Types of Programming Languages

- imperative (how to solve the problem)
    - procedural
    - object-oriented
- declarative (description of the problem and what constitutes a solution)
    - functional
    - logical

Prolog

- Prolog is a *declarative language*
- A program is a collection of *axioms* from which *theorems* can be proven.
  - Rather than describing how to compute a solution, a program consists of a data base of facts and logical relationships (rules).
  - Rather then running a program to obtain a solution, the user asks a question. When asked a question, the run time system searches through the data base of facts and rules to determine (by logical deduction) the answer.

We will be using SWI-Prolog for our assignments.

Horn clauses

A clause is Horn if it contains at most one positive literal.

$$(L \vee \neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_n) \equiv L_1 \wedge L_2 \wedge \dots \wedge L_n \rightarrow L \qquad n \geq 0$$

$$(\neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_n) \equiv \neg(L_1 \wedge L_2 \wedge \dots \wedge L_n) \qquad n \geq 0$$

A definite clause is a Horn clause with exactly one positive literal.

When $n = 0$ a definite clause is called a fact.

When $n \geq 0$ a definite clause is called a rule.

A Horn clause with no positive literals is called a goal.

Prolog programs consist of definite clauses and questions to a program is a goal.

Logic Programming

- Axiom are written in a standard form known as ***Horn clauses***
  - A Horn clause consists of a *consequent* (*head* H) and a *body* (*terms* B_i)

    $\mathbf{H \leftarrow B_1, B_2, \ldots, B_n}$
  - Semantics: when all B_i are true, we can deduce that H is true

Prolog

- *PROgramming in LOGic*
- It is the most widely used logic programming language
- Its development started in 1970 and it was result of the collaboration between researchers from Marseille, France, and Edinburgh, Scotland
- Main applications:
  - Artificial intelligence: expert systems, natural language processing, …
  - Databases: query languages, data mining,…
  - Mathematics: theorem proving, symbolic packages,…

History

- Kowalski: late 60's Logician who showed logical proof can support computation.
- Colmerauer: early 70's Developed early version of Prolog for natural language processing, mainly multiple parses.
- Warren: mid 70's First version of Prolog that was efficient.

Example

Program

```
location(desk, office).
location(apple, kitchen).
location(flashlight, desk).
location('washing machine', cellar).
location(nani, 'washing machine').
location(broccoli, kitchen).
location(crackers, kitchen).
location(computer, office).
```

Query

```
?- location(apple, kitchen).
yes
```

Prolog Programming Model

- A program is a *database of (Horn) clauses* that are assumed to be true.
- The clauses in a Prolog database are either *facts* or *rules*. Each ends with a period.
- A *fact* is a clause without a right-hand side.
  - rainy(asheville).
- A *rule* has a right-hand side.
  - snowy(X) :- rainy(X), cold(X).
- The token :- is the implication symbol.
- The comma (,) indicates “and”

Prolog Clauses

- A *query* or *goal* is a clause with an empty left-hand side. Queries are given to the prolog interpreter to initiate execution of a program.
- Each clauses is composed of *terms*:
  - *Constants* (atoms, that are identifier starting with a lowercase letter, numbers, punctuation, and quoted strings)
    - » *E.g.* `curry, 4.5, +, 'Hi, Mom'`
  - *Variables* (identifiers starting with an uppercase letter)
    - » *E.g.* `Food`
  - *Structures* (predicates or data structures)
    - » *E.g.* `indian(Food), date(Year,Month,Day)`

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*
- 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

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
```

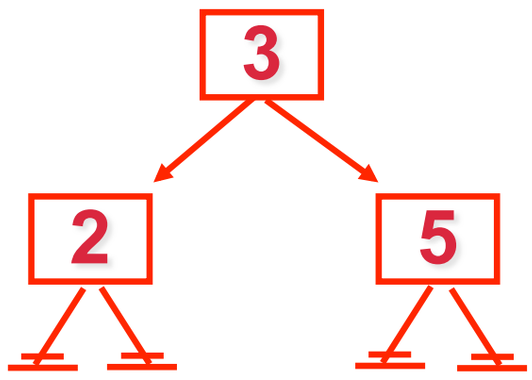
```
?- add(s(s(0)), s(s(s(0))), X).
X = s(s(s(s(s(0))))) ;
No

