

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

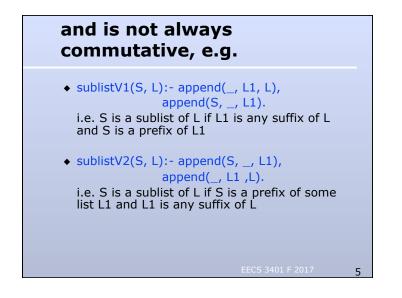
procedural and declarative semantics

- 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.

CS 3401 F 2017

and & or

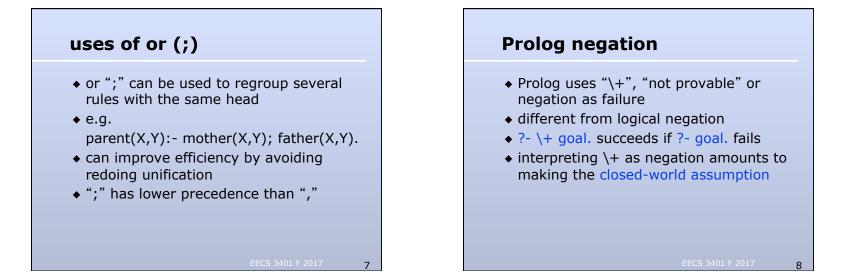
- Prolog's and (,) & or (; and alternative facts and rules that match a goal) are not purely logical operations
- often important to consider the order in which goals are attempted
 - left to right for "," and ";"
 - top to bottom for alternative facts/rules

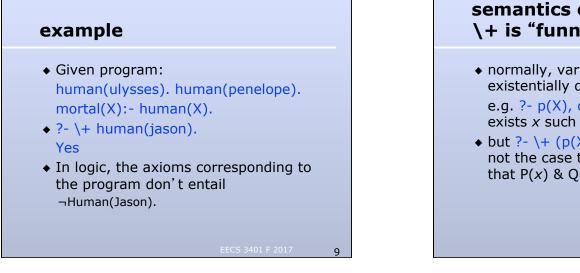


and is not always commutative, e.g.

- ?- sublistV1([c,b], [a, b, c, d]).
 false.
- sublistV2([c,b], [a, b, c, d]). ERROR: Out of global stack why?

EECS 3401 F 2017

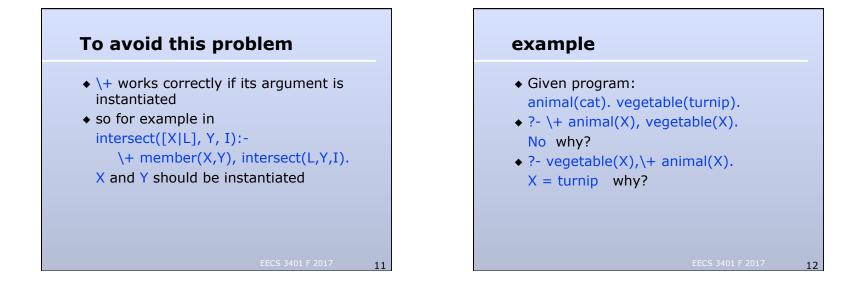


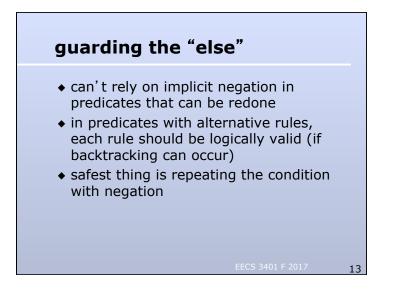


semantics of free variables in \+ is "funny"

- normally, variables in a query are existentially quantified from outside
 e.g. ?- p(X), q(X). represents "there
- 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)"

EECS 3401 F 2017

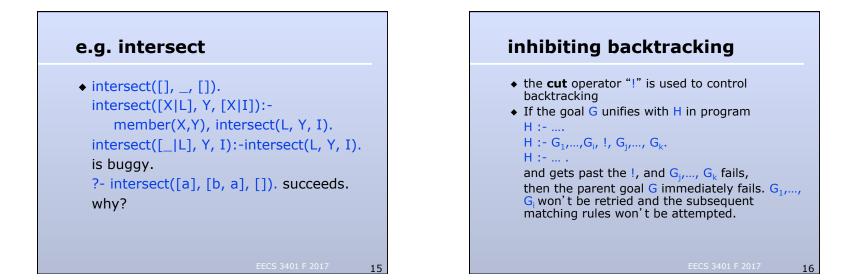


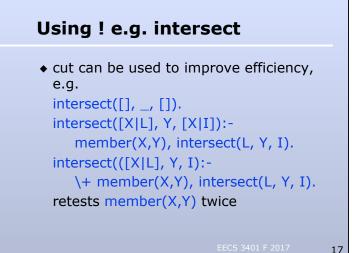


e.g. intersect

 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.

