I/O in Prolog

- ✔ To write a term `X` to the currently open file, execute the goal `write(X)`
- ✔ To read a term from the currently open file, execute the goal `read(X)`
  - ✗ `X` is instantiated to the term read
  - ✗ on end of file, `X` is instantiated to the atom `end_of_file`
- ✔ To write a newline to the currently open file, execute the goal `nl`
- ✔ To write `N` spaces to the currently open file, execute the goal `tab(N)`

I/O in Prolog

- The terminal is the default file for reading and writing

- To open a file named **`filename`** for reading , execute the goal

  `see(filename)`

- To open a file named **`filename`** for writing, execute the goal

  `tell(filename)`

- To close a file opened for reading, execute the goal **`seen`**

- To close a file opened for writing, execute the goal **`told`**

- The terminal corresponds to a file named **`user`**

I/O in Prolog

✔ Using **read** and **write** to interact with the user

```
dosquares :-
    write(`Next item please: `),
    read(X),
    process(X).

process(stop) :- !.

process(N) :-
    C is N * N,
    write(`Square of `),write(N),write(` is `),
    write(C), nl,
    dosquares.
```

```
?- dosquares.
Next item please: 5.
Square of 5 is 25.
Next item please: 12
Square of 12 is 144
Next item please: stop
yes
```

I/O in Prolog

- Why wouldn't the following work:

```
dosquares :-
    write(`Next item please: `),
    read(stop), !.

dosquares :-
    read(N),
    S is N * N,
    write(`Square of `),write(N),write(` is `),
    write(S), nl,
    dosquares.
```

Manipulating characters

- ✔ To write a character to the currently open file, execute the goal `put(C)` where `C` is instantiated to the ASCII code of the desired character

```
?- put(65),put(66),put(67).

ABC
yes
```

- ✔ To read the next character from the currently open file, execute the goal `get0(C)` ... This instantiates `C` to the ASCII code of the character read

- ✔ To read the next printable character from the currently open file, execute the goal `get(C)` ... This instantiates `C` to the ASCII code of the character read

Constructing and decomposing atoms

- ✔ `name(Atom,CList)` is true if `CList` is a list of ASCII codes of the characters in the atom `Atom`

  - ✗ useful for figuring out what characters make up an atom:

```
?- name(foobar,L).
L = [102,111,111,98,97,114]
yes
```

  - ✗ useful for making an atom out of a list of characters:

```
?- name(A,[122,122,121,122,120,68,77]).
A = zzyzxDM
yes
```

Get a sentence as a list of words...

```
getsentence(Wordlist) :-
    get0(Char),
    getrest(Char,Wordlist).

getrest(46,[]) :- !. % end of sentence: ascii 46 = .

getrest(32,Wordlist) :- % ascii 32 = space
    getsentence(Wordlist). % skip the space

getrest(Letter,[Word|Wordlist]) :- % read letters of current
word
    getletters(Letter,Letters,Nextchar),
    name(Word,Letters),
    getrest(Nextchar,Wordlist).

getletters(46,[],46) :- !. % end of word: ascii 46 = .

getletters(32,[],32) :- !. % end of word: ascii 32 = space

getletters(Let,[Let|Letters],Nextchar) :-
    get0(Char), getletters(Char,Letters,Nextchar).
```

Constructing and decomposing terms

- ✔ `Term =.. L` is true if `L` is a list whose head is the principal functor of `Term`, and whose tail is a list of the arguments to `Term`

```
?- f(a,b) =.. L.

L = [f,a,b]
```

```
?- T =.. [append,[a,b],[c,d],Result].

T = append([a,b],[c,d],Result).
```

Constructing and decomposing terms

- ✔ `=..` is useful for creating goals at run time

```
obtain(Functor),
compute(Arguments),
Goal =.. [Functor|Arguments],
call(Goal).
```

Manipulating the Prolog database

- Facts and rules can be added and deleted from the Prolog database at runtime by executing goals with the functors `assert, retract`

- `assert(C)` always succeeds, and causes the clause `C` to be added to the database

- `retract(C)` succeeds if a clause matching `C` exists in the database, and removes it

Manipulating the Prolog database

```
?- assert(fast(ann)).
yes
?- fast(ann).
yes
?- retract(fast(ann)).
yes
?- fast(ann).
no
assert(fast(ann)),assert(slow(tom)),assert(slow(pat)).
yes


?- assert((faster(X,Y) :- fast(X),slow(Y))).
yes

?- faster(A,B).

A = ann
B = tom;

A = ann
B = pat;

no

?- retract(slow(X)).
X = tom;
X = pat;
no

?- faster(A,B).
no
```

Manipulating the Prolog database

- `asserta` and `assertz` are like `assert` except they are guaranteed to place the clause at the start or end of the database, respectively

- Placing a new rule representing a special-case solution at the beginning of the database can improve efficiency

```
?- solve(problem1,Solution),
        asserta(solve(problem1,Solution)).
```

- The next time a problem matching `problem1` is encountered, finding a solution will be fast

- “Memoizing” or “caching”

Collecting multiple solutions

- Backtracking can provide several instantiations of variables that satisfy a goal

- Instead of prompting for each solution with ";", all solutions can be collected into a list using the predicates `bagof` and `setof`

- `bagof(X,G,L)` will produce a list `L` of all the instantiations of `X` that satisfy the goal `G`

- `setof(X,G,L)` will produce a list `L` of all the instantiations of `X` that satisfy the goal `G`, where `L` is ordered alphabetically or numerically, and with duplicates removed

Collecting multiple solutions

```
age(peter,7).
age(ann,5).
age(pat,8).
age(tom,5).
```

```
?- bagof(Child,age(Child,5),List).

List = [ann,tom]
```

```
?- bagof(Child,age(Child,Age),List).

Age = 7
List = [peter];

Age = 5
List = [ann,tom];

Age = 8
List = [pat];

no
```

```
?- bagof(Child, Age ^ age(Child,Age), List).

List = [peter,ann,pat,tom]
```

Iteration control in Prolog

✔ `repeat` is a nondeterministic goal that always succeeds, defined as

```
repeat.
repeat :- repeat.
```

✔ `repeat` can be used for iteration

```
dosquares :-
    repeat,
    write(`Next item please: `),
    read(X),
    (X = stop, !
     ;
     write(`Square of `), write(X), write(` is `),
     Y is X * X, write(Y), nl,
     fail
    ).
```