Symbolic Programming
CS3011

INSTRUCTOR: Tim Fernando
Tim.Fernando@tcd.ie

4) Lists

Lists in prolog are just plain old lists of finite elements:

- [mia, vincent, jules, yolanda]
- [mia, robber(honey_bunny), X, 2, mia]

Lists can be empty lists, can contain nested lists, complex terms, atoms and variables.

Lists have both a head and a tail - consider the following list:

- [mia, vincent, jules, yolanda]
  - head -> mia
  - tail -> [vincent, jules, yolanda]

Lists - Head & Tail

We can execute statements on lists in the following manner:

```
?- [Head|Tail] = [mia, vincent, jules, yolanda].

  Head = mia

  Tail = [vincent,jules,yolanda]

  yes
```

The above statement also directly translates through to:

```
?- [X|Y] = [mia, vincent, jules, yolanda].
  X = mia
  Y = [vincent,jules,yolanda]
  Yes
```

We can do a lot more with the | operator such as finding the first two elements of a list:

```
?- [X,Y | W] = [[], dead(z), [2, [b, c]], [], Z].
  X = []
  Y = dead(z)
  W = [[2,[b,c]],[],_8327]
  Z = _8327
  Yes
```

Similarly to Haskell we can also use the **anonymous variable (_)** to extract only elements we are interested in:

```
?- [_,X,_,Y|_] = [[], dead(z), [2, [b, c]], [], Z].
  X = dead(z)
  Y = []
  Z = _9593
  Yes
```

Lists - Members

When dealing with lists one of the most common requirements is figuring out whether X is an element of list L - this can be done by recursively checking X with the tail of a list:

```
member(X,[X|T]).

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

This can be queried as follows (immediately finds yolanda at head of list - rule 1):

```
?-  member(yolanda,[yolanda,trudy,vincent,jules]).

Yes
```

This can also be queried as follows (recursively finds jules):

```
?-  member(jules,[yolanda,trudy,vincent,jules]).

Yes
```

We can also introduce the **anonymous variable (_)** into this clause to simplify things:

```
member(X,[X|_]).

member(X,[_|T])  :-  member(X,T).
```

This is the exact same as the original rules however in rule 1 we don't care about the tail of the list and in rule 2 we don't care about the head of the list so there is no need to instantiate them to a variable.

Lists - Recursing Down Lists

Consider we wanted a predicate to verify that two lists of a's and b's had the same amount of each letter:

```
-? a2b([a,a,a,a],[b,b,b,b]).

Yes

-? a2b([a,a],[b,b,b,b]).

No
```

The best way to face this task is to consider the most simplest option (base case) such as two empty lists matching being true.

```
a2b([],[]).
```

We can then proceed with the verification by ensuring that the head of List1 is an a and the head of List2 is a b. If this is true, recursively verify the remaining tails of the lists.

```
a2b([a|Ta],[b|Tb])  :-  a2b(Ta,Tb).
```

If we were to pose this against the lists below it would **PASS** and take the following steps:

1.	a2b([a,a,a],[b,b,b]).	=> Yes since head of [a,a,a] is an a and head of [b,b,b] is a b
2.	a2b([a,a],[b,b]).	=> Yes since head of [a,a] is an a and head of [b,b] is a b
3.	a2b([a],[b]).	=> Yes since head of [a] is an a and head of [b] is a b
4.	a2b([],[]).	=> Yes since this is satisfied by rule 1

If we were to pose this against the lists below it would **FAIL** and take the following steps:

5.	a2b([a,a,a,a],[b,b,b]).	=> Yes since head of [a,a,a,a] is an a and head of [b,b,b] is a b
6.	a2b([a,a,a],[b,b]).	=> Yes since head of [a,a,a] is an a and head of [b,b] is a b
7.	a2b([a,a],[b]).	=> Yes since head of [a,a] is an a and head of [b] is a b
8.	a2b([a],[]).	=> No since head of [a] is a and head of [] is undefined

5) Arithmetic

Prolog provides many built in arithmetic operators such as:

Arithmetic examples	Prolog Notation
6 + 2 = 8	8 is 6+2.
6 ∗ 2 = 12	12 is 6*2.
6 − 2 = 4	4 is 6-2.
6 − 8 = − 2	-2 is 6-8.
6 ÷ 2 = 3	3 is 6/2.
7 ÷ 2 = 3	3 is 7/2.
7 mod 2	1 is mod(7,2).

We can use this to work with variables and perform arithmetic in programs:

```
?-  X  is  6+2.

  X  =  8

?-  X  is  6*2.

   X  =  12

?-  R  is  mod(7,2).

    R  =  1
```

This can similarly be used to define functions to perform arithmetic:

```
add_3_and_double(X,Y)  :-  Y  is  (X+3)*2.


 ?-  add_3_and_double(1,X).

  X  =  8
```

Arithmetic - Lists

Prolog also provides many built in arithmetic functions which can be used with lists:

```
len([],0).

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

----------------------------------------------------------

?-  len([a,b,c,d,e,[a,b],g],X).

  X  =  7
```

This method can be changed/reversed to introduce the use of **accumulators** to calculate the length of a list:

```
accLen([_|T],A,L)  :-   Anew  is  A+1,  accLen(T,Anew,L).

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

This allows you to use the accumulator (A) to house the current value of the length of the list in between each intermediate pass off the accLen call until finally you have recursed the tail to the empty list. When you reach the empty list the accumulator is unified to L so that it is returned as L = A.

Below is a trace of a call to accLen which shows how the accumulator is used between calls. It includes 4 x Calls and 4 x Exits which recurse down and back up a stack:

```
?- accLen([a,b,c],0,L).

    Call: (6) accLen([a, b, c], 0, _G449) ?

    Call: (7) _G518 is 0+1 ?

    Exit: (7) 1 is 0+1 ?

    Call: (7) accLen([b, c], 1, _G449) ?

    Call: (8) _G521 is 1+1 ?

    Exit: (8) 2 is 1+1 ?

    Call: (8) accLen([c], 2, _G449) ?

    Call: (9) _G524 is 2+1 ?

    Exit: (9) 3 is 2+1 ?

    Call: (9) accLen([], 3, _G449) ?

    Exit: (9) accLen([], 3, 3) ?

    Exit: (8) accLen([c], 2, 3) ?

    Exit: (7) accLen([b, c], 1, 3) ?

    Exit: (6) accLen([a, b, c], 0, 3) ?
```

Arithmetic - Comparing Integers

Arithmetic examples	Prolog Notation
x < y	X < Y.
x ≤ y	X =< Y.
x = y	X =:= Y.
x ∕= y	X =\= Y.
x ≥ y	X >= Y
x > y	X > Y

These operators can be used to create a recursive function which uses an accumulator to find the maximum element within a list:

```
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).
```

This first checks if the head (H) is greater than the current max element, then calls accMax again with the tail to see if it can find anything higher.

If the head is less than current max, ignore and call accMax with the tail again.

If the list is empty, return A.