

A Rule-Based Learning Poker Player

For this assignment, teams of two students will write a program to play poker. We will play a somewhat simplified version of poker: five-card stud with a single betting interval and with declarations.

Your program must be a rule-based agent, i.e. the actions of your poker player must be determined by a list of rules such as the following:

```
((forall x in other-active-players
  ((poker:> myHand (highHand x)) ==> (Raise maxRaise)))
 (exists x in other-active-players
  (exists y in other active players
   ((not (equal? x y))
    (poker:> myHand (lowHand x))
    (poker:< myHand (highHand y)) ==> (Pass))))
 (#t ==> (Call)))
```

On the left of the ==> symbol are *predicates*, on the right are *actions*. We will provide a number of predicates and actions that you can use, but you will also be able to write your own. We are also providing a function that will execute the rules that implement your player.

Your program must also incorporate some learning element. You can use learning to create some of rules for your player, or you can have your program attempt to learn the strategies of other players.

Your player will play several rounds of poker against some combination of other students' players and my players. We are still working out the exact details of play; the current plan for conducting a round is as follows. Five players will be seated at a table in random order, three student players and two players from a set of simple players that we will provide. Each player will start with 1,000 clams. In each hand, players can bet some number of clams against the other players. If a player loses all its clams, it does not participate in the remaining hands; the other players continue to play. The round is over when either only one player remains or when 200 hands have been played.

There are a lot of details to this assignment, so please read this handout carefully. Information on the required activities for this assignment appear at the end of this handout.

1 The game of poker

This section will cover the basic rules of poker and then the details of the specific variety of poker we'll play. We are using the basic rules described in "Hoyle's Rules of Games." Some of these rules may be different than the rules of poker you might know.

1.1 Basic rules

Poker is played with a regular deck of playing cards. There are 52 cards, 13 in each of the four suits: spades, diamonds, clubs, hearts. The 13 values in each suit are (in increasing order) 2 through 10, Jack, Queen, King, and Ace.

Poker is a game played in "hands." There are many different variations of poker; a hand of the basic game (straight poker) is played as follows. Five cards are dealt to each player face-down. Each player may look at his own cards but cannot see his opponents' cards. Next, there is a *betting interval* during which players place their wagers in the center of the card table. The pile of wagers (usually in the form of "chips") is called the *pot*. Once the bet has equalized, there is a *showdown* in which players show their cards, and the highest hand wins the pot.

The two essential features of all varieties of poker are betting intervals and the showdown; these are described in the following sections.

1.1.1 Betting intervals

At the start of a hand, all players are considered *active*. Each active player in turn (starting to the left of the dealer) has the opportunity to bet and must declare one of the following actions:

- *Pass*: The player does not add an additional wager to the pot and becomes inactive. The player forfeits the bet he has placed in the pot.
- *Call*: The player places an additional wager in the pot to match the maximum total bet of any other player.
- *Raise*: The player first calls and then places an additional wager in the pot.

The bet continues to rotate among all active players until it has *equalized*, i.e. when all active players have made identical total bets. This occurs when all other players call or pass after a raise.

1.1.2 The Showdown

At the showdown, the active players show their cards, and the player with the highest poker hand wins the entire pot. In high-low versions of poker (such as we will play), the highest hand and the lowest hand split the pot. Poker hands are classified into one of the nine categories below; a hand of a higher category beats a hand of a lower category. Within each category, the values of cards in the hand determine the higher hand; for some categories, there may be ties. The suit of the cards (spades, diamonds, clubs, hearts) does not make a difference under these rules!

