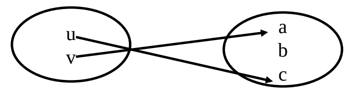
Logic Programming & Prolog

IP, OOP & FP

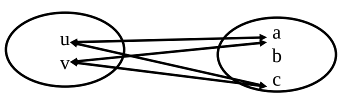
- All based on the notion that a program implements a **mapping** (**function**) M from input to output.
- Given a, determine the value of M(a)?



Mapping: many to one relationship

Logic Programming

- Based on the notion that a program implements a **relation** R.

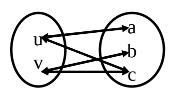


Relation : many to many relationship

Logic Programming

- Given u, find all y s.t. R(u, y) is true?
  - y=a, y=c

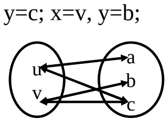
R:



Logic Programming

- Given u and a whether R(u, a) is true ?
  - Yes
- Given c, find all x s.t. R(x,c) is true?
  - x=u, x=v
- Find all x and y s.t. R(x, y) is true.
  - x=u, y=a; x=u, y=c; x=v, y=b; x=v, y=c

R:



Relations

- Relations:
  - Like a table.
  - Unary relation R (a)
  - Binary relation R(a, b)
  - Ternary relation R(a, b, c)
  - ……
- Relations treat arguments and results uniformly.
  - No distinction between input and output.
- Relations are specified by rules and facts.
  - Based on formal symbolic logic (**Predicate calculus**)

Describing Relations - Rules

- A rule
  - R0 **if** R1 **and** R2 **and** … **and** Rn**.**
  - `R0 :- R1, R2, …, Rn.`
  - Called **Horn clauses**
- If R1, …, Rn are all true, then we can infer that R0 is also true.

Horn Clauses

- Why use **Horn clauses** for rules?
  - Most (not all) logical statements can be described by Horn clauses.
  - The program can be implementable.
  - The program can be tolerably efficient.

Describing Relations - Facts

- A **fact** is a special rule.
  - `R0.`
  - R0 is unconditionally true.

Example: Rules and Facts

```
link(fortran, algol60).
link(algol60, cpl).
link(cpl, bcpl).
link(bcpl, c).
link(c, C++).
link(algol60, simula).
link(simula, c++).
link(simula, smalltalk).

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


Computing with Relations

- Use **queries (goals)** about relations – **a database**.
  - A goal with multiple subgoals
- The language system explore **all possible** solutions to the query (goal).
  - Uses **backtracking**

Facts and Queries

```
link(fortran, algol60).
link(algol60, cpl).
link(cpl, bcpl).
Link(bcpl, c).
link(c, c++).
link(algol60, simula).
link(simula, c++).
link(simula, smalltalk).
```

```
?- link (cpl, bcpl).
yes
?- link (cpl, bcpl), link (bcpl, c).
yes
?- link (simula, L).
L= c++;
L= Smalltalk;
?- link (L, c++).
L= c;
L= simula;
```

Facts and Queries

```
link(fortran, algol60).
link(algol60, cpl).
link(cpl, bcpl).
Link(bcpl, c).
link(c, c++).
link(algol60, simula).
link(simula, c++).
link(simula, smalltalk).
```

```
?- link (L1, L2).
L1 = fortan;
L2 = algol60;
...
?- link (algol60, L), link (L, M).
L = cpl, M=bcpl;
L= simula, M=c++;
L= simula, M=smalltalk;
?- link (lisp, simula).
no
```


Closed World Assumption

```
?- link (simula, java).
no
```

- **no** means
  - I can't prove it.
  - It can not be inferred to be true.
  - Unknown.
  - If we extend the fact later, it could be **yes**!

Example: Facts, Rules and Queries

```
link(fortran, algol60).
link(algol60, cpl).
link(cpl, bcpl).
Link(bcpl, c).
link(c, c++).
link(algol60, simula).
link(simula, c++).
link(simula, smalltalk).

