Lecture 5: Arithmetic & Terms

- 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
  - Introduce the == predicate
  - Take a closer look at term structure
  - Introduce strings in Prolog
  - Introduce operators

Arithmetic in Prolog

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

Arithmetic	Prolog
2 + 3 = 5	?- 5 is 2+3.
3 x 4 = 12	?- 12 is 3*4.
5 – 3 = 2	?- 2 is 5-3.
3 – 5 = -2	?- -2 is 3-5.
4 : 2 = 2	?- 2 is 4/2.
1 is the remainder when 7 is divided by 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.
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

Length of a list in Prolog

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

```
?-
```

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.

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

?-
```

Adding a wrapper predicate

```
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

Comparison

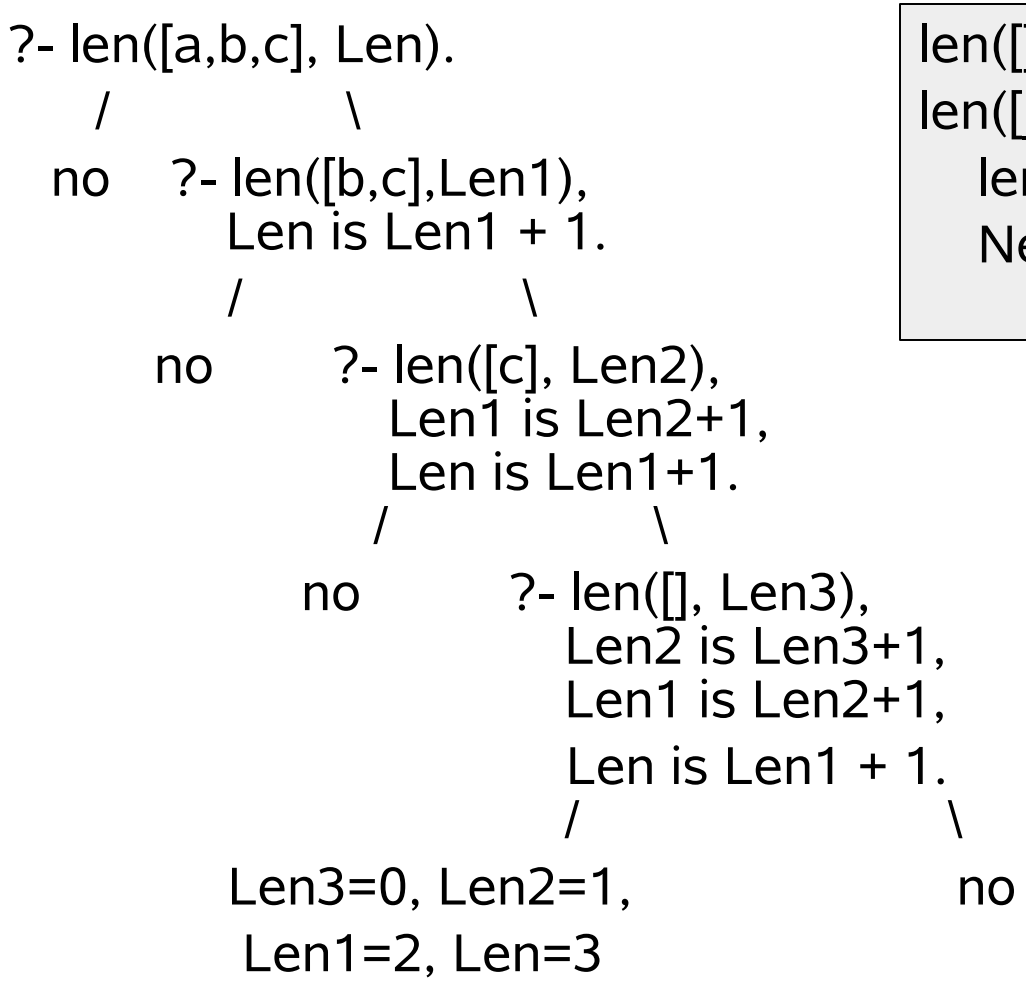
Not tail-recursive

```
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).
```