The categories of poker hands, from lowest to highest are:

1. *high card*: The hand consists of five unmatched cards. The higher of two “high cards” is determined by comparing the value of the remaining cards in each hand in decreasing order.
2. *one pair*: Two cards have the same value; the other three are unmatched. The higher of two “one pairs” is determined by comparing the value of the pairs. If identical, then the value of the unmatched cards from each hand are compared in decreasing order.
3. *two pair*: There are two pairs of cards which have the same value and one unmatched card. The higher of two “two pairs” is determined by comparing the value of the higher pair in each hand. If identical, the value of the other pair in each hand is compared, and if they are identical, then the value of the remaining card is compared.
4. *three of a kind*: Three cards have the same value; the other two are unmatched. The higher of two “threes of a kind” is determined by comparing the value of the three matched cards.
5. *straight*: The card values are consecutive (with no duplicates) with the exception that the ace may be used as the lowest card (i.e. i.e. the card values would be Ace, 2, 3, 4, 5). The higher of two “straights” is determined by comparing the value of the highest card in each hand.
6. *flush*: All five cards are of the same suit. The higher of two “flushes” is determined by the value of the highest card. If identical, then the value of the remaining cards are compared in decreasing order.
7. *full house*: Three cards have the same value, and the other two cards have the same value. The higher of two “full houses” is determined by comparing the value of the three matched cards.
8. *four of a kind*: Four cards have the same value; the remaining card is unmatched. The higher of two “fours of a kind” is determined by comparing the value of the four matched cards.
9. *straight flush*: The five cards are both a “straight” and a “flush.” The higher of two “straight flushes” is determined by the value of the highest card.

1.2 Simplified five-card high-low stud poker

We will play a simplified version of five card high-low stud poker with declarations. At the beginning of each hand, each active player is dealt one card face-down and four cards face-up. The face-down card is known as the *hole card*. A player can see all opponents' face-up cards, but no other player can see his hole card. There is a single betting interval. Before the showdown, each player must declare whether he is contending for the "high" or the "low" half of the pot. A player is only eligible for the half of the pot for which he declares. If all players declare for the same half of the pot, the other half is not awarded. Clams cannot be divided, so if the pot cannot be split evenly (or if a half of the pot cannot be split evenly between two or more tied players), the exact amount awarded to each player will be rounded to the nearest integer.

1.2.1 Raises, limits, and other details

The minimum raise is 5 clams; the maximum raise is 50 clams. So that players cannot be shut out of the betting in a hand for lack of clams, your balance may go negative. Each player's maximum total bet for a hand is limited only by the maximum number of clams of the active players. In other words, the active player with the most number of clams can bet them all on a hand; the other players can go into debt to match that bet, but no player may raise beyond that amount.

Your player will start each new round with the same number of clams as the other players (even if you went into debt in the previous round). However, any measure of performance of your player will take the cumulative amount of winnings and losses into account.

Each hand will be dealt from a complete deck. Note that there will not always be five hands dealt from the deck. The opportunity to bet first will cycle among the players at the table after some number (10?) of hands.

2 Representing poker in Scheme

We make the following definitions to represent cards and poker hands in Scheme.

- A *card* consists of a list of two elements:
 1. the *value* of the card: 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace
 2. the *suit* of the card: spades, hearts, clubs, diamonds
- Your *cards* consists of a list of five cards. The first card in this list is your hole card (the face-down card). The remaining four cards are your *public cards*, i.e. the cards that all other players can see.
- A *hand* is a list where:
 - the first element is one of the nine categories of hands: straight-flush, four-ofakind, full-house, flush, straight, three-ofakind, two-pairs, one-pair, and high-card
 - the remaining elements, if present, contain information needed to compare two hands of the same category

The following table shows the representations of the generic hand, and the lowest and highest hands for each category. Refer to comments in the support code for details of the values stored for each category (or infer from the category descriptions above).

generic	lowest such hand	highest such hand
(high-card)	(high-card 7 5 4 3 2)	(high-card A King Queen Jack 9)
(one-pair)	(one-pair 2 5 4 3)	(one-pair Ace King Queen Jack)
(two-pairs)	(two-pairs 3 2 4)	(two-pairs Ace King Queen)
(three-ofakind)	(three-ofakind 2)	(three-ofakind Ace)
(straight)	(straight 5)	(straight Ace)
(flush)	(flush 7 5 4 3 2)	(flush Ace King Queen Jack 9)
(full-house)	(full-house 2)	(full-house Ace)
(four-ofakind)	(four-ofakind 2)	(four-ofakind Ace)
(straight-flush)	(straight-flush 5 4 3 2 Ace)	(straight-flush Ace King Queen Jack 10)

3 Information available to your rule-based player

A great deal of information is available to your player, some of it through variables, the rest through functions. The following variables and functions are available for your rules. There is additional information available to user-defined functions that you might write which will be covered in a later section.

