COMP3031: Logic Programming

Qiong Luo

HKUST

Introduction

- A program in an imperative programming language implements a function from input values to output values through control structures and assignments that change the state of the machine.
- A program in a functional programming language defines a function in terms of simpler, primitive functions.
- A program in a logic programming language (partially) specifies a set of relations.

Relations in Logic Programming

- Computing with relation is more flexible than with function:
  - If a program implement a function `foo(inputs)`, then this program can also be taken as a specification of the relation:

    `{ (inputs,output) | output = foo(inputs) }`
  - Relations treat arguments (inputs) and results (outputs) uniformly, they have no sense of direction, no prejudice about who is computed from whom.
  - If a program specifies the relation `R(x,y)`, then we can supply an x, say a, and ask it to find some y such that R(a,y). We can also supply a y, say b, and ask it to find some x such that R(x,b) holds. For example, if we define `R(x,y) iff square(x) = y`, then we can ask for some or all y such that `R(2,y)` holds, and we can also ask for some or all x such that `R(x,4)` holds.

Specifying a Relation (Facts)

- A concrete view of a relation is as a table.
- For example, the following table specifies a flight connection relation:

```
direct_flight
-------------------------
hongkong    | tokyo
hongkong    | beijing
hongkong    | sanfrancisco
tokyo       | hongkong
tokyo       | vancouver
-------------------------
```

- A table completely specifies a relation: a tuple (X1,...,Xn) is in the relation iff it is in the table for this relation.
- For example, a tuple `(X,Y)` is in the relation `direct_flight` if it is in the above table.

Specifying a Relation (Continued)

- A relation is often called a *predicate*. E.g., instead of saying the tuple `(HongKong,Tokyo)` is in the relation `direct_flight`, we say that the atomic formula `direct_flight(hongkong,tokyo)` is true.
- So the table for `direct_flight` would be represented as the following set of *facts* in a LP such as Prolog:

```
direct_flight(hongkong, tokyo).
direct_flight(hongkong, beijing).
direct_flight(hongkong, sanfrancisco).
direct_flight(tokyo, hongkong).
direct_flight(tokyo, vancouver).
```

Specifying a Relation (Rules)

- Sometimes it is difficult or even impossible to give a table for a relation. Instead, we have some *rules* about the relation.
- In particular, we frequently use recursive rules in defining a relation.
- Example:
  - If there is a direct flight from X to Y, then there is a flight from X to Y.
  - Recursively, if there is a direct flight from X to Y, and there is a flight from Y to Z, then there is a flight from X to Z.
- In Prolog, the relation `flight` can be written as the following two rules:

```
flight(X,Y) :- direct_flight(X,Y).
flight(X,Z) :- direct_flight(X,Y), flight(Y,Z).
(Read ":-" as "if".)
```

Queries

- Logic programming is driven by queries about relations.
- The simplest queries ask whether a particular tuple belongs to a relation. Examples: Is `(hongkong,tokyo)` in the relation `direct_flight`? Is `(hongkong,vancouver)` in the relation `direct_flight`?
- Queries containing variables are more interesting. For example, if you're looking for a flight from Hong Kong to Vancouver, you may first ask: Is `(hongkong,vancouver)` in the relation `direct_flight`? Failing that, you then ask: Is there a city X such that both `(hongkong,X)` and `(X,vancouver)` are in the relation `direct_flight`?

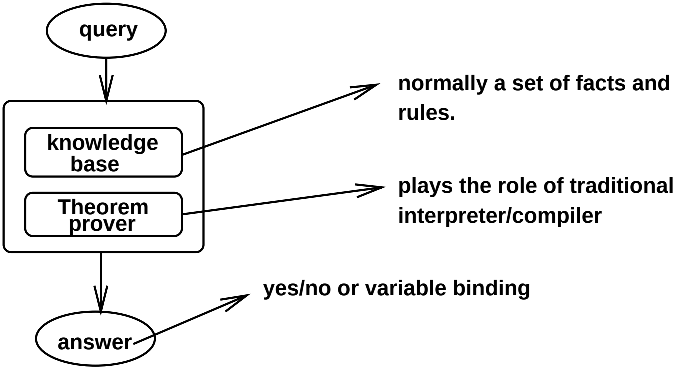
How To Answer Queries?

- Generally, a logic program is a set of sentences in a logic.
- If K is a logic program, Q is a query, then the answer to the query is "yes" if Q is entailed by K in the given logic.
- For example, our program about `flight` is the following knowledge base (database):

```
direct_flight(hongkong, tokyo).
direct_flight(hongkong, beijing).
direct_flight(hongkong, sanfrancisco).
direct_flight(tokyo, hongkong).
direct_flight(tokyo, vancouver).
direct_flight(X,Y) -> flight(X,Y)
direct_flight(X,Y), flight(Y,Z) -> flight(X,Z).
```

- To answer, for example, the query `flight(hongkong,vancouver)`, we check to see if the sentence corresponding to this query is a logical consequence of the above knowledge base.