EECS 3401 F 2017





e.g. intersect

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

EECS 3401 F 2017

18

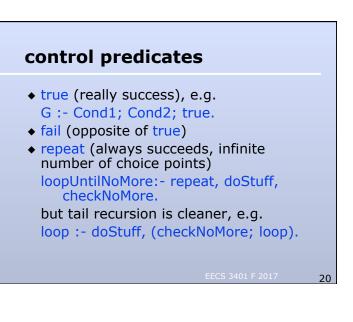
cut can be used to define useful features

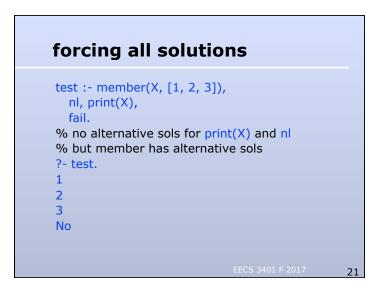
 ◆ If goal G should be false when C₁,..., C_n holds, can write

G :- C₁,..., C_n, !, fail.

not provable can be defined using cut
 + G :- G, !, fail.
 + G.

efined using cut





2nd order features: bagof & setof

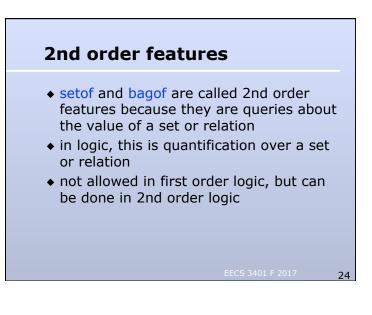
?- bagof(T,G,L). instantiates L to the list of all instances of T such for which G succeeds, e.g.
?- bagof(X,(member(X,[2,5,7,3,5]),X >= 3),L).
X = _G172
L = [5, 7, 3, 5]
Yes

ECS 3401 F 2017

22

2nd order features: bagof & setof

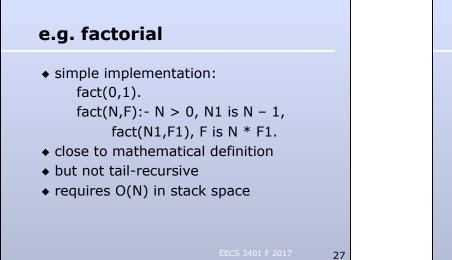
setof is similar to bagof except that it removes duplicates from the list, e.g.
setof(X,(member(X,[2,5,7,3,5]),X >= 3),L).
X = _G172
L = [3, 5, 7]
Yes
can collect values of several variables, e.g.
bagof(pair(X,Y),(member(X,[a,b]),member(Y,[c,d])),
L).
X = _G157
Y = _G158
L = [pair(a, c), pair(a, d), pair(b, c), pair(b, d)]
Yes

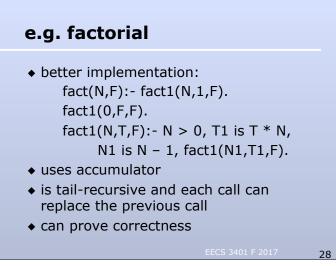


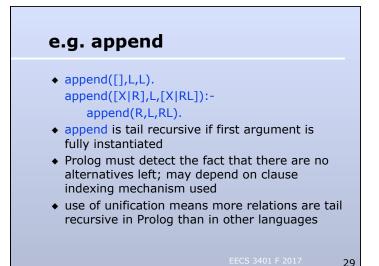


Tail recursion optimization in Prolog

- ◆ suppose have goal A and rule A' :- B₁, B₂, ..., B_{n-1}, B_n. and A unifies with A' and B₂, ..., B_{n-1} succeed
- if there are no alternatives left for A and for B_2 , ..., B_{n-1} then can simply replace A by B_n on execution stack
- in such cases the predicate A is tail recursive
- nothing left to do in A when B_n succeeds or fails/backtracks, so we can replace call stack frame for A by B_n 's; recursion can be as space efficient as iteration ECS = 3401 + 2017 26







split

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

Tail recursive!

