

Table 1: *append*

X	Y	Z
[]	[]	[]
[a]	[]	[a]
...
[a, b]	[c, d]	[a, b, c, d]
...

Table 2: *member*

X	Y
a	[a]
a	[a, b]
b	[a, b]
...	...
a	[b, [c], d, [a]]
...	...

Logic Programming

- deals with relations rather than functions.
- relations treat arguments and results uniformly.
- no sense of direction, no prejudice about who is computed from whom.

Most operations done on lists, e.g.

$[b, c]$ → list with symbols b, c

[] → empty list

$[H|T]$ → list with head H (a symbol) and tail T (a list)

Hence, $[a, b, c] = [a|[b, c]]$

Relation is a table with $n \geq 0$ columns and a possibly infinite set of rows. A tuple (a_1, a_2, \dots, a_n) is in a relation if a_i appears in column i , $1 \leq i \leq n$, of some row in the table.

Relations are also called predicates because any relation *rel*, can be thought of as a test of the form:

Is a given tuple in the relation *rel* ?

Table 1 and Table 2 depicts the relations *append* and *member*.

So we can write, $([a], [b], [a, b])$ is in relation *append* while $([a], [b], [])$ is not.

Relations are specified by “rules” also known as Horn clause, e.g.

P if Q_1 and Q_2 and ... and Q_k . where P, Q_1 , Q_2 etc. are “terms”.

Terms are constants (starts with a lower case letter) or variables (starts with a lower case letter) or has the form $rel(T_1, T_2, \dots, T_n)$ for $n \geq 0$ (*rel* starts with a lower case letter).

Fact is a special case of a rule where $k = 0$ and P holds without any condition.

“append” is specified by two rules :

- 1) It is a fact stating that triples of the form $([], Y, Y)$ are in the relation.
- 2) $append[H|X_1]$ and Y to get $[H|Z_1]$ if $append X_1$ and Y to get Z_1 .

“member” is specified by two rules :

- 1) It is a fact that X is a member of a list with head X .
- 2) X is a member of a list if X is a member of the tail.

Logic programming is driven by “queries” about relations.

Simple query : whether a particular tuple belongs to a relation, e.g.

Does $([a, b], [c, d], [a, b, c, d])$ belong to the relation `append` ?

Queries with variables : Is there a Z such that $append[a, b]$ and $[c, d]$ to get Z ? (in other words, request for Z such that tuple $([a, b], [c, d], Z)$ is in relation `append`)

Yes, when $Z = [a, b, c, d]$.

Neat thing about relations is that if X, Y , etc. are used to get Z , then any one of X, Y (, etc.) and Z can be computed from the others.

In the case of `append`,

X can be computed from Y and Z

Y can be computed from X and Z

Z can be computed from X and Y .

New relations can be defined from old ones,

prefix X of Z if for some Y , `append X and Y` to get Z .

suffix Y of Z if for some X , `append X and Y` to get Z .

Prolog control proceeds from left to right.

Simple Term : Simple terms are numbers, variables and atoms, e.g.

0, 1972 (numbers), X , Source (variables), `lisp`, `algol60` (atoms).

Compound Term : Atom followed by a parenthesized sequence of subterms. In this case, the atom is called the *functor* and the subterms are called the *arguments*.

Some operators can be prefix or infix; (X, Y) is the same as $X = Y$

“_” is a special variable, a placeholder for an unnamed term.

Existential Queries

All variables in a query are implicitly existentially quantified. I gave a few examples of these in class.

Given the following database of facts:

```
link(algol60, cpl).
link(cpl, bcpl).
link(bcpl, c).
link(c, cplusplus).
```

Now if we query the system

```
[- link(algol60, L), link(L, M)].
```

It actually implies:

$\exists L, M. \text{link}(\text{algol60}, L)$ and $\text{link}(L, M)$?

Universal Facts and Rules

All variables in facts and rules are implicitly universally quantified.

Given the following fact and rules:

```
path(L,L).
path(L,M) :- link(L,X), path(X,M).
```

It actually implies

$\forall L, M, X. \text{path}(L, M)$ if $(\text{link}(L, X)$ and $\text{path}(X, M)$)

Since X does not appear in the head, it is equivalent to:

$\forall L, M. \text{path}(L, M)$ if $(\exists X. \text{link}(L, X)$ and $\text{path}(X, M)$)

Unification

Process of pattern matching to make statements identical is called unification.

$! ? - f(X, b) = f(a, Y)$

$X = a$

$Y = b$

$g(a, a)$ is an instance (obtained by substituting subterms) of $g(X, X)$

$f(a, b)$ is an instance of $f(X, b)$

$g(h(b), h(b))$ is an instance of $g(X, X)$

$g(a, b)$ is NOT an instance of $g(X, X)$

Deduction in Prolog is based on the concept of unification. Two terms T_1 and T_2 unify if they have a common instance U . If a variable occurs in both T_1 and T_2 , then the same subterm must be substituted for all occurrences of the variable in both T_1 and T_2 .