Logic Programming Framework


query
knowledge base
Theorem prover
answer
normally a set of facts and rules.
plays the role of traditional interpreter/compiler
yes/no or variable binding

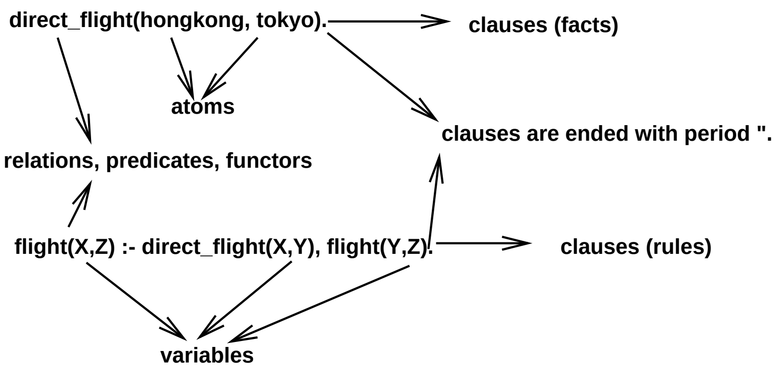
Prolog

- Prolog is the most widely used LP language.
- PROLOG stands for "PROgramming in LOGic".
- Prolog only implements a subset of logic : first order Horn clause logic. Because of this many people call Prolog a "Relational Programming" language.

An SWI-Prolog Session

```
csl2wk12:luo:114> pl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.6.64
Copyright (c) 1990-2008 University of Amsterdam.
....
| ?- [flight].
% flight compiled 0.00 sec, 2,416 bytes
true. <-- response from the system
| ?- df(hk,tk). <-- I typed the query
true.
| ?- df(hk,mc).
false.
| ?- df(hk,X).
X = tk ;  <-- I typed ";" and "return"
X = bj ;  <-- I typed ";" and "return"
X = sf.
```

Prolog Syntax Illustrated


direct_flight(hongkong, tokyo).
clauses (facts)
atoms
clauses are ended with period ".
relations, predicates, functors
flight(X,Z) :- direct_flight(X,Y), flight(Y,Z).
clauses (rules)
variables

All but clauses and separators are terms!

Atoms

Atoms. There are three types of atom:

1. Alphanumerics: Strings of letters, digits and "_". Must start with a lower case letter:

   `hongkong   tokyo   fred_Bloggs   a_Really_Silly_Atom`

2. Special character strings: Strings of the allowed special characters. May not contain letters, digits or "_": `>==>`, `----`, `<<<>>>`.
3. Quoted character strings: Strings of any character enclosed between ': `'Fred Bloggs'`, `'An atom with spaces'`. These are very useful for atoms beginning with upper case letters: `emperor('Octavius')`.

Variables

- Variables are strings of letters, digits and "_". Must start with an upper case letter or an "_":
```
X   Variable   L1_1   Fred   X_3   _23
```
- Prolog variables are *logical* variables, not *store* variables as in C++/Pascal. They are given values by *instantiation* rather than by *assignment*.
- *Anonymous variables*: Sometimes we need a variable as a place holder but its value is not important. We can use anonymous variables for the purpose, denoted by "_"
- Anonymous variables are common in queries:
```
| ?- parent('John',_). /*does John has a child?*/
true.
| ?-
```
  The "_" means : find a value for "_" which satisfies the query, but don't bother to tell me what that value is.

Variables (Continued)

- Anonymous variables are also useful in rules:

  ```
  killer(X) :- murdered(X,_).
  ```

- Can use more than one anonymous variable in a clause:

  ```
  | ?- parent(_,_).
  true.
  | ?-
  ```

  Each "_" means a different logical variable. In other words, this query is equivalent to the query `parent(X,Y).` except that Prolog does not list the pairs for which "parent" can be proved, it just says yes (true).

Numbers

- Prolog allows both integers and real.

```
23 0.23 12.3e34
10243 1.0234 -11.2e-45
-23 -12.23 1e45
```

- Prolog provides the bare minimum of numeric operations:

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`div`	Integer Division
`mod`	Integer Remainder

- Prolog is not good for numeric computations.

Structures

- `<atom> | <variable> | <number>` are simple terms.
- `<functor> (<terms>)` are *structures* or *compound terms*.
- *Functors.* These are names for relations. Must be alphanumeric atoms:
  `flight, direct_flight, emperor.`
- Examples of structures:
  ```
  flight(hongkong,tokyo)  flight(1,2)  emperor(qing)
  flight(hongkong, emperor(qing))
  book(moby_dick, author(herman,melville))
  ```
- Functors can be overloaded:
  ```
  emperor(qing) emperor(qing, "Qing Dynasty")
  emperor(X) :- emperor(X,_).
  ```

Clauses (Facts)

There are two types of clauses: facts and rules.
Generally, a fact has the form:

```
(1)   r(t1,...,tn).
```

where r is a functor (a relation name), and `t1,...,tn` are terms. If n=0, then this become a relation with no argument:

```
sick.
```

- Logically, (1) asserts the truthfulness of the corresponding sentence with all variables universally quantified:

  ```
  mortal(X). /* for all X, mortal(X) is true */
  ```

- In terms of relations, it says that all instantiations of the tuple `(t1,...,tn)` are in the relation r.

