U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTMENT OF INFORMATICS ENGINEERING

U.PORTO
FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2024/2025 – 1st Semester

Prolog

Database Modification / Cycles

Daniel Castro Silva
dcs@fe.up.pt

Agenda

- Database Modification
  - Memoization
- Cycles

Database Modification

- Prolog allows clauses to be dynamically added or removed from a program
  - This provides great flexibility
  - However, modifying the program is costly, as it requires re-indexing
- To add or remove clauses from an existing predicate, it first needs to be declared as dynamic
  - Several predicates can be declared dynamic at once

```
:- dynamic male/1, female/1, parent/2.
```

Adding Clauses

- ***assert/1*** adds a new clause to the program

```
ask_and_add_to_kb:-
      write('Insert Parent-Child to add'),nl,
      read(P-C),
      assert(parent(P, C)).
```

  - When adding a rule, an additional pair of parentheses is required

```
| ?- assert(( father(X,Y):- male(X), parent(X,Y) )).
yes
| ?- father(homer, X).
X = bart ?
yes
```

Adding Clauses

- ***assert/1*** inserts the new clause in an arbitrary position within the predicate
- Two preferred variations exist:
  - ***asserta/1*** – the new clause is added before all existing predicate clauses (if any)
  - ***assertz/1*** – the new clause is added after all existing predicate clauses (if any)

```
| ?- asserta( parent(abe, homer) ).
yes
| ?- parent(X, Y).
X = abe,
Y = homer ? ;
X = homer,
Y = bart ?
```

Removing Clauses

- ***retract/1*** removes a clause from the program (the first that matches the given clause)

```
replace_name:-
        retract( parent(abe, homer) ),
        asserta( parent(abraham, homer) ).
```

  - When removing a rule, an additional pair of parentheses is required

```
| ?- retract(( father(X,Y):- male(X), parent(X,Y) )).
yes
| ?- father(X, Y).
no
```

Removing Clauses

- Successive calls to *retract/1* can remove all predicate clauses but not the predicate's properties and definition

```
remove_fathers:-
        retract( father(X,Y) ),
        fail.
remove_fathers.
```

```
| ?- remove_fathers.
yes
| ?- father(X,Y).
no
| ?- mother(X,Y).
! Existence error in user:mother/2
! procedure user:mother/2 does not exist
! goal:  user:mother(_357,_359)
| ?-
```

Removing Clauses

- ***retractall/1*** retracts all clauses matching the specified head
  - Even if retracting rules, only the head is specified
- ***abolish/1*** removes all clauses and properties of the specified predicate

```
retractall(ancestor(_X, _Y)).
abolish(parent/2).
```

```
| ?- retractall(father(X,Y)).
yes
| ?- father(X,Y).
no
| ?- abolish(father/2).
yes
| ?- father(X,Y).
! Existence error in user:father/2
! procedure user:father/2 does not exist
! goal:  user:father(_357,_359)
| ?-
```

Predicate Listing

- ***listing/0*** lists all clauses from the currently loaded program
- ***listing/1*** lists all clauses from a given predicate
- These predicates list the code in the current output stream
  - Note that variable naming and code formatting are not preserved

```
a(X, Y):- b(X), !, b(Y).
a(3, 4).
b(2).
b(3).
```

```
| ?- listing.
a(A, B) :-
        b(A), !,
        b(B).
a(3, 4).

b(2).
b(3).

yes
| ?- listing(a/2).
a(A, B) :-
        b(A), !,
        b(B).
a(3, 4).

yes
```

Accessing Clauses

- ***clause(+Head, ?Body)*** allows access to the clauses of a given predicate in the knowledge base

```
a(X, Y):- b(X), !, b(Y).
a(3, 4).
b(2).
b(3).
```

```
| ?- clause( a(X,Y), _Body ),
     retract(( a(X,Y):- _Body )),
     a(A, B),
     asserta(( a(X,Y):- _Body )).
A = 3,
B = 4 ?
yes
| ?- listing(a/2).
a(A, B) :-
        b(A), !,
        b(B).
a(3, 4).

yes
| ?- clause( a(X,Y), Body ), retract(( a(X,Y):-Body )).
Body = (b(X),!,b(Y)) ?
yes
| ?- listing(a/2).
a(3, 4).

yes
```

Database Modification

- Changes to the predicate being executed only take effect after the predicate finishes execution

```
| ?- assert(( test_retract:- write(before), nl,
 retractall(test_retract), write(after) )).
yes
| ?- listing.
test_retract :-
        write(before),
        nl,
        retractall(test_retract),
        write(after).

yes
| ?- test_retract.
before
after
yes
| ?- listing.
yes
```

```
| ?- assert(( test_retract_2(N):- write(here),
N1 is N-1, retractall( test_retract_2(_) ),
test_retract_2(N1) )).
yes
| ?- listing(test_retract_2/1).
test_retract_2(A) :-
        write(here),
        B is A-1,
        retractall(test_retract_2(_)),
        test_retract_2(B).

yes
| ?- test_retract_2(3).
here
no
```

