Principles of Programming Languages
COMP251: Logic Programming in Prolog

Prof. Dekai Wu

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China

Fall 2007

Part I

Introduction

A Typical AI Problem

There are three musician: Alan, Bill, and Carl.

One of them is a guitarist, one of them is a drummer, and one of them is a pianist.

One day, the drummer would like to hire the guitarist to do a recording, but somebody told him that the guitarist and the pianist had gone out of town for performance together. The drummer then went to their performance and really impressed with the show. There are more facts:

- guitarist earns more money then drummer.
- Alan earns less than Bill.
- Carl had never heard of Bill.

Question: What instruments do Alan, Bill, and Carl play?

Logic Programming

- *Imperative programming* implements a function using control structures and assignments which change the state of the machine (computation).

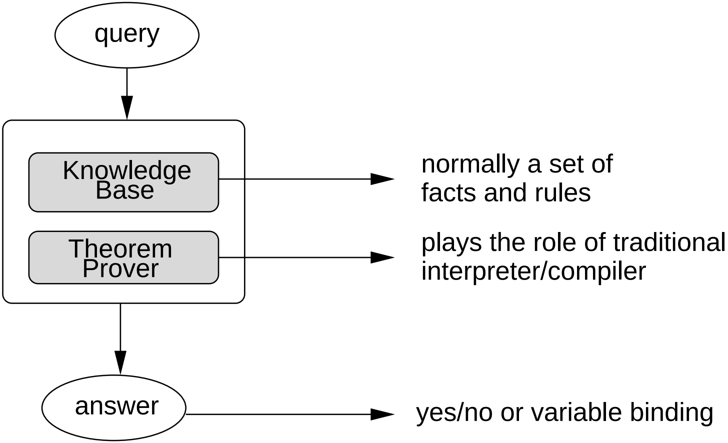
- *Functional programming* implements a function using function composition of simpler or primitive functions, and function applications.

- *Logic programming* specifies a set of relations among the objects of interest — the logic part of an algorithm.

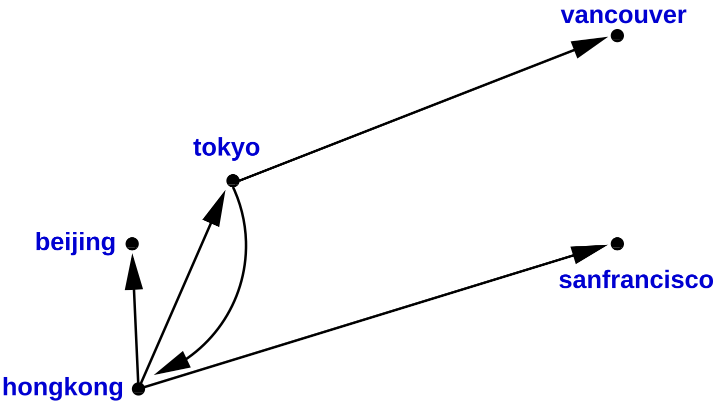
$$\mathit{Algorithm} = \mathit{Logic} + \mathit{Control}$$

  - A program is an algorithm.
  - We programmers specify the logic.
  - Logic programming language supplies the control.

Logic Programming Framework



Example: flights



Example: Specify a Relation in a Table

- A concrete view of a relation is as a table.
- e.g. the following table specifies a flight connection relation:

direct_flight	
DEPARTURE	ARRIVAL
hongkong	tokyo
hongkong	beijing
hongkong	sanfrancisco
tokyo	hongkong
tokyo	vancouver

- A table completely specifies a relation: a tuple $(X_1, \ldots, X_n)$ is in the relation *iff* it is in the table of the relation.
- A tuple `(X,Y)` is in the relation `direct_flight` if it is in the above table.

Specify a Relation By Facts

- A relation is often called a predicate.
- e.g. Instead of saying that the tuple `(hongkong,tokyo)` is in the relation `direct_flight`, we say that the boolean predicate `direct_flight(hongkong,tokyo)` is *true*.
- The above table for `direct_flight` is represented as a set of *facts* in Prolog as:

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

Specify a Relation By Rules

- Sometimes it can be difficult or even impossible to give a table for a relation — e.g. an infinite relation. Instead, we use *rules* to describe a relation.
- In particular, recursive rules are usually used to define a relation.
- Examples:
  - 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` is given by the two rules:

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

- ":-" is read as "if".

Queries About a Relation

Logic programming is driven by queries about relations.

- The simplest queries ask if a tuple belongs to a relation. e.g.

  Is `(hongkong,tokyo)` in the relation `direct_flight`?
  Is `(hongkong,vancouver)` in the relation `direct_flight`?

- Queries containing variables are more interesting.
  e.g., you're looking for a flight from HK to Vancouver.
  - you may first ask:
    Is `(hongkong,vancouver)` in the relation `direct_flight`?
  - If it fails, you then ask:
    Is there a flight from HK to Vancouver via some city X?

    i.e. 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.

- For example, our logic program about `flight` consists of the following knowledge database:

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

- If K is a logic program and Q is a query, then the answer to the query is "yes" if Q is entailed by K.
- Thus, to answer the query `flight(hongkong,vancouver)`, we check to see if the sentence corresponding to this query is a logical consequence of the above knowledge database.

