Lecture notes 10.1

COMP 2411, Session 1, 2004

1 Built-ins for arithmetical computations

To compute `2+3` and assign the result to variable `X`, we cannot use `X = 2+3`: the built-in `=` tries and unify the argument on the left hand side with the argument on the right hand side. Hence the query `X = 2+3` yields the solution `X = 2+3`:

```
?- X = 2+3.

X = 2+3
```

A query such as `5 = 2+3` fails:

```
?- 5 = 2+3.

No
```

In Prolog, `=` enforces pattern matching, not arithmetical computation. The built-in `is` is used for arithmetical computations:

```
?- X is 2+3.

X = 5
```

The expression on the right hand sign of `is` is evaluated, and then assigned to the expression on the left hand side if this expression is a variable. For example:

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

Y = 2
X = 5

?- X is Y + 3.
ERROR: Arguments are not sufficiently instantiated
```

`is` can also be used when the expression on the left hand side contains no variable:

```
?- 5 is 2+3.

Yes
?- 2+3 is 2+3.

No
```

Arithmetic operators include `*` (mutiplication), `//` (integer division), `abs` (absolute value), etc. See SWI-Prolog manual (typing the "`help.`" query).

Relational operators include `=:=` (equality), `=\=` (inequality), `<` (less than), `=<` (less than or equal to), etc. See SWI-Prolog's manual. Both expressions on the left hand side and on the right hand side have to be evaluated to an integer for the goal to succeed or fail:

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

X = 4

?- X is 2 + 2, 3 >= X.

No
?- 3+4 > 3*2.

Yes

?- 3 > X, X is 2.
ERROR: Arguments are not sufficiently instantiated
```

Apart from `=` and `=:=`, there is a third built-in, `==` that checks whether both arguments are the same structures, without applying any substitution:

```
?- X = Y.

X = _G151
Y = _G151 ;

?- X =:= Y.
ERROR: Arguments are not sufficiently instantiated
?- X == Y.

No
?- X == X.

X = _G151
```

The negation of `==` is `\==`, not to be confused with `=\=`, the negation of `=:=`:

```
?- X \== Y.

X = _G151
Y = _G152

?- X =\= Y.
ERROR: Arguments are not sufficiently instantiated
```

Of course, there are cases where `=`, `=:=` and `==` are equivalent:

```
?- X is 2 + 3, X = 5.

X = 5

?- X is 2 + 3, X =:= 5.

X = 5

?- X is 2 + 3, X == 5.

X = 5
```

2 Examples of programs

We can computing the factorial of a number recursively:

```
factorial(0,1).

factorial(N,F) :- N > 0, N1 is N-1, factorial(N1,F1), F is N*F1.
```

Note that the previous program cannot be used to compute which number has a factorial equal to some number: a query such as `factorial(X,6)` generates an error.

It is better to compute the factorial of a number iteratively, using a technique based on *accumulators*:

```
factorial(N,F) :- factorial(0,N,1,F).

factorial(Count,N,Acc,F) :-
          Count < N, Count1 is Count+1,
          Acc1 is Acc*Count1,
          factorial(Count1,N,Acc1,F).

factorial(N,N,F,F).
```

`Count` is a counter, first initialized to `0`, that gets incremented by `1` until it becomes equal to

N. Then the last clause is used, and the result that is stored in the accumulator—the third argument of factorial—is passed to the fourth argument of factorial. Note that this argument is only used in the last clause. Each step of the computation yields partial results that are stored in the accumulator.

The previous program multiples all numbers from 1 to N in increasing order. It is also possible to multiply all numbers from N to 1 in decreasing order. It is then no longer necessary to keep track of "N", since the test for the counter is changed from being equal to "N" to being equal to 0. This reduces the arguments in the main procedure from 4 to 3:

```
factorial(N,F) :- factorial(N,1,F).

factorial(N,Acc,F) :-
          N > 0, Acc1 is Acc*N,
          N1 is N-1, factorial(N1,Acc1,F).

factorial(0,F,F).
```

There is a buit-in length to compute the length of a list. It could be implemented recursively as follows:

```
length([],0).

length([_|T],L) :- length(T,L1), L is L1+1.
```

Here as well, another version that uses an accumulator would be a better choice:

```
length(List,L) :- length(List,0,L).

length([],L,L).

length([_|T],Acc,L) :- Acc1 is Acc+1, length(T,Acc1,L).
```

There is a built-in between that generates all integers in a given range by backtracking. It could be implemented as follows:

```
between(I,J,I) :- I =< J.

between(I,J,K) :- I < J, I1 is I+1, between(I1,J,K).
```