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

```
?- path (cpl, cpl).
yes
?- path (cpl, c).
yes
?- path (cpl, L)
L= cpl;
L=bcpl;
L= c;
L= c++;
```


Example: Rules and Queries with Lists

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

?- append([a,b], [c,d], [a,b,c,d]).
yes
?- append([a,b], [c,d], Z).
Z= [a,b,c,d]
?- append( [a,b], Y, [a,b,c,d]).
Y = [c,d]
?- append( X, [c,d], [a,b,c,d]).
X= [a,b]
?- append( X, [d,c], [a,b,c,d]).
no
```


The Structure of Logic Programs

- A logic program consists of
  - A collection of relations defined by rules and facts (Horn clauses)
  - A query
- Use facts and rules to represent information
  - Provided by the programmer
- Use deduction to answer queries
  - Provided by the programming language

Answering A Query (Satisfying a Goal)

- How a language computes a response to a query?
- To prove that a goal is true:
  - Must find a chain of inference rules and facts in the database that connect the goal to one or more facts in the database.

Control in LP

- How a language computes a response to a query?
  - Computation (answering to the queries) is based on
    - **Resolution**
    - **Unification**.
  - Can be expressed through a sequence of resolutions and unifications.

Resolution

- Given two rules
  - C1 **if** a1 **and** a2 **and** … **and** am**.**
  - C2 **if** C1 **and** b1 **and** b2 **and** … **and** bn**.**
- A new rule can be derived:
  - C2 **if** a1 **and** a2 **and** … **and** am **and** b1 **and** b2 **and** … **and** bn**.**

Unification

- The derivation of a new rule from a given rule through the binding of variables to values.
  - The process of making two terms "the same".
  - A pattern matching process
  - **Instantiation**

f(a, X) unifies with f(Y, b) by instantiating X to b and Y to a.

Substitutions

- A function that maps variables to terms
  - $\sigma = \{$**`X`**$\rightarrow$**`a`**, **`Y`**$\rightarrow$**`f(a,b)`**$\}$
- The result of applying a substitution to a term is an *instance* of the term
  - σ(**`g(X,Y)`**) = **`g(a,f(a,b))`**
  - **`g(a,f(a,b))`** is an *instance* of **`g(X,Y)`**

Unification

- Two terms t1 and t2 unify if there is some substitution σ that makes them identical:
  - $\sigma(t1) = \sigma(t2)$

Unification Examples

- a and b do not unify
- f(X,b) and f(a,Y) unify: a unifier is {X→a, Y→b}
- f(X,b) and g(X,b) do not unify
- a(X,X,b) and a(b,X,X) unify: a unifier is {X→b}
- a(X,X,b) and a(c,X,X) do not unify
- a(X,f) and a(X,f) do unify: a unifier is {}

Multiple Unifiers

- `parent(X,Y)` and `parent(fred,Y)`:
  - σ_1 = {`X`→`fred`}
  - σ_2 = {`X`→`fred`, `Y`→`mary`}
- Prolog chooses unifiers like σ_1 that do just enough substitution to unify, and no more.
  - **Most General Unifier** (**MGU**)

Most General Unifier

- Term x_1 is *more general than* x_2 if x_2 is an instance of x_1 but x_1 is not an instance of x_2
  - **`parent(fred,Y)`** is more general than **`parent(fred,mary)`**
- A unifier σ_1 of two terms t_1 and t_2 is an MGU if there is no other unifier σ_2 such that $\sigma_2(t_1)$ is more general than $\sigma_1(t_1)$.

Two Search Strategies

- **Forward Chaining**
  - Start with existing rules attempting to derive the goal.
- **Backward Chaining**
  - Start with the goal attempting to unresolve it into a set of existing rules.
  - **Prolog!**

Example: Search Strategies

```
father(bob).
man(X) :- father(X).
?- man(bob).
```

- Forward Chaining
  - Start with existing rules attempting to derive the goal.
- Backward Chaining
  - Start with the goal attempting to unresolve it into a set of existing rules.

Backtracking

- The process of returning back to a previously proven subgoal.
  - In order to pursue a different path through the search tree.
  - The unification (instantiation) is undone.
- **Backtracking is very time & space consuming**!

Example: Backtracking

