**DATA S1**

**Computational Model**

;; UTILITIES  
(define (count-satisfying x l)  
  (length (filter (lambda (r) (equal? r x)) l)))  
  
(define (percentage-satisfying x l)  
  (/ (count-satisfying x l) (length l)))  
  
(define (aggregate-samples lst)  
  (list '(left right) (list (percentage-satisfying 'left lst) (percentage-satisfying 'right lst))))  
  
;; BEGIN MODEL SPECIFICATION  
  
(define duck-box-num-ducks 45)  
(define duck-box-num-balls 15)  
  
(define ball-box-num-ducks 15)  
(define ball-box-num-balls 45)  
  
(define (make-number-container dd db bd bb)  
  (lambda (box object)  
    (case box  
      (('duck) (case object  
                 (('duck) dd)  
                 (('ball) db)))  
      (('ball) (case object  
                 (('duck) bd)  
                 (('ball) bb))))))  
  
(define nums  
  (make-number-container  
   duck-box-num-ducks  
   duck-box-num-balls  
   ball-box-num-ducks  
   ball-box-num-balls))  
  
(define (random-sample-from-box box)  
  (if (flip (/ (nums box 'duck) (+ (nums box 'duck) (nums box 'ball))))  
      'duck  
      'ball))  
  
(define (sample-from-box box sampling-manner likes)  
  (case sampling-manner  
    (('random)  
     (random-sample-from-box box)  
     )  
    (('selective)  
     likes)))  
  
(define (sample-n-from-box box sampling-manner likes n)  
  (let ((f (lambda () (sample-from-box box sampling-manner likes))))  
    (repeat n f)))  
  
  
(define (make-boxes l r)  
  '((left ,l) (right ,r)))  
  
(define (left boxes)  
  (assoc 'left boxes))  
  
(define (right boxes)  
  (assoc 'right boxes))  
  
(define (box-majority-object box) (last box))  
(define (left-box-primary-object boxes) (last (left boxes)))  
(define (right-box-primary-object boxes) (last (right boxes)))  
  
(define (which-box object boxes)  
  (cond  
   ((equal? (left-box-primary-object boxes) object) (left boxes))  
   ((equal? (right-box-primary-object boxes) object) (right boxes))))  
  
(define (which-side box) (first box))  
  
(define (where-is ob boxes)  
  (which-side (which-box ob boxes)))  
  
(define (box-on-side left-or-right boxes)  
  (case left-or-right  
    (('left) (left boxes))  
    (('right) (right boxes))))  
  
(define (switch-boxes boxes)  
  (make-boxes (right-box-primary-object boxes) (left-box-primary-object boxes)))  
  
(define (other-box side)  
  (if (equal? side 'left) 'right 'left))  
  
;; model of the child and the experiment (external to the frog)  
(define (child-sample)  
  
  (rejection-query  
   (define pre-boxes (make-boxes 'duck 'ball)) ;; arrangement of boxes before  
   ;; frog leaves (observed by frog)  
  
   (define actual-switch? true) ;; whether the boxes are switched or kept in place  
  
   (define num-draws 3)  ;; how many objects are sampled (3 or 5)  
   (define sampling-manner 'random) ;; whether sampling is random or selective  
   (define experimenter-preference 'duck) ;; experimenter's target during selective sampling  
  
   (define post-boxes (if actual-switch? (switch-boxes pre-boxes) pre-boxes)) ;; box locations when frog returns (unknown to frog)  
   (define which-box-drawn-from (where-is 'duck post-boxes)) ;; which box is being sampled from  
  
   (define observed-sample (repeat num-draws (lambda () 'duck))) ;; the set of objects sampled (observed by frog)  
  
   ;; child's model of the frog  
  
   ;; BEGIN MODEL-SPECIFIC CODE – set which-model to 'a-'f to enable each alternative model  
   (define which-model 'a)  
   (define (frog-sample)  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       ;; (a) full model  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
     (case which-model  
       (('a)  
        (rejection-query  
         (define frog-switch-prior .159) ;; how much frog expects the 'switch' trick  
         (define frog-thinks-switch? (flip frog-switch-prior)) ;; whether the frog guesses the boxes were switched, before sample  
  
         (define frog-belief-post-boxes (if frog-thinks-switch? (switch-boxes pre-boxes) pre-boxes)) ;; where the frog thinks the boxes are, before observing sample  
  
         (define imagined-sample (sample-n-from-box  
                                  (box-majority-object (box-on-side which-box-drawn-from frog-belief-post-boxes)) ;; which-box-drawn-from is directly observed  
                                  sampling-manner         ;; directly observed by frog  
                                  experimenter-preference ;; directly observed by frog  
                                  (length observed-sample)))  ;; sample frog imagines might be seen, conditioned on his beliefs about the boxes  
  
         (define frog-belief-where-duck-is (where