2 Terms

1. Typeless language
2. Compound terms
3. Variables in terms
4. Groundness
5. Recursive structures
6. Lists

2.1 Typeless language

Prolog is a **typeless language**, which means you do not declare types

Prolog has one type: every Prolog data structure is a **term**

Every term is one of:

- An *atom* (what we've used so far)
- A *number* (integer or float)
- A *variable*
- A *compound term*

2.1.1 Atoms and Numbers

Atoms are like strings in other languages, except:

- Can be quickly compared for equality
- Only one copy of characters in memory

Usual syntax: begins with a small letter, followed by any number of letters, digits, and underscores

Any characters allowed in atoms, if surrounded by single quotes, e.g.:

```
'This is one atom!'
```

To include single quote in atom, double it:

```
'Doesn''t this look odd?'
```

Numbers are simple: float if it has a decimal point or exponential notation; otherwise integer

Atoms and numbers are called **atomic terms**

2.1.2 **Variables**

How can a *variable* be a data structure?

A Prolog **variable** is just a term whose value you don't know yet

Don't *assign* to variables, *constrain* them

A variable can be bound to any term — even another variable

Can't change value of variable once bound

More like a variable in algebra than a conventional programming language

2.2 Compound Terms

The **Compound term** is Prolog's sole data structuring abstraction

A compound term has:

- a **functor**, which is an atom
- one or more **arguments**, which can be any terms

Syntax: functor first, then arguments in parentheses, separated by commas

Compound terms look just like predicate invocations

The number of arguments is called the **arity**

No need to declare it — just use it

2.2.1 Compound Terms

Use compound terms where you would use (a pointer to) a struct in C

Use functor to indicate the kind (type?) of term

No need to allocate a structure and assign to members: just write it down

E.g., to create a term holding an (X,Y) position of (20,10):

```
position(20,10)
```

To create an employee data structure:

```
employee(1234, 'Jones', 'James', 100000)
```

2.3 Variables in Terms

E.g., to create a term holding an (X,Y) position of (`X`,`Y`) where `X` and `Y` are variables:

```
position(X,Y)
```

To create an employee data structure:

```
employee(Num, Surname, Firstname, Salary)
```

Valid whether or not values of variables are determined

In code: (assume there is a predicate `line_length` that relates two `position`/2 terms and the distance between them)

```
% Draw line from (0,0) to (X,Y)
    inside_circle(X, Y, Radius) :-
        line_length(position(0, 0), position(X, Y), Len),
        Len =< Radius.
```

Prolog comments: `%` to end of line; and `/*` comments `*/`

2.3.1 Anonymous Variables

```
wealthy(employee(_, _, _, Salary)) :-
        Salary >= 100000.
```

Each _ is a separate **anonymous variable**; it means we don't care about its value

```
?- wealthy(employee(1234,'Jones', 'James', 100000)).

Yes
?- wealthy(employee(5678,'Smith', 'Sam', 20000)).

No
```

2.3.2 **Unification**

In Prolog, = is used for both giving values to variables and comparing terms for equality

```
?- 4 = 4.

Yes
?- 4 = 5.

No
?- X = 5.

X = 5 ;

No
```

```
?- 5 = V.

V = 5 ;

No
?- X = Y.

X = _G168
Y = _G168 ;

No
```

N.B. Variables appearing in different queries are unrelated.

:- Unification (2)

= performs **unification**, tries to make two terms same by binding variables appearing in the terms

Unification is a powerful sort of pattern matching

```
?- foo(A,2,C,D) = foo(1,X,Y,X).
A = 1
C = _G302
D = 2
X = 2
Y = _G302 ;
No
?- bar(X, X) = bar(1, 2).
No
```

Prolog also uses unification for argument passing

More on unification later

Exercise: Unification

We haven't formally defined unification yet. Nevertheless, try to find bindings for these variables to make these pairs of terms the same.

1. `f(X, 17) = f(Y, Y)`

2. `f(42, U, a) = f(V, f(V,W), W)`

2.3.3 Construction and Deconstruction

Prolog also uses unification for constructing and deconstructing data structures

```
P = position(10, 20)
```

binds `P` to a data structure `position(10, 20)`

```
P = position(10, 20),
P = position(X, Y)
```

binds `X` and `Y` to 10 and 20

Implicit use of unification leads to more elegant code

2.3.4 Argument Passing

Prolog uses unification for argument passing

```
transpose(position(X, Y), position(Y, X)).
```

```
?- transpose(position(10, 20), Pos).
Pos = position(20, 10) ;
No
?- transpose(Pos, position(20, 10)).
Pos = position(10, 20) ;
No
?- transpose(position(X,Y), position(20, 10)).
X = 10
Y = 20 ;
No
?- transpose(P1, P2).
P1 = position(_G302, _G303)
P2 = position(_G303, _G302) ;
No
```

2.4 Groundness

A variable is **bound** if it has been unified with an atom, number, or compound term (anything but a variable)

An unbound variable denotes a term we don't know yet, and could eventually be bound to any term

A compound term containing variables denotes a term we don't fully know; it could be further *refined* by binding variables

e.g. `X` could be any term, but `p(X)` can be any term whose functor is `p` and arity is 1

A term containing no unbound variables is said to be **ground**

Ground terms cannot be further refined, and always denote only one term

Atomic terms are always ground

2.5 Recursive Structures

Any argument of any compound term can be any other term — including a term with the same functor

This is how Prolog implements recursive structures, such as lists, trees, etc.

