Prolog - Cheat Sheet

A quick reference guide to Prolog programming, covering syntax, data types, list manipulation, control flow, and built-in predicates.

Syntax Fundamentals

Clauses: Prolog programs are built from clauses, which are either facts or rules.

Facts: Declare a relationship. `parent(john, mary).` (John is a parent of Mary)

Rules: Define relationships based on other relationships.
`ancestor(X, Y) :- parent(X, Y).`
`ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).`

Queries: Ask questions about the defined relationships.
`?- parent(john, mary).`

Variables: Start with an uppercase letter or underscore. `X`, `Result`, `_value`

Atom: Constant values, starting with lowercase letter. `john`, `mary`, `apple`

Comments: `%` for single-line comments.
`/* ... */` for multi-line comments.

Compound Terms: Structures built from a functor and arguments.
`book(title, author)`

Lists: Ordered collections of items.
`[apple, banana, cherry]`
`[1, 2, 3]`

Operators: Symbols for arithmetic, comparison, and logical operations.
`+`, `-`, `*`, `/`, `=`, `\=` (not equals), `<`, `>`

Unification: The process of matching terms.
`X = 5.` (Assigns 5 to X)
`term1 = term2.` (Attempts to unify term1 and term2)

Anonymous Variable: Represented by `_`, used when a variable is needed but its value is not important.

Data Types

Atoms: Symbolic constants.	`hello`, `world`, `atom_name`
Numbers: Integers and floating-point numbers.	`123`, `-45`, `3.14`, `-0.001`
Variables: Represent unknown values. Must start with an uppercase letter or underscore.	`X`, `Result`, `_value`
Structures: Complex terms built from functors and arguments.	`book(title, author)`, `point(1, 2)`
Lists: Ordered collections of elements (see next page).	`[1, 2, 3]`, `[a, b, c]`, `[head | tail]`

Operators

`+`, `-`, `*`, `/`
Arithmetic operators.
	`X + Y`
	Addition.
	`X / Y`
	Division.

`//` Integer division.	`X // Y` Integer division of X by Y.
`mod` Modulo operator.	`X mod Y` Remainder of X divided by Y.
`is` Arithmetic evaluation.	`X is Y` Evaluates Y and unifies the result with X.
`=:=` Arithmetic equality.	`X =:= Y` True if X and Y evaluate to the same number.
`=\=` Arithmetic inequality.	`X == Y` True if X and Y evaluate to different numbers.

`>`, `<`, `>=`, `=<`
Comparison operators.
	`X > Y`
	True if X is greater than Y.
	`X =< Y`
	True if X is less than or equal to Y.

`=` Unification operator.	`X = Y` True if X and Y can be unified.
`\=` Not unifiable.	`X = Y` True if X and Y cannot be unified.
`==` Term equality (identical).	`X == Y` True if X and Y are identical terms.
`\==` Term inequality (not identical).	`X == Y` True if X and Y are not identical terms.
`:-` Rule definition.	`head :- body.` Defines a rule where `head` is true if `body` is true.

List Representation

Empty List: `[]`

List with elements: `[a, b, c]`

Head and Tail: `[Head | Tail]` where `Head` is the first element and `Tail` is the rest of the list.

Accessing elements: Prolog lists are typically accessed via unification and recursion, not direct indexing.

List concatenation: Use `append(List1, List2, Result)` to concatenate two lists.

Example:

```
append([1, 2], [3, 4], X).
% X = [1, 2, 3, 4].
```

Membership test: Use `member(Element, List)` to check if an element is in a list.

Example:

```
member(3, [1, 2, 3]). % true
member(4, [1, 2, 3]). % false
```

Anonymous variable in lists: `[_ | Tail]` ignores the head of the list.

Example:

```
?- [ _ , _ , X] = [1,2,3].
X = 3.
```

Common List Operations

Membership (member/2): Checks if an element is in a list.	`member(X, [a, b, c]).`
Concatenation (append/3): Concatenates two lists.	`append([1, 2], [3, 4], Result).` `Result = [1, 2, 3, 4].`
Prefix (prefix/2): Checks if a list is a prefix of another list.	`prefix([1, 2], [1, 2, 3, 4]).`
Suffix (suffix/2): Checks if a list is a suffix of another list.	`suffix([3, 4], [1, 2, 3, 4]).`
Sublist (sublist/2): Checks if a list is a sublist of another list.	`sublist([2, 3], [1, 2, 3, 4]).`
Length (length/2): Determines the length of a list.	`length([a, b, c], Length).` `Length = 3.`
Reverse (reverse/2): Reverses the order of elements in a list.	`reverse([a, b, c], Result).` `Result = [c, b, a].`
nth0/3: Access element at index (0-based).	`nth0(1, [a, b, c], Element).` `Element = b.`
nth1/3: Access element at index (1-based).	`nth1(2, [a, b, c], Element).` `Element = b.`
select/3: Select an element from a list, resulting in a new list without that element.	`select(b, [a, b, c], Result).` `Result = [a, c].`
last/2: Retrieves the last element of a list.	`last([a, b, c], Last).` `Last = c.`
delete/3: Deletes all occurrences of an element from a list.	`delete(a, [a, b, a, c], Result).` `Result = [b, c].`