?- 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
```

Structures

- Structures consist of an atom called the *functor* and a list of arguments
  - *E.g.* `date(Year,Month,Day)`
  - *E.g.*

Functors

```
T = tree(3, tree(2,nil,nil), tree(5,nil,nil))
```



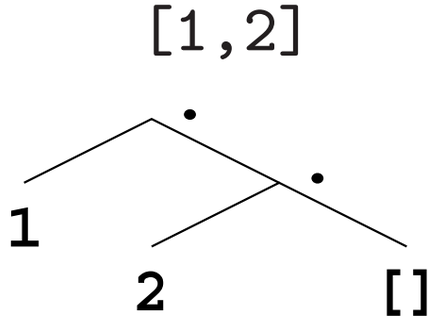
2.6 Lists

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

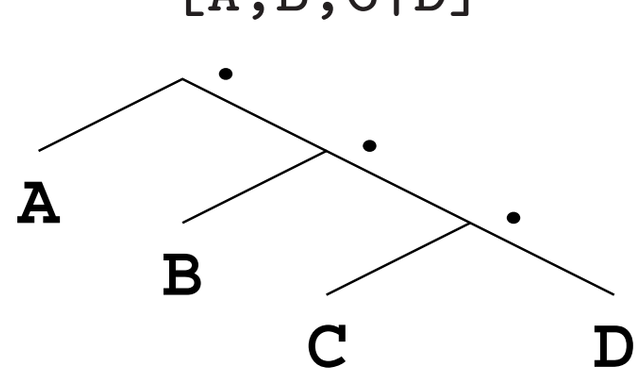
```
list([]).
list([_|Tail]) :- list(Tail).
```

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

`[1,2]`



`[A,B,C|D]`



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
```

Unification

- Prolog associates variables and values using a process known as *unification*
  - Variable that receive a value are said to be *instantiated*
- Unification rules
  - A constant unifies only with itself
  - Two structures unify if and only if they have the same functor and the same number of arguments, and the corresponding arguments unify recursively
  - A variable unifies to with anything

