P.1

PROLOG:

Programming in Logic

Prolog is a language for automatic theorem proving.

Suppose we have

P(a)

P(x) $\longrightarrow$ Q(x) for all x

and want to prove Q(a) .

In Prolog

```
|        p( a ) .

|        q( x ) : --    p( x )


| ? --    q( a ) .

yes
```

P.2

How do theorem provers work

- They use predicate logic.
- They input axioms, rules, and a theorem to be proved.
- They employ techniques of pattern matching.
- They perform a search to determine a sequence of

  axioms
  rules
  intermediate results

  that prove the theorem.

P.3

Predicate Logic

constants **a, b, c, 123, -6.5, John, Mary**

variables **x, y, z**

functions **f, g, h, broiled**

predicates **P, Q, R, S, EATS, LOVES, SLEEPS**

P(a, 123)

LOVES(John, Mary)

EATS(Mary, broiled(x))

Sleeps(John)

logic operators **V,** Λ **,** $\neg$ **,** $\rightarrow$

quantifiers $\forall$ **(for every)**

$\exists$ **(there exists)**

$\forall$**x** $\exists$**y** **EATS(y)** Λ **SLEEPS(Y)** Λ **LOVES(x, y)**

P.4

Examples

1) man(Marcus)

2) Pompeian(Marcus)

3) born(Marcus, 40)

4) $\forall x$ man(x) $\longrightarrow$ mortal(x)

5) $\forall x$ Pompeian(x) $\longrightarrow$ died(x, 79)

6) erupted(volcano, 79)

7) $\forall x \ \forall t_1 \ \forall t_2$ mortal(x) Λ born(x, t_1) Λ gt(t2 - t1, 150) $\longrightarrow$ dead(x, t2)

8) now = 1998

9) $\forall x \ \forall t$ [alive(x, t) $\longrightarrow$ $\neg$dead(x, t)] Λ [$\neg$dead(x, t) $\longrightarrow$ alive(x, t)]

10) $\forall x \ \forall t_1 \ \forall t_2$ died(x, t_1) Λ gt(t_2, t_1) $\longrightarrow$ dead(x, t_2)

P.5

Resolution theorem provers work with clauses that are derived from arbitrary predicate calculus expressions.

PROLOG works with a restricted kind of clauses called

Horn Clauses

which have three possible forms:

1) $P \longrightarrow Q$

2) $P_1 \wedge P_2 \wedge \ldots\ldots \wedge P_k \longrightarrow Q$

3) P

If we group 1) and 2) together there are two ways to represent knowledge in PROLOG

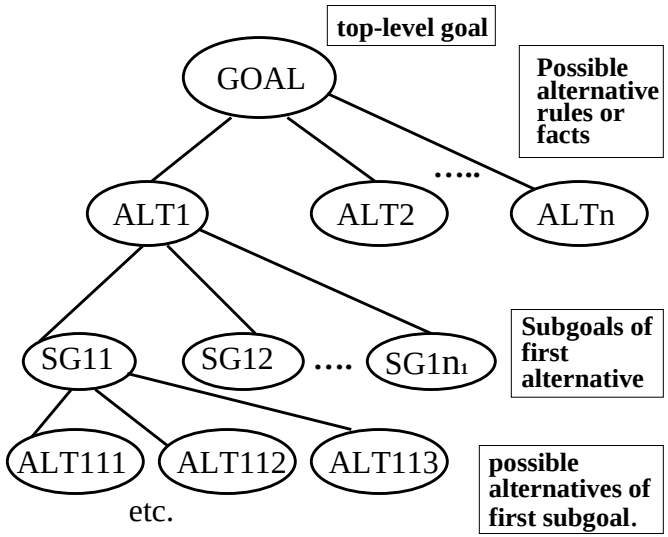
- **rules**
- **facts**

P.6

Resolution theorem provers produce an explosion of unnecessary results while trying to prove a theorem.

Prolog, with its restricted clause forms, is GOAL-ORIENTED.

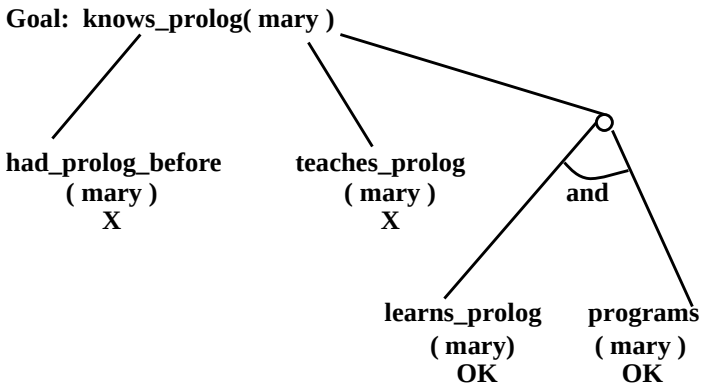
Its control strategy is a straightforward BACKTRACKING SEARCH.



