Modeling Games with Prolog Expert Systems

- **Outline:**
  - Modeling Games with Expert Systems
  - Hearts
    - Domain Model
    - Encoding Rules
    - Encoding Player Strategies
  - Adventure Game
    - Domain Model
    - Encoding World Knowledge
    - Encoding Monster AI Strategies

Games as Expert Systems

Types of Game-Playing Systems

- **Search Based**
  - Use an evaluation function to evaluate moves
  - Use minimax to search for the best possible move
  - Assumes that both players are identical
- **Expert System Based**
  - Encode "expert" knowledge about what moves to make.
  - Can respond to different opponent strategies

Goals for Expert System Games

- **Two different tasks:**
  - Encode expert knowledge about a complex domain
    - Deduce complex information about objects in the world
  - Encode expert knowledge about strategies
    - Deduce the best move
- **Four examples:**
  - Hearts
    - Encode the rules of hearts
    - Encode player strategies
  - Adventure game
    - Encode knowledge about the adventure game world.
    - Encode monster AI strategies

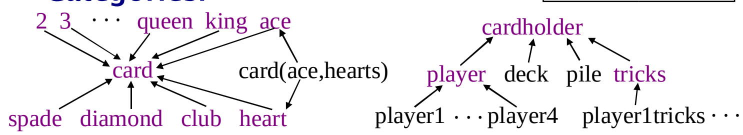
Hearts: Abridged Rules

- 4-player card game. Each player draws 13 cards.
- Each player plays a card, in clockwise order.
  - First card played by the player who took the previous trick
  - Each player must follow suit if possible.
  - If a player is out of cards in the suit, they may play any card.
- The highest ranked card in the initially lead suit takes the trick.
- Play continues until all card have been played.
- Scoring (lower scores are better)
  - Each player gets 1 point for each heart they took
  - The player that took the queen of spades gets an additional 13 points

Modeling the Hearts Domain

- **Objects:**
  - Cards: e.g., card(queen, spades)
  - Players: e.g., player2
  - Tricks taken by a player: e.g., player2tricks
  - The undealt cards: deck
  - The cards on the table: pile
- **Categories:**

Purple: category
Black: object



Aside: Propositions vs Structures

- **We must decide how to represent knowledge:**
  - *directly*, using propositions
    `deck([card(3,hearts), card(2,spades), …])`
  - *indirectly*, using structures
    `prop(deck, has, [card(3,hearts), card(2,spades), …])`
  - Direct representation is simpler
  - Indirect representation is more flexible
    - We can keep track of multiple decks.
    - Decks can inherit properties (e.g., size)
- **For this example, we will represent all knowledge indirectly.**

Modeling the Hearts Domain (2)

- **Properties:**
  - prop(Player, turn, Bool).
    - Is it the player's turn to play?
  - prop(Cardholder, has, [Card1, Card2, …]).
    - The cards held by a player, deck, or pile.
  - prop(Player, tricks, PlayerTricks).
    - The tricks taken by a player.
  - prop(Card, points, N).
    - Number of points associated with a card (1 for hearts, 13 for Queen of spades)
  - prop(Card, rankval, N).
    - Rank value of a card, used to decide highest card (2-14).

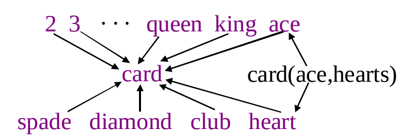
Modeling the Hearts Domain (3)

- **We can use inheritence to define points:**

```
prop(card, points, 0).
prop(heart, points, 1).
prop(Card(queen,spades), points, 13).
```

- **We can use inheritence to define rankval:**

```
prop(N, rankval, N) :- number(N).
prop(jack, rankval, 11).
prop(queen, rankval, 12).
prop(king, rankval, 13).
prop(ace, rankval, 14).
```



Using the Hearts Domain Model

Two uses for the hearts domain model:

1) Encode knowledge about the rules of the game

2) Encode knowledge about strategies for playing the game

Encoding Hearts Rules

- Now that we have a basic domain model, we can start encoding expert knowledge.
- Define two top-level predicates:
  - start.
  - play(card).