Clauses (Rules)

A rule has the form:

```
r(t1,...,tn) :- r1(t11,...,t1k),...,rm(tm1,...,tmj).
```

- Logically, it says that if the sentences on the right side (the *body of the rule) are true, then the sentence on the left side (the* head of the rule*) is also true. Again, variables are universally quantified.*

```
  brother(X,Y) :- brother(X,Z), brother(Y,Z).
/* for all X,Y, and Z, if X is Z's brother, and Y is
   also Z's brother, then X is Y's brother */
```

- *In terms of relations, it says that for any instantiation of the rule:*

```
(t11,...,tik) in r1
 ...                  implies (t1,...,tn) in r
(tm1,...,tmj) in rm
```

Clauses (Procedural Interpretation)

Given a query Q:

- If p is a fact, then to prove Q, try to match it with p, and return the binding, if any, as an answer. For example, given the query `flight(hk,sf)`, if `flight(hk,sf)` is a fact, then the two match exactly, and a yes (true) answer is returned. For the same query, if `flight(hk,X)` is a fact, then bind X to sf, and return the binding together with a yes answer.
- If `p := p1, ..., pn` is a rule, then to prove Q, try to match it with p, the head of the rule. If there is a match, then proceed to prove p1,...,pn. For example, given the query `flight(hk,sf)`, if we have the following rule:

  ```
   flight(hk,X) :- direct_flight(hk,Y), flight(Y,X).
  ```

  we'll prove the following to answer the query
  `direct_flight(hk,Y), flight(Y,sf).`

The Family Tree Example

```
mother('Elizabeth', 'Susan').
mother('Elizabeth', 'James').
mother('Elizabeth', 'Joyce').
mother('Susan', 'Kathryn').
mother('Sue', 'Robert').
father('Robert', 'Kathryn').
female('Kathryn').
male('James').
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
ancester(X,Y) :- parent(X,Y).
ancester(X,Z) :- parent(X,Y), ancester(Y,Z).
```

Now let's consider the query: `| ?- ancester(X,'Kathryn').`

List Structures

- A list is a sequence of items. In Prolog, a list is written using "square bracket" notation:

```
[]     the empty list
[1]
[1, 2]
[6, *, [9], is]
```

  Notice that the components of a list can be any type of object.

- To further simplify the notation, Prolog uses "|":

```
[1 | [5, 6]] <=> [1, 5, 6]
[1, 2, 3 | [4, 5]] <=>  [1, 2, 3, 4, 5]
```

  In general: `[ V1, V2, ... Vn | Tail ]` denotes a list containing "V1" ... "Vn" followed by whatever is in the list "Tail".

Relations On Lists

- cons constructs a new list from the given head and tail:

  `cons(1, [2], X) is true iff X is [1, 2]`

- Generally, `cons(X, Y, Z)` is true if Z is the list whose head is X, and whose tail is Y, that is, `[X | Y]`:

  ```
   cons(X,Y,Z) :- Z = [X | Y].
  ```

- There is another, more concise way of defining cons:

  ```
  cons(X, Y, [ X | Y ]).
  ```

- We can similarly define:

  ```
  head([ X | Y ], X).
  tail([ X | Y ], Y).
  ```

List Relations (Continued)

- We can now ask queries about cons any way we want:

```
| ?- cons(1,[ 2,3,4 ], L).
L = [1,2,3,4].

| ?- cons(Head, Tail, [ 1,2,3,4,5 ]).
Tail = [2,3,4,5],
Head = 1.

| ?- cons(1, X, Y).
Y = [1|X].
```

List Relations (Member)

Another interesting relation on lists is to test whether a given element is in a list: How do we define the `member` relation so that `member(X,Y)` holds iff X is one of the elements in the list Y?

```
  member(X, Y) :- Y = [ X | Z].
  member(X, Y) :- Y = [ Z1 | Z2], member(X,Z2).
or
  member(X, [ X | Y ] ).
  member(X, [ Y | Z ] ) :- member(X, Z).

 | ?- member(X, [ 1,2,3 ]).
 X = 1 ;
 X = 2 ;
 X = 3 ;
 false.
```

List Relations (Append)

`append(L1,L2,L3)` concatenate (append) L1 and L2 into L3:

```
append([],L, L).
append([X | X1], Y, [X | L]) :- append(X1, Y, L).
```

In SWI-Prolog, append is a built-in predicate.