Execution Order

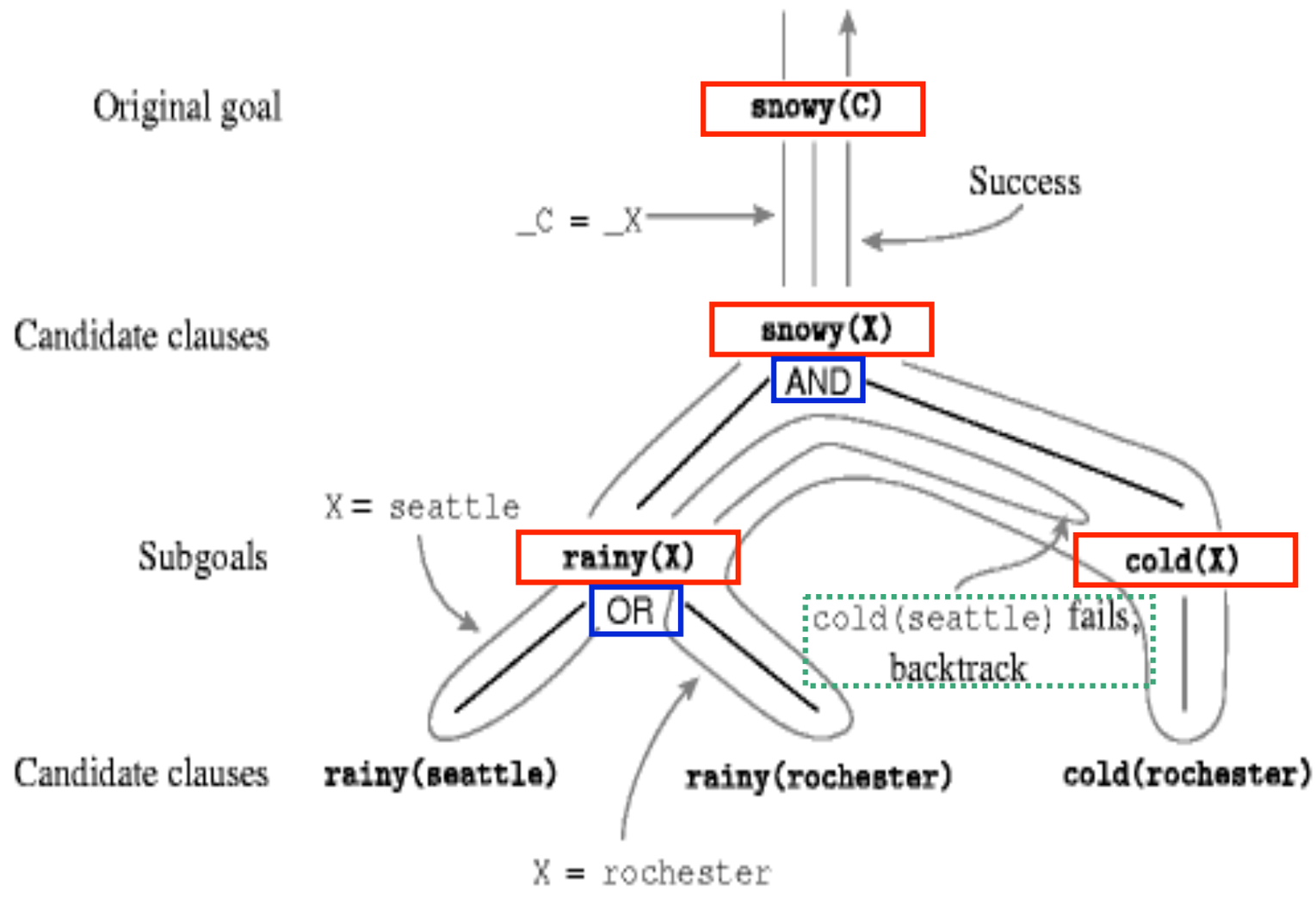
- Prolog searches for a resolution sequence that satisfies the goal
- In order to satisfy the logical predicate, we can imagine two search strategies:
  - *Forward chaining*, derived the goal from the axioms
  - *Backward chaining*, start with the goal and attempt to resolve them working backwards
- Backward chaining is usually more efficient, so it is the mechanism underlying the execution of Prolog programs
  - Forward chaining is more efficient when the number of facts is small and the number of rules is very large

Backward Chaining in Prolog

- Backward chaining follows a classic depth-first backtracking algorithm
- Example
  - Goal:

    `Snowy(C)`

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X)
```



Depth-first backtracking

- The search for a resolution is ordered and depth-first
  - The behavior of the interpreter is predictable
- Ordering is fundamental in recursion
  - Recursion is again the basic computational technique, as it was in functional languages
  - Inappropriate ordering of the terms may result in non-terminating resolutions (infinite regression)
  - For example: Graph

```
edge(a,b). edge(b, c). edge(c, d).
edge(d,e). edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :- edge(Z, Y), path(X, Z).
```

Correct

Depth-first backtracking

- The search for a resolution is ordered and depth-first
  - The behavior of the interpreter is predictable
- Ordering is fundamental in recursion
  - Recursion is again the basic computational technique, as it was in functional languages
  - Inappropriate ordering of the terms may result in non-terminating resolutions (infinite regression)
  - For example: Graph

```
edge(a,b). edge(b, c). edge(c, d).
edge(d,e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```

Incorrect

Infinite Regression

```
edge(a, b).   edge(b, c).   edge(c, d).
edge(d, e).   edge(b, e).   edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```