Sample Game

| ?- start. Move: player3 Pile: (empty) Cards: Card(3, hearts) Card(7, spades) … Card(2, clubs) yes	| ?- play(Card(2,clubs)). Move: player4 Pile: Card(2,clubs) Cards: Card(5, diamonds) Card(ace, clubs) … Card(10, hearts) yes	| ?- …

Non-Monotonic Logic (Review)

- "assert(…)" adds a fact or rule.
- "retract(…)" removes a fact or rule.
- Assert and retract be included in rules:

```
go(north) :- at(X), path(X,Y,north),
             retract(at(X)), assert(at(Y)).
```

- ":- dynamic …" declares what facts can change.
  - If we plan to modify prop and at:

    ```
    :- dynamic prop/3, at/1.
    ```

  - Put "dynamic" statements at the top of your Prolog source file.

Hearts: Starting the Game

- Starting the game:

```
start :- reset, shuffle, deal.
```

- Dealing cards to players:

```
deal :- prop(deck, has, [C1,C2,C3,C4|Cards]),
        give(player1, C1), give(player2, C2),
        give(player3, C3), give(player4, C4),
        retract(prop(deck,has,[C1,C2,C3,C4|Cards]),
        assert(prop(deck,has,Cards)),
        deal.
deal :- prop(deck, has, []).
```

- Giving cards to cardholders:

```
give(X, Card) :- prop(X, has, Cards),
                 retract(prop(X, has, Cards)),
                 assert(prop(X, has, [Card|Cards])).
```

Hearts: Resetting the Game

- Reset the game in two steps:

```
reset :- clear, setup.
```

  - First, clear all temporary assertions:

```
clear :- retract(prop(_, has, _)), clear.
clear :- retract(prop(_, turn, _)), clear.
clear.
```

  - Then, set up initial conditions:

```
setup :- assert(prop(deck, has, [])),
         assert(prop(pile, has, [])),
         assert(prop(player1, has, [])),
         assert(prop(player2, has, [])),
         assert(prop(player3, has, [])),
         assert(prop(player4, has, [])).
```


Nondeterminism

- `shuffle` is nondeterministic.
- Implement it using the random library, which provides a basic random number generator.
  - Loading the random library:

```
:- use_module(library(random)).
```

  - `random(Lower, Upper, N)` binds `N` to a random number in the interval [Lower, Upper)
- Use `random` to implement `permute`; and use `permute` to implement `shuffle`:

```
shuffle :- prop(deck, has, Cards),
           permute(Cards, ShuffledCards),
           retract(prop(deck, has, Cards)),
           assert(prop(deck, has, ShuffledCards)).
```


Choose and Permute

- Two useful nondeterministic functions:
  - Choose a random element from a list:

```
choose(List, Elt) :- length(List, Len),
                     Bound is Len+1,
                     random(1, Bound, Index),
                     nth(Index, List, Elt).
```

  - Permute a list:

```
permute(L1, [Elt|L3]) :- choose(L1, Elt),
                         delete(L1, Elt, L2),
                         permute(L2, L3).
```


Aside: Libraries

- Libraries extend the set of built-in functions.
- "`:- use_module(...).`" loads libraries

```
:- use_module(library(lists)).
:- use_module(library(random)).
```

- Some useful libraries:
  - lists: provides basic list operations
  - random: provides a random number generator
  - queues: defines operations on queues
  - tcltk: Tcl/Tk graphical interfaces
  - timeout: run goals with execution time limits

Hearts: Playing the Game

- **Basic algorithm for `play(Card)`:**
  - Check who the current player is
  - Check that the play is valid
  - Remove the card from the current player's hand
  - Add the card to the pile
  - If the pile contains 4 cards:
    - Decide who won the round
    - Add the pile to the winner's tricks
    - Clear the pile
    - Set the next player to the winner
  - Otherwise:
    - Set the next player (rotate clockwise).

Hearts: Playing the Game (2)

```
play(C) :- prop(P, turn, 1), validmove(P, C),
           prop(pile, has, PileCards),
           give(pile, C), take(P, C),
           finishplay, printstatus.

finishplay :- prop(pile, has, [C1,C2,C3,C4]),
              winner([C1,C2,C3,C4], Winner),
              prop(Winner, tricks, Tricks),
              give(Tricks, C1), …, give(Tricks, C4),
              clear(deck), prop(P, turn, 1),
              retract(prop(P,turn,1)),
              asssert(prop(Winner,turn,1)), !.

finishplay :- prop(P, turn, 1),
              clockwise(P, P2),
              retract(prop(P,turn,1)),
              asssert(prop(P2,turn,1)).
```


Hearts: Playing the Game (3)

- **`play` is based on functions that encode information about the rules of Hearts:**
  - winner([C1,C2,C3,C4], P): Player P wins the given round.
  - validmove(P,C): Player P may play card C at this time.
  - hearts_broken: At least one heart has been played.
  - void(P, S): Player P is void in suit S.
  - highcard([C1, C2, ...], C): Card C has the highest rankval.
  - score(Tricks, S): The total score for the given tricks is S.
  - suit_lead(S): S was the suit lead.

