flow of control, negation, cut, 2nd order programming, tail recursion

Yves Lespérance
Adapted from Peter Roosen-Runge

simplicity hides complexity

- simple and/or composition of goals hides complex control patterns
- not easily represented by traditional flowcharts
- may not be a bad thing
- want important aspects of logic and algorithm to be clearly represented and irrelevant details to be left out

procedural and declarative semantics

- Prolog programs have both a declarative/logical semantics and a procedural semantics
- declarative semantics: query holds if it is a logical consequence of the program
- procedural semantics: query succeeds if a matching fact or rule succeeds, etc.
  - defines order in which goals are attempted, what happens when they fail, etc.

and & or

- Prolog’s and (,) & or (; and alternative facts and rules that match a goal) are not purely logical operations
- often important to consider the order in which goals are attempted
  - left to right for “,” and “;”
  - top to bottom for alternative facts/rules
and is not always commutative, e.g.

- sublistV1(S, L):- append(_, L1, L),
  append(S, _, L1).
  i.e. S is a sublist of L if L1 is any suffix of L and S is a prefix of L1

- sublistV2(S, L):- append(S, _, L1),
  append(_, L1 ,L).
  i.e. S is a sublist of L if S is a prefix of some list L1 and L1 is any suffix of L

uses of or (;)

- or “;” can be used to regroup several rules with the same head
- e.g.
  parent(X,Y):- mother(X,Y); father(X,Y).
- can improve efficiency by avoiding redoing unification
- “;” has lower precedence than “,”

Prolog negation

- Prolog uses “\+”, “not provable” or negation as failure
- different from logical negation
- \?- \+ goal. succeeds if \?- goal. fails
- interpreting \+ as negation amounts to making the closed-world assumption
example

- Given program:
  - `mortal(X):- human(X).`  
- `?- \+ human(jason).`  
  - Yes  
- In logic, the axioms corresponding to the program don’t entail  
  - `¬Human(Jason)`.

semantics of free variables in \+ is “funny”

- normally, variables in a query are existentially quantified from outside  
  - e.g. `?- p(X), q(X).` represents “there exists x such that P(x) & Q(x)”  
- but `?- \+ (p(X), q(X)).` represents “it is not the case that there exists x such that P(x) & Q(x)”

To avoid this problem

- \+ works correctly if its argument is instantiated  
- so for example in  
  - `intersect([X|L], Y, I):-`  
  - `\+ member(X,Y), intersect(L,Y,I).`  
  - X and Y should be instantiated

example

- Given program:
  - `animal(cat). vegetable(turnip).`  
- `?- \+ animal(X), vegetable(X).`  
  - No  
  - Why?  
- `?- vegetable(X),\+ animal(X).`  
  - X = turnip  
  - Why?
guarding the “else”

- can’t rely on implicit negation in predicates that can be redone
- in predicates with alternative rules, each rule should be logically valid (if backtracking can occur)
- safest thing is repeating the condition with negation

![Example](image)

e. g. intersect

- intersect([], _, []).
  intersect([X|L], Y, [X|I]):- member(X,Y), intersect(L, Y, I).
  intersect([X|L], Y, I):- \+ member(X,Y), intersect(L, Y, I).
  is OK.

- intersect([], _, []). intersect([X|L], Y, [X|I]):- member(X,Y), intersect(L, Y, I).
  intersect([X|L], Y, I):- \+ member(X,Y), intersect(L, Y, I).
  is buggy.

?- intersect([a], [b, a], []). succeeds. why?

inhibiting backtracking

- the **cut** operator “!“ is used to control backtracking
- If the goal \( G \) unifies with \( H \) in program
  \[ H ::= \ldots, \]
  \[ H ::= G_1, \ldots, G_i, I, G_j, \ldots, G_k, \]
  \[ H ::= \ldots, \]
  and gets past the \( ! \), and \( G_1, \ldots, G_k \) fails,
  then the parent goal \( G \) immediately fails. \( G_1, \ldots, G_i \) won’t be retried and the subsequent matching rules won’t be attempted.
Using ! e.g. intersect

