Programming Languages

Prolog

CSCI-GA.2110-001
Summer 2011

Prolog overview

- Stands for **Pro**gramming in **Log**ic.
- Invented in approximately 1972.
- Belongs to the logical and declarative paradigms.
- Based on first order predicate calculus.
- Used for artificial intelligence, theorem proving, expert systems, and natural language processing.
- Used as a standalone language or complements traditional languages.
- Radically different than most other languages.
- Each program consists of 2 components:
  - database (*program*): contains facts and rules
  - query : ask questions about relations

Stating Facts

Two ways to state facts:

```
?- [user].                                   consult user
sunny.                                       state the fact
% user://1 compiled 0.00 sec, 408 bytes
true.
```

(same as `?- consult(user).`)

Or:

```
?- assert(sunny).   state the fact
true.
```

Stating Facts 2

What facts can we describe?

1. Items ?- `assert(sunny).`
2. Relationships between items:
   ?- `assert(likes(john,mary)).`

Query the database:

```
?- likes(john,mary).
true.
?- likes(mary,john).
false.
?- likes(john,sue).
false.
```

Relations

- Relations take an arbitrary number of parameters.
- Arguments can be legal Prolog *terms* : integer, atom, variable, structure.
- Atoms: letters, digits, underscore with lowercase characters, or *anything* in quotes.
  - Legal: hello, hi123, two_words, “G_1)!#)@blah”
  - Illegal: Hello, 123hi, _hello, two-words
- Variables: Any word beginning with a capital letter.
- Structures: Functors with a list of arguments.

Note: variables bind to values, not memory locations.

```
?- likes(john,Who).
Who = mary
```

Prolog will display one *instantiation*. Type a semicolon for more.

More Relations

All satisfying `likes` relations:

```
?- likes(Who1,Who2).
Who1 = john; Who2 = mary
```

Constrain queries using variables:

```
?- likes(Who,Who).
false.
```

(People who like themselves.)

Use wild card to determine if some instantiation exists:

```
?- likes(john,_).
true.
```

(That is, john likes *someone*—we don't care who.)

Wild cards can be used in conjunction with variables:

```
?- likes(Who,_).
Who = john
```

Rules

Rules express conditional statements about our world.
Consider the assertion: "All men are mortal."

Expressible as modus ponens: `human` $\rightarrow$ `mortal`.

`mortal` is a *goal* (or *head*), and `human` is a *subgoal* (or *body*).

```
?- assert(mortal(X) :- human(X)).
true.
?- assert(human(socrates)).
true.
```

Now we query:

```
?- mortal(socrates).
true.
```

You can also ask who is mortal:

```
?- mortal(X).
X = socrates
```

Conjunction and Disjunction

Conjunction is expressed using commas:

```
?- fun(X) :- red(X), car(X).
```

Disjunction is expressed with semicolons or separate clauses:

```
?- fun(X) :- red(X); car(X).
```

. . . is the same as

```
?- fun(X) :- red(X).
?- fun(X) :- car(X).
```

Order of rules matters!

Consider:

```
?- assert(car(jeep)). assert(car(ford)). assert(red(jeep)).
?- fun(jeep).
true.
?- fun(ford).
false.
```

Multi-Variable Rules

```
daughter(X,Y) :- mother(Y, X), female(X).
```

```
grandfather(X,Y) :- male(X), parent(X,Z), parent(Z,Y).
```

Quantification:

- Variables appearing in the goal are *universally* quantified.
- Variables appearing only in the subgoal are *existentially* quantified.

The `grandfather` goal reads as:
$\forall_{X,Y} \exists_{Z} : \texttt{grandfather}(X,Y) \Leftarrow \texttt{male}(X), \texttt{parent}(X,Z), \texttt{parent}(Z,Y).$

Resolution Principle

Prolog responds to queries using the *resolution principle*:

If C_1 and C_2 are rules and the head of C_1 matches one of the terms in the body of C_2, then replace the term in C_2 with the body of C_1.