3.1 Variables

Many (but not all) of these variables will have changed every time your betting rules are called.

Variables pertaining to your hand

- `my/hand`: the evaluation of your cards
- `my/hi-hand`, `my/lo-hand`: the highest and lowest hand you can have, not considering your hole card but taking into account the all the face-up cards. This is what your opponents know about your hand (except that your opponents also know their respective hole cards).

Variables pertaining to the bet

- `call-amount`: the number of clams needed to call the bet
- `total-bet`: your total bet so far
- `previous-raise`: the amount of the most recent raise
- `pot-total`: the number of clams in the pot
- `bet-round`: the number of times (inclusive) you've bet (i.e. set to 1 when you are asked to make your first bet, etc.)
- `hand-number`: how many hands (inclusive) have been played in this round
- `min-raise`, `max-raise`: the minimum and maximum raise permitted

Variables pertaining to clams

- `my/clams`: how many clams your player currently has

Variables pertaining to who is playing

- `me`: your player's name
- `num-players`: the number of players that were dealt cards
- `num-active-players`: the number of players that are still active
- `other/active-players`, `all/active-players`: lists of the names of the active players. If a player becomes inactive, its name is removed from the list. The `all/` list includes your player; the `other/` list does not.
- `other/players`, `all/players`: lists of the names of the players who were dealt cards for the current hand. The `all/` list includes your player; the `other/` list does not.
- `other/seated-players`, `all/seated-players`: lists of the names of the other players seated at the table (regardless of their solvency). The `all/` list includes your player; the `other/` list does not.

3.2 Functions

- `(clams player-name)`: the number of clams the given player currently has
- `(lo-hand player-name)`, `(hi-hand player-name)`: the lowest/highest hand a player can possibly have given its public hand and the other cards you know about (your hole card and all public cards are taken into consideration)
- `(last-bet player-name)`: returns one of the symbols `pass`, `call`, or `raise`. If the given player hasn't bet yet, it returns `#f`.
- `(last-raise player-name)`: returns the amount of the last raise made by the given player. If the player made no raise, it returns 0. This function is most useful after ascertaining (with the `lastBet` function) that the player actually made a raise as its last bet.

4 Writing a rule-based learning agent

Your poker player will have three components:

1. betting rules
2. declaration rules
3. a learning function

Your betting rules will be run each time it is your player's turn to bet. After the bet has equalized, if your player is still active, your declaration rules will be run. After the showdown, your learning function will be called (regardless of whether your player was active at the end of the hand or not); it can attempt to learn from the history of the round.

4.1 Rule format

The rules govern the behavior of your player in the following way. Before any of your rules are called, certain variables and predicates will be defined to reflect the current state of the hand and the round. Your rules will be repeatedly evaluated in order until they produce a *terminal action*. For the betting rules, the terminal actions are passing, calling, and raising. For the declaration rules, the terminal action is a declaration of "low" or "high."

The basic format of a rule is:

```
( predicate . . . ==> action . . . )
```

The list of predicates are evaluated in order. If they all return `#t`, then all the actions are executed (in order). When this happens, we say that the rule has *fired*.

A rule may also use a quantifier. The formats for quantified rules are:

- `(forall var in set rule)`
- `(exists var in set rule)`

Both quantifiers attempt to execute the rule with the variable *var* bound to each value in the list *set* in turn. The `foreach` quantifier will attempt to execute the rule for all values in *set*. The `exists` quantifier will keep trying values until it finds a value for which the rule fires; it does not try the remaining values. You can use any list for *set*, however you will probably find these qualifiers most useful with the predefined variables `other/active-players` and `all/players`.

Quantified rules can be nested, but the variable substitution is done with a simple text substitution, so no nested quantifier can use the same variable!

