CSCI 4150 Introduction to Artificial Intelligence, Fall 2000
Assignment 6 Support code release notes

1 Overview

There are three files of Scheme code:

- `a6code.com` which contains the play-hand and play-round function and other stuff to administer the game. The replay-hand function is not available yet.
- `infofunc.scm` which contains a few functions that provide information to your poker player. You can look at these for examples of functions that access hand- and card-information data structures.
- `poker.scm` which contains support code for cards, dealing, and evaluating hands

There are also several sample poker players that you can look at. These were really written more for testing the support code than as serious poker players. They illustrate the syntax and some of the features of this rule-based implementation. The files are:

- `player1.scm` provides the player Ace
- `player5.scm` provides the player Sharky
- `human.scm` provides the human interface so you can play against your programs

More sample players will be available soon.

There are about 2200 lines (so far) in the source code files for this support code, which has all been written in the past week and a half. So, although I hate to admit it, there are probably some bugs that I haven't found yet. If you find a problem, please let me know. In the meantime, I'll continue my own testing.

1.1 Getting started

Download all the files into the same directory and try playing against one or both of the sample players I am providing:

```
(load "a6code")
(play-hand "human" "player1")
```

You will only ever need to load the `a6code.com` file — it will load the other two source code files. The `play-hand` (or `play-round` function will load your player files.

The defaults are set to print out a lot of information to the screen. You can learn how to change this later in the document.

2 Important changes and clarifications

- There is now an ante of 5 clams, i.e. all players will put 5 clams into the pot before the cards are dealt. This is to encourage players that try to win rather than decide when to participate.
- If you want to call a user defined action, you do not use the "call" keyword now. This is because that name is used by one of the terminal betting actions.
- There is only a `forall` rule quantifier (not an `exists`), and there is additionally a `forall` and a `exists` predicate. You can think of the rule quantifier as a looping construct; it will try to fire the subrule once for every value in the given set.

  The `forall` and `exists` predicates are (more properly) a conjunction and disjunction (respectively) over all values in the set. The syntax for these quantifiers is as follows:

```
(forall <var> in <list> predicate ...)
(exists <var> in <list> predicate ...)
```

Note that the entire quantifier serves as a predicate in a rule

- In addition to declaring high or low, you can also declare high/low. If you declare high/low, you must win or tie for both the high and low halves of the pot. If not, you don't win anything. This would be particularly useful if you find yourself the only active player at the showdown. The terminal action for making this declaration is `(declare-high/low)`.
- The `last-raise` function can be called with 0 arguments (in which case it returns the amount of the last raise) or with a single player-name as the argument (in which case it returns the amount of the last raise made by that player). The original handout only specified that a player-name had to be given.
- The `previous-raise` variable is not provided. I don't think this information is important, but if it is, you can get it using the `last-raise` function with 0 arguments.
- The following parameters have been set in the source-code. I have set them to what I think are reasonable values, but I will reserve final judgement until after you have turned in your preliminary players and we can play some rounds with them. The parameters are: ante = 5, min-raise = 5, max-raise = 50, starting-clams = 1000, round-hands = 200, dealer-switch = 10.
- So that you can play multiple "copies" of your player against each other without creating separate player files, player names will be changed to make them unique. The first instance of a player name will have "`-1`" appended to it, with other instances numbered sequentially. This means that you code should not assume that you know your player's name! You can get your players name through the `me` variable.
- Your poker player should be contained in a single file! Because I am loading each player into its own environment, if you `load` code from your player file, it probably won't work properly. You'll have to put your code in a single file anyway to submit it.

3 Running a game of poker

This section describes some details about running a game of poker with `play-hand` and `play-round`. Some of these details might not make sense until you read the rest of the document.

The `play-hand` and `play-round` functions take two to five filenames. The same file may be given more than once, and multiple copies of that player will be created. As described in the previous section, players will be renamed so that they all have unique names.

Although for the official rounds of poker, players will be seated randomly, these functions seat players in the order that you specify the filenames. The player in the first file gets to bet first.

The files are loaded in the order given. The one implication of this is that if you use any of the functions to control what output is printed to the screen (see Section 5.1), the last call to one of these functions will have the final say on what gets printed!

Because files are loaded into their own environment, any global varibles you define in your file are only accessible to you functions (and not to other players). "Private" information about your cards and such is made available to your function by setting global variables in your player's environment.

You may not change the value of any variable I provide! Doing so could possibly mess up running the game.

3.1 Rule interpretation

Any arguments to actions and predicates are evaluated! Therefore, you must quote any argument that you don't want evaluated. For example,

```
((exists x in other/active-players
         (or (poker:> (lo-hand x) one-pair)
             (equal? x 'Ace)))
 ==> (pass))
```

This rule illustrates several points about quoting. The symbol `Ace` must be quoted so that it is not evaluated. In specifying hand "constants" for comparing hands, you would normally have to write

```
(poker:> (lo-hand x) '(one-pair))
```

but I have made the following definition in the `poker.scm` file:

```
(define one-pair '(one-pair))
```

Similar definitions exist for the other hands. Of the other symbols that are not quoted, the keyword "`in`" and the variable `x` are exceptions to the rule about quoting; the rest are either defined functions or variables.

3.2 Other rule stuff

I suggest you write your rules so that they can produce a terminal action in a single pass. You should ensure that they will produce a terminal action.

