```
prod(X, 0, 0).
prod(X, s(Y), Z) :- prod(X, Y, W), sum(X, W, Z).
```

- Non practicable...

```
:- T is 5+3.
Yes, T=8.
```

```
:- X is, T is 5+X.
what happens is:
:- is(X,3), is(T,+(5,X)).
:- X/3, is(T,+(5,X)).
:- is(T,+(5,3)).
:- is(T,8).
:T/8
```

## Evaluation of expressions

However, when we write an expression, we would like to have it evaluated...

- Special pre-defined predicat `is`.

`T is Expr` ( `is(T,Expr)` )

- `T` can be a numerical atom or a variable

2020/6/12 10:51

- where **REL** is a relational operator, and **Expr1** and **Expr2** are expressions

- **Expr1** and **Expr2** are evaluated. Mind it! They both have to be completely instantiated.
- The results are then compared on the basis of **REL**

2020/6/12 10:52

Do a program that receives N as input, and prints N times "Hello world!"

↳ recursive calls

```
chesani(0.)
chesani(N):-
    print('Hello world'), nl,
    N1 is N-1,
    chesani(N1).
```

nl → newline

print → predicate of arity 0

instead of this works also write

last => tail-recursive

## Iteration and recursion in Prolog

- In Prolog there is no iteration (no while, for, repeat...)

... as a concept
exchanged and exploited in diff

- There are several pre-defined pr
  to deal with these structures, an

```
p(father(federico,francesco)).
p(father(federico,chiara)).
female(chiara)
test(X):-
    p(father(X,Y)),
    female(Y).

father(X,Y):- son(Y,X).
son(francesco,federico).
```

} KB

father in
as

```
test_if_x_is_father(X):-
    p(father(X,Y)),
    call(son(X,Y)).
```

passed as a term

```
test2(X):-
    p(X), call(X).
```

call all the term unified with X from

interpreted

```
p(a).
```

2020/6/12 10:52

When exec

s a procedural

)

al1 is evaluated

al2, is evaluated

. Apply a predicate

ne) :

- `fail` takes no arguments
- Its evaluation always fails
- As a consequence, it forc other alternatives...
- ... in other words, it explic backtracking
- Why on the earth should v proof?
  - To obtain some form of iter
  - To implement the negation
  - To implement a logical imp

```
p(1).
p(2).
p(3).
p(4).
p(5).
p(6).
p(a).
```

KB

```
q(X):-
    integer(X),
    write(X),
    write('is an integer'),nl.
```

```
q(X):- \+ integer(X),
    write(X),
    write('is not an integer).
```

negation in prolog "\+"

iterate.

output:

```
1 is an integer
2 "  "  "
3 "  "  "
4 "  "  "
5 "  "  "
6 "  "  "
a IS NOT an integer
```

othw:

```
?- p(X),q(X).
```

because ; will pick just the first instance of p(X) → so we can iterate!

should ask next solution with ";".

**The `fail` predicate – the ne**

- Implement the negation predicate `not(P)`
- `not(P)` is true if `P` is not a program

```
not(P) :- call(P), !,
not(P).
```

call

if I wrote

```
test(Body) :- clause(q(_X,_Y), Body).
?- test(Body).
Body = (p(_6020), r(a));
Body = d(_6020).
```

We can inspect our program through clause!

Other

- Mod hug
- To l use_

p(1). is a fact => p(1) :- True

Exa :- Loa

2020/6/12 10:53

```
true .
```
Return true if the sentence can be derived by our grammar

```
?- phrase(e, [a,+,+,a]).
false.
```

e when DCGBody applies to List.

entence, and determines if it
e or not.

nstruct the parse tree (indeed,
branch of success)

e tree and reuse it?

↳ SLD resolution tree

have arguments, that will be
ers

e parameters are
fully control of the abstract
rocess

n be called in the right part

```
article --> [the].
article --> [a].
name --> [dog].
name --> [cat].
verb --> [loves].
verb --> [eats].
```

```
?- phrase(sentence, [the,cat,eats,the,dog]).
true.
?- phrase(sentence, X).
X = [the,dog,loves,the,dog];
X = [the,dog,loves,the,dog,that,loves,the,dog];
```

infinite # of sentence ← ...

## DCG – Example – parsing integer expressions...

simple grammar that allow one digit mathem expression

$G = (V_n, V_t, P, S)$

$V_n = \{E, T\}$

$V_t = \{ +, \{digit\} \}$

$P = \{$

E ::= T "+" E
E ::= T "-" E
E ::= T
T ::= factor "*" T
T ::= factor "/" T
T ::= factor
factor ::= digit
factor ::= "(" E ")"

$\}$

$S = E$

"+" can be used instead of [+]

div is an default oper. of digit

Before no argument!
↳ these are Terms

```
e(plus(Op1,Op2)) --> t(Op1), plus_sign, e(Op2).
e(minus(Op1,Op2)) --> t(Op1), minus_sign, e(Op2).
e(Op) --> t(Op).

t(mult(Op1, Op2)) --> factor(Op1), mult_sign, t(Op2).
t(div(Op1, Op2)) --> factor(Op1), div_sign, t(Op2).
t(Op) --> factor(Op).

factor(X) --> digit(Code), {number_string(X,[Code])}.
% factor ::= "(" E ")"

digit(Digit) --> [Digit], {
        char_code('0',Zero),
        char_code('9',Nine),
        Zero =< Digit,
        Digit =< Nine}.

plus_sign --> "+".
minus_sign --> "-".
mult_sign --> "*".
div_sign --> "/".
```

transform in

char-code gets
For ASCI conversion

of

transform into ASCII

```
?- string_codes("1+5", Codes), phrase(e(X),Codes).
Codes = [49,43,53],
X = [illegible].
```

```
?- string_codes("1", Codes), phrase(e(X), Codes).
Codes = [49]
X = 1.
```

```
?- Term =..[boh].
   Term = boh.
?- Term =.. [boh,1].
   Term = boh(1).
?- Term=.. [father, mario,
            aldo], call(Term).
   Term = father(mario,
                 aldo).
?- boh(p, q(f(r(1)))) =.. L.
   L=[boh, p,
      q(f(r(1)))].
```

## Accessir

- Term =
  [SWI do
  - List is
    remo
  - Eithe

```
?- foo(
List =
```

```
?- Term
```

```
Term = b
```