CS 61A Spring, 2010

Programming Project 1: Twenty-One
(Scratch version)

For our purposes, the rules of twenty-one ("blackjack") are as follows. There are two players: the "customer" and the "dealer". The object of the game is to be dealt a set of cards that comes as close to 21 as possible without going over 21 ("busting"). A card is represented as a word, such as `10s` for the ten of spades. (Ace, jack, queen, and king are `a`, `j`, `q`, and `k`.) Picture cards are worth 10 points; an ace is worth either 1 or 11 at the player's option. We reshuffle the deck after each round, so strategies based on remembering which cards were dealt earlier are not possible. Each player is dealt two cards, with one of the dealer's cards face up. The dealer always takes another card ("hits") if he has 16 or less, and always stops ("stands") with 17 or more. The customer can play however s/he chooses, but must play before the dealer. If the customer exceeds 21, s/he immediately loses (and the dealer doesn't bother to take any cards). In case of a tie, neither player wins. (These rules are simplified from real life. There is no "doubling down," no "splitting," etc.)

The customer's *strategy* of when to take another card is represented as a predicate function (the hexagonal-block kind that reports true or false). The function has two arguments: the customer's hand so far, and the dealer's card that is face up. The customer's hand is represented as a list in which each element is a card in the form of a word like `3H` for the three of hearts, or `QC` for the queen of clubs. the dealer's face-up card is a single word (not a list). The strategy function should report a true or false output, which tells whether or not the customer wants another card.

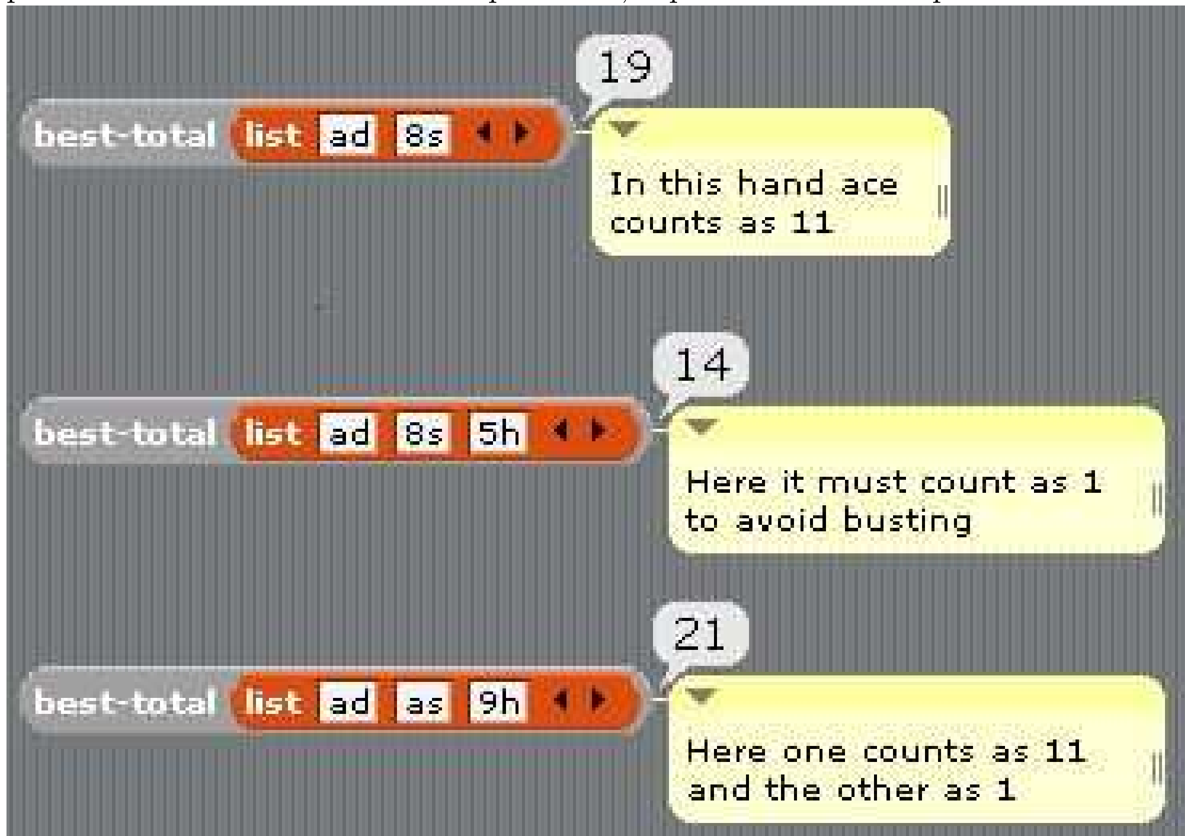
We provide a library of functions to help in dealing with words (in the Operations menu) and lists (in the Variables menu).