```
female(shelley).
female(mary).
male(mike).
male(bill).
male(jake).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father( X, Y).
?- male(X), parent(X, shelley).
```

Control in LP

- In principle,
  - The order of goals within a query and the order of rules and facts **should not** matter.
- In practice,
  - The response to a query is **affected** by
    - The **goal order** within the query
    - The **rule order** within the facts and rules..

CS216 31

Control in Prolog

- Prolog applies resolution in a linear fashion:
  - **Goal order**
    - Choose the leftmost subgoal. (**from left to right**)
  - **Rule order**
    - Choose the first rule. (**from first(top) to last (bottom)**)

- Every Prolog program is deterministic.

CS216 32

Control in Prolog

- Prolog uses a **depth-first search** on a tree of possible choices!
- Can be implemented in a stack-based or recursive fashion.
- Solutions may not be found if the search tree has branches that have infinite depth.

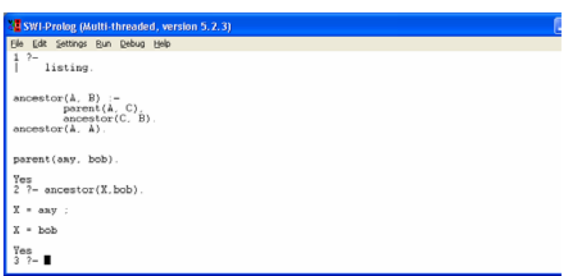
CS216 33

Example: Control in Prolog

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
ancestor(X, X).
parent(amy, bob).

?- ancestor(X, bob).
X = amy;
X = bob
```

CS216 34

Example: Control in Prolog



CS216 35

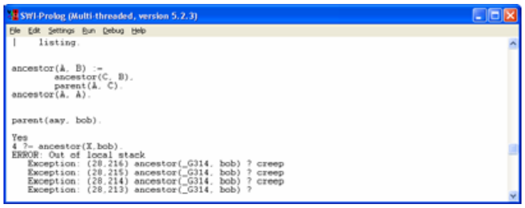
Example: Control in Prolog

```
ancestor(X, Y) :- ancestor(Z, Y), parent(X, Z).
ancestor(X, X).
parent(amy, bob).

?- ancestor(X, bob).
[infinite loop]
```

Left-recursive

CS216 36

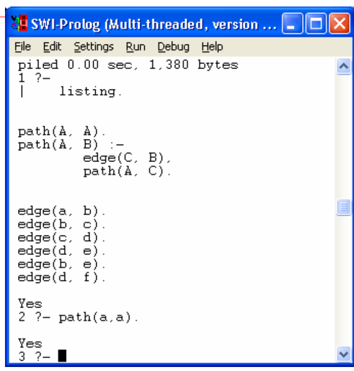
Example: Control in Prolog



Example: Control in Prolog

```
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).

?- path(a, a).
yes
```

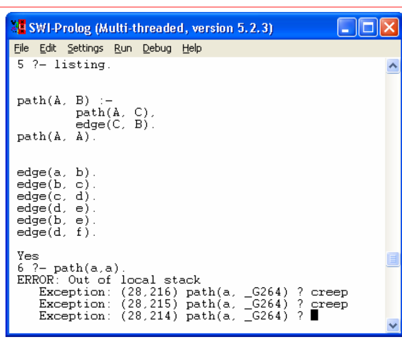

Example: Control in Prolog



Example: Control in Prolog

```
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).

?- path(a, a).
[infinite loop]
```


Example: Control in Prolog



Cuts – Explicit Control of Backtracking

- The cut operator **!**
  - Cuts out an unexplored part of the search tree.
  - **Imperative control**
- A :- C1, C2, …, Ci, **!**, Ci+1, …, Cn
  - Backtrack past Ci, …, C2, C1, A without considering any remaining rules for them.