Logic Programming With Relations

Computing with relation is more flexible than with function.

- If a program implements a function `foo(x)`, then this program can also be taken as a specification of the relation:

  `{ (x, y) | y = foo(x) }`

- Relations treat arguments `x` and results `y` 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 x_1, and ask it to find some y_1 such that $\mathtt{R}(x_1, y_1)$ holds.
  - We can also supply a y_2, and ask it to find some x_2 such that $\mathtt{R}(x_2, y_2)$ is true.
- e.g., if we define `R(x,y)` holds iff `square(x) = y`, then
  - we can ask for some or all `y` such that `R(2,y)` holds; or,
  - we can also ask for some or all `x` such that `R(x,4)` holds.

Prolog

- Although LP is more than Prolog, it 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.
- It has always been an ambition of the Mathematics and Computing Science communities to construct systems that would prove theorems automatically.
- The first actual implementation was done by Alain Colmerauer in collaboration with Kowalski at Marseille University where it was used for (among other things) natural language processing and AI.

Prolog ..

- Widespread interest in Prolog really began when David Warren of Edinburgh University produced the first efficient implementation based on the Warren Abstract Machine.
- Prolog was a major component of the Japanese 5th Generation Project which seems to have had mixed fortunes.
- Prolog is widely used in industry for
  - expert systems,
  - artificial intelligence
  - natural language processing & computational linguistics
- It has also found some use as a
  - relational database prototyping language
  - rapid prototyping systems of industrial software

Main References

- W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Library open reserve.
- *Using SWI Prolog – The Basics*. Available on the course website.
- The following websites contain everything that you'll want to know about Prolog. It includes pointers to free Prolog interpreters and compilers for PC.
  - `http://www.cs.cmu.edu/Groups/AI/lang/prolog/0.html`
  - `http://www.swi-prolog.org`

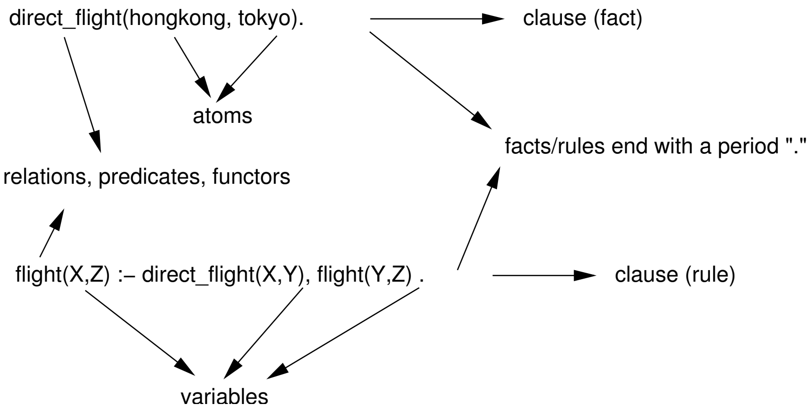
A SWI Prolog Session

```
Welcome to SWI-Prolog (Multi-threaded, Version 5.4.7)
Copyright (c) 1990-2003 University of Amsterdam. ...
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- [flight].      /* Load the program "flight.pl" */
% flight compiled 0.01 sec, 1,576 bytes
Yes

?- direct_flight(hongkong,tokyo). /* I type this */
Yes
?- direct_flight(hongkong,seoul).
No
?- direct_flight(hongkong,X).
X = tokyo ;         /* System response, then I typed ";" */
X = beijing ;
X = sanfrancisco ;
No
?- ^D
% halt
```

Part II

Syntax, Fact, Rule, Program

Prolog Syntax Illustrated



- All but clauses and separators are terms.

Prolog Syntax Defined

```
<fact>    ::= <atom> | <functor> (<terms>)
<rule>    ::= <term> :- <terms>.
<query>   ::= <terms>.

<terms>   ::= <term> | <terms>, <term>
<term>    ::= <atom> | <variable> | <number> | <functor> (<terms>)
<functor> ::= <atom>
```

- All Prolog data objects are terms.
- Prolog is weakly typed.

Three types of atom

1. Alphanumerics: Strings of letters, digits and "_". It must start with a lower case letter. e.g.

   `hongkong` `tokyo` `fred_Bloggs` `a_Really_Silly_Atom`

2. Special character strings: Strings of the allowed special characters. May not contain letters, digits or "_". e.g. `>==>`, `----`, `<<<>>>`.

3. Quoted character strings: Strings of any character enclosed between `'`. e.g. `'Fred Bloggs'`, `'An atom with spaces'`.

   These are very useful for atoms beginning with upper case letters. e.g. `emperor('Octavius')`.

Variables

Variables are strings of letters, digits and "_". It must start with an upper case letter or a "_".

```
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* as denoted by "_" are "don't care" variables. e.g.

```
| ?- parent('John', _). /* Does John have a kid? */
yes
| ?-
```

The "_" means : find a value for "_" which satisfies the query, but don't bother to tell me what that value is.

Variables ..

- Anonymous variables are also useful in rules:

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

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

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

Each of the "_" 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".

Numbers

Prolog allows both integers and real numbers. One restriction: there must be at least one digit before and after a decimal for a real number.