P.7

A Simple Example

```
knows_prolog( X ) : -   had_prolog_before( X ).
knows_prolog( X ) : -   teaching_prolog( X ).
knows_prolog( X ) : -   learns_prolog( X ), programs( X ).
learns_prolog( mary ).
programs( mary ).
```



P.8

What is Prolog?

- Prolog is a relational language.
- The Prolog programmer designs objects and relations over objects.
- The statements of the language are declarative, not imperative.
- It is more than just a language; it is a theorem prover.

What do Prolog programs do?

They declare facts about objects and their relationships.

They define rules about objects and their relationships.

They ask questions about objects and their relationships.

P.9

FACTS

Facts are relations on objects that are known to be true.

leftof(redcircle, bluestar).

ta(joanna).
ta(patricia).

triangle(p1, p2, p3).

madeof(moon, greencheese).

The Prolog system works with symbols, like Lisp.

The symbols have no particular meaning to Prolog, only to the programmer.

The facts in a Prolog program form a database.

P.10

A fact has the syntax

< relation name >(< argument list >).*

A relation may have a variable number of arguments.

address(bill, seattle, washington).
address(mary, seattle, washington, 98195).

Facts don't have to make sense.

address(tom, blue, triangle, dog)

But they ought to mean something to the programmer.

* Some Prologs use Lisp format

(< relation name > < arguments >)

QUERIES

Suppose the database of facts is:

```
|    teaches( shapiro,  cse341 ).
|    teaches( notkin,  cse341 ).
|    teaches( young,  cse322 ).
|    teaches( hanks,  cse 473 ).
```

Then we can query the system and Prolog will search for a match to our queries, starting at the top.

```
| ? -    teaches( notkin, cse341 ) .
         yes

| ? -    teaches( young, cse341 ) .
         no

| ? -    teaches( notkin, bluecheese ) .
         no

| ? -    teaches( watermelon, strawberries ) .
         no
```


Facts are about relations on objects which are constants.

Queries can also contain variables.

```
| ? -    teaches( shapiro ,  X ) .
         X = cse341

| ? -    teaches( X ,  cse341 ) .
         X = shapiro ;   X = notkin ;  no
```

Queries with variables are called existential queries in logic.
The query

teaches(shapiro , X) .

corresponds to asking Prolog to prove the theorem

$\exists$ X: teaches(shapiro , X)

Prolog proves the theorem by searching the database to find a matching fact.

P.13

Conjunctive Queries

Queries may have several terms.

The terms may share variables.

The query is solved in left-to-right order.

The query succeeds only if ALL the terms succeed.

```
owns( glen , stereo ) .
owns( glen , car ) .
owns( glen , tv ).

type( stereo , machine ) .
type( car , vehicle ) .
type( tv , machine ) .
type( computer , machine ) .
```

```
| ? -   owns( glen ,  X ) ,    type( X ,  machine ) .
      X =  stereo ;    X =  tv ;    no
```

P.14

RULES

Rules define new relationships in terms of existing ones.

A rule has the syntax

< head > : - < body >.

The head contains 1 predicate.

The body is a conjunction of predicates, separated by commas.

bird(X) : - animal(X), haswings(X) .

means $\forall X(\text{animal}(X) \wedge \text{haswings}(X) \rightarrow \text{bird}(X))$

can be read

" X is a bird if X is an animal
and X has wings.

```
Sister( X, Y )   : -   female( X ) ,
                       sameparents( X , Y ) ,
                       notequal( X , Y ).
```

P.15

USING RULES TO ANSWER QUERIES

```
male( albert ) .
male( edward ) .

female( alice ) .
female( victoria ) .

parents( edward ,  victoria ,  albert ) .
parents( alice ,  victoria ,  albert ) .

sisterof( X , Y )   : -   female( X ) ,
                          parents( X ,  M ,  F ) ,
                          parents( Y ,  M ,  F ) .

| ? -     sisterof( alice ,  edward ) .
          yes

| ? -     sisterof( alice ,  X ) .
          X = edward

| ? -     sisterof( alice ,  alice ) .
          yes
```

P.16

RECURSIVE RULES

```
parent( john , mary ) .
parent( mary , bill ) .
parent( bill , sue  ) .
ancestor( X , Y )   : -  parent( X , Y ) .
ancestor( X , Y )   : -  parent( X , Z ) ,
                         ancestor( Z , Y ) .

| ? -      ancestor( john , sue ) .
           Parent( john , sue )  X
           parent( john,  Z ) , ancestor( Z , sue )
           parent( john , mary ) ,  ancestor( mary , sue )
                                    parent( mary , sue )  X
                parent( mary , Z ) ,  ancestor( Z , sue )
                parent( mary , bill ) ,  ancestor( bill , sue )
                                    parent( bill , sue )

          yes
```