

Lecture 5: Arithmetic

- Theory
 - Introduce Prolog's built-in abilities for performing **arithmetic**
 - Apply these to simple list processing problems, using **accumulators**
 - Look at **tail-recursive** predicates and explain why they are more efficient than predicates that are not tail-recursive
- Exercises
 - Correction exercises LPN chapter 4
 - Exercises LPN chapter 5

Arithmetic in Prolog

- Prolog provides a number of basic arithmetic tools
- Integer and real numbers

Arithmetic

$$2 + 3 = 5$$

$$3 \times 4 = 12$$

$$5 - 3 = 2$$

$$3 - 5 = -2$$

$$4 : 2 = 2$$

1 is the remainder when 7 is
divided by 2

Prolog

?- 5 is 2+3.

?- 12 is 3*4.

?- 2 is 5-3.

?- -2 is 3-5.

?- 2 is 4/2.

?- 1 is mod(7,2).

Example queries

?- 10 is 5+5.

yes

?- 4 is 2+3.

no

?- X is 3 * 4.

X=12

yes

?- R is mod(7,2).

R=1

yes

Defining predicates with arithmetic

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

Defining predicates with arithmetic

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

```
?- addThreeAndDouble(1,X).
```

```
X=8
```

```
yes
```

```
?- addThreeAndDouble(2,X).
```

```
X=10
```

```
yes
```

A closer look

- It is important to know that $+$, $-$, $/$ and $*$ do not carry out any arithmetic
- Expressions such as $3+2$, $4-7$, $5/5$ are ordinary Prolog terms
 - Functor: $+$, $-$, $/$, $*$
 - Arity: 2
 - Arguments: integers

A closer look

$$?- X = 3 + 2.$$

A closer look

?- X = 3 + 2.

X = 3+2

yes

?-

A closer look

$$?- X = 3 + 2.$$

$$X = 3 + 2$$

yes

$$?- 3 + 2 = X.$$

A closer look

?- $X = 3 + 2.$

$X = 3 + 2$

yes

?- $3 + 2 = X.$

$X = 3 + 2$

yes

?-

The is/2 predicate

- To force Prolog to actually evaluate arithmetic expressions, we have to use

is

just as we did in the other examples

- This is an instruction for Prolog to carry out calculations
- Because this is not an ordinary Prolog predicate, there are some restrictions

The is/2 predicate

?- X is 3 + 2.

The is/2 predicate

```
?- X is 3 + 2.
```

```
X = 5
```

```
yes
```

```
?-
```

The is/2 predicate

?- X is 3 + 2.

X = 5

yes

?- 3 + 2 is X.

The is/2 predicate

?- X is 3 + 2.

X = 5

yes

?- 3 + 2 is X.

ERROR: is/2: Arguments are not sufficiently instantiated

?-

The is/2 predicate

?- X is 3 + 2.

X = 5

yes

?- 3 + 2 is X.

ERROR: is/2: Arguments are not sufficiently instantiated

?- Result is 2+2+2+2+2.

The is/2 predicate

?- X is 3 + 2.

X = 5

yes

?- 3 + 2 is X.

ERROR: is/2: Arguments are not sufficiently instantiated

?- Result is 2+2+2+2+2.

Result = 10

yes

?-

Restrictions on use of is/2

- We are free to use variables on the right hand side of the **is** predicate
- But when Prolog actually carries out the evaluation, the variables must be instantiated with a variable-free Prolog term
- This Prolog term must be an arithmetic expression

Notation

- Two final remarks on arithmetic expressions
 - $3+2$, $4/2$, $4-5$ are just ordinary Prolog terms in a user-friendly notation:
 $3+2$ is really $+(3,2)$ and so on.
 - Also the **is** predicate is a two-place Prolog predicate

Notation

- Two final remarks on arithmetic expressions
 - $3+2$, $4/2$, $4-5$ are just ordinary Prolog terms in a user-friendly notation:
 $3+2$ is really **$+(3,2)$** and so on.
 - Also the **is** predicate is a two-place Prolog predicate

```
?- is(X,+(3,2)).
```

```
X = 5
```

```
yes
```

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).                %% not a Prolog native predicate!  
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
 - Introduce the idea of 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.
```

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

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

?-

Length of a list in Prolog

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

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

```
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).
```

?-

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([a,b,c],0,Len).

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

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

Search tree for acclen/3

?- acclen([a,b,c],0,Len).
/ \

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

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


Search tree for acclen/3

?- acclen([a,b,c],0,Len).

/ \

no

?- acclen([b,c],1,Len).

/ \

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

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

Search tree for acclen/3

?- acclen([a,b,c],0,Len).

/

\

no

?- acclen([b,c],1,Len).

/

\

no

?- acclen([c],2,Len).

/

\

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

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

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).

/

\

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

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

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).
```

Adding a wrapper predicate

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

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

```
length(List,Length):-                %% native Prolog predicate  
    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

Comparison

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).

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


Search tree for len/2

```
?- len([a,b,c], Len).  
  /      \  
no  ?- len([b,c],Len1),  
      Len is Len1 + 1.
```

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

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.