```
23      0.23      12.3e34
10243   1.0234    -11.2e-45
```

- Prolog provides the bare minimum of numeric operations:

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

- Prolog is very lousy for numeric computations.

Structures

1. Simple Terms: `<atom> | <variable> | <number>`.
2. Functors: Names for relations/predicates. They must be alphanumeric atoms. e.g.
   `flight, direct_flight, emperor`.
3. Structures or Compound Terms: `<functor>(<terms>)`.

Examples of structures:

```
flight(hongkong,tokyo)
flight(hongkong, emperor(qing))
book('Programming Languages', author(sethi))
```

Functors can be overloaded:

```
emperor(qing).
emperor(qing, 'Qing Dynasty').
emperor(X) :- emperor(X,_).
```

Variable Instantiation

- A variable in a term is instantiated when it is bound to some value.
- For example, when you input the query,

  ```
  flight(hongkong, X)
  ```

  the variable `X` is not bound to any value. Thus, `X` is not *instantiated*.
- Prolog replies to the query with,

  ```
  X = tokyo
  ```

  By then, variable `X` is *instantiated* or *bound* to the atom "`tokyo`".

Facts, Rules

- Fact = something that is *unconditionally* true.
- A fact is written as: $r(t_1, ..., t_n)$,
  where `r` is a *functor*, and $t_1, \ldots, t_n$ are terms.
- A rule contains at least one condition.
- A rule has the form: `<head> :- <body>`.

$$r(t_1, \ldots, t_n) \; :- \; r_1(t_{11}, \ldots, t_{1k}), \ldots, r_m(t_{m1}, \ldots, t_{mj}).$$

Logically, it says that if the terms in the right side are ALL true, then the term on the left side is also true. Variables are universally quantified. e.g.

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

Procedural Interpretation

To prove a query `Q`:

1. If `P` is a fact:

   Try to match it with `P`, and return the variable bindings, if any, as an answer.

   - e.g. Given the query `direct_flight(hongkong, sanfrancisco)`, if `P` = `direct_flight(hongkong, sanfrancisco)` is a fact, then the two match exactly, and a "`Yes`" answer will be returned.
   - e.g. If the query is `direct_flight(hongkong, X)`, then `X` is bound to `sanfrancisco`, and the system returns the binding together with a "`Yes`" answer.

Procedural Interpretation ..

1. If $P \;:-\; P_1, ..., P_n$ is a rule:
   - Try to match it with P — the head of the rule.
   - If there is a match, then recursively call to prove $P_1, \ldots, P_n$.

   e.g. Given the query `flight(hongkong,vancouver)`, it will match hongkong with the following rule:

```
flight(hongkong,Z) :- direct_flight(hongkong,Y), flight(Y,Z).
```

   - Then the system will recursively call to answer the query

```
direct_flight(hongkong,Y), flight(Y,vancouver).
```

   by trying to match the variable Y.
   (after putting Z = vancouver)

What is Logic Programming?

- Logic program = facts + rules: represents the knowledge/information.
- Based on the knowledge, answer queries by deduction.
- Closed world assumption:

  Anything that cannot be deduced from the given facts and rules is false!

Part III

Prolog Programming with Lists

In Prolog, a list is a built-in datatype (again, from Lisp).

- A Prolog list is a structure (compound term) using the binary *prefix* functor "." (c.f. list constructor cons in Scheme/Lisp and :: in ML)
- An empty list is pre-defined as : `[]`.
- Examples:

```
.(1, .(2, []))
.(hello, .(world, []))
.(6, .(*, .(9, .(is, []))))
```

- Since Prolog is weakly typed or latently typed, items in a list can be of mixed types.

List Structures ..

- Writing lists using the "." functor is unwieldy and error-prone.
- Prolog provides syntactic sugar by allowing the "," notation for lists (c.f. ML): e.g.

```
.(1, .(2, [])) <=> [1, 2]
.(hello, .(world, [])) <=> [hello, world]
.(6, .(*, .(9, .(is, [])))) <=> [6, *, 9, is]
.(1, .(.(two, .(three,[])), .(4,[]))) <=> [1, [two,three], 4]
```

- To make things even easier (especially when using pattern matching), Prolog allows another notation with "|". e.g.

```
[1 | [2]] <=> [1, 2]
[6, * | [9, is]] <=> [6, *, 9, is]
```

- In general, $[V_1, V_2, \ldots, V_n | Tail]$ means a list containing items, "V_1", ..., "V_n", followed by whatever is in the sub-list "Tail".

Predicates On Lists: cons, head, tail

- **cons** constructs a new list from a given head and a tail.

  `cons(Head, Tail, New_List)` is true if New_List is the list whose head is Head, and whose tail is Tail. That is, `[Head|Tail]`.

  ```
  cons(Head, Tail, New_List) :- New_List = [Head|Tail].
  ```

  For example:

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

- A more concise way of defining **cons**:

  ```
  cons(H, T, [H|T]).
  ```

- We can similarly define:

  ```
  head([H|T], H). /* or even:  head([H|_], H). */
  tail([H|T], T). /* or even:  tail([_|T], T). */
  ```

List Predicates

We can now ask queries about cons:

```
| ?- cons(1, [2,3,4], L).
L = [1,2,3,4];
No

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

| ?- cons(1, X, Y).
X = _7423,
Y = [1|X];
No
```

List Predicate: member

SWI-Prolog has a built-in predicate, member(X, List): X is a member of List.

