built-in predicates

- `sort(List, Sorted)`
- `length(List, Len)`
- `member(E, L)`
- `append(L1, L2, L3)`
- `findall(+Template, +Goal, -List)`

  example: `findall(F, friend(F, clare), Friends)`
- `setof(+Template, +Goal, -List)`

  compared with `findall`, `List` is sorted and duplicate-free.

  At the same time, `Goal` can contains free variables, and for every value of every variable, a result `List` would be given
- `functor(Term, Functor, Arity)`

  example: `Term:fact(3,6)` ⇔ `Functor:fact,Arity:2`
- `arg(ArgNum, Term, Arg)` extracts an argument from a term

  example: `arg(2, fact(3, 6), 6)` succeeds
- `=../2`, `fact(3,6) =.. [fact,3,6]`
- add/remove clauses: `assert/1, retract/1, retractall/1`, see 7.2.3

operators

- `=`, unifying terms(counter part: `\=` means cannot unify)
- `==`, comparing terms
- `=:=`, the same value after evaluation(counter part: `=\=`)
- `\+`, "not"
- `>, <, =<, >=`, classical comparing
- `@>, @<, @=<, @>=`, standard comparing
- `+, -, *, /, //, div, rem, mod, **`

  `//` truncated, `div` rounded, `rem`: `X rem Y` = `X-Y*(X//Y)`, `mod`: `X mod Y` = `X-Y*(X div Y)`
- `is/2`, evaluate the second term and unify the result with the first term

tail recursion

2 conditions:

1. recursive calls only occur at the very end
2. when a recursive call happens no further backtracking

definitions

terms are a piece of data in Prolog, they include variables, atoms and numbers

compound terms: `f(t1,t2,...,tN)` where `t1,t2,...,tN` are also terms

predicates can only be applied to terms

Lists

predicate `./2`, `[1,2,3]` is `.(1,.(2,.(3,[])))`, `[]` is also an important atom

`.(H,T)` can also be written as `[H|T]`

in-order list mapping:

```
%% double(+L, -Result), in-order version.
double([], []).
double([X|LTail], [Y|ResultTail]) :-
  Y is X * 2,
  double(LTail, ResultTail).

%% reversed version
double(L, Result) :-
         double(L, [], Result).
     double([], Result, Result).
     double([X|Tail], ResultAcc, Result) :-
         Y is X * 2,
         double(Tail, [Y|ResultAcc], Result).
```

Definitive clause grammar

`sentence([the,man,eats,an,apple], R)` succeeds if list is a valid sentence after removing a list `R` at the end of the given list, which means that `R` is a suffix of the given list.

```
sentence(S, R) :-
  noun_phrase(S, VP),
  verb_phrase(VP, R).

determiner([the|R], R).
```

Process: first find a prefix noun phrase, after that try to find a prefix verb phrase.

Simplified version:

```
sentence -->
  noun_phrase,
  verb_phrase.

determiner --> [the].
determiner --> [a].
determiner --> [an].
```

Mention that `sentence` is still a predicate whose arity is 2.

Related built-in predicate: `phrase/[2,3]`, example:

```
phrase(sentence, [the,man,eats,an,apple]).
phrase(sentence, [the,man,eats,an,apple], R).
```

Add some extra conditions:

```
determiner(det(D)) --> /* here we can not only check the syntax but
also produce some outputs */
  [D],             /* describe the construction */
  {determiner(D)}. /* extra condition, will be kept intact */

/* transferred */
determiner(det(D), X, Y) :- /* there are always 2 automatically
added arguments: List and R */
  X = [D|Y],
  determiner(D).

/* determiner dictionary */
determiner(the).
determiner(an).
determiner(a).
```