- cut can be used to improve efficiency, e.g.
  \[\text{intersect}([], _, []).\]
  \[\text{intersect}([X|L], Y, [X|I]):-\]
  \[\text{member}(X,Y), \text{intersect}(L, Y, I).\]
  \[\text{intersect}([X|L], Y, I):-\]
  \[\text{\textbackslash+ member}(X,Y), \text{intersect}(L, Y, I).\]
  retests \text{member}(X,Y)\] twice

\[17\]

\[18\]
e.g. intersect

- using cut, we can avoid this
  \[\text{intersect}([], _, []).\]
  \[\text{intersect}([X|L], Y, [X|I]):-\]
  \[\text{member}(X,Y), !, \text{intersect}(L, Y, I).\]
  \[\text{intersect}([X|L], Y, I):-\]
  \[\text{\textbackslash+ member}(X,Y), \text{intersect}(L, Y, I).\]
- means that the last 2 rules are a conditional branch

\[19\]
cut can be used to define useful features

- If goal \(G\) should be false when \(C_1, \ldots, C_n\) holds, can write
  \[G :- C_1, \ldots, C_n, !, \text{fail}.\]
- not provable can be defined using cut
  \[\text{\textbackslash+ G} :- \text{G}, !, \text{fail}.
  \text{\textbackslash+ G}.\]

\[20\]
control predicates

- true (really success), e.g.
  \[G :- \text{Cond1}; \text{Cond2}; \text{true}.\]
- fail (opposite of true)
- repeat (always succeeds, infinite number of choice points)
  \[\text{loopUntilNoMore}:- \text{repeat}, \text{doStuff}, \text{checkNoMore}.\]
- but tail recursion is cleaner, e.g.
  \[\text{loop} :- \text{doStuff}, (\text{checkNoMore}; \text{loop}).\]
forcing all solutions

```prolog
test :- member(X, [1, 2, 3]),
      nl, print(X),
      fail.
% no alternative sols for print(X) and nl
% but member has alternative sols
?- test.
1
2
3
No
```

2nd order features: bagof & setof

- `?- bagof(T,G,L).` instantiates L to the list of all instances of T such for which G succeeds, e.g.
  `?- bagof(X,(member(X,[2,5,7,3,5]),X >= 3),L).`
  `X = _G172`
  `L = [5, 7, 3, 5]`
  Yes

- `setof` is similar to `bagof` except that it removes duplicates from the list, e.g.
  `?- setof(X,(member(X,[2,5,7,3,5]),X >= 3),L).`
  `X = _G172`
  `L = [3, 5, 7]`
  Yes

- can collect values of several variables, e.g.
  `?- bagof(pair(X,Y),(member(X,[a,b]),member(Y,[c,d])), L).`
  `X = _G157`
  `Y = _G158`
  `L = [pair(a, c), pair(a, d), pair(b, c), pair(b, d)]`
  Yes

2nd order features

- `setof` and `bagof` are called 2nd order features because they are queries about the value of a set or relation
- in logic, this is quantification over a set or relation
- not allowed in first order logic, but can be done in 2nd order logic
entering and leaving

- Trace steps are labelled:
  - Call: enter the procedure
  - Exit: exit successfully with bindings for variable
  - Fail: exit unsuccessfully
  - Redo: look for an alternative solution
- 4 ports model

Tail recursion optimization in Prolog

- Suppose have goal A and rule A' :- B₁, B₂, ..., Bₙ₋₁, Bₙ, and A unifies with A' and B₂, ..., Bₙ₋₁ succeed
- If there are no alternatives left for A and for B₂, ..., Bₙ₋₁ then can simply replace A by Bₙ on execution stack
- In such cases the predicate A is tail recursive
- Nothing left to do in A when Bₙ succeeds or fails/backtracks, so we can replace call stack frame for A by Bₙ's; recursion can be as space efficient as iteration

e.g. factorial

- Simple implementation:
  - fact(0,1).
  - fact(N,F):- N > 0, N₁ is N - 1, fact(N₁,F₁), F is N * F₁.
- Close to mathematical definition
- But not tail-recursive
- Requires O(N) in stack space

e.g. factorial

- Better implementation:
  - fact(N,F):- fact1(N,1,F).
    fact1(0,F,F).
    fact1(N,T,F):- N > 0, T₁ is T * N, N₁ is N - 1, fact1(N₁,T₁,F).
- Uses accumulator
- Is tail-recursive and each call can replace the previous call
- Can prove correctness
e.g. append

- append([],L,L).
- append([X|R],L,[X|RL]):-
  append(R,L,RL).
- append is tail recursive if first argument is fully instantiated.
- Prolog must detect the fact that there are no alternatives left; may depend on clause indexing mechanism used.
- use of unification means more relations are tail recursive in Prolog than in other languages.

split

split([],[],[]).
split([X],[X],[X]).
split([X1,X2|R],[X1|R1],[X2|R2]):- split(R,R1,R2).