List Relations (Append, Continued)

```
| ?- append([1,2,3],[4,5,6],L).
L = [1,2,3,4,5,6]
true.

| ?- append(L1, L2, [1,2,3,4]).
L1 = [],
L2 = [1,2,3,4] ;
L1 = [1],
L2 = [2,3,4] ;
L1 = [1,2],
L2 = [3,4] ;
L1 = [1,2,3],
L2 = [4] ;
L1 = [1,2,3,4]
L2 = [] ;
false.
```

List Relations (Split)

- A type of useful list operators are filters : find all the elements of a list that satisfy a certain condition.
- The filter "split" separates a list into two other lists, Less and NotLess:
  - Less: All values in the original list less than a given value.
  - NotLess: All values in the original list not less than a given value.
- `split(X,L,Less,NotLess)`: split L into Less (those less than X) and NotLess (those not less than X).

```
split(X, [], [], []).
split(X, [Y | Y1], [ Y | Less ], NotLess) :-
     Y < X, split(X, Y1, Less, NotLess).
split(X, [Y | Y1], Less, [ Y | NotLess ]) :-
     Y >= X, split(X, Y1, Less, NotLess).
```

List Relations (Split, Continued)

```
| ?- split(3, [1,2,3,4,5], L, NL).
L = [1,2],
NL = [3,4,5] ;
false.

| ?- split(hello, [hello, mum], L, NL).
ERROR: </2: Arithmetic: ‘hello/0’ is not a function
   Exception: (7) split(hello, [hello, mum], _G211, _G212) ?

(Type ’h’ for a list of commands, and ’a’ to abort it.)
```

qSort

- Using "split" and "append" sorting lists of numbers is now straight forward.
- `qSort(X,Y)` if Y is a permutation of X, and Y is sorted.

```
qSort([],[]).
qSort([ X | X1 ], Result) :-
        split(X, X1, Less, NotLess),
        qSort(Less, SortLess),
        qSort(NotLess, SortNotLess),
        append(SortLess, [ X | SortNotLess ], Result).
```

qSort (Continued)

```
| ?- qSort([9,1,10,45,33,2], L).

L = [1,2,9,10,33,45]

| ?-
```

qSort cannot be run "backwards" to find which list could be sorted into another list because split relies on arithmetic operators which require that their arguments are already instantiated.

```
| ?- qSort(L, [1,2,3]).
ERROR: </2: Arguments are not sufficiently instantiated
   Exception: (9) split(_G289, _G290, _L145, _L146) ?
```

Substitutions and Unification

- *Unification* is central to Prolog: How do we match a query with a given fact? How do we match a query with the head of a rule?
- Unification is defined in terms of *substitutions*.
- A substitution is a finite set of the form:

$$\{v_1/t_1, ..., v_n/t_n\}$$

  where v_i's are *distinct* variables, and t_i's are terms. The empty set is a substitution: $n = 0$.
- Each v_i/t_i is called a *binding*: the variable v_i is bound to t_i (replace t_i for all occurrences of v_i).
- Examples: `{X/a}` `{X/a, Y/f(a)}` `{X/Y, Y/X}`.
- Non-examples: `{X/a, X/b}` `{a/X}` `{f(X)/f(a)}`.

Substitutions and Unification (Continued)

- If t is a term, and σ a substitution, then $t\sigma$ is the standard notation for the result of applying substitution σ to term t: if the binding v/t' is in σ, then all occurrences of v in t are replaced by t'.

```
mother(X,a){X/b, Y/c} = mother(b,a)
mother(X,a){Y/b, Z/c} = mother(X,a)
append([],Y,Y){Y/[a,b,c]} =
              append([],[a,b,c],[a,b,c])
mother(X,Y){X/Y,Y/X} = mother(Y,X)
```

- A term u is an *instance* of t, if $u = t\sigma$, for some substitution σ: The following are all instances of `mother(X,a)`

```
mother(b,a), mother(c,a), mother(Y,a),
mother([a,b,c],a), ...
```

Most General Unifiers (MGUs)

- Two terms t_1 and t_2 *unify* if $t_1\sigma = t_2\sigma$ for some substitution σ, which is call a *unifier*.

  ```
  mother(X,a) and mother(b,a) unify: {X/b}
  cons(X,Y,[X | Y]) and cons(a, [b,c], [a,b,c]]) unify
  f(X) and f(Y) unify: {X/Y}, {Y/X}, {X/a, Y/a},
    {X/f(a),Y/f(a)}, {X/g(Z), Y/g(Z)}, ...
  ```

- *Most General Unifier.* A unifier σ of t_1 and t_2 is called a most general unifier (mgu) if for all other unifier σ', $t_1\sigma'$ is an instance of $t_1\sigma$. (This means that $t_2\sigma'$ is an instance of $t_1\sigma$ as well.)

  ```
  f(X) and f(Y): {X/Y} is an mgu. So are: {Y/X},
  {X/Z, Y/Z}. But not {X/a, Y/a} nor {X/f(Z),Y/f(Z)}
  ```

- So mgu is not unique. But all mgus of two terms are equivalent in a sense.