Database Modification

- Predicate are assumed not to be dynamic if they exist in the code without the *:-dynamic* declaration

```
try_assert:-
      assertz( guess(1) ),
      assertz( guess(2) ).
```

```
| ?- guess(X).
! Existence error in user:guess/1
! procedure user:guess/1 does not exist
! goal:  user:guess(_357)
| ?- try_assert.
yes
| ?- guess(X).
X = 1 ? ;
X = 2 ? ;
no
```

```
guess(0).

try_assert:-
      assertz( guess(1) ),
      assertz( guess(2) ).
```

```
| ?- guess(X).
X = 0 ? ;
no
| ?- try_assert.
! Permission error: cannot assert static user:guess/1
! goal:  assertz(user:guess(1))
| ?-
```

Database Modification

- Assert and retract should be used sparingly (ideally only for things that do not change often)
  - They are slow operations
  - It can make programs harder to understand / debug

- The effect of database modification predicates is not undone in backtracking (just like input/output)

See section 4.12 of the SICStus Manual for more information

Memoization

- Modifying the database can be used to save partial results, resulting in a dynamic programming approach

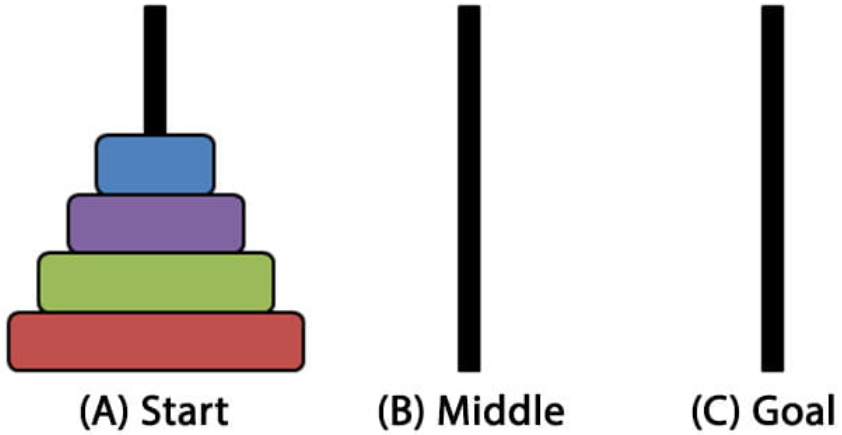
```
fib(0, 0):- !.
fib(1, 1):- !.
fib(N, F):-
      N2 is N-2, N1 is N-1,
      fib(N2, F2), fib(N1, F1),
      F is F2 + F1.
```

```
fib(0, 0):- !.
fib(1, 1):- !.
fib(N, F):-
      N2 is N-2, N1 is N-1,
      fib(N2, F2), fib(N1, F1),
      F is F2 + F1,
      asserta(( fib(N, F):-! )).
```

Could we use *assertz* instead?

Games and Memoization

- Example: Tower of Hanoi
  - Goal: move stack from pole 1 to pole 3
  - Rules:
    - Can only move one disk at a time
    - Disks can only be placed on top of a larger disk

