```
;; authors: Richard Banister, Reeti Kumar, Karl Shouler
;; for CSC4500 Artificial Intelligence, Villanova University, Spring 2008

(defparameter *second-draft-agent*
  (make-holdemagent
   :namestring "BILLY" ;works by utility
   :ID 3 ;; change to some number in [1..4] if you're using this in a game
   :agentfunction #'(lambda (roundstate id)
                      (let ((round nil) (numplayers 1))

                        (setf round (currentround roundstate)) ;;what round are we in?

                        (format *debug* "We're in ~A~%" round)

                        (cond ((eq round *round*) ;; round unchanged, just determine-action
                               ;(determine-action *strength* roundstate id))
                               (determine-utility #'utility2 *strength*
                                                  (aref (holdemround-playerbanks roundstate) id)
                                                  (holdemround-pot roundstate)
                                                  (actionable-players roundstate)
                                                  (zerop (holdemround-bet roundstate))
                                                  (number-of-raises roundstate)
                                                  (holdemround-blind roundstate)
                                                  5)
                               (format *debug* "Best Action: ~A~%" *best-action*)
                               *best-action*)
                              ((eq round :cleanup) ;; hand over, do nothing
                               (format t "Billy Mad Bank: ~D~%" (aref (holdemround-playerbanks roundstate) id))
                               (setf *round* nil)
                               (setf *strength* nil))

                              (t ;; round has changed, reformulate strength and det action
                                 (setf *round* round)

                                 ;; (holdemround-playercards)
                                 ;; (declare (ignore roundstate id))

                                 (setf numplayers (- (actionable-players roundstate) 1))
                                 (if (zerop numplayers) (setf numplayers 1))

                                 ;;(let ((strength 0) (action "fold") (raiseamount 0))
                                 (setf *strength* (hse  (aref (holdemround-playercards roundstate) id)
                                                        (holdemround-commoncards roundstate)
                                                        numplayers))

                                 ;; now determine action
                                 ;(determine-action *strength* roundstate id))
                                 (determine-utility #'utility2 *strength*
                                                    (aref (holdemround-playerbanks roundstate) id)
                                                    (holdemround-pot roundstate)
                                                    (actionable-players roundstate)
                                                    (zerop (holdemround-bet roundstate))
                                                    (number-of-raises roundstate)
                                                    (holdemround-blind roundstate)
                                                    5)
                                 (format *debug* "Best Action: ~A~%" *best-action*)
                                 *best-action*)
                              );end cond

                        )))) ;;end agent BILLY


;; chance-of-win should be from -0.5 to 0.5
;; commitment is amount of money put into this hand
;; pot is amount we stand to win
(defun determine-utility (utility-function chance-of-win bank pot number-of-players check-allowed numraises current-bet depth)
  (cond
   ((zerop depth)
    ;; return the base estimation of utility
    (funcall utility-function chance-of-win pot bank))

   (t
    (let ((foldu 0)(checku 0)(callu 0)(raiseu 0)(allinu 0) ;;expected utility of actions
          (new-depth (- depth 1)) (raise-amount 0)
          (chance-of-losing (- 1 chance-of-win))
          (expected-utility 0))

      (setf *best-action* (list :fold))

      ;; EU for each action is sum of (probabilities of results of actions * utilities of resulting situations)
      ;; Note: this is real simple, and not all possible actions are accounted for. Is that a problem with
      ;; my algorithm, or just a matter of trimming the possibilities?
      ;;
      ;; possible actions:
      ;; :fold
      ;; fold is definite: (P = 1)
      ;; current commitment stays the same
      ;; chance-of-win = 0

      (setf foldu (funcall utility-function chance-of-win pot bank))

      ;; :check
      ;; possibilities:
      ;; 1. will sometimes definitely be coerced into a fold:
      ;; current commitment stays the same
      ;; chance-of-win = 0

      ;; 2. someone else opens betting
      ;; current commitment stays the same
      ;; 3. no one bets
      ;; current commitment stays the same

      (cond (check-allowed
             ;; then a check is possible
             ;; chances are someone will open betting
             (setf checku (+
                           (* .7 (determine-utility utility-function
                                                    chance-of-win bank (+ pot 700) number-of-players
                                                    nil 1 current-bet new-depth))
                           (* .3 (determine-utility utility-function
                                                    chance-of-win bank pot number-of-players
                                                    check-allowed numraises current-bet new-depth))))
                           ; wow, that was some arbitrary stuff
             )
            (t
             ;; coerce to fold
             (setf checku foldu)))

      ;; :allin
      ;; possible results of allin:
      ;; 1. everyone else allins (P = 1/numplayers+1)
      ;; commitment = entire bank
      ;; pot += everyone's bank
      ;; chance of winning stays the same

      ;; 2. one to n-1 people fold (P = numplayers-1/numplayers+1)
      ;; commitment = entire bank
      ;; pot += (numplayers - numfolds)x raiseamount
      ;; chance of winning increases by numfolds * (1 - chance-of-winning)/numplayers)
      ;; 3. n people fold (P = 1/numplayers+1)
      ;; commitment = entire bank
      ;; pot += allin
      ;; chance of winning = 1

      (setf allinu (+