```
member(X, List) :- List = [ X | Tail ].
member(X, List) :- List = [ Y | Tail ], member(X, Tail).
```

Or more concisely,

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

```
| ?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

List Predicate: delete

SWI-Prolog has a built-in predicate, delete(Old_List, X, New_List): delete an occurrence of X in Old_List to obtain New_List.

```
mydelete([], X, []).
mydelete([ X | Tail ], X, Tail).
mydelete([ H | Tail ], X, [H | L]) :-
    mydelete(Tail, X, L), L \== Tail
```

delete: Examples

```
| ?- mydelete([1,2,3], 1, L).
L = [2,3]
Yes
| ?- mydelete([1,2,3], 5, L).
No
| ?- mydelete([1,2,3], X, [1,3]).
X = 2
Yes
| ?- mydelete([1,a,1,b,1], 1, L).
L = [a,1,b,1];
L = [1,a,b,1];
L = [1,a,1,b];
No
```

List Predicate: append

SWI-Prolog has a built-in predicate, append(L1, L2, L3): concatenate (append) L1 and L2 into L3.

```
append([], L, L).
append([H | L], L2, [H | Tail]) :- append(L, L2, Tail).
```

append: Examples

```
| ?- append([1,2,3],[4,5,6],L).
L = [1,2,3,4,5,6]
Yes
| ?- 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 = [];
No
```

List Predicate: split

split(X, List, Less, Not_Less): split the list, List, into 2 smaller lists:

- Less list — containing those items $<$ item X
- Not_Less list — those items $\geq$ item X

```
split(X, [], [], []).
split(X, [H | Tail], [H | Less], Not_Less) :-
      H < X, split(X, Tail, Less, Not_Less).
split(X, [H | Tail], Less, [H | Not_Less]) :-
      H >= X, split(X, Tail, Less, Not_Less).
```

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

| ?- split(hello, [hello, mum], L, NL).
ERROR: Arithmetic: ‘hello/0’ is not a function
 Exception: (7) split(hello, [hello, sum], _G341, _G342)?
```

List Predicate: qsort

- Using **split** and **append** to implement qsort:
- qsort(X,Y) if Y is a permutation of X, and Y is sorted.

```
qsort([], []).
qsort([H | Tail], Sorted) :-
    split(H, Tail, Less, Not_Less),
    qsort(Less, Sorted_Less),
    qsort(Not_Less, Sorted_Not_Less),
    append(Sorted_Less, [H | Sorted_Not_Less], Sorted).
```

qsort: Examples

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

- 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: Arguments are not sufficiently instantiated
   Exception: (8) split(_G308, _G309, _G320, _G321) ?
```

Part IV

Substitutions and Unification

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:

$$\sigma = \{v_1|t_1, ..., v_n|t_n\}$$

  where v_i's are *distinct* variables, and t_i's are terms.
- The empty set is also 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}`.
- Wrong: `{X|a, X|b}` `{a|X}` `{f(X)|f(a)}`.

- 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.

$$t \xrightarrow{\sigma} t\sigma$$

- If the binding $v|t_1$ is in σ, then all occurrences of v in t are replaced by t_1.

```
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), ...
```

Unifier

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

t_1	t_2	UNIFIERS
mother(X,a)	mother(b,a)	{X|b}
cons(X,Y,[X|Y])	cons(a, [b,c], [a,b,c]])	{X|a, Y|[b,c]}
f(X)	f(Y)	{X|Y}, {Y|X}, {X|a,Y|a}, {X|f(a),Y|f(a)}, {X|g(Z),Y|g(Z)}, ...

Most General Unifier (MGU)

$$\begin{tikzcd}[column sep=huge, row sep=huge]
t_1 , t_2 \arrow[r, "\sigma'"] \arrow[d, "\sigma"'] & t_1\sigma' = t_2\sigma' = t' \\
t_1\sigma = t_2\sigma = t \arrow[ur, "\sigma''", "t\sigma'' = t'"'] &
\end{tikzcd}$$

- 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 $t_2\sigma$ as well.)
- For example, for the 2 terms: f(X) and f(Y):
  - {X|Y} is an mgu.
  - So are: `{Y|X}, {X|Z, Y|Z}`.
  - But not these: {X|a, Y|a}, nor `{X|f(Z),Y|f(Z)}`.

Example: Most General Unifier

Example: $t_1 = X, t_2 = Y$.

One possible MGU is {X|Y}.
One unifier is {X|a, Y|a}.

Therefore,

```
            {X|a, Y|a}
X , Y    ------------> a = a
                        ^
  |                     |
  | {X|Y}               |   {Y|a}
  v                     |
Y = Y    ---------------
```

Most General Unifiers ..

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

- $\sigma_1 = \{X|a, Y|a\}$ is not an mgu
  because $\sigma_2 = \{X|Y\}$ is a unifier for f(X) and f(Y),
  but $f(X)\sigma_2 = f(Y)$ is not an instance of $f(X)\sigma_1 = f(a)$.
- $\{X|Y\}$ is a mgu for f(X) and f(Y)
  because for any other unifier σ,
  $f(X)\sigma = f(t)$, for some term t, is an instance of f(Y).
- mgu is not unique. But all mgu's of two terms are equivalent in a sense.

