X|R],R) \leftarrow \text{element found}
\]

\[
take_out \ (X,[F|R], [F|S]) \leftarrow \text{traverse the list, at the same time create new list}
\]

**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 (Exercise)

- Write a code/rule to print out as follows:

```prolog
?- print_splits("prolog").
p r o l o g
```

```prolog
?- print_splits("cars").
```

```prolog
yes
```

Character Strings (Exercise)

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

```prolog
?- end_with_s("flowers").
yes
?- end_with_s("car").
n0
?- end_with_s("cars").
yes
```
### Operators Notation

- **Data structure in Prolog:**
  
  ```prolog
  functor(arg1, arg2, ..., argN).
  ```

- **Functor can become operator**
  
  ```prolog
  eat(ali, rice).
  ```

- **Example:**
  
  ```prolog
  eat(ali, rice).
  ```

### Linguistic Operator

- **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!)</td>
</tr>
</tbody>
</table>

- **Example:**
  
  ```prolog
  def op(Precedence, Type, Name) is 1.
  ```

- **Defining operator “is_in” and “room”**
  
  ```prolog
  use display/1
  To view the standard syntax
  ```

- **Example:**
  
  ```prolog
  undef op(33,xfx,is_in).
  ```

- **Wrongly use operator**
  
  ```prolog
  ?- op(36,fx,room).
  ```

- **Error**
  
  ```prolog
  Error 42 : Syntax Error
  ```

- **Change the precedence value**
  
  ```prolog
  ?- op(35,xfx,is_in).
  ```

- **To view the standard syntax**
  
  ```prolog
  ! Goal : eread(_11848,_11850)
  ! Error 42 : Syntax Error
  ! ----------------------------------------
  ```

- **Defining operator “is_in” and “room”**
  
  ```prolog
  undef op(33,xfx,is_in).
  ```
Linguistic Operator – Precedence Value

Example

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

?- display(apple is in room kitchen).

Using Linguistic Operator

Examine

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

Standard Syntax
if then (and (binatang(X), makan(X, ikan)), kucing(X))

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

- fx: non-associative prefix operator
- fy: right associative prefix operator
- xf: non-associative postfix operator
- yf: left associative prefix operator
- xfx: non-associative infix operator
- xfy: left associative infix operator
- yfx: right associative infix operator

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

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).

Arithmetic Operation

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>.
Wrong expression:

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

2 is 1 + X \(\rightarrow\) Instantiation error – value of X is unknown.

Different with “is/2” and “=/2”:

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

=/> is to test if both arguments on the left and right are unify – do unification.

The precedence of operators:

Example:

\(X = (1 - (2 * (5 ^ 2)) / 8) + 6\)

Equivalent to

\(X = (1 - (2 * (5 ^ 2))) / (8 + 6)\)

Using Arithmetic Operators

Example:

?- papar(5).
1 21 321 4321

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. 5</td>
</tr>
<tr>
<td>-</td>
<td>X is 3 - 2. 1</td>
</tr>
<tr>
<td>*</td>
<td>X is 3 * 2. 6</td>
</tr>
<tr>
<td>/</td>
<td>X is 3 / 2. 1.5</td>
</tr>
<tr>
<td>//</td>
<td>X is 3 / 2. 1</td>
</tr>
</tbody>
</table>
Constructing Expression

- Different between Prolog and other programming languages is:
  - Other programming languages evaluate arithmetic expressions whenever they occur.
  - Prolog evaluates them only in specific places.

Practical Calculation

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

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal syntax:</td>
<td>? - X is 2 + 3.</td>
</tr>
</tbody>
</table>

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 results of two expressions are equal.</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 the result of one expression is greater than another.</td>
</tr>
<tr>
<td>Ex1 &lt;= Ex2</td>
<td>Test if one expression is less than or equal to another.</td>
</tr>
<tr>
<td>Ex1 &gt;= Ex2</td>
<td>Test if one expression is greater 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.