Encoding Strategies for Hearts

- **Use the same domain model that we used for *rules* to encode knowledge about *strategies*.**
  - Define `pick(P,C)`
    - True if player `P` chooses to play card `C`.
  - Strategies need new types of information:
    - Has the queen of spades been played yet?
    - who has the queen of spades?
    - is someone trying to shoot the moon?
    - what strategy is each player currently using?
    - what strategies does each player tend to use?
  - Define a new predicate `thinks(Player, (X,Prop,Y))`
    - True if `Player` thinks that `X`'s property `prop` has value `Y`.
    - Example:
      `thinks(player1,(player2,has,[card(queen,spades)])).`

Encoding Strategies for Hearts (2)

- Some inferences depend on transient information:
  - Once a round is complete...
    - There is no record of who played which card.
    - There is no record of who led the round.
  - Once a game is complete...
    - There is no evidence of who played what.
- Make inferences when the information is available, and store the results.
  - Define a new predicate, `examine_play(P)`, that is called for each player after each move.

Encoding Strategies for Hearts (3)

- examine_play(P) consists of a set of clauses that are executed for side effect.
  - All clauses except the last one will always fail.
  - This ensures that every clause gets evaluated.

```
examine_play(P) :- (conditions),
      assert(thinks(P, (P2, has, card(queen, spades)))),
      fail.

examine_play(P) :- (conditions),
      assert(thinks(P, (P2, strategy, shoot_the_moon))),
      fail.
...
examine_play(P).
```


Encoding Strategies for Hearts (4)

- `pick` uses `think` and information about the current game state to choose which card to play.
  - `Pick` is implemented with an ordered list of conditions.

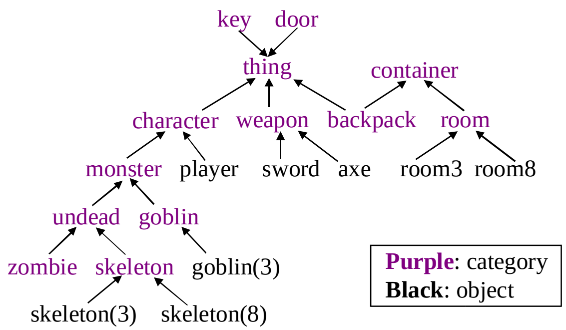
```
pick(P, card(R,heart)) :-
      validmove(P, Card(R,heart)),
      think(P, (P2, strategy, shoot_the_moon),
      P \== P2, +\hearts_broken.

pick(P, card(R,S)) :-
      validmove(P, Card(R,S)),
      ...
...
```


A Simple Adventure Game

- The player controls a character that can:
  - Move around a map.
  - Pick up and drop objects.
  - Fight monsters.
  - Open and close doors.
  - Look at rooms and objects.
  - etc.
- For examples and detailed descriptions, see:
  - http://www.csc.vill.edu/~dmatusze/resources/prolog/spider.html
  - http://www.csc.vill.edu/~dmatusze/resources/prolog/sleepy.html
  - http://www.csc.vill.edu/~dmatusze/resources/prolog/prolog.ppt
  - http://www.csc.vill.edu/~dmatusze/resources/prolog/prolog1.ppt
  - http://www.csc.vill.edu/~dmatusze/resources/prolog/prolog2.ppt
  - http://www.csc.vill.edu/~dmatusze/resources/prolog/prolog3.ppt

Modeling the Adventure Game



Modeling the Adventure Game (2)

- Properties:
  - prop(Thing, in, Container).
  - prop(Character, health, Number).
  - prop(Monster, attack, Number).
  - prop(Room, description, String).
  - prop(Thing, description, String).
  - prop(Character, has, Backpack).
  - prop(Door, locked, Boolean).
  - prop(Key, unlocks, Door).
  - prop(Door, connects, (Room1, Room2)).
  - prop(Room, has_door, (Door, Direction).
  - etc.