Most General Unifiers: Examples

To find the MGU of 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 $\{W|X,\ Y|g(Z),\ Z|h(X)\}$?
- No. It is NOT even a unifier.
- Possible solutions:
  - $\{X|W,\ Z|h(W),\ Y|g(h(W))\}$
  - $\{W|X,\ Z|h(X),\ Y|g(h(X))\}$

Quiz

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

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

3. f(X,h(b,X)) and f(g(P,a), h(b, g(Q,Q)))
```

MGU and Prolog

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

```
?- f(W,g(Z),Z) = f(X,Y,h(X)).
W = X = _G189, Z = h(_G189), Y = g(h(_G189)) ;
No
?- append([b],[c,d],L) = append([X|L1],L2,[X|L3]).
L = [b|_G197], X = b, L1 = [], L2 = [c, d], L3 = _G197;
No
?- X = 3+2.
X = 3+2 ;
No
?- 5 = 3+2.
No
?- X is 3+2.
X = 5 ;
No
?- 5 is 3+2.
Yes
```

Part V

Prolog Search

Prolog Search Tree: Introduction

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

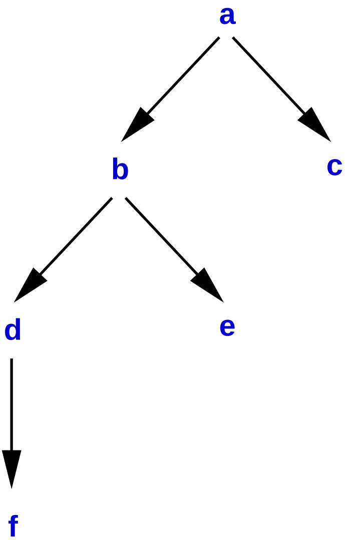
1. A Prolog program, which is a sequence of clauses (facts and rules). (As we shall see later, the order of clauses matters.)
2. A query, which is a sequence of terms $G_1, \ldots, G_k$, $k \geq 1$.

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 Tree: Introduction ..



Prolog Search Tree: Goals and Subgoals

- To study 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:

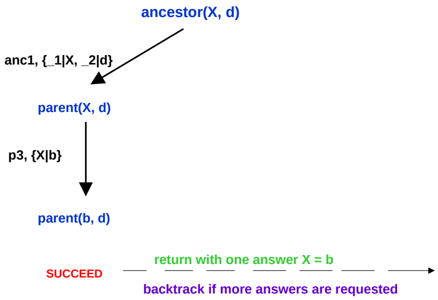
$$G_1 :- B_1, \ldots, B_n$$

  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 and Unification: Example

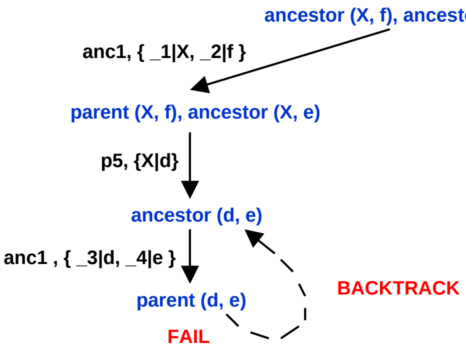


- Every time a clause is matched with a query(goal), the variables in the clause are renamed to avoid conflicts with variables in the goal.
- Here, we rename the variables X, Y in anc1 into _1, _2:

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

- `{_1|X, _2|d}` is the MGU of ancestor(X, d), the goal, and ancestor(_1, _2), the head of the clause.

Prolog Search and Backtracking: Example



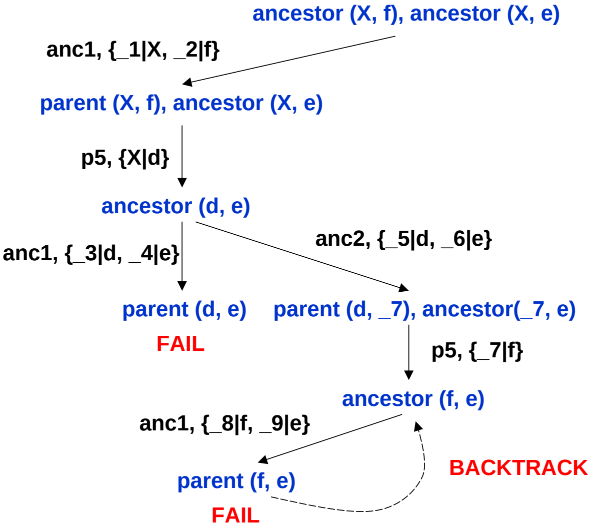
- To simplify the presentation of search trees, we only label arrows with rules and the bindings for variables appearing in the parent nodes. (The bindings for other variables are not significant and will not be shown.)

- If a node N1 is the only child of the node N2, then the problem of proving the goal for N2 can be reduced to (solved by) proving the goal for N1.
- The empty goal means nothing to prove, and it always "succeeds".
- 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 it always "fails".
- Final complication: rename variables whenever necessary. Variables in a goal (query) may happen to have the same name as those in a clause, but they are different variables.

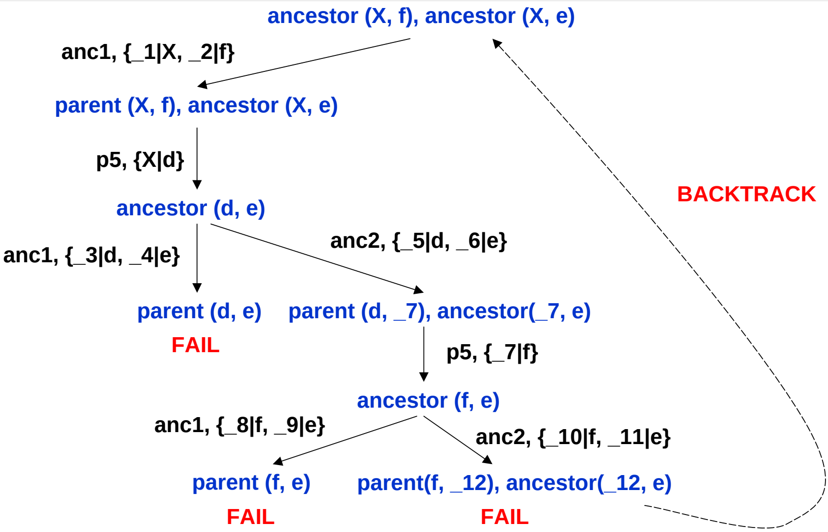
Prolog Search Strategy

- Given a query, the Prolog interpreter does *not* generate the whole search tree.
- It employs depth-first search, and expands the tree as it goes along.
- Starting at the root, it generates the first 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 ancestor node for which another child node can be generated, and the process continues.

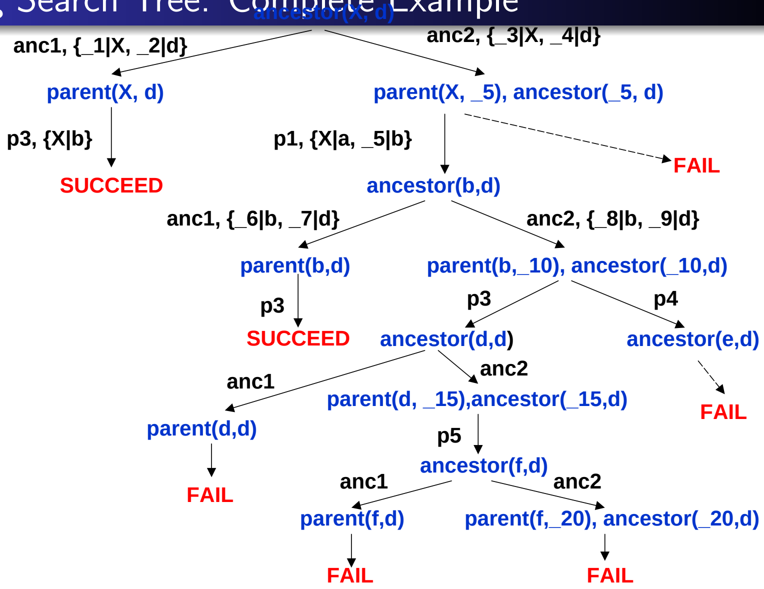
The next couple of slides illustrate this search strategy, and the process of backtracking.


ancestor (X, f), ancestor (X, e)
anc1, {_1|X, _2|f}
parent (X, f), ancestor (X, e)
p5, {X|d}
ancestor (d, e)
anc1, {_3|d, _4|e}
anc2, {_5|d, _6|e}
parent (d, e)
FAIL
parent (d, _7), ancestor(_7, e)
p5, {_7|f}
ancestor (f, e)
anc1, {_8|f, _9|e}
BACKTRACK
parent (f, e)
FAIL

Prolog Search Strategy: Example ..



Prolog Search Tree: Complete Example



Infinite Search Tree: Example

- A search tree may be infinite.
- The following program consists of a single clause:

```
p :- p.
```

- The following is the search tree for the query p.

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 first 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: Example

Compare the following two programs:

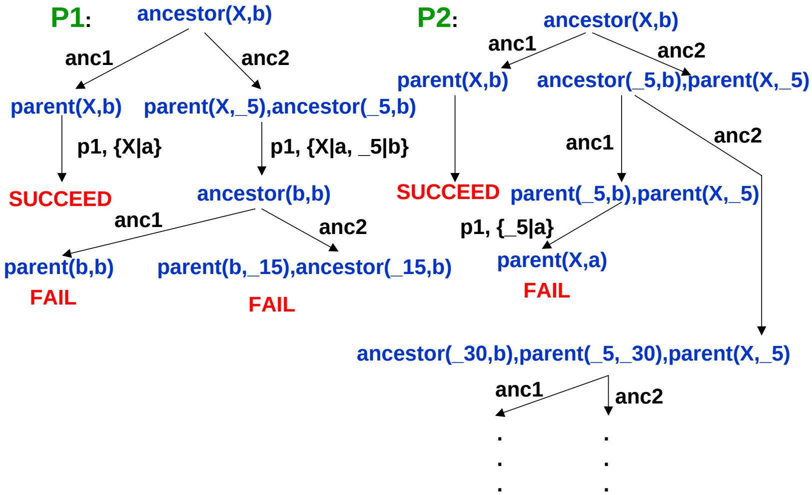
P1:

```
  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: Example ..



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 first, which is created by applying the first applicable rule.

Consider the following two simple programs:

```
P1:

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

```
P2:

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

For the query p(a), P1 will answer "Yes", but P2 will go into an infinite loop.