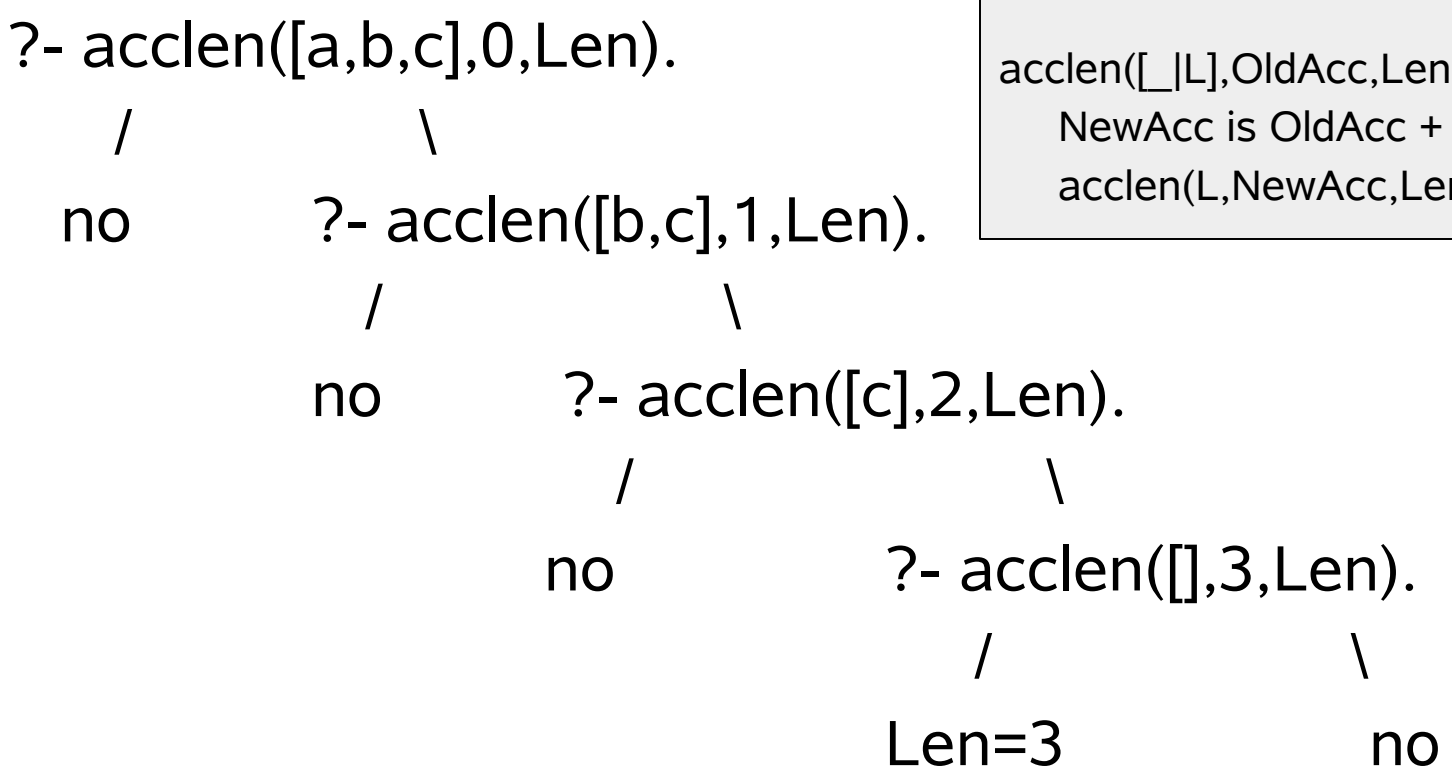
Search tree for len/2




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

Search tree for acclen/3

```
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

?-
```

Comparing terms: ==/2

- Prolog contains an important predicate for comparing terms
- This is the identity predicate ==/2
- The identity predicate ==/2 does not instantiate variables, that is, it behaves differently from =/2

Comparing terms: ==/2

- Prolog contains an important predicate for comparing terms
- This is the identity predicate ==/2
- The identity predicate ==/2 does not instantiate variables, that is, it behaves differently from =/2

```
?- a==a.
yes

?- a==b.
no

?- a=='a'.
yes

?- a==X.
X = _443
no
```

Comparing variables

- Two different **uninstantiated** variables are not identical terms
- Variables **instantiated** with a term *T* are identical to *T*

Comparing variables

- Two different **uninstantiated** variables are not identical terms
- Variables **instantiated** with a term *T* are identical to *T*

```
?- X==X.
X = _443
yes

?- Y==X.
Y = _442
X = _443
no

?- a=U, a==U.
U = _443
yes
```