4.1.1 Predicates

A predicate is a function that returns #t or #f. Here are the poker-specific predicates we will provide:

- (poker:> a b), (poker:>= a b), (poker:= a b), (poker:<= a b), (poker:< a b)

These functions compare two *hands* (mind the difference between *hands* and *cards*). When one or both of the *hands* is generic, the comparison is based only on the category of the hand. For example:

```
(poker:> '(full-house 7) '(full-house 5)) ==> #t
(poker:> '(full-house 7) '(full-house))    ==> #f
(poker:= '(full-house)   '(full-house 5)) ==> #t
```

- (active? player-name)

Returns #t if the given player is still active in this hand.

- (attribute ...)

You can define and test for your own attributes using the `assert` action. For example:

```
((forall x in other/active-players
  ((equal? (last-bet x) raise) (poker:< (hiHand x) myHand)
   => (assert Bluffing x)))
...
(exists z in other/active-players
  ((Bluffing z) => (Call))))
```

If you test for an attribute that has not been asserted, it will evaluate to #f.

In addition to the predicates above, you can use any Scheme predicate. The ones I'd expect to be commonly used are for comparing numbers: <, <=, =, >=, >, for comparing symbols: `equal?`, and the boolean operators: `and`, `or`, `not`. Remember that there is an implicit `and` over the list of predicates in a rule.

You may also write your own predicates. See the next section on functions available for doing so.

4.1.2 Actions

Your set of rules must ultimately produce some *terminal action*. For the betting rules, the terminal actions are:

- (call)
- (pass)
- (raise n)

where *n* is the number of clams of the raise, subject to the limits of `minRaise` and `maxRaise`.

For your declaration rules, the terminal actions are:

- (declare-high)
- (declare-low)

In addition, you can use any of the following actions:

- (assert *attribute*)
- (assert *attribute object* ...)
- (assert ~ *attribute*)
- (assert ~ *attribute object* ...)

The `assert` action defines a predicate (possibly with arguments which might represent a player name) which can be tested as a predicate in your rules. If the first argument to `assert` is the symbol `~`, any previous assertion is removed.

Assertions persist for the entire hand but are cleared at the start of a new hand.

- `(set var value)`

Sets the value of the variable *var*, creating it if necessary. *Variables you create persist for the entire round.* If you want them to be reset at the start of each hand, you will have to add a rule such as:

```
((= betRound 1) ==> (set myVar1 ()) (set myVar2 ()))
```

before any other use of the variable.

- `(call function arg ...)`

Calls a user-defined function. This function may make calls to the `assert` and `set` actions. See the next section on writing functions.

4.2 The Learning function

After the showdown, your learning function is called. This should be a function which does not take any arguments. You do not need to do learning here although you must demonstrate some learning component as part of this assignment. (See the Assignment activities section.) As a practical matter, there will be CPU time limits imposed on players, so you may only want to do learning periodically (say after every 10 hands). See the next section on what information is available to your functions.

5 Information available to your functions

The variables and functions described in a previous section are a subset of the information available to your player. By writing your own predicates, action functions, or learning function, you can access all this data and do arbitrary computations with it. You can see the individual cards in your hand and the public cards of your opponents. You can also access the entire betting history for the hand. You can also access any of this information for any previous hand.

The functions and variables previously described may be used in your functions just as you would use any other function or (global) variable. In addition, you have the following functions available. Note that the last `hand-number` argument to these functions is an optional argument. If not specified, `hand-number` defaults to the number of the current hand.

- `(my/cards . hand-number)`

Returns a list of your five cards; the first card is your hole card.

- `(initial-clams player-name . hand-number)`
`(final-clams player-name . hand-number)`

Returns the number of clams the given player had at the beginning/end of the hand. It is an error to call `final-clams` before the showdown on the current hand.

- `(public-cards player-name . hand-number)`

Returns a list of the four public cards for the given player.

- `(betting-history . hand-number)`

Returns a list containing the betting history for the current hand (or the given hand number if specified). The first element is the most recent bet; the last element is the first bet. Each element has one of the following forms:

```
(player-name pass)
(player-name call)
(player-name raise amount)
```

- `(hole-card player-name . hand-number)`

Returns the given player's hole card if known. A player's hole card is revealed only if it participates in the showdown for that hand. This function returns `()` if the hole card is not known.