                    (* .5 (funcall utility-function chance-of-win (+ pot (* number-of-players bank)) 1)) ;everyone all in
                    (* .1 (determine-utility utility-function ;everyone folds
                                             1 1 (+ pot bank)
                                             0 nil (+ numraises 1) current-bet 0))
                    (* .4 (determine-utility utility-function ;at least one player folds ;;TO DO better
                                             (+ chance-of-win (* 2 (/ chance-of-losing number-of-players))) ; just guessing here
                                             1 (+ pot (* (/ number-of-players 2) bank)); and here
                                             (/ number-of-players 2) nil (+ numraises 1) current-bet 0))))

      ;; :call
      ;; possibilities
      ;; 1. everyone else calls or raises
      ;; commitment += call amount
      ;; pot += 4x call amount
      ;; chance-of-win stays same

      ;; 2. one to n-1 people fold (P = numplayers-1/numplayers+1)
      ;; commitment += call amount
      ;; pot += (numplayers - numfolds)x call
      ;; chance of winning increases by numfolds * (1 - chance-of-winning)/numplayers)

      ;; 3. n people fold (P = 1/numplayers+1)
      ;; commitment += call amount
      ;; pot += call amount
      ;; chance of winning = 1
      (cond ((< bank current-bet) ; can't make the bet
             (setf callu allinu))
            (t
             (setf callu (+
                          (* .7 (determine-utility utility-function ;everyone calls
                                                   chance-of-win (- bank current-bet) (+ pot (* number-of-players current-bet))
                                                   number-of-players check-allowed numraises current-bet new-depth))
                          (* .1 (determine-utility utility-function ;everyone folds
                                                   1 (- bank current-bet) (+ pot current-bet)
                                                   0 check-allowed numraises current-bet 0))
                          (* .2 (determine-utility utility-function ;at least one player folds ;;TO DO better
                                                   (+ chance-of-win (* 2 (/ chance-of-losing number-of-players))) ; just guessing here
                                                   (- bank current-bet) (+ pot (* (/ number-of-players 2) current-bet)); and here
                                                   (/ number-of-players 2) check-allowed numraises current-bet new-depth))
                          ))))


      ;; :raise
      ;; possible results of raising:
      ;; 1. everyone else calls (P = 1/numplayers+1)
      ;; commitment += raise amount
      ;; pot += 4x raise amount
      ;; chance of winning stays the same

      ;; 2. one to n-1 people fold (P = numplayers-1/numplayers+1)
      ;; commitment += raise amount
      ;; pot += (numplayers - numfolds)x raiseamount
      ;; chance of winning increases by numfolds * (1 - chance-of-winning)/numplayers)

      ;; 3. n people fold (P = 1/numplayers+1)
      ;; commitment += raise amount
      ;; pot += raise amount
      ;; chance of winning = 1

      (setf raise-amount (+ current-bet (* current-bet (/ (+ (random 50) 10) 10))))

      (cond ((= numraises 3)
             ;; that's a fold
             (setf raiseu foldu))
            ((< bank raise-amount) ; can't make the raise
             ;; we would have to allin
             (setf raiseu allinu))
            (t
             (setf raiseu (+ ;complete with totally faked probabilities
                           (* .6 (determine-utility utility-function ;everyone calls
                                                    chance-of-win (- bank raise-amount) (+ pot (* number-of-players raise-amount))
                                                    number-of-players nil (+ numraises 1) current-bet new-depth))
                           (* .1 (determine-utility utility-function ;everyone folds
                                                    1 (- bank raise-amount) (+ pot raise-amount)
                                                    0 nil (+ numraises 1) current-bet 0))
                           (* .3 (determine-utility utility-function ;at least one player folds ;;TO DO better
                                                    (+ chance-of-win (* 2 (/ chance-of-losing number-of-players))) ; just guessing here
                                                    (- bank raise-amount) (+ pot (* (/ number-of-players 2) raise-amount)); and here
                                                    (/ number-of-players 2) nil (+ numraises 1) current-bet new-depth))
```

```
                   ))))

      ;(format t ":fold utility: ~D~%" foldu)
      ;(format t ":check utility: ~D~%" checku)
      ;(format t ":allin utility: ~D~%" allinu)
      ;(format t ":call utility: ~D~%" callu)
      ;(format t ":raise utility: ~D~%" raiseu)

      (setf expected-utility (extremum (list foldu checku allinu callu raiseu)))
      ;(format t "Best expected utility: ~D~%" expected-utility)

      (cond ((= foldu expected-utility)
             (setf *best-action* (list :fold)))
            ((= checku expected-utility)
             (setf *best-action* (list :check)))
            ((= callu expected-utility)
             (setf *best-action* (list :call)))
            ((= raiseu expected-utility)
             (setf *best-action* (list :raise raise-amount)))
            ((= allinu expected-utility)
             (setf *best-action* (list :allin))))

      (format *debug* "Best Action: ~F~%" *best-action*)

      (incf *count*)

      expected-utility
      )))
)

(defun utility1 (chance-of-win pot bank)
  ;;
  (* chance-of-win (log (+ pot bank))))


;; this is a little messed up
(defun utility2 (chance-of-win pot bank)
  ;; high cow ^ high gain > high cow ^ low gain > low cow ^ low gain > low cow ^ high gain
  ;;
  (let ((gain 1))
    (if (not (zerop bank)) (setf gain (/ pot bank)))
    (* chance-of-win (abs (log gain)))))
```

```
(defun utility3 (chance-of-win pot bank)
  ;; high cow ^ high gain > high cow ^ low gain > low cow ^ low gain > low cow ^ high gain
  ;;
  (let ((chance-of-lose (- 1 chance-of-win)) (commitment (/ 1 pot)) (gain (/ pot bank)))
    (- (* chance-of-win (abs (log gain))) (* chance-of-lose commitment) )))

(defun utility4 (chance-of-win pot bank)
  (let ((chance (- chance-of-win .5)))
    (format t "~F~%" (* chance pot))
    ))
```