Part VI

Cut & Negation

Cuts: Motivation

- In practice, we need to *limit* the size of search space to do any useful computation without running out of memory.
- This can be done to a certain degree by re-ordering clauses and goals.
- 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.
```

And the query:

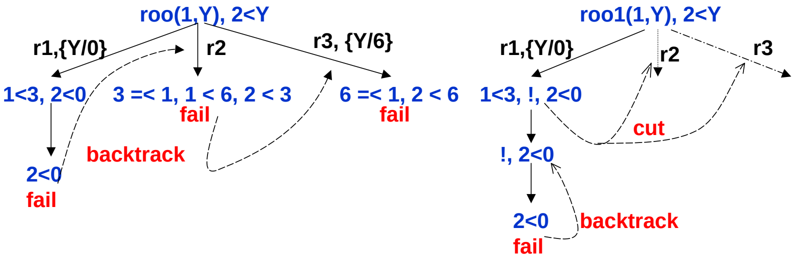
```
?- roo(1, Y), 2 < Y.
```

Cuts: Motivation ..

- The query will fail.
- 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.
  - if r1 or r2 succeed then r3 will fail.
- It is desirable such pointless 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.
```

- "!" (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 or prunes the search space.



Cuts ..

Given:

```
p :- q, !, r.
p :- t.
```

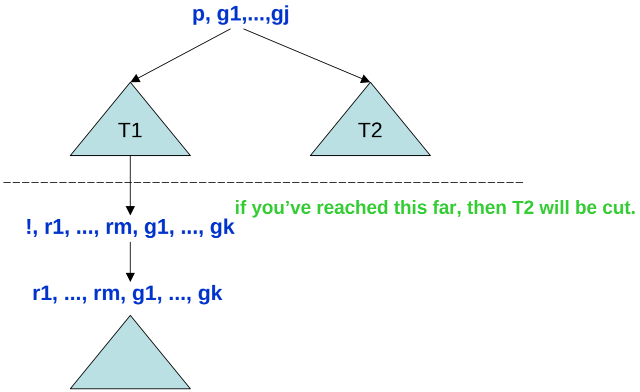
To prove p:

- If q succeeds, then p succeeds *only if* r succeeds. The alternative t will never be attempted even if r fails.
- If q fails, then t will be attempted.
- Effectively, "!" means that if you get this far, then you've made the only correct choice, and you succeed or fail with this choice.
- The above rules behave like an "if-then-else" expression:

in SML:	p = if q then r else t;
in C++:	p = (q) ? r : t;

The above interpretatio ed as follows:

p :- q1, …, qk, !, r1, …, rm.
p :- t1, …, tn



Cuts: Example 1

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 ;
No
| ?- num_parent(fred,X).
X = 2 ;
No
| ?- num_parent(eve,2).
Yes
```

Cuts: Example 2

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 ;
No
?- num_parent(fred,X).
X = 2 ;
No
?- num_parent(eve,2).
No
?- num_parent(X,0).
X = adam ;
No  /* Quiz: how to have it also return X = eve */
```

Cuts: Example 3

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

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

```
?- member1(X,[1,2,3]).
X = 1 ;
No
```

Recall the problem we had with our delete relation:

```
mydelete([], X, []).
mydelete([ X | Tail ], X, Tail).
mydelete([ H | Tail ], X, [H | L]) :-
    mydelete(Tail, X, L), L \== Tail
```

```
?- mydelete([1,2,1], 1, L).
L = [2,1] ;
L = [1,2] ;
No
```

Quiz: How do we write a delete function that delete only the first occurrence of the given item?

```
delete1([], _, []).
delete1([X|Y], X, Y) :- !.
delete1([Y1|Y2], X, [Y1|L]) :- delete1(Y2, X, L).
```

```
?- delete1([1,2,1], 1, L).
L = [2,1] ;
No
```

Negation As Failure

- What does "no" in Prolog mean?

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


?- president(clinton, usa).
no
```

- A "no" does not means 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(_).
```

Negation Example 1

```
?- not(president(clinton, usa)).
yes
?- not(president(washington, usa)).
no
```

```
?- X = 2, not(X = 1).
X = 2
yes
```

```
?- not(X = 1), X = 2.
no
```

Negation Example 2

- Bachelors?

```
bachelor(X) :- male(X), not(married(X)).
```

- Should we define married in terms of single or the other way around?

```
married(X) :- not(single(X)).
single(X) :- not(married(X)).
```

- When are two lists disjoint?

```
joint(L1,L2) :- member(X, L1), member(X, L2).
disjoint(L1, L2) :- not(joint(L1, L2)).
```

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

Cuts: Quiz

How many answers for the query s(X,Y) to the following program? And what are they?

```
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).
```

Part VII

Open List

Open List

```
[ a, b | X ]
```

- `[a, b |X]` is an open list.
- It ends in a variable, thus allowing its structure for further expansion.
- The variable X is referred to as the end marker of the open list.

```
?- L = [a, b|X].

L = [a, b|_G161]
X = _G161
```

- _G161 is a temporary variable generated by Prolog corresponding to the end marker X.
- An open list can be modified by unifying its end marker with new data.