Inheritance and Defaults

- Every thing has a location:
  - prop(thing, in, container).
  - prop(character, in, room).
- Use defaults to specify "normal" attributes for different kinds of characters:
  - prop(character, health, 10).
  - prop(character, attack, 5).
  - prop(undead, health, 6).
  - prop(skeleton, attack, 8).
- Doors are usually unlocked:
  - prop(door, locked, 0).
- Give default descriptions of objects:
  - prop(thing, description, "It's nondescript")

Using the Adventure Game Domain Model

- We will consider two uses for the adventure game domain model:

2) Encode knowledge about the how the world works.
   - What are the effects of various actions?
   - What can we deduce about the state of the world?

3) Encode knowledge about strategies for monsters.
   - What should a monster do in a given situation?

Game Commands

- n: go through the door to the north
- s: go through the door to the south
- e: go through the door to the east
- w: go through the door to the west
- look: look at the current room
- look_at(Thing): look at a given object
- take(Thing): Put Thing in your backpack
- drop(Thing): Remove Thing from your backpack.
- use(Key, Door): Use a key to open a door
- attack(Character): Attack a character
- inv: Display the contents of the your backpack.
- restart: Reset the game to its initial state
- etc.

Generalized Game Commands

- Define basic commands as special cases of more general commands, that take a Character as their first argument:
  ```
  • n :- go(n, player).    s :- go(s, player).
    e :- go(e, player).    w :- go(w, player).
  • look :- look(player).
  • take(Think) :- take(player, Thing).
  • use(Key, Door) :- use(player, Key, Door)
  ```
- This will allow our monster AI strategies to use these commands.

Giving Feedback: Prolog I/O

- Adventure game commands produce output for the player to read.
  - Use write to display strings:
    ```
    look(C) :- prop(C, in, R), prop(R, description, S),
               write(S), nl.
    ```
  - write can also display numbers and symbols:
    ```
    health(C) :- prop(C, health, H), write("You have "),
                 write(H), write(" hit points."), nl.
    ```
  - nl prints a newline.

Inventory Listing: findall

- Use findall to list all values that satisfy a given predicate.
  - Example uses:
    ```
    | ?- findall(X,(member(X,[5,-2,4,1]), X >= 2),L).
    L = [5,4]
    | ?- findall((N,V), nth(N,[7,5,3],V), L).
    L = [(1,7),(2,5),(3,3)] ?
    ```
- We can use findall to define inv:
  ```
  inv(C) :- prop(C, has, BP),
            findall(X, prop(X, in, BP), Items),
            write("You are carrying: "),
            write(Items), nl.
  ```

More Adventure Game Commands

- **Movement:**

```
go(C,Dir) :- prop(C,in,R),
             prop(Room,has_door,(Dir,Door)),
             prop(Door,connects,(R,R2)),
             retract(prop(C,in,R)),
             assert(prop(C,in,R2)), look(C), !.
go(_,_) :- write("You can't go that way"), nl.
```

- **Getting and dropping objects:**

```
get(C, Obj) :- prop(C,in,R), prop(Obj,in,R),
               prop(C,has,BP),
               assert(prop(Obj,in,BP)),
               retract(prop(Obj,in,R)),
               write("You pick up the",
               write(Obj), nl, !.
get(_, _) :- write("You can't get that"), nl.
```


Reasoning About the World

- **Define predicates that derive information about the world:**
  - `connected(R1, R2)` is true if rooms R1 and R2 are connected by some path.
  - `shortest_path(R1, R2, P)` is true if P is the shortest path from R1 to r2.
  - `sees(C, Thing, D)` is true if character C can see Thing in direction D.
  - `smells(C, Thing, D)` is true if character C can smell Thing in direction D.

Monster Strategies

- **We can also use the domain model to encode strategies for monsters.**
  - Define a predicate `go(C)` that performs a single action for character C.
  - Use the world model and world knowledge to decide what the mosnster should do.

Monster Strategies (2)

- **A simple monster strategy:**
  - Attack the player if you think you can win:

```
go(C) :- prop(C,in, R), prop(player,in,R),
         prop(C,health,CH), prop(player,health,PH),
         CH > PH, attack(C,player), !.
```

  - Otherwise, run away:

```
go(C) :- prop(C,in,R), prop(player,in,R),
         prop(R,has_door, (Dir,_)),
         go(C,Dir), !.
```

  - Go towards the player if you can smell her:

```
go(C) :- smell(C,player,Dir), go(C,Dir), !.
```

  - Otherwise, do nothing:

```
go(C).
```

- **Note the use of cut (!) to ensure that the monster only performs one action.**