ECS 3401 F 2017

30

32

merge

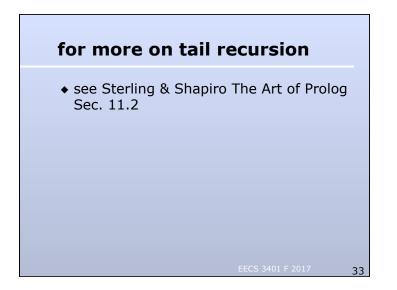
merge([],L,L).
merge(L,[],L).
merge([X1|R1],[X2|R2],[X1|R]):order(X1,X2), merge(R1,[X2|R2],R).
merge([X1|R1],[X2|R2],[X2|R]):not order(X1,X2), merge([X1|R1],R2,R).

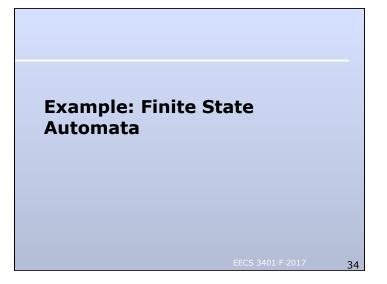
Tail recursive, but lack of alternatives may be hard to detect (can use cut to simplify).

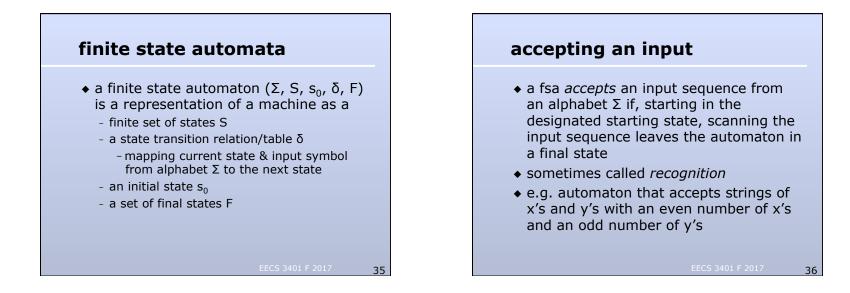
31

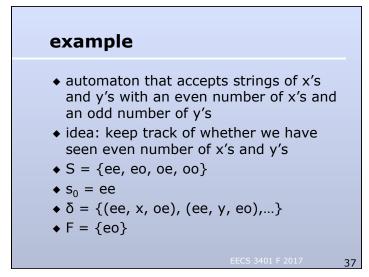
merge sort

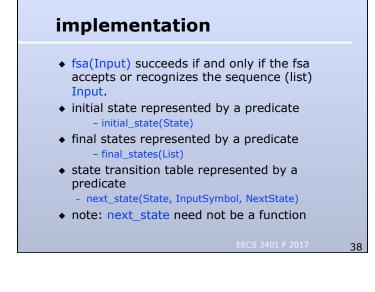
EECS 3401 F 2017

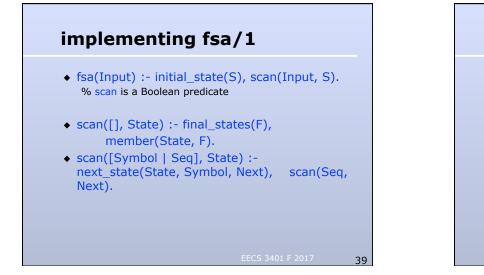












scan uses pumping/result propagation carries around current state and remainder of input sequence

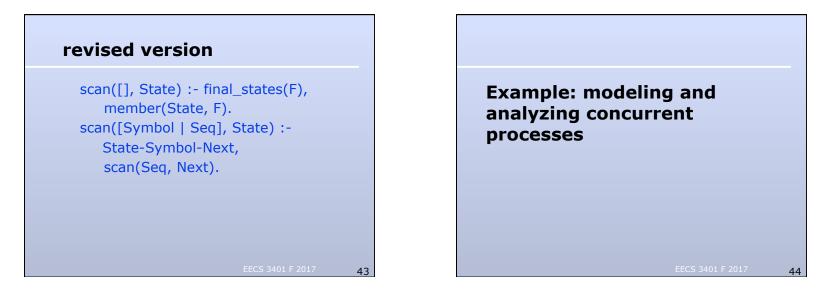
- if FSA is deterministic, when end of input is reached, can make an accept/reject decision immediately; tail recursion optimization can be applied
- if FSA is nondeterministic, may have to backtrack; must keep track of remaining alternatives on execution stack