Comparing terms: \==/2

- The predicate \==/2 is defined so that it succeeds in precisely those cases where ==/2 fails
- In other words, it succeeds whenever two terms are **not identical**, and fails otherwise

Comparing terms: \==/2

- The predicate \==/2 is defined so that it succeeds in precisely those cases where ==/2 fails
- In other words, it succeeds whenever two terms are **not identical**, and fails otherwise

```
?- a \== a.
no

?- a \== b.
yes

?- a \== 'a'.
no

?- a \== X.
X = _443
yes
```

Terms with a special notation

- Sometimes terms look different, but Prolog regards them as identical
- For example: **a** and **'a',** but there are many other cases
- Why does Prolog do this?
  - Because it makes programming more pleasant
  - More natural way of coding Prolog programs

Arithmetic terms

- Recall lecture 5 where we introduced arithmetic
- +, -, <, >, etc are functors and expressions such as 2+3 are actually ordinary complex terms
- The term 2+3 is identical to the term +(2,3)

Arithmetic terms

- Recall lecture 5 where we introduced arithmetic
- +, -, <, >, etc are functors and expressions such as 2+3 are actually ordinary complex terms
- The term 2+3 is identical to the term +(2,3)

```
?- 2+3 == +(2,3).
yes

?-  -(2,3) == 2-3.
yes

?-  (4<2) == <(4,2).
yes
```

Summary of comparison predicates

=	Unification predicate
\=	Negation of unification predicate
==	Identity predicate
\==	Negation of identity predicate
=:=	Arithmetic equality predicate
=\=	Negation of arithmetic equality predicate

Lists as terms

- Another example of Prolog working with one internal representation, while showing another to the user
- Using the | constructor, there are many ways of writing the same list

```
?- [a,b,c,d] == [a|[b,c,d]].
yes
?- [a,b,c,d] == [a,b,c|[d]].
yes
?- [a,b,c,d] == [a,b,c,d|[]].
yes
?- [a,b,c,d] == [a,b|[c,d]].
yes
```

Prolog lists internally

- Internally, lists are built out of two special terms:
  - [] (which represents the empty list)
  - '.' (a functor of arity 2 used to build non-empty lists)
- These two terms are also called *list constructors*
- A recursive definition shows how they construct lists

Definition of prolog list

- The empty list is the term []. It has length 0.
- A non-empty list is any term of the form .(*term*,*list*), where *term* is any Prolog term, and *list* is any Prolog list. If *list* has length *n*, then .(*term*,*list*) has length *n*+1.

A few examples...