Unification occurs implicitly when a rule is applied. Relation identity is defined by the fact, $\text{identity}(Z, Z)$.

Unification is used to compute the response of the query

$! ? - \text{identity}(f(X, b), f(a, Y))$

$X = a$

$Y = b$

Response is computed by unifying $\text{identity}(Z, Z)$ with $\text{identity}(f(X, b), f(a, Y))$ which leads to unification of Z with $f(X, b)$ and with $f(a, Y)$. In effect $f(X, b)$ is unified with $f(a, Y)$.

Arithmetic

$=$ operator stands for unification, so

$! ? - X = 2 + 3.$

$X = 2 + 3$

simply binds X to the term $2 + 3$

infix operator **is** evaluates an expression

$! ? - X \text{ is } 2 + 3.$

$X = 5$

So,

$[\text{?} - X \text{ is } 2 + 3, X = 5.$

$X = 5$

is satisfied, but

$[\text{?} - X \text{ is } 2 + 3, X = 2 + 3.$

no

i.e. $2 + 3$ does not unify with 5 .

Data Structures

Basic data structure of Prolog is a list; e.g.

$[a, b, c], []$

$[a, b, c] \equiv [a, b, c|[]]$

$[a, b, c] \equiv [a, b|[c]]$

$[a, b, c] \equiv [a|[b, c]]$

Can use unification to get data,

$[\text{?} - [H|T] = [a, b, c].$

$H = a$

$T = [b, c]$

$[\text{?} - [a|T] = [H, b, c].$

$T = [b, c]$

$H = a$

Another way of specifying lists is using “.” operator, similar to *cons* in Lisp.

|? - $.(H, T) = [a, b, c]$.

$H = a$

$T = [b, c]$

Hence, $.(a, .(b, .(c, []))) \equiv [a, b, c]$.

Definition of *append* and *member*.

```
append([], Y, Y).
```

```
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

```
member(X, [X|_]).
```

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

Order of subgoals important. (Guess and Verify Techniques)

|? - $X = [1, 2, 3], member(a, X)$.

no

But if we interchange the subgoals then we are in trouble

|? - $member(a, X), X = [1, 2, 3]$.

infinite computation

The guess goal $member(a, X)$ has an infinite no. of solutions

$X = [a|_]$; a can be the first element of X

$X = [_|a|_]$; a can be the second element of X

$X = [_|_|a|_]$; a can be the third element of X

none of which binds X to the list $[1, 2, 3]$.

Search Trees

$prefix(X, Z) : -append(X, Y, Z).$

$suffix(Y, Z) : -append(X, Y, Z).$

Say we have the following query with 2 subgoals

$! ? - suffix([a], L), prefix(L, [a, b, c]).$

Tree with no backtracking

$|\{ ? - \text{suffix}([b], L), \text{prefix}(L, [a, b, c]) \}|$.

Tree with backtracking

Cuts

Cuts are used to prune unexplored part of search tree.

Say if we have a rule,

$B : -C_1, C_2, \dots, C_{j-1}, !, C_{j+1}, \dots, C_k.$

Cut is denoted by ! and it tells the control to backtrack past $C_j - 1, \dots, C_1, B$ if C_{j+1} fails.

If we have the following rules and facts

$a(1) : -b.$

$a(2) : -e.$

$b : -c.$

$b : -d.$

$d.$

$e.$

We query the system,

$! ? - a(X).$

$X = 1;$

$X = 2;$

no

Let us change the rule $b : -c.$ to $b : -!c.$, i.e. we introduce a cut.

Now, if we query the system,

```
|? - a(X).
```

```
X = 2;
```

```
no
```

Green cuts are used to reduce computation.

The relation `member` can be represented as follows:

```
member(K, node(K, _, _)).  
member(K, node(N, S, _)) :- K < N, member(K, S).  
member(K, node(N, _, T)) :- K > N, member(K, T).
```

where, `node(K, _, _)` represents a binary search tree with K at the root and unnamed left and right subtrees and,
`member(K, U) :- U = node(N, S, T), K = N` and,
rule `member(K, node(N, S, _) :- K < N, member(K, S)` implies K is a tree node `node(N, S, _)` if $K < N$ and K is in the left subtree S .

Now, introduce “green” cuts to reduce computation.

```
member(K, node(N, S, _)) :- K < N, !, member(K, S).
```

if $K < N$ but $member(K, S)$ fails because K is not in the tree, then there is no need to try the third rule. Hence the introduction of the cut.

Negation

How to define negation using cuts ?

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

Rule 1) if goal X succeeds, then cut and fail are reached.

Rule 2) if goal X fails then this rule succeeds.

$! ? - X = 2, not(X = 1).$

subgoal $X = 2$, unified X with 2, leaving the current goal to be $not(2 = 1)$, i.e.

$2 = 1, !, fail$

Now, $2 = 1$ fails, cut is not reached, Rule 2 for not is tried; goal is $not(2 = 1)$, succeeds.