Download `http://byob.berkeley.edu/21.ypr` which contains definitions for several blocks needed in the project, most notably the `Dealer` sprite's `game` block, which takes a strategy function as its argument. Calling `game` plays a game using the given strategy and a randomly shuffled deck, and reports 1, 0, or −1 according to whether the customer won, tied, or lost.

Don't invent global variables. Each sprite has its own `hand` variable; make sure you are running each block in the appropriate sprite! Use block variables rather than globals for temporary storage.

1. The program in the library is incomplete. It has a "stub" procedure `best-total` that always reports 0. It is supposed to take a hand (a list of card words) as argument, and report the total number of points in the hand. It's called *best*-total because if a hand contains aces, it may have several different totals. The procedure should report the largest

possible total that's less than or equal to 21, if possible. For example:



Finish writing `best-total`.

2. Define a strategy procedure `stop-at-17` that's identical to the dealer's, i.e., takes a card if and only if the total so far is less than 17. (Note: It's okay to use `best-total` to calculate the user's point count, even though a serious blackjack player might use a more complicated rule to decide on the "best" value for a hand.)

3. Write a procedure `play-n` such that

plays `n` games with a given strategy and reports the number of games that the customer won minus the number that s/he lost. Use this to exercise your strategy from problem 2, as well as strategies from the problems below.

Don't forget: a "strategy" is a procedure! We're asking you to write a procedure that takes another procedure as an argument. This comment is also relevant to parts 6 and 7 below.

4. Define a strategy named `dealer-sensitive` that "hits" (takes a card) if (and only if) the dealer has an ace, 7, 8, 9, 10, or picture card showing, and the customer has less than 17, or the dealer has a 2, 3, 4, 5, or 6 showing, and the customer has less than 12. (The idea is that in the second case, the dealer is much more likely to "bust" (go over 21), since there are more 10-pointers than anything else.)

5. Generalize part 2 above by defining a function `stop-at`.

should *report a strategy* that keeps hitting until a hand's total is `n` or more. For example,

is equivalent to the strategy in part 2.

6. On Valentine's Day, your local casino has a special deal: If you win a round of 21 with a heart in your hand, they pay double. You decide that if you have a heart in your hand, you should play more aggressively than usual. Write a `valentine` strategy that stops at 17 unless you have a heart in your hand, in which case it stops at 19.

7. Generalize part 6 above by defining a function `suit-strategy` that takes three arguments: a suit (`h`, `s`, `d`, or `c`), a strategy to be used if your hand *doesn't* include that suit, and a strategy to be used if your hand *does* include that suit. It should report a strategy that behaves accordingly. Show how you could use this function and the `stop-at` function from part 5 to redefine the `valentine` strategy of part 6.

8. Define a function `majority` that takes three strategies as arguments and produces a strategy as a result, such that the result strategy always decides whether or not to "hit" by consulting the three argument strategies, and going with the majority. That is, the result strategy should report `true` if and only if at least two of the three argument strategies do. Using the three strategies from parts 2, 4, and 6 as argument strategies, play a few games using the "majority strategy" formed from these three.

9. Some people just can't resist taking one more card. Write a procedure `reckless` that takes a strategy as its argument and reports another strategy. This new strategy should take one more card than the original would. (In other words, the new strategy should stand if the old strategy would stand on the `butlast` of the customer's hand.)

10. That guy at the other end of the table isn't doing anything. Write a strategy for him that *asks the user* whether to hit or stand, after displaying the player's hand and the dealer's up card. Modify the `game` procedure so that both players play, one with a programmed strategy and the other with this interactive strategy. Keep scores for both players.