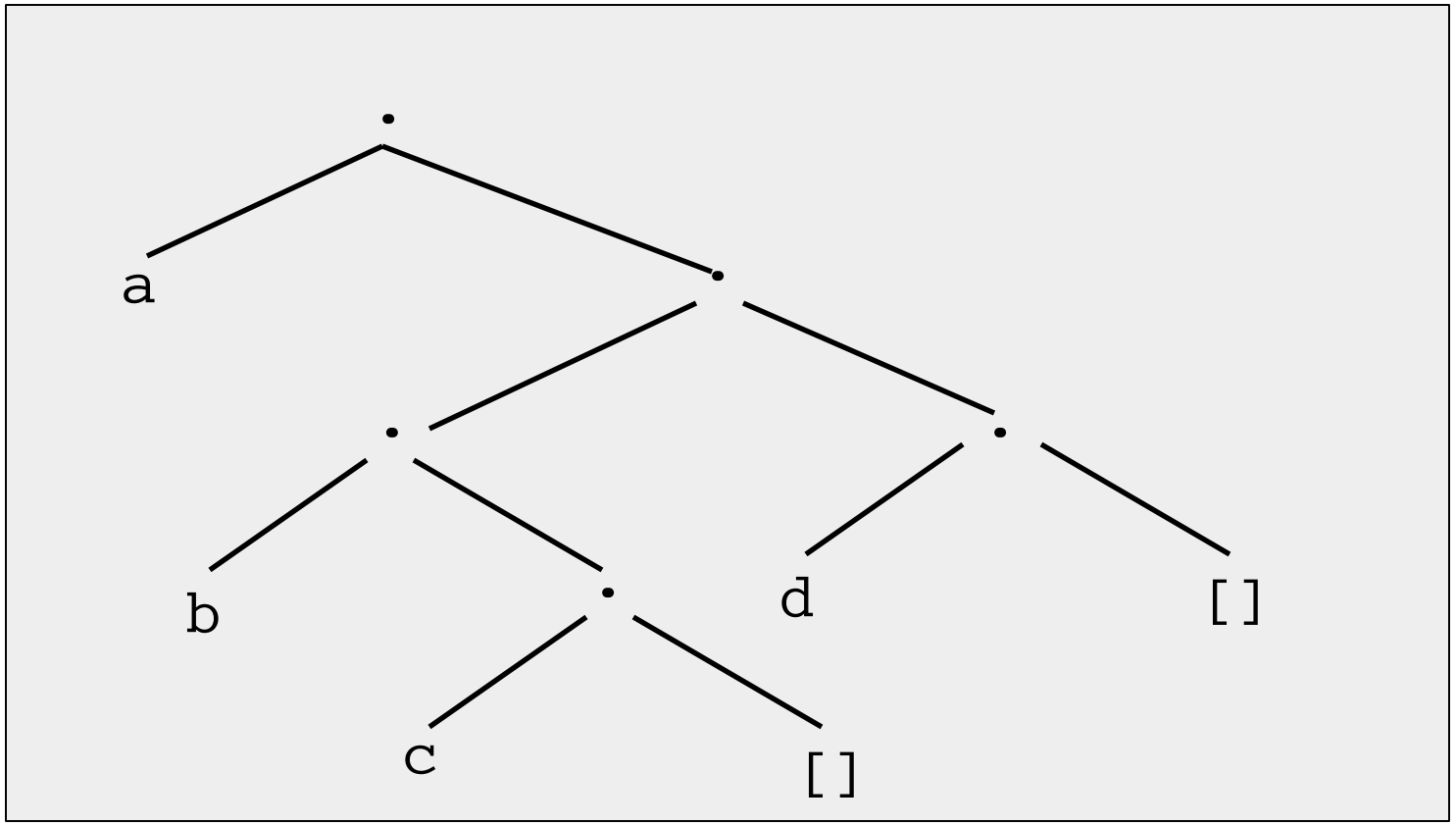
```
?- .(a,[]) == [a].
yes

?- .(f(d,e),[]) == [f(d,e)].
yes

?- .(a,.(b,[])) == [a,b].
yes

?- .(a,.(b,.(f(d,e),[]))) == [a,b,f(d,e)].
yes
```

Internal list representation

- Works similar to the | notation:
- It represents a list in two parts
  - Its first element, the *head*
  - the rest of the list, the *tail*
- The trick is to read these terms as trees
  - Internal nodes are labeled with **.**
  - All nodes have two daughter nodes
    - Subtree under left daughter is the head
    - Subtree under right daughter is the tail

Example of a list as tree

- Example: [a,[b,c],d]



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

Exercises (1)

Write a predicate addone/2 whose first argument is a list of integers, and whose second argument is the list of integers obtained by adding 1 to each integer in the first list. For example, the query

```
addone([1,2,7,2],X).
```

should give

```
X = [2,3,8,3].
```

Exercises (2)

A fundamental operation on vectors is the dot product. This operation combines two vectors of the same dimension and yields a number as a result.
For example, the dot product of [2,5,6] and [3,4,1] is 6+20+6, that is, 32.

Write a 3-place predicate dot.

dot([2,5,6],[3,4,1],Result).

should yield

Result = 32

Exercises (3)

How does Prolog respond to the following queries? WHY?

```
?- .(a,.(b,.(c,[]))) = [a,b,c]
?- .(a,.(b,.(c,[]))) == [a,b,c]
?- .(a,.(b,.(c,[]))) = [a,b|[c]]
?- .(.(a,[]),.(.(b,[]),.(.(c,[]),[]))) = X
?- .(.(a,[]),.(.(b,[]),.(.(c,[]),[]))) == X
?- .(.(a,[]),.(.(b,[]),.(.(c,[]),[]))) =:= X
?- .(a,.(b,.(.(c,[]),[]))) = [a,b|[c]]
?- X = a, X ==a
?- X == a, X = a
```

Exercises (4)

Define a predicate average/2, which calculates the average of a list of integers. Make sure your solution is tail recursive (hint: use accumulator(s)).
(hint: average == (sum) / (nr. els))

?- average([1,2,3],A).
?- A = 2