Tail recursive!

merge

merge([],L,L).
merge([L],[],L).
merge([X1|R1],[X2|R2],[X1|R]):-
  order(X1,X2), merge(R1,[X2|R2],R).
merge([X1|R1],[X2|R2],[X2|R]):-
  not order(X1,X2), merge([X1|R1],R2,R).

Tail recursive, but lack of alternatives may be hard to detect (can use cut to simplify).

merge sort

mergesort([],[]).
mergesort([X],[X]).
mergesort(L,S): - split(L,L1,L2),
  mergesort(L1,S1),
  mergesort(L2,S2),
  merge(S1,S2,S).
for more on tail recursion

- see Sterling & Shapiro The Art of Prolog Sec. 11.2

Example: Finite State Automata

finite state automata

- a finite state automaton \((\Sigma, S, s_0, \delta, F)\)
  is a representation of a machine as a
  - finite set of states \(S\)
  - a state transition relation/table \(\delta\)
    - mapping current state & input symbol
      from alphabet \(\Sigma\) to the next state
  - an initial state \(s_0\)
  - a set of final states \(F\)

accepting an input

- a fsa accepts an input sequence from an alphabet \(\Sigma\) if, starting in the
designated starting state, scanning the
input sequence leaves the automaton in a final state
- sometimes called recognition
- e.g. automaton that accepts strings of x’s and y’s with an even number of x’s and an odd number of y’s
example

- automaton that accepts strings of x’s and y’s with an even number of x’s and an odd number of y’s
- idea: keep track of whether we have seen even number of x’s and y’s
- \( S = \{ ee, eo, oe, oo \} \)
- \( s_0 = ee \)
- \( \delta = \{(ee, x, oe), (ee, y, eo), \ldots \} \)
- \( F = \{ eo \} \)

implementation

- \( fsa(Input) \) succeeds if and only if the \( fsa \) accepts or recognizes the sequence (list) \( Input \).
- initial state represented by a predicate
  - \( initial_state(State) \)
- final states represented by a predicate
  - \( final_states(List) \)
- state transition table represented by a predicate
  - \( next_state(State, InputSymbol, NextState) \)
- note: \( next_state \) need not be a function

implementing \( fsa/1 \)

- \( fsa(Input) :- initial_state(S), scan(Input, S). \)
  - \( scan \) is a Boolean predicate
- \( scan([], State) :- final_states(F), member(State, F). \)
- \( scan([Symbol | Seq], State) :- next_state(State, Symbol, Next), scan(Seq, Next). \)

result propagation

- \( scan \) uses pumping/result propagation
- carries around current state and remainder of input sequence
- if FSA is deterministic, when end of input is reached, can make an accept/reject decision immediately; tail recursion optimization can be applied
- if FSA is nondeterministic, may have to backtrack; must keep track of remaining alternatives on execution stack
**non-determinism**

- A non-deterministic FSA accepts an input sequence if there exists at least one sequence which leaves the automaton in one of its final states.
- `?- fsa(Input).`
- `scan` searches through all possible choices for `Symbol` at each state;
- `fails` only if no sequence leads to a final state.

**representing tables**

- Can use binary connector, e.g., `A-B-C` instead of `next_state(A,B,C)`
  - Reduces typing;
  - Can make it easier to check for errors.
- `ee-x-oe. ee-y-eo.`
- `oe-x-ee. oe-y-oo.`
- etc.

**revised version**

```prolog
scan([], State) :- final_states(F),
              member(State, F).
scan([Symbol | Seq], State) :-
    State-Symbol-Next,
    scan(Seq, Next).
```