# The Semantics of Prolog Programs

- Prolog clauses can have a “declarative” or a “procedural” interpretation
  - facts: `<term>`
    - declarative interpretation: The statement expressed by `<term>` is true
    - procedural interpretation: A goal matching `<term>` is achievable
  - rules: `<term-0> :- <term-1>, ..., <term-n>.`
    - declarative interpretation: If the statements expressed by `<term-1>, ..., <term-n>` are true, then the statement expressed by `<term-0>` is true
    - procedural interpretation: To achieve a goal matching `<term-0>`, first achieve the subgoals `<term-1>, ..., <term-n>` in order
  - questions: `?- <term-1>, ..., <term-n>.`
    - declarative interpretation: Do the statements expressed by `<term-1>, ..., <term-n>` logically follow from the current facts and rules? That is, given the current facts and rules, are the statements expressed by `<term-1>, ..., <term-n>` true? If so, for what values of the variables are they true?
    - procedural interpretation: Given the current facts and rules, are the goals `<term-1>, ..., <term-n>` achievable, in order? If so, for what values of the variables are they achievable?

# Declarative Semantics of Prolog

- In the declarative semantics, rules and facts are interpreted as statements in logic
- In the declarative semantics, as in logic, the order of the rules and facts in the Prolog program are irrelevant
- In the declarative semantics, a term G is true given the program if and only if
  - there is a clause C in the program such that there is an instance I of C such that
    - the head of I is identical to G, and
    - all the terms in the body of I are true given the program.

An *instance* of a clause is the clause with each of its variables substituted with some term.

# Procedural Semantics of Prolog

- The question: 'Does a given predicate calculus formula logically follow from a given set of predicate calculus formulas?" is undecidable
- Therefore, any logic programming language must fail to implement perfectly the declarative semantics of the predicate calculus
- The resulting compromises can be understood in part by studying the procedural semantics of the language
- In the procedural semantics, the order of rules and facts in a Prolog program are relevant

# Prolog in LISP (almost)

```
;;; SATISFY
;;; goal-list is a list of terms
;;; program is a list of clauses
;;; return T if goal-list can be satisfied
;;; given the program, else NIL
;;;
(defun satisfy  (goal-list
                 program
                 &optional (clauseptr 0))
 (let ((goal (first goal-list))
       (clause (nth clauseptr program )))
    (cond
     ((null goal)
     ;; if goal-list is empty, success
       T)
     ((null clause)
     ;; if no clauses left, failure
       NIL)
     ((and (match goal (head clause))
           (satisfy
            (new-goal-list goal-list clause)
            program))
     ;; if goal matches head of clause...
     ;; and can satisfy a new goal list
     ;; based on the body of the clause...
     ;; then success
       T)
     (t
     ;; else try to match the next clause
     ;; in the program with the old goal list
       (satisfy goal-list program (1+ clauseptr))))))
```

# Prolog in LISP (almost)

```
;;; NEW-GOAL-LIST
;;; goal-list is a list of terms
;;; clause is a clause
;;; return a new goal list in which
;;; the first goal is replaced by the
;;; body of the clause, and variables have
;;; been instantiated as necessary to
;;; match (first goal-list) and (head clause)
(defun new-goal-list (goal-list clause)
 (let ((goal (first goal-list))
       (newgoals (body clause))
       (head (head clause)))
   (if (empty newgoals)
       (instantiate
          (rest goal-list)
          goal head)
    (instantiate
        (append newgoals (rest goal-list))
        goal head))))
```

```
;;; INSTANTIATE
;;; termlist is a list of terms
;;; term1, term2 are terms
;;; return a new term list
;;; obtained from termlist by
;;; figuring out what variable instantiations
;;; are required to make term1, term2 match
;;; and substitute these for variables in termlist
(defun instantiate (termlist term1 term2)
 )
```

# Using Prolog to solve state-space search problems

- In a state-space search problem...
  - The world starts in some initial state
  - There are actions that move the world from one state to another
  - There is a desired goal state
  - The problem is to come up with a sequence of actions that get the world from the initial state to the goal state
- Many problems can be cast in the form of state-space search

# Using Prolog to solve state-space search problems

- The "monkey and the bananas" problem
- A classic problem in AI state-space search
- The state of the world is characterized by 4 components:
  - the horizontal position of the monkey
  - the vertical position of the monkey
  - the horizontal position of a box
  - whether or not the monkey has the bananas
- In the initial state, the monkey does not have the bananas
- In the desired goal state, the monkey has the bananas
- There are four kinds of actions:
  - walking (changing horizontal position of the monkey)
  - climbing on the box (changing vertical position of the monkey)
  - pushing the box (changing the horizontal position of the box)
  - grasping (getting the bananas)
- Only some actions apply in some states: for example, grasping of the bananas can only be performed when the monkey is on the box in the middle of the room
- Question: can the goal state be achieved from some given initial state?

# The monkey and the bananas program

```
% Move in State1 results in State2
% move(State1, Move, State2)
%
% A state is represented by a term
% state(MonkeyHoriz,MonkeyVert,BoxHoriz,HasBanana)

move(state(middle,onbox,middle,hasnot),
       grasp,
       state(middle,onbox,middle,has)).

move(state(P,onfloor,P,H),
       climb,
       state(P,onbox,P,H)).

move(state(P1,onfloor,P1,H),
       push(P1,P2),
       state(P2,onfloor,P2,H)).

move(state(P1,onfloor,B,H),
       walk(P1,P2),
       state(P2,onfloor,B,H)).


canget(state(_,_,_,has)).

canget(State1) :-
    move(State1,Move,State2),
    canget(State2).
```

# A monkey and the bananas solution

```
?- canget(state(atdoor,onfloor,atwindow,hasnot)).
```

fig. 2.15

Bratko

# Another monkey and the bananas program

```
% Move in State1 results in State2
% move(State1, Move, State1)
%
% A state is represented by a term
% state(MonkeyHoriz,MonkeyVert,BoxHoriz,HasBanana)

move(state(P1,onfloor,B,H),
       walk(P1,P2),
       state(P2,onfloor,B,H).

move(state(middle,onbox,middle,hasnot),
       grasp,
       state(middle,onbox,middle,has)).

move(state(P,onfloor,P,H),
       climb,
       state(P,onbox,P,H)).

move(state(P1,onfloor,P1,H),
       push(P1,P2),
       state(P2,onfloor,P2,H)).


canget(state(_,_,_,has)).

canget(State1) :-
    move(State1,Move,State2),
    canget(State2).




?- canget(state(atdoor,onfloor,atwindow,hasnot)).
```

# Procedural semantics and the ordering of clauses

```
predecessor(X,Z) :- parent(X,Z).

predecessor(X,Z) :- parent(X,Y), predecessor(Y,Z).


?- predecessor(tom,pat).
```

fig. 1.11

Bratko

# Procedural semantics and the ordering of clauses

```
pred2(X,Z) :- parent(X,Y), pred2(Y,Z).

pred2(X,Z) :- parent(X,Z).

?- pred2(tom,pat).
```

fig 2.17a

Bratko

# Procedural semantics and the ordering of clauses

```
pred3(X,Z) :- parent(X,Z).

pred3(X,Z) :- pred3(X,Y), parent(Y,Z).

?- pred3(tom,pat).
```

fig. 2.17b

Bratko

# Procedural semantics and the ordering of clauses

```
pred4(X,Z) :- pred4(X,Y), parent(Y,Z).

pred4(X,Z) :- parent(X,Z).

?- pred4(tom,pat).
```

fig. 2.17c

Bratko