One **excellent** strategy for writing predicates that process recursive structures:

1. write predicate that recognizes recursive data structure
2. use this definition as a "skeleton" for other predicates, adding extra arguments and goals as needed

If you are ever stuck with Prolog programming, try this

2.5.1 Programming Numbers as Terms

For example, we could represent a natural number as either `0` or a term `s(X)` (1 more than `X`) where `X` represents a natural number

NB: This is not really how Prolog does maths

```
nat(0).                  % 0 is a natural number
nat(s(X)) :- nat(X).     % s(X) is a nat if X is

add(0, Y, Y).            % adding 0 to Y gives Y
add(s(X), Y, s(Z)) :-    % adding X+1 to Y gives Z+1
        add(X, Y, Z).    %         where Z is X + Y
```

Note that structure of `add/3` follows that of `nat/1`

(We refer to predicates as $name/arity$ since Prolog allows different predicates with same name and different arity)

2.5.2 Example Queries

```
?- nat(s(s(0))).
Yes

?- nat(X).
X = 0 ;
X = s(0) ;
X = s(s(0))
Yes

?- add(s(s(0)), X, Y).
X = _G254
Y = s(s(_G254)) ;
No
```

```
?- add(s(s(0)), s(s(s(0))), X).
X = s(s(s(s(s(0))))) ;
No

?- add(s(s(0)), X, s(s(s(s(s(0)))))).
X = s(s(s(0))) ;
No

?- add(X, s(s(s(0))), s(s(s(s(s(0)))))).
X = s(s(0)) ;
No

?- add(X,Y,s(s(0))).
X = 0, Y = s(s(0)) ;
X = s(0), Y = s(0) ;
X = s(s(0)), Y = 0 ;
No
```

Exercise

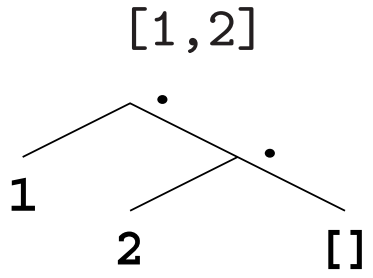
Define a predicate `mult(X,Y,Z)` such that `X` times `Y` equals `Z`. Use the `add/3` predicate defined above.

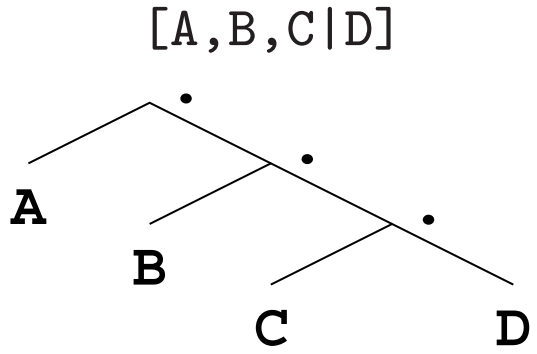
2.6 Lists

Lists are very widely used in Prolog — they have a special syntax

```
list([]).
list([_|Tail]) :- list(Tail).
```

- `[]` is the empty list
- `[H|T]` is the list with head `H` and tail `T`.





List functor is `.` (dot), arity is 2

Bracket notation is special syntax, but lists are ordinary terms

:- Limitations

- Only one | in a list term
- Left of | are one ore more list *elements*, separated by commas
- Right of | is the single list *tail* — a list, not an element
- `[H|T]` in Prolog is like `h:t` in Haskell

2.6.1 List Membership

The `member/2` predicate can check list membership

It is a built-in of SWI Prolog, but could be defined as:

```
member(X, [X|_]).    % X is member of list beginning with X
member(X, [_|L]) :-  % X is member of list
       member(X, L). %        if it is member of tail
```

2.6.2 Member in Action

Member can be used in different ways:

```
?- member(c, [a,b,c,d]).
Yes
?- member(e, [a,b,c,d]).
No
?- member(X, [a,b,c]).
X = a ;
X = b ;
X = c ;
No
?- L = [_,_,_], member(a, L).
L = [a, _G291, _G294] ;
L = [_G288, a, _G294] ;
L = [_G288, _G291, a] ;
No
```

2.6.3 List Concatentation

The `append/3` predicate concatenates (++ in Haskell) two lists

Again a built-in, but could be defined as:

```
append([], L, L).                % [] ++ L gives L
append([J|K], L, [J|KL]) :-      % to ++ a list with first element J
                                 % and remainder K to list L
        append(K, L, KL).        %     determine K ++ L and
                                 %     then add J to front
```

Recall Haskell definition:

```
append [] l = l
append (j:k) l = j : append k l
```

The same, but can only append lists

2.6.4 Append in Action

Prolog's `append` is more flexible:

```
?- append([a,b,c], [d,e], L).
L = [a, b, c, d, e] ;
No

?- append(L, [d,e], [a,b,c,d,e]).
L = [a, b, c] ;
No

?- append([a,b,c], L, [a,b,c,d,e]).
L = [d, e] ;
No
```

```
?- append(K, L, [a,b]).
K = []
L = [a, b] ;

K = [a]
L = [b] ;

K = [a, b]
L = [] ;

No
```