Most General Unifiers (Continued)

How do we prove that `{X/Y}` is a mgu for `f(X)` and `f(Y)`, but `{X/a,Y/a}` is not?

- σ_1 = `{X/a,Y/a}` is not an mgu because σ_2 = `{X/Y}` is a unifier for `f(X)` and `f(Y)`, but `f(X)`σ_2 = `f(X)` is not an instance of `f(X)`σ_1 = `f(a)`.
- `{X/Y}` is a mgu for `f(X)` and `f(Y)` because for any other unifier σ, `f(X)`σ = `f(t)`, for some term `t`, is an instance of f(X).

Most General Unifiers (Continued)

Some exercises on mgu:

```
   f(X,a) and f(a,Y)

we need X=a and a=Y, so an mgu is {X/a, Y/a}

   f(h(X,a),b) and f(h(g(a,b),Y),b)

we need h(X,a)=h(g(a,b),Y) and b=b,
so an mgu is {X/g(a,b), Y/a}

   f(W,g(Z),Z) and f(X,Y,h(X))

we need W = X, g(Z) = Y, Z = h(X)
so an mgu is {W/X, Y/g(Z), Z/h(X)}?
```

Mgu and Prolog

Unification (mgu) is the central operation in Prolog. In fact, the operator "$=$" computes mgu (somtimes).

```
| ?- f(W,g(Z),Z) = f(X,Y,h(X)).
W = X, Z = h(X), Y = g(h(X)).

| ?- append([b],[c,d],L) = append([X|L1],L2,[X|L3]).
L = [b|L3], X = b, L1 = [], L2 = [c,d].

| ?- X = 3+2.
X = 3+2.

| ?- 5 = 3+2.
false.

| ?- X is 3+2.
X = 5.
```

Prolog Search Trees

A Prolog search tree is conditioned on the following two inputs:

1. A Prolog program, which is a sequence of clauses (facts and rules). (As we shall see later, the order of clauses matters.)
2. A query, which is a sequence of terms $G_1, \ldots, G_k$, $k \geq 1$.

A Prolog program:

```
  p1:  parent(a,b).
  p2:  parent(a,c).
  p3:  parent(b,d).
  p4:  parent(b,e).
  p5:  parent(d,f).
anc1:  ancestor(X,Y) :- parent(X,Y).
anc2:  ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

A query: `ancestor(X,f), ancestor(X,e).` (Find e's and f's common ancestors.)

Prolog Search Trees

- To appreciate Prolog search trees (procedural interpretation of Prolog programs), it helps to understand first the logical meaning of Prolog programs and queries.
- A Prolog program is like a logical theory, and a query is like a goal to prove from the logical theory.
- The key with Prolog search trees is that if you want to prove the goal $G_1, G_2, \ldots, G_k$, and you have a rule of the form:

  $$B_1, \ldots, B_n \longrightarrow G_1$$

  then the problem of proving the original goal can be reduced to proving the following new goal:

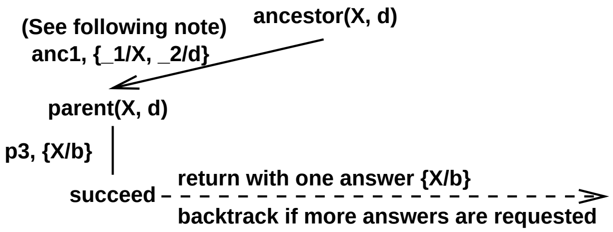
  $$B_1, \ldots, B_n, G_2, \ldots, G_k$$

- In a Prolog search tree, nodes represent goals to prove: the root is the original query, the top goal to prove.

Prolog Search Trees (Cont'd)

- If a node N1 is a child of the node N2, then the problem of proving the goal for N2 can be solved by (reduced to) proving the goal for N1.
- The empty goal means nothing to prove, thus is "succeeded". A leaf, which is a node without children, with non-empty goal is a dead-end: there is no way to prove the goal, and is thus "failed".
- Final complication: rename variables when necessary. Variables in a goal (query) may happen to have the same name as those in a clause, but they are different variables.

Prolog Search Trees (An Example)



Note:

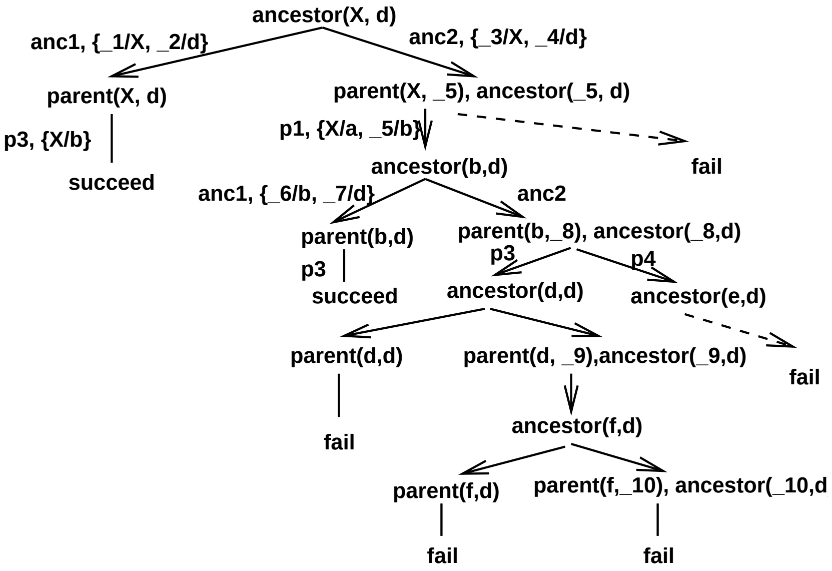
Every time a clause is matched with a query (goal), the variables in th clause are renamed to avoid conflicts with variables in the goal.

In our example, we have renamed the variables in anc1 into:

ancestor(_1, _2) :- parent(_1, _2).

{_1/X, _2/d} is the mgu of ancestor(X,d), the goal, and ancestor(_1,_2), t head of the clause.

Prolog Search Trees (Example, Continued)



Prolog Search Trees (Continued)

A search tree may be infinite:

Given the following program that consists of a single claus