Example:

C_1: `happy(X) :- workday(Z),day_off(X,Z).`

C_2: `go_walking(X) :- happy(X).`

1. Query: `?- go_walking(emily).`
2. Instantiate the rule: `go_walking(emily) :- happy(emily).`
3. Apply resolution principle:

   `go_walking(emily) :- workday(Z),day_off(emily,Z).`

Unification

Consider again:

C_1: `happy(X) :- workday(Z),day_off(X,Z).`

C_2: `go_walking(X) :- happy(X).`

When the user queries `?- go_walking(emily)`, How does Prolog make the connection? `go_walking(emily)` `go_walking(X)`

Answer: *unification*.

Unification Algorithm

1. Constants: Any constant unifies with itself.
2. Structures: Same functor, same arity, arguments unify recursively.
3. Variables: Unify with anything.
   (a) Value: variable takes on the value.
   (b) Another Variable: unify by reference.

Some examples:

`21`	`21`	`21`
`X`	`5`	`X=5`
`love(X,me)`	`love(you,Y)`	`X=you,Y=me`
`love(X,Y)`	`love(you,Y)`	`X=you,Y=Y`
`8`	`15`	error
`love(X,Y)`	`foobar(you,Y)`	error
`c(X,c(Y,c(Z,n)))`	`c(he, c(she, c(it,n)))`	`X=he, Y=she, Z=it`
`love(X,Y)`	`love(you,f(Y))`	`X=you,Y=??`

Occurs Check

Consider:

`equal(Y, f(Y)).`

Let's try unifying `Y=f(Y)`. We have:

`equal(Y, f(Y))`	no match
`equal(f(Y), f(f(Y)))`	no match
`equal(f(f(Y)), f(f(f(Y))))`	no match
`equal(f(f(f(Y))), f(f(f(f(Y)))))`	no match

Infinite recursion!

This situation can be caught with an *occurs check*.

More on Occurs Check

An *occurs check* causes unification to fail if variable is contained within a structure.

- Prevents infinite loops or unsoundness.
- Inefficient to implement (linear in the size of the largest term).
- Most implementations of Prolog (like SWI Prolog) omit it.

Therefore, in SWI Prolog:

```
?- equal(Y, f(Y)).
Y = f(Y).
```

If you insist on the occurs check, you can force it in SWI:

```
?- unify_with_occurs_check(X,f(X)).
false.
```

Execution Order

There are two ways to answer a query:

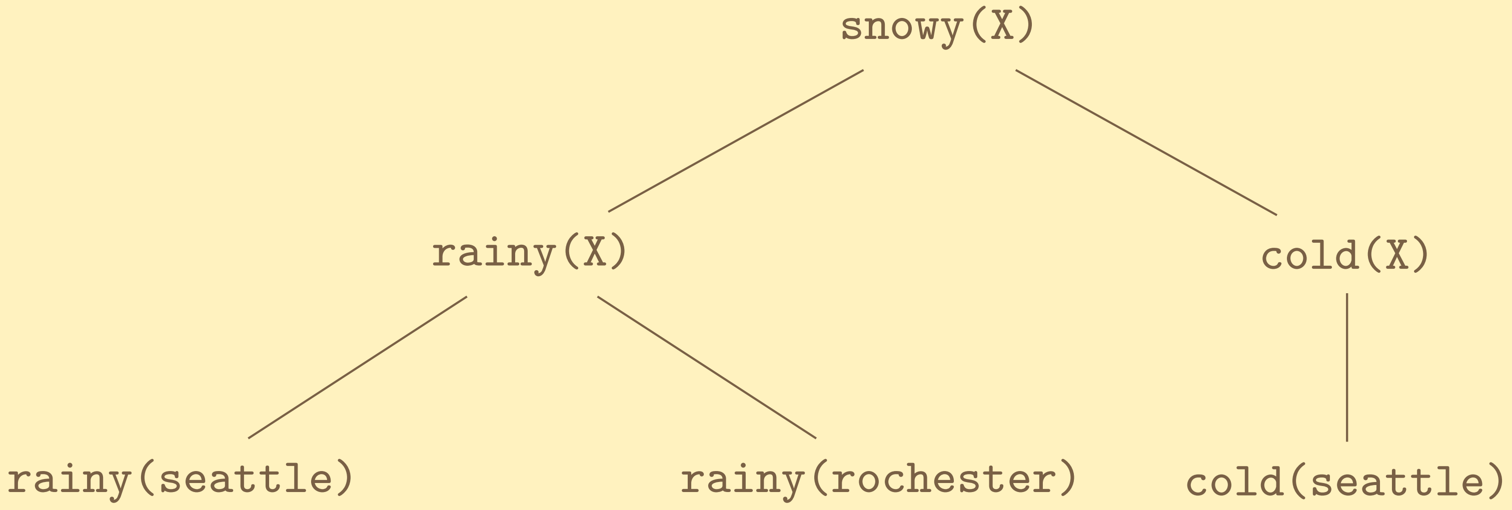
1. *Forward chaining*: start with existing clauses and work forward.
2. *Backward chaining*: start with goal and work backward. (Used by Prolog).