Cuts

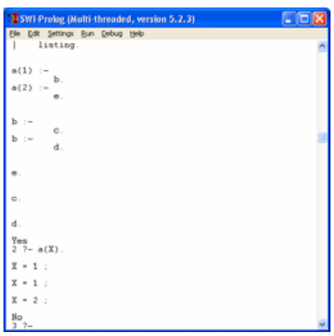
```
A0 :- A1, !, A2.
A0 :- …
A0 :- …
?- A0.
```

- If **A1** fails, then backtrack & try to find another rule for **A0**.
- If **A1** succeeds, then accept the first answer yielded by **A1**, pass the cut, go on to test **A2**.
  - If **A2** subsequently fails, then we immediately conclude that **A** fails!
    - ✓Make no attempt to find any further answer from **A1**.
    - ✓Make no attempt to find any further rule for **A0**.

Example: No Cut

```
a(1) :- b.
a(2) :- e.
b :- c.
b :- d.
c.
d.
e.

?- a(X).
X = 1;
X = 1;
X = 2;
no
```

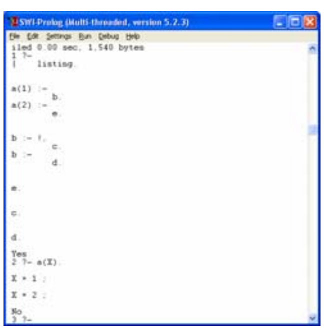

Example: No Cut



Example: Cut

```
a(1) :- b.
a(2) :- e.
b :- !, c.
b :- d.
c.
d.
e.

?- a(X).
X = 1;
X = 2;
no
```


Example: Cut



Cut vs No Cut

```
a(1) :- b.
a(2) :- e.
b :- c.
b :- d.
c.
d.
e.

?- a(X).
X = 1;
X = 1;
X = 2;
no
```

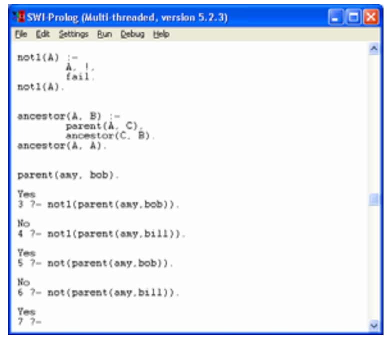
```
a(1) :- b.
a(2) :- e.
b :- !, c.
b :- d.
c.
d.
e.

?- a(X).
X = 1;
X = 2;
no
```

Negation

```
not1(X) :- X, !, fail.
not1(X).
```

CS216 49

Negation



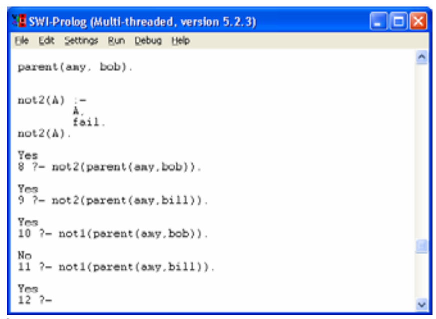
CS216 50

Negation

```
not2(X) :- X, fail.
not2(X).
```

CS216 51

Negation



CS216 52

Negation and Cut

```
not1(X) :- X, !, fail.
not1(X).
```

```
not2(X) :- X, fail.
not2(X).
```

CS216 53

Input and Output

```
?- write('Hello world').
Hello world

Yes
?- read(X).
|    hello.

X = hello

Yes
```

CS216 54

Debugging With `write`

```
append ( [], Y, Y).
append ( [H|X], Y, [H|Z] ) :- append (X, Y, Z).
p :-
  append(X,Y,[1,2]),
  write(X), write(' '), write(Y), write('\n'),
  X=Y.
```

```
?- p.
[] [1, 2]
[1] [2]
[1, 2] []

No
```

CS216 55

The `assert` Predicate

```
?- parent(joe,mary).

No
?- assert(parent(joe,mary)).

Yes
?- parent(joe,mary).

Yes
```

- Adds a fact to the database (at the end).

CS216 56

The `retract` Predicate

```
?- parent(joe,mary).

Yes
?- retract(parent(joe,mary)).

Yes
?- parent(joe,mary).

No
```

- Removes the first clause in the database that unifies with the parameter.

CS216 57