```
p :- p.
```

The following is the search tree for the query p:

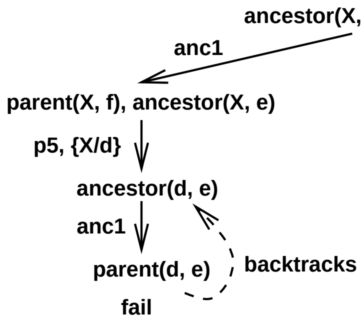
$$\begin{tikzcd} p \arrow[d] \\ p \arrow[d] \\ p \arrow[d] \\ \vdots \end{tikzcd}$$

Prolog Search Strategy

- Given a query, the Prolog interpreter does not generate the whole search tree.
- It employs a "depth-first" search strategy, and expands the tree as it goes along.
- Starting at the root, it generates first the leftmost child of a node.
- Once a child node is generated, it immediately moves on to the newly generated child node.
- Only when a node fails (a node with non-empty goal, but has no children), it *backtracks* to the nearest ancester node for which another child node can be generated, and the process continues.

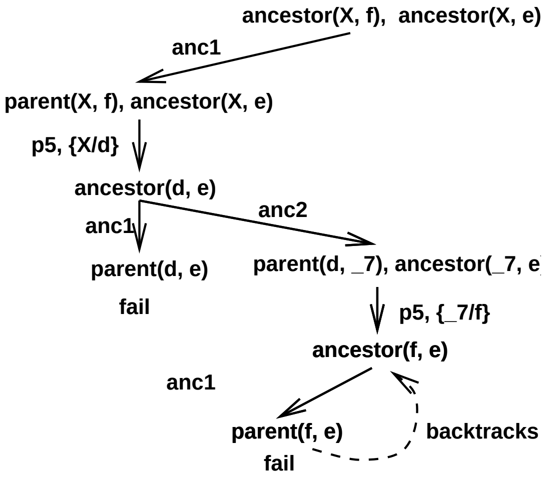
The next couple of slides shows illustrate this search strategy, and the process of backtracking.

Prolog Search Strategy (Continued)

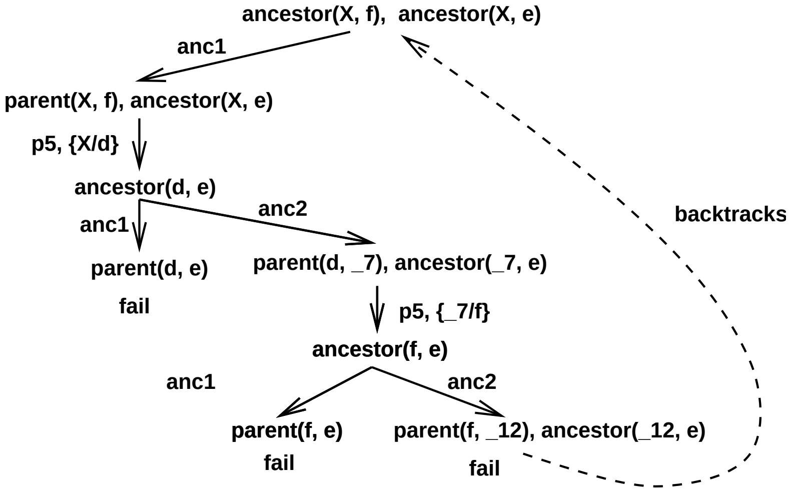


Note: To simplify the presentation of search tree, we only label arrows wi
rules and the bindings for the variables appearing in the parent nodes.
The bindings for other variables are not significant, and will not be show

Prolog Search Strategy (Continued)



Prolog Search Strategy (Continued)


ancestor(X, f), ancestor(X, e)
anc1
parent(X, f), ancestor(X, e)
p5, {X/d}
ancestor(d, e)
anc1
anc2
parent(d, e)
fail
parent(d, _7), ancestor(_7, e)
p5, {_7/f}
ancestor(f, e)
anc1
anc2
parent(f, e)
fail
parent(f, _12), ancestor(_12, e)
fail
backtracks

Goal Order Changes Solutions

- Recall that a goal is a sequence of terms: $G_1, \ldots, G_k$.
- For each $1 \leq i \leq k$, G_i is called a *subgoal*.
- In Prolog search trees, a rule is always applied to the leftmost subgoal. In other words, to prove the goal $G_1, \ldots, G_k$, Prolog always tries to prove the leftmost subgoal G_1 first.
- This means that the order of subgoals matters.
- The order for subgoals comes from two sources: the order of terms in the original query and the order of terms in the body of a rule.
- Change either of them, you may also change the answer to the query.

Goal Order Changes Solutions (Continued)

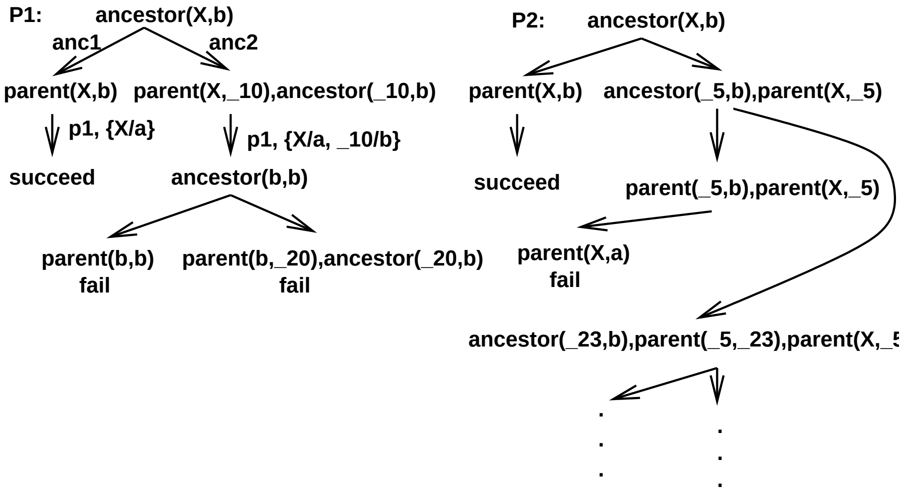
Compare the following two programs:
P1:

```
  p1:  parent(a,b).
anc1:  ancestor(X,Y) :- parent(X,Y).
anc2:  ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

P2:

```
  p1:  parent(a,b).
anc1:  ancestor(X,Y) :- parent(X,Y).
anc2:  ancestor(X,Y) :- ancestor(Z,Y), parent(X,Z).
```

Goal Order Changes Solutions (Continued)


P1:
ancestor(X,b)
anc1
anc2
parent(X,b)
parent(X,_10),ancestor(_10,b)
p1, {X/a}
p1, {X/a, _10/b}
succeed
ancestor(b,b)
parent(b,b)
fail
parent(b,_20),ancestor(_20,b)
fail
P2:
ancestor(X,b)
parent(X,b)
ancestor(_5,b),parent(X,_5)
succeed
parent(_5,b),parent(X,_5)
parent(X,a)
fail
ancestor(_23,b),parent(_5,_23),parent(X,_5

Rule Order Changes Solutions

- Recall that a Prolog program is a sequence of clauses (rules).
- The order of rules matters because Prolog uses a search strategy that always visit the leftmost child, which is created by applying the first applicable rule, first.

Consider the following two simple programs:

```
P1:                     P2:

p(a).                   p(X) :- p(X).
p(X) :- p(X).           p(a).
```

For the query `p(a)`, P1 will answer "yes", but P2 will go into a loop.

Cuts (Motivation)

- We need to limit the size of search spaces to do any useful computation without running out of memory.
- This can be done to a certain degree using clause and goal ordering.
- However, often the problem is with backtracking which a lot of time is pointless, and it is a waste of memory to store the choice points. Consider the following program:

```
r1:  roo(X, 0) :- X < 3.
r2:  roo(X, 3) :- 3 =< X, X < 6.
r3:  roo(X, 6) :- 6 =< X.

    | ?- roo(1, Y), 2 < Y.
```

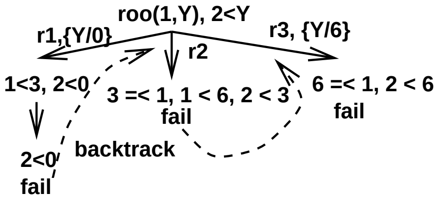
Cuts (Motivation, Continued)