Obviously this information is not available for the current hand until after the showdown. It is an error for this function to be called on the current hand from your player's betting or declaration rules. (It will not even return your own hole card.)

- `(declaration player-name . hand-number)`

Returns one of the symbols `low` or `high` if the player participated in the showdown and returns `()` otherwise. It is an error for this function to be called on the current hand from your player's betting or declaration rules.

- `(high-winners . hand-number)`
`(low-winners . hand-number)`

Returns a list of player names that won the high/low half of the pot. Note that the list may contain from 0 to 4 names.

The following functions were described in an earlier section but also take an optional second argument to specify a hand number.

- `(hi-hand player-name . hand-number)`
`(lo-hand player-name . hand-number)`

The following functions correspond to information available through variables for the current hand.

- `(get-my/hand . hand-number)`
- `(get-my/hi-hand . hand-number)`
`(get-my/lo-hand . hand-number)`

Any of your functions can call the `assert` and `set` actions. They cannot use the `pass`, `call`, `raise`, `declare-high`, or `declare-low` actions. Your functions can also maintain global state.

6 Support tools

Because there is a large number of variables and other pieces of information associated with a betting or declaration situation, it will be difficult to test your player in isolation. The advantage of a rule-based agent is that the rules should make it easier to understand the agent's behavior. Even so, the support tools we will provide are aimed at enabling you to see how your player handles certain situations and to recreate those situations for your player. Here are brief descriptions of the support tools for these purposes:

- `play-hand`, `play-round`

These functions will allow you to play your strategy against other player programs (and perhaps even allow you to play as one of the players). You will be able to have your program "think aloud" by showing which rules fire when your player is called upon to bet or declare. These functions will generate a full transcript for you to analyze afterwards. You will also be able to save the information for a hand (or match?) to recreate situations for testing your player.

- `replay-hand`

Loads in saved information for a hand and allows you to "fast forward" to a certain point in the hand to debug your player or to test a revised player. One shortcoming of this tool is that it cannot recreate the internal state of your player (if you use any global state).

To support learning, both from a learning function during play and for learning rules (offline) from a human player, we will provide some functions (or at least some examples) for extracting training data from a number of hands. In addition there are a few support functions, such as:

- `(deal n)` which deals `n` hands from a shuffled deck
- `(evaluate-cards c)` which returns a *hand* given a list of five cards

that you may find useful, so they will be released as part of the support code.

7 Details and logistics

The preliminary method for turning in your player is to define the following list in your file:

```
(define poker-player (list 'your-players-name
                           "string with your real names"
                           betting-rules
                           declaration-rules
                           learning-function))
```

We will provide a sample player along with the support code.

You will turn in your poker player to the web page for this assignment. It will at least verify that your poker player works, and we will use it to distribute data on the preliminary and final rounds.

There will be some reasonable CPU time limit for your player so that we can actually run all the rounds of poker before the semester is over! This should be high enough for your player to do some amount of learning during a match. We should be able to provide your functions the elapsed CPU time so that they can determine if they have time to do more learning.

8 Assignment activities

The activities for this assignment include the following:

- Turn in an original working player by Friday November 10.
We will run a few rounds and post all the results and data to the course web pages.
- Implement some user-defined predicate, action, or learning function(s).
The basic functions we have provided are just that: basic, so I will require that you implement some user-defined functionality.
- Implement some learning component.
You can either have your player attempt to learn other players' behavior during a round, or you can attempt to learn rules for your player based on human play. You can use decision tree learning, or some other learning technique that we will cover.
- Turn in a final player and a written report by Friday November 17.
Specific guidelines for the written report will be issued at a later date. Generally speaking, this report should document and explain your player's strategy, describe your user-defined predicate, action, or learning functions, describe your learning component, and give some analysis of your player's behavior with specific examples.

Your grade for this assignment will be tentatively determined as follows:

- 15 points for turning in an original working player by Friday November 10
- 40 points based on the performance of your final player
- 30 points based on evaluation of your learning component through your written report and code
- 50 points based on evaluation of your strategy and implementation through your written report and code

Except in extenuating circumstances, both students working on a player will receive the same grade for this assignment.