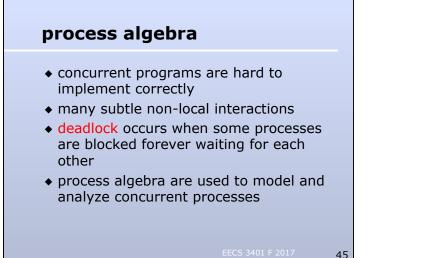




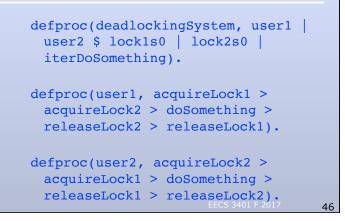
- can make it easier to check for errors
- ◆ ee-x-oe. ee-y-eo.
- ◆ 0e-x-ee. 0e-y-00.

CS 3401 F 2017



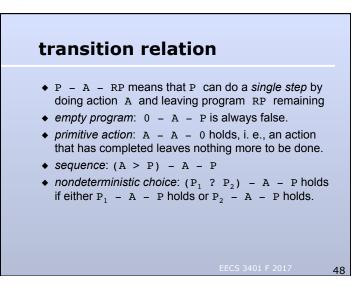


deadlocking system example



deadlocking system example

```
defproc(lock1s0,
    acquireLock1 > lock1s1 ? 0).
defproc(lock1s1, releaseLock1 > lock1s0).
defproc(lock2s0,
    acquireLock2 > lock2s1 ? 0).
defproc(lock2s1,releaseLock2 > lock2s0).
defproc(iterDoSomething,
    doSomething > iterDoSomething ? 0).
EECS 3401 F 2017 47
```





- interleaved concurrency: (P₁ | P₂) A P holds if either P₁ - A - P₁₁ holds and P = (P₁₁ | P₂), or P₂ - A - P₂₁ holds and P = (P₁ | P₂₁)
- synchronized concurrency: $(P_1 \ \ P_2) A P$ holds if both $P_1 - A - P_{11}$ holds and $P_2 - A - P_{21}$ holds and $P = (P_{11} \ \ P_{21})$
- recursive procedures: ProcName A P holds if ProcName is the name of a procedure that has body B and B – A – P holds.

S 3401 F 2017

49

51

can check properties by searching process graph

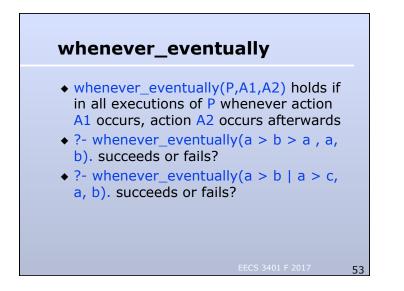
- a process has an *infinite execution* if there is a cycle in its configuration graph
- e.g. defproc(aloop, a > aloop)
- has_infinite_run(P):- P _ PN, has_infinite_run(PN,[P]).
- has_infinite_run(P,V) holds if process P has an infinite run when it has already visited configurations in the list V

EECS 3401 F 2017

50

checking properties by searching process graph cannot_occur(P,A) holds if no execution of P where action A occurs search graph for a transition P1 - A - P2 useful built-in predicate: forall(+Cond, +Action) holds iff for all bindings of Cond, Action succeeds e.g. forall(member(C,[8,3,9]), C >= 3) succeeds

cannot_occur(a > b | a > c, b). succeeds or fails?
?- cannot_occur((a > b | a > c)\$(a > c), b). succeeds or fails?



whenever_eventually examples

?- whenever_eventually(loop1, a, b). succeeds or fails, where defproc(loop1, a > b > loop1)?
?- whenever_eventually(loop1, b, a). succeeds or fails, where defproc(loop1, a > b > loop1)?
?- whenever_eventually(loop2, b, a). succeeds or fails, where defproc(loop2, a > b > (loop2 ? 0)).

EECS 3401 F 2017