Modifying an Open List

```
?- L = [a, b|X], X = [c|Y].

L = [a, b, c|_G167]
X = [c|_G167]
Y = _G167
```

- In the above example, the open list L is extended from 2 known items to 3 known items followed by a new end marker Y.
- An advantage of working with open lists is that the end of a list can be accessed quickly, in constant time, through its end marker.
- It is often used to represent data structures that require fast access at their ends.

Difference Lists

- A difference list is made up of two lists. e.g. L and E, where E unifies with a suffix of L.
- The contents of the actual list is

  L - E

  i.e. L after the removal of the suffix part represented by E.
- Examples of difference lists with contents [a,b]:

```
[a,b] - []
[a,b,c] - [c]
[a,b|E] - E
[a,b,c|F] - [c|F]
```

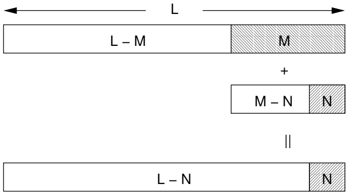
Example 1: append_dl

- Let's first recall the following recursive append predicate.

```
append([], L, L).
append([H | L], L2, [H | Tail]) :- append(L, L2, Tail).
```

- The difference list may be used to implement a non-recursive append_dl predicate that runs in constant time as follows.

```
append_dl(L-M, M-N, L-N).
```



- Notice how the end markers M and N are used as place-holders for pattern matching or unifications.

Example 1: append_dl ..

- To use the append_dl predicate, the difference lists must be used as follows:

```
append_dl([First_List | Dummy_Var1] - Dummy_Var1,
          [Second_List | Dummy_Var2] - Dummy_Var2,
          Result_List).
```

```
/* Output is another difference list */
?- append_dl([a,b|X]-X, [c,d|Y]-Y, R).
X = [c, d|_G193]
Y = _G193
R = [a, b, c, d|_G193]-_G193

/* By using R-[], R now can't be further extended */
?- append_dl([a,b|X]-X, [c,d|Y]-Y, R-[]).
X = [c, d]
Y = []
R = [a, b, c, d]
```

Example 2: append_dl2

Another way to write the append_dl predicate:

```
append_dl2(L-M, M, L).
```

And it is used as follows:

```
append_dl2([First_List | Dummy_Var] - Dummy_Var,
           Second_List, Result_List).
```

```
?- append_dl2([a,b|X]-X, [c,d], R).

X = [c, d]
R = [a, b, c, d]
```

append_dl or append_dl2 ?

- In append_dl, all 3 arguments are difference lists.
- Thus, in general, the result of append_dl consists of an open list that may be used for further processing.
- For example, we may append 3 lists in constant time as follows:

```
?- append_dl([a,b|X]-X, [c,d|Y]-Y, L1),
   append_dl(L1, [e,f|Z]-Z, R-[]).

X = [c, d, e, f]
Y = [e, f]
L1 = [a, b, c, d, e, f]-[e, f]
Z = []
R = [a, b, c, d, e, f]
```

- On the other hand, you can't cascade several calls of append_dl2.

Example 3: Recursive shift

```
shift([], []).
shift([H|T], L) :- append(T, [H], L).
```

```
?- shift([1,2,3,4], L1), shift(L1, L2).
L1 = [2, 3, 4, 1]
L2 = [3, 4, 1, 2]
```

```
nshift(0, L, L) :- !.
nshift(N, L1, L2) :-
    N1 is N-1, shift(L1, L), nshift(N1, L, L2).
```

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

Example 3: Non-Recursive shift_dl2

```
shift_dl2([]-[], []).
shift_dl2([H|T]-[H], T).
```

To use the shift_dl2 predicate, it may be called as follows:

```
shift_dl2([First_List|Dummy_Var]-Dummy_Var, Result_List).
```

```
?- shift_dl2([1|X]-X, L).

X = [1]
L = [1]
```

Example 3: Non-Recursive shift_dl2 ..

```
?- shift_dl2([1,2,3,4|X]-X, L).

X = [1]
L = [2, 3, 4, 1]

?- shift_dl2([1,2,3,4|X1]-X1, L1),
   append(L1, X2, L2), shift_dl2(L2-X2, L).

X1 = [1]
L1 = [2, 3, 4, 1]
X2 = [2]
L2 = [2, 3, 4, 1, 2]
L = [3, 4, 1, 2]
```

Example 4: Non-Recursive shift_dl

```
shift_dl(A-B, []-[]) :- A==B.
shift_dl([H|T]-[H|E], T-E).
```

To use the shift_dl predicate, it may be called as follows:

```
shift_dl([First_List|Dummy_Var]-Dummy_Var, Result_List).
```

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

```
?- shift_dl([a]-[a], L-[]).
L = []
```

Example 4: Non-Recursive shift_dl ..

- There is no need to use the append predicate to shift twice with shift_dl:

```
?- shift_dl([1,2,3,4|X]-X, L1), shift_dl(L1, L2-[]).
X = [1, 2]
L1 = [2, 3, 4, 1, 2]-[2]
L2 = [3, 4, 1, 2]
```

- In general, to shift *n* times:

```
nshift_dl(_, X-Y, []-[]) :- X==Y, !.
nshift_dl(0, L, L) :- !.
nshift_dl(N, L, R) :- N1 is N-1,
      shift_dl(L, L2), nshift_dl(N1, L2, R).
```

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