Five-Card Draw

Jack Scott

June 2019

1 Features

For my project I created a worksheet for a Five-Card Draw video poker game. The program allows the user to adjust their bet by increments of 1 or 10 before dealing the initial hand of five cards, one at a time, all of which are displayed using actual card images. Then the user selects and deselects which cards to "hold" by clicking on them, before trading the remaining cards for new ones. Then, the user receives a payout proportional to their bet according to what kind of poker hand they have (two pair, full house, etc.), and the cards are reshuffled and the process repeated. I used the most common scoring method that most video poker games do, Jacks or Better. The payouts for each hand are displayed on the bottom right, and they change to reflect changes in the user's bet, as well as light up according to which hand the user ended up with. The buttons dynamically become enabled and disabled as the state of the game changes, such as not allowing the user to lower their bet below 1, deal while in the holding stage, or bet more than their current balance.

2 Some Implementation Decisions

- Initially, I chose to represent each card as its own constant, but I quickly realized that a constructor `card(value, suit)` was a much more useful representation, since I only use one 52 card deck and thus every card can be defined exactly as its value-suit pair. Instead of needing data like `suit(2_clubs, clubs)` or `value(7_hearts, 7)`, the value and suit of cards are immediately available. This was most useful in defining the hand types, allowing for very concise and elegant rules like `hand_type(card(V0, _), card(V0, _), card(V0, _), card(V1, _), card(V1, _), "Full House")`.
- The only relevant information about each card is its current location in the game: it can either be in the deck, hand, or discarded after not being held. I began by using predicates like `in_deck(card(3, hearts))`, `in_hand(card(12, spades))`, which was motivated by intuitions from iterative programming where I would put the cards into different data structures corresponding to different locations. However, I decided to use

a binary predicate `loc` instead, e.g. `loc(card(3, hearts), deck)`. This allowed for a more elegeant approach to a couple parts of the game, such as reshuffling the cards: instead of negating the facts about each location and adding new ones for the deck, I could simply change the location of every card to the deck in one rule.

- I decided to use a decent number of static CSS rules to position elements along with many dynamic CSS rules using the `worksheets.js` interface. The look and feel of any casino game is as important as the game itself, so I wanted to make sure the interface was clean and fun to use, hence the dramatic one-at-a-time dealing and the way the balanced goes up when scoring.

3 Optimizations

The only real performance issue that came up was calculating the payout for the user. My approach test each of the 5! = 120 possible permutations for the user's hand against each different hand type, like `hand_type(card(V, _), card(V, _), card(V, _), _, _, "Three of a Kind")`. While there surely are more efficient ways of finding the best hand, this allowed for a particularly clean way of defining each hand type, and I think it exemplifies the kind of elegance logic programming is often conducive of. I made sure to only do this calculation once per round, materializing the rule as a fact instead of recalculating as the value was needed so that the user does not experience any delays.

4 Logic Programming

Clearly, logic programming is not necessary to implement video poker: it's been done countless times using iterative languages. There are many places in the code where rules are effectively just changing variables, e.g. `==> ~bet(Bet) & bet(Bet_) & ~balance(Bal) & balance(Bal_)`, which, while not necessarily bad, is less elegant and much more bug prone than the usual iterative approach. Additionally, video poker is at its core a very sequential experience, so handling the different state transitions was a bit tricky at some points. However, I do think that logic programming had many strengths here. As mentioned previously, calculating the best hand was very easy. Enabling and disabling the buttons according to different rules was also perfectly suited to epilog. Styling the various HTML elements was helped greatly, too, as it allowed for a much cleaner approach than my previous experiences with javascript styling. Overall, I think logic programming was a good fit for the problem, as I rarely had to worry about the underlying system, which is usually the strongest indicator that logic programming is not well-suited to the problem. I also ended up with much fewer lines of code compared to what I would have with an iterative approach.