The interpreter will make multiple passes over your rules until it reaches a terminal action. If there is any pass where no rule fires, then the interpreter will stop, assuming that there is an error. However, it is more likely that you have a rule that will always fire each pass through your rules, so it's quite possible you'll end up in an infinite loop.

To address this problem, I've included the variable `rule-iteration-limit` which will stop interpretation of your rules once it reaches that many iterations. You can change the value of this variable by redefining it. It will initially be set to 10. Let me know if your rules require a higher limit so we can set in appropriately for running rounds of poker.

4 Hand information

4.1 Public hand information

Quite a bit of information about the current hand is stored in a large list data structure which is available to you as the global variable `public/hand-information`. The contents of this data structure are available through a number of accessor functions:

- `(puhi/seated-players public/hand-information)`

  Returns a list of names of players that are seated at the table.

- `(puhi/seated-players-clams public/hand-information)`

  Returns a list of clams of players that are seated at the table (in the same order as `puhi/seated-players`.

- `(puhi/public-card-information public/hand-information)`

  Returns a list where each element is a "public card information" (abbreviated "puci"). Each element of the list has the following form:

  ```
  (player-name public-cards hi-hand lo-hand)
  ```

  This list can be used as a association list. The `public-cards` function and the `my/lo-hand` and `my/hi-hand` variables use information from this data structure through the accessors:

  - `(puci/public-cards public/hand-information)`
  - `(puci/hi-hand public/hand-information)`
  - `(puci/lo-hand public/hand-information)`

- `(puhi/betting-history public/hand-information)`

  Returns a list of the betting history to date. Each element is a list of the form: `(player-name pass)`, `(player-name call)`, or `(player-name raise N)` where `N` is an integer. The first element of the list is the most recent bet.

- `(puhi/bets public/hand-information)`

  Returns a list of the total bet each player made. Each element corresponds to the player in the list returned by `puhi/seated-players`.

  `(puhi/showdown-information public/hand-information)`

  Returns a list of the following form:

  ```
  ((player-name cards hand declaration winnings) ...)
  ```

  Only players that participated in the showdown are listed here. The `cards` element is a list of all five of that player's cards (the first card was the hole card). The declaration may be any of the symbols `high`, `low`, and `high/low`. If the player didn't win anything, `winnings` will be 0.

4.2 Private card information

There is also a data structure available to your player (and not shared with any other player) which has information on your hand and on your opponents (public) cards. This is stored in the global variable `private/card-information`. It has the form:

```
(your-player-name cards hand ((other-player-name hi-hand lo-hand)
                              (other-player-name hi-hand lo-hand)
                              ...))
```

This data structure is used to provide information to the `hi-card` and `lo-card` functions. The information in this structure can be accessed with the following functions.

- `(prci/cards private/hand-information)`
- `(prci/hand private/hand-information)`
- `(prci/other-player-list public/hand-information)`

  This list can be used as an association list to search for a player's entry by name. The following accessors operate on one of the elements of this list:

  - `(opl/hi-hand public/hand-information)`
  - `(opl/lo-hand public/hand-information)`

Here is the source code for some of the functions provided to you that accesses this data structure:

```
(define (my/cards)
  (second private/card-information))

(define (lo-hand player-name)
  (let ((other-player-info
         (assoc player-name (prci/other-player-list
                             private/card-information))))
    (if (null? other-player-info)
        (error "Invalid player name given to function lo-hand" player-name)
        (opl/lo-hand other-player-info))))

(define (hi-hand player-name)
  (let ((other-player-info
         (assoc player-name (prci/other-player-list
```

```
                              private/card-information))))
    (if (null? other-player-info)
        (error "Invalid player name given to function hi-hand" player-name)
        (opl/hi-hand other-player-info))))
```

5 Printing to the screen and to files

In developing your poker player, you will probably want to run, test, and debug it at several levels, so I've tried to design a flexible system for controlling what output is printed to the screen. You can also have output directed to a file.

5.1 Hand information

For now, the main control you'll probably want to have is whether the hole cards are printed to the screen or not. If you're playing against the computer, you then have the option of cheating! Anyway, here are three functions that you can use to control this:

- `(print-everything)`
- `(print-only-public)`
- `(set-printing . args)`

These functions can be executed after you have loaded the `a6code` file, or they can be put in your players file (see the `human.scm` player for an example). The first two cause basically everything to be printed. The last function lets you specify exactly what you want or don't want. Here are your choices:

- `table-info`: who is seated at the table
- `all-cards`: shows all the cards that are dealt (including the hole cards)
- `public-cards`: shows only the face-up cards that are dealt
- `betting`: shows the detailed betting sequence
- `declarations`: prints some messages while it gets declarations from the players
- `showdown`: prints the results of the showdown
- `learning`: tells you whose learning function is being called
- `debug-info`: prints out messages that I was using in debugging

5.2 Rule execution information

Independent of the above, you can control what information is printed about what rules of your player are firing. To control this, you must define the variable `*print-to-screen*` in your player's file. When the `play-hand` procedure loads your file, it will look for this variable. Its value should be a list with any combination of the following symbols as elements

- `state-info`: prints out all the standard variables and attributes available to your player
- `predicate-matching`: prints details of testing predicates
- `rule-firings`: prints which rules fire
- `action-execution`: prints details of what actions are executed

5.3 Printing to files

You can also have any of this information printed out to a file for later review. See the `player5.scm` file for an example. You need to define the variable `*print-to-file*`.