```
len([],0).
```

```
len(_|L,NewLength):-
```

```
len(L,Length),
```

```
NewLength is Length + 1.
```

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.

```
len([],0).
```

```
len(_|L,NewLength):-
```

```
len(L,Length),
```

```
NewLength is Length + 1.
```

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,
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 Numbers

Arithmetic

$x < y$

$x \leq y$

$x = y$

$x \neq y$

$x \geq y$

$x > y$

Prolog

$X < Y$

$X = < Y$

$X =: = Y$

$X = \backslash = Y$

$X > = Y$

$X > Y$

Comparison Operators

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

?- 2 < 4+1.

yes

?- 4+3 > 5+5.

no

Comparison Operators

- Have the obvious meaning
- Force both left and right hand arguments 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
```

```
?-
```

Summary of this lecture

- In this lecture we showed how Prolog does arithmetic
- We demonstrated the difference between tail-recursive predicates and predicates that are not tail-recursive
- We introduced the programming technique of using accumulators
- We also introduced the idea of using wrapper predicates

Solution Exercise 4.1

How does Prolog respond to the following queries?

1. $[a, b, c, d] = [a, [b, c, d]]$. ✗
2. $[a, b, c, d] = [a | [b, c, d]]$. ✓
3. $[a, b, c, d] = [a, b, [c, d]]$. ✗
4. $[a, b, c, d] = [a, b | [c, d]]$. ✓
5. $[a, b, c, d] = [a, b, c, [d]]$. ✗
6. $[a, b, c, d] = [a, b, c | [d]]$. ✓
7. $[a, b, c, d] = [a, b, c, d, []]$. ✗
8. $[a, b, c, d] = [a, b, c, d | []]$. ✓
9. $[] = _$. ✓
10. $[] = [_]$. ✗
11. $[] = [_ | []]$. ✗

Solution Exercise 4.3

Write a predicate `twice(In,Out)` whose left argument is a list, and whose right argument is a list consisting of every element in the left list written twice.

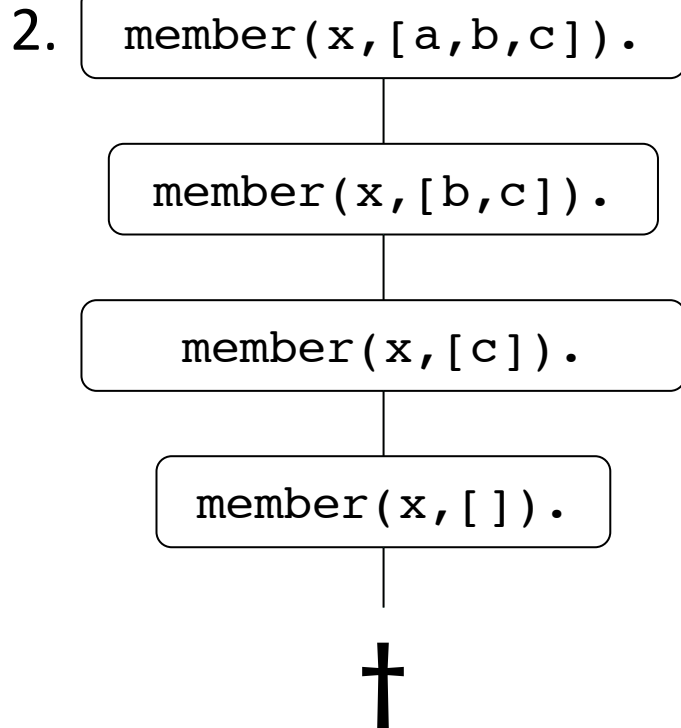
```
twice([], []).  
twice([H|T], Out) :- Out = [H, H | TailOut], twice(T, TailOut).
```

```
?- twice([a,4,buggle],X).  
X = [a,a,4,4,buggle,buggle].
```

```
?- twice([1,2,1,1],X).  
X = [1,1,2,2,1,1,1,1].
```

Solution Exercise 4.4

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



1. member(a, [c,b,a,y]).

member(a, [b,a,y]).

member(a, [a,y]).



3. member(X, [a,b,c]).

X = a

member(a, [a|[b,c]]).



Exercise LPN Chapter 5

- 5.1, 5.2, 5.3

Next lecture

- Yes, more lists!
 - Defining the `append/3`, a predicate that concatenates two lists
 - Discuss the idea of reversing a list, first naively using `append/3`, then with a more efficient way using accumulators