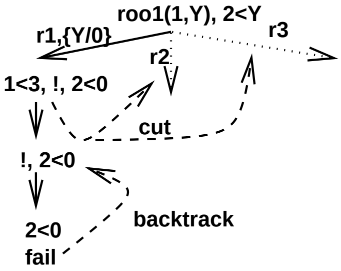
- The query will fail. Furthermore, we know that as soon as the first subgoal: `roo(1,Y)` matches with the first clause `r1` because if r1 succeeds then r2 and r3 will fail, and if r1 or r2 succeed then r3 will fail.
- It is desirable such pointerless backtracking be avoided:
  - Query will run faster.
  - Query will use less memory since the additional search space for r1 and r2 will not be generated.
- But Prolog is not smart enough to know that, we need a way to tell it.
- This is where *cuts* come in:

```
r1:  roo1(X, 0) :- X < 3, !.
r2:  roo1(X, 3) :- 3 =< X, X < 6, !.
r3:  roo1(X, 6) :- 6 =< X.
```

Cuts

- "!" (cut) is a special symbol in Prolog.
- It can appear only in the body of a clause as a subgoal. (Actually, it is legal to include it in a query. But this is pointless so we'll ignore this case.)
- As a goal, it always succeeds. What's interesting is its *side effect*: it "cuts" (prunes) the search space.





Cuts (Examples)

"Everyone has two biological parents, except Adam and Eve who have none."

```
num_parent(adam, 0) :- !.
num_parent(eve, 0) :- !.
num_parent(X, 2).

| ?- num_parent(eve,X).
X = 0.

| ?- num_parent(fred,X).
X = 2.

| ?- num_parent(eve,2).
true.
```

Cuts (Examples)

A better solution?

```
num_parent(adam, X) :- !, X = 0.
num_parent(eve, X) :- !, X = 0.
num_parent(X, 2).
```

```
| ?- num_parent(eve,X).
X = 0.

| ?- num_parent(fred,X).
X = 2.

| ?- num_parent(eve,2).
false.

| ?- num_parent(X,0).
X = adam ;
false.  /* Quiz: how to have it also return X = eve */
```

Cuts (Examples)

Recall our membership relation:

```
member(X, [ X | Y ] ).
member(X, [ Y | Z ] ) :- member(X, Z).
```

What if we change it into:

```
member1(X, [ X | Y ] ) :- !.
member1(X, [ Y | Z ] ) :- member1(X, Z).
```

- This is ok when both arguments are instantiated.
- Can't be used for finding all members of a list:

```
| ?- member(X,[1,2,3]).  | ?- member1(X,[1,2,3]).
X = 1 ;                  X = 1.
X = 2 ;
X = 3 ;
```

Check your knowledge of cut: given the following program

```
q(1).
q(2).
r(a).
r(b).
p(X,Y) :- q(X),!,r(Y).
p(3,c).
s(X,Y) :- p(X,Y).
s(4,d).
```

what are the answers to the query $s(X, Y)$?

Negation As Failure

What does "false" in Prolog mean?

```
president(bush,usa).
president(lincoln,usa).
president(washington,usa).

| ?- president(clinton,usa).
false.
```

A "false" does not mean that the assertion corresponding to the query is false; it means that it is not in our database. We can easily implement a version of such negation using cuts:

```
not(X) :- X, !, fail.
not(_).
```

The SWI-Prolog's built-in operator \+ is defined exactly this way.

Negation As Failure (Continued)

```
| ?- \+ president(clinton,usa).
true.
| ?- \+ president(washington,usa).
false.

| ?- X = 2, \+ X = 1.
X = 2.

| ?- \+ X = 1, X = 2.
false.
```

Negation As Failure (Continued)

When are two lists disjoint?

```
disjoint(L1,L2) :- \+ (member(X,L1), member(X,L2)).
```

(L1 is disjoint from L2 if there is no element X that is a member of both L1 and L2.)

Assert and Retract

These are special Prolog predicates that add or delete clauses from the current knowledge base (program).

- `asserta(C)`: add the clause `C` at the beginning of the knowledge base.
- `assertz(C)`: add the clause `C` at the end of the knowledge base.
- `assert(C)`: the same as `assertz(C)`.
- `retract(C)` delete the first clause in the knowledge base that unifies with `C`.
- `retractall(C)` delete all clauses in the knowledge base that unify with `C`.

Assert and Retract (Continued)

```
  p1:  parent(a,b).
  p2:  parent(a,c).
  p3:  parent(b,d).
  p4:  parent(b,e).
  p5:  parent(d,f).
anc1:  ancestor(X,Y) :- parent(X,Y).
anc2:  ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

  | ?- asserta(parent(a,l)).
  true. /*parent(a,l) is added before p1*/
  | ?- retract(parent(X,Y)).
  X = a, Y = l ; /* the newly added is first to go*/
  X = a, Y = b.   /* p1 is gone now*/

  | ?- parent(a,b).
  false.
```

Computing the length of a list with retract and assert

```
:- dynamic counter/1.

my_length(L,N) :- assert(counter(0)),
   member(X,L), counter(M), M1 is M+1,
   retract(counter(M)), assert(counter(M1)), fail.
my_length(L,N) :- counter(N), retract(counter(N)).
```