If the body of a rule unifies with the heads of other rules in some particular order, it can be expressed as a tree.

Execution Order

Consider:

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



Reflexive Transitive Closure

More than one “application” of a rule:

```
?- connect(Node,Node).
?- connect(N1,N2) :- edge(N1,Link), connect(Link,N2).
```

Now add some edges:

```
?- assert(edge(a,b)).   ?- assert(edge(c,d)).
?- assert(edge(a,c)).   ?- assert(edge(d,e)).
?- assert(edge(b,d)).   ?- assert(edge(f,g)).
```

```
?- connect(a,e).
true.

   connect(a,e) :- edge(a,b), connect(b,e)
   connect(b,e) :- edge(b,d), connect(d,e)
   connect(d,e) :- edge(d,e), connect(e,e)

?- connect(d,f).
false.
```

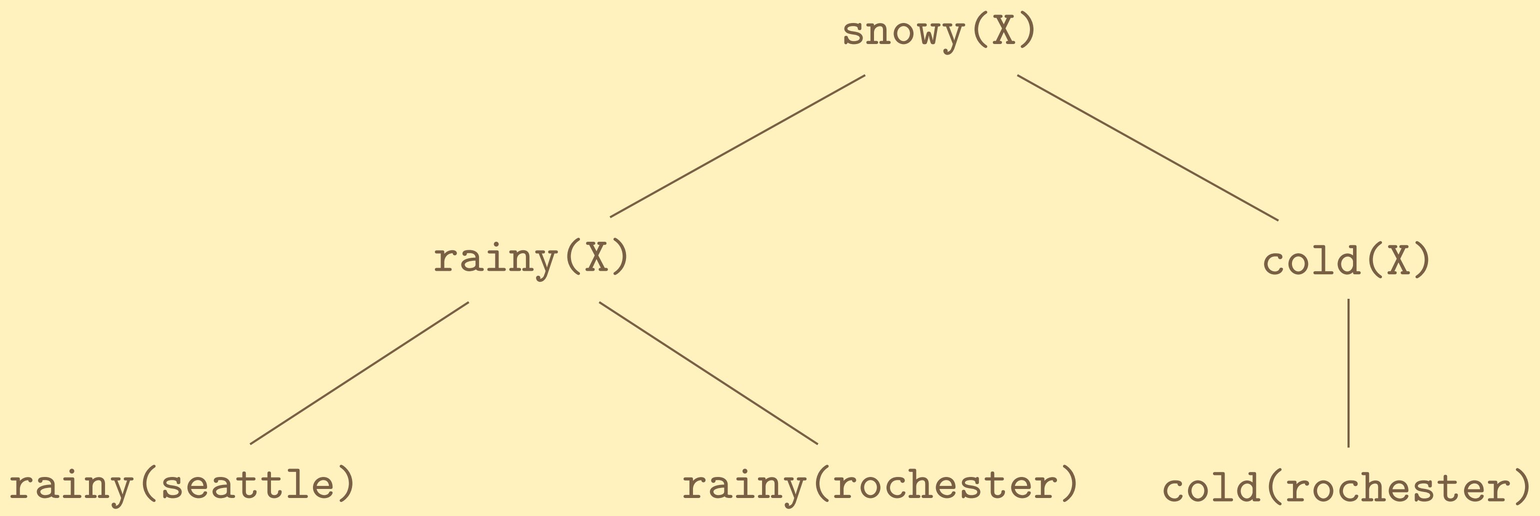
Backtracking

- Prolog maintains a list of goals to be satisfied.
- When a goal is queried, all *subgoals* of the goal are added to the list.
  - `goal(X,Y) :- subgoal1(X), subgoal2(Y).`
- Prolog will try to satisfy *all* subgoals.
- If a subgoal cannot be satisfied, Prolog will try another way.
  - `subgoal1(X) :- subsubgoal1(X).`
  - `subgoal1(X) :- subsubgoal2(X),subsubgoal3(X).`
- This is called *backtracking*.
- Carried out through a tree data structure:
  - Goal is a node.
  - Subgoals are children of the node.

Backtracking Example

Consider:

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



Backtracking in Prolog

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

Print the backtrace by invoking `trace.`, then `snowy(X).`

```
Call: (6) snowy(_G466) ? creep
Call: (7) rainy(_G466) ? creep
Exit: (7) rainy(seattle) ? creep
Call: (7) cold(seattle) ? creep
Fail: (7) cold(seattle) ? creep
Redo: (7) rainy(_G466) ? creep
Exit: (7) rainy(rochester) ? creep
Call: (7) cold(rochester) ? creep
Exit: (7) cold(rochester) ? creep
Exit: (6) snowy(rochester) ? creep
X = rochester
```

Lists

Lists are denoted by `[ a, b, c ]`.

A *cons pair* is denoted `[X|Y]` where `X` is the *head* and `Y` is the *tail*.

Rules for testing list membership:

```
?- assert(member(X, [X|Xs])).
?- assert(member(X, [Y|Ys]) :- member(X,Ys)).
```

Testing membership:

```
?- member(b,[a,b,c]).
true.
?- member(b,[a,c]).
false.
```

You can also extract list membership:

```
?- member(X,[a,b,c]).
X = a; X = b; X = c.
```

Reversing Lists

Consider a list reverse rule:

```
reverse([],[]).
reverse([X|Xs],Zs) :- reverse(Xs,Ys), append(Ys,[X],Zs).
```

Reverse-accumulate:

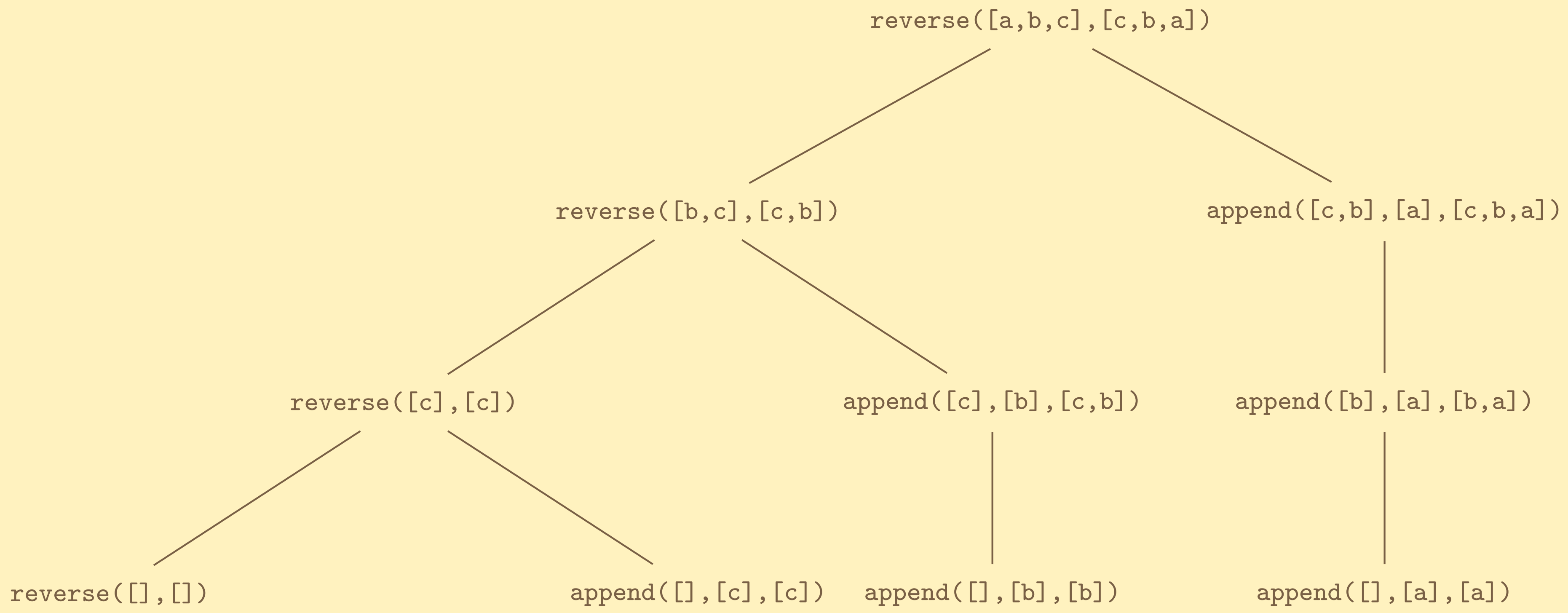
```
reverse(Xs,Ys) :- reverse(Xs,[],Ys).
reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

Invoking the reverse rule:

```
?- reverse([a,b,c], X).
X = [c, b, a].
?- reverse([a,b,c], [a,c,b]).
false.
```

Tree for Reverse

The `reverse` rule at work:



Cut Operator

You can tell Prolog to stop backtracking using the *cut* operator, `!`.

- Used to "commit" all unifications up to the point of the `!`
- Will never backtrack through any subgoal to the left of `!`
- Done to optimize performance.
- Generally requires intuition about the program.

Consider:

```
prime_candidate(X) :- member(X, candidates), prime(X).
```

- Variable X may appear several times in `candidates`.
- Once X is found to be in candidates, no need to try other possibilities.
- Solution: use the cut operator.
  - `member(X, [X|_]) :- !.`
  - `member(X, [_|T]) :- member(X, T).`

More on Cut

The cut operator can also serve as an if-then-else construct:

```
statement :- condition, !, then_part.
statement :- else_part.
```

- Cut prevents the condition from being retested.
- If `condition` is true, subgoal `then_part` will be attempted.
- If `then_part` fails, the system will not backtrack into the condition.
- If first goal fails, the second goal will be tried.

Negation

One way to negate a subgoal is using predicate `not`:

```
unmarried_student(X) :- not(married(X)), student(X).
```

Definition of `not` (also known as \+):

```
not(Goal) :- call(Goal), !, fail.
not(Goal).
```

- Predicate `fail` unconditionally fails.
- Predicate `call` treats the input term as a goal and attempts to satisfy it.

Example:

```
single(Person) :- \+ married(Person,_), \+ married(_,Person).
```

Note: $\backslash +$ indicates *inability to prove*—**not** falsehood.