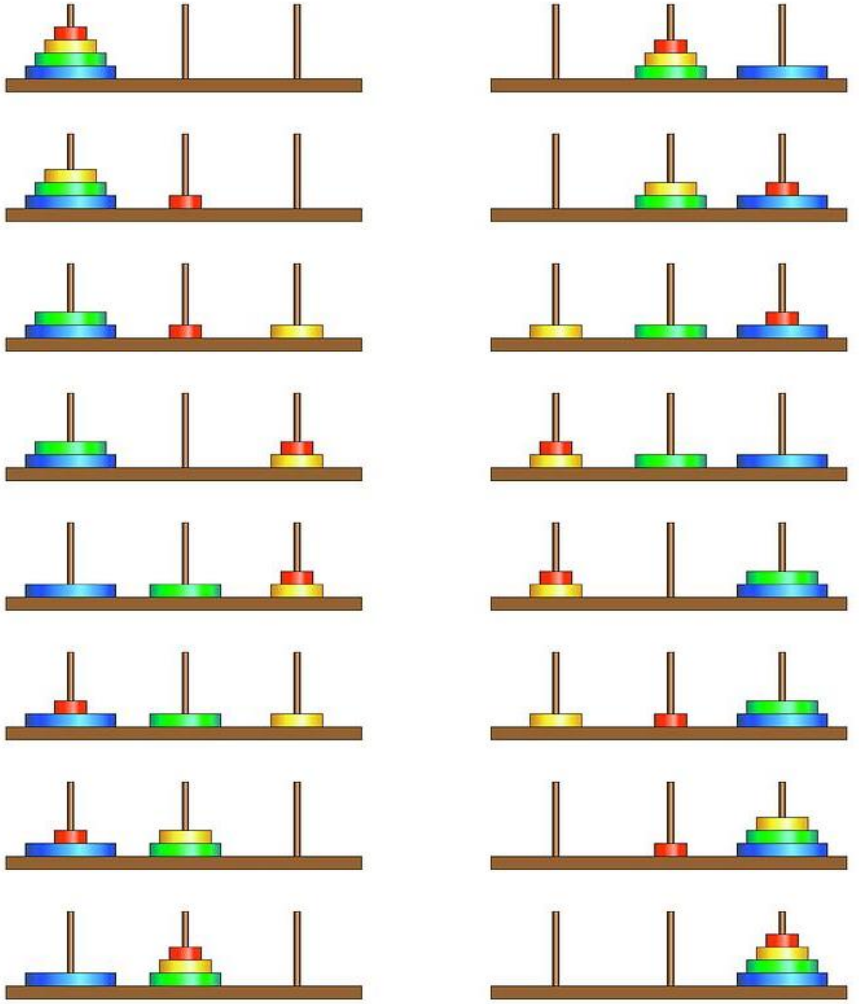


Games and Memoization

- To move a stack of size N from pole 1 to pole 3, first move stack of size N-1 to pole 2, move base piece, and then move N-1 stack from pole 2 to pole 3

```
hanoi(1, A, B, C, [A-C]).
hanoi(N, A, B, C, Moves):-
    N > 0, N1 is N-1,
    hanoi(N1, A, C, B, First),
    hanoi(N1, B, A, C, Last),
    append(First, [A-C|Last], Moves),
    asserta((hanoi(N, A, B, C, Moves):-!)).

test_hanoi(N, A, B, C, M):-
      hanoi(N, X, Y, Z, M), X-Y-Z=A-B-C.
```



Database Modification

- We can also use database modification as an alternative to finding all answers to a query

```
get_all_children(Parent, _Children):-
      assert( children(Parent, []) ),
      fail.
get_all_children(Parent, _Children):-
      parent(Parent, Child),
      retract( children(Parent, Current) ),
      assert( children(Parent, [Child|Current]) ),
      fail.
get_all_children(Parent, Children):-
      retract( children(Parent, Children) ).
```

Why is this approach inefficient?

Agenda

- Database Modification
  - Memoization
- Cycles

Failure Driven Loops

- The example above for finding all answers is a failure driven loop
  - The *fail* forces Prolog to backtrack until all solutions are found

```
failure_driven_loop:-
      find_solution(X),
      do_something_with_solution(X),
      fail.
failure_driven_loop.          % ensure predicate succeeds
```

  - Efficient in terms of memory use
  - Usually only used in situations when only side effects are important (results are not kept)

Failure Driven Loops

- Failure driven loops are an alternative to recursive ones
  - Compare the following two approaches to implement a predicate *print_n(+N, +C)*, which prints a character *C* to the terminal *N* times

```
print_n(0, _C):- !.
print_n(N, C):-
      write(C),
      N1 is N-1,
      print_n(N1, C).
```

```
print_n(N, C):-
      between(1, N, _T),
      write(C),
      fail.
print_n(_N, _C).
```

Which approach is more efficient?

Failure Driven Loops

- Another example: consulting a program

```
consult(File):-
      see(File),
      read_loop,
      seen.

read_loop:-
      repeat,
      read(Clause),
      process(Clause), !.

process(end_of_file):- !.
process(Clause):-
      assertz(Clause),
      fail.
```

Generic Game Program

- A recursive loop can be used to code a generic 2-player game:

```
play_game:-
        initial_state(GameState-Player),
        display_game(GameState-Player),
        game_cycle(GameState-Player).

game_cycle(GameState-Player):-
        game_over(GameState, Winner), !,
        congratulate(Winner).
game_cycle(GameState-Player):-
        choose_move(GameState, Player, Move),
        move(GameState, Move, NewGameState),
        next_player(Player, NextPlayer),        % could be done in move/3
        display_game(NewGameState-NextPlayer), !,
        game_cycle(NewGameState-NextPlayer).
```

Generic Game Program

```
choose_move(GameState, human, Move):-
      % interaction to select move
choose_move(GameState, computer-Level, Move):-
      valid_moves(GameState, ValidMoves),
      choose_move(Level, GameState, ValidMoves, Move).

valid_moves(GameState, Moves):-
      findall(Move, move(GameState, Move, NewState), Moves).

choose_move(1, _GameState, Moves, Move):-
      random_select(Move, Moves, _Rest).
choose_move(2, GameState, Moves, Move):-
      setof(Value-Mv, NewState^( member(Mv, Moves),
            move(GameState, Mv, NewState),
            evaluate_board(NewState, Value) ), [_V-Move|_]).

% evaluate_board assumes lower value is better
```

Q & A

When you're writing Prolog and it succeeds on the first try