Example: List processing

```
% Sum of elements in a list
sum_list([], 0).
sum_list([H|T], Sum) :-
    sum_list(T, RestSum),
    Sum is H + RestSum.
```

```
% Membership test
member(X, [X|_]).
member(X, [_|T]) :-
    member(X, T).
```

```
% List concatenation
append([], L, L).
append([H|T], L, [H|Result]) :-
    append(T, L, Result).
```

```
% Reversing a list
reverse(L, R) :-
    reverse_helper(L, [], R).

reverse_helper([], Acc, Acc).
reverse_helper([H|T], Acc, R) :-
    reverse_helper(T, [H|Acc], R).
```

```
% Find the last element of a list
last([X], X).
last([_|T], X) :-
    last(T, X).
```

```
% Remove duplicates from a list
remove_duplicates([], []).
remove_duplicates([H|T], Result) :-
    member(H, T),
    remove_duplicates(T, Result).
remove_duplicates([H|T], [H|Result]) :-
    \+ member(H, T),
    remove_duplicates(T, Result).
```

Control Flow

Conjunction (`,`): `A, B` - A and B must both be true.

Disjunction (`;`): `A; B` - A or B must be true.

Negation (`\+`): `\+ A` - A must be false.

If-Then-Else (`-> ;`): `(Condition -> Then ; Else)` - If Condition is true, Then is executed; otherwise, Else is executed.

Call: `call(A)` - Executes the goal `A`. Useful for meta-programming.

Once: `once(A)` - Executes goal `A`, but only the first solution is returned. Prevents Prolog from backtracking to find alternative solutions.

Repeat: `repeat.` - Always succeeds and can be used to generate multiple solutions through backtracking. Must be used with a cut (`!`) to avoid infinite loops.

Example:

```
repeat.
process.
!. % Cut to prevent infinite backtracking
```

Fail: `fail.` - Always fails, forcing backtracking. Useful for creating loops and testing.

True: `true.` - Always succeeds. Useful as a placeholder or to satisfy a condition.

Useful Built-in Predicates

`true/0` : Always succeeds.	`true.`
`fail/0` : Always fails.	`fail.`
`not/1` : Negation (same as `\+`).	`not(member(4, [1, 2, 3])).`
`=/2` : Unification (attempts to make two terms equal).	`X = 5.`
`is/2` : Evaluates an arithmetic expression.	`X is 2 + 3.`
`write/1` : Prints a term to the console.	`write('Hello, world!').`
`nl/0` : Prints a newline character.	`nl.`
`read/1` : Reads a term from the console.	`read(X).`
`display/1` : Displays a term using prefix notation.	`display(+(1,2)).` % Displays: +(1,2)
`atom_string/2` : Converts between atoms and strings.	`atom_string(hello, "hello").`
`number_string/2` : Converts between numbers and strings.	`number_string(123, "123").`
`current_predicate/1` : Checks if a predicate is currently defined.	`current_predicate(member/2).`
`arg/3` : Accesses the arguments of a compound term.	`arg(2, f(a, b, c), X). % X = b`
`functor/3` : Retrieves or defines the functor and arity of a term.	`functor(term(a,b), F, N). % F = term, N = 2`

Family Relationships

```
parent(john, mary).
parent(john, peter).
parent(susan, mary).
parent(susan, peter).

male(john).
female(susan).

father(X, Y) :- parent(X, Y), male(X).
mother(X, Y) :- parent(X, Y), female(X).
sibling(X, Y) :- parent(Z, X), parent(Z,
Y), X \= Y.
```

Queries:

```
?- father(john, mary).
?- sibling(mary, peter).
```

List Manipulation

```
% Reverse a list
reverse_list(List, Reversed) :-
    reverse_list(List, [], Reversed).

reverse_list([], Acc, Acc).
reverse_list([H|T], Acc, Reversed) :-
    reverse_list(T, [H|Acc], Reversed).
```

Queries:

```
?- reverse_list([1, 2, 3], Reversed).
```

Simple AI: Expert System

```
% Rules for diagnosing a problem
symptom(cough).
symptom(fever).
symptom(runny_nose).

disease(flu) :-
    symptom(fever),
    symptom(cough),
    symptom(runny_nose).

disease(cold) :-
    symptom(cough),
    symptom(runny_nose),
    \+ symptom(fever).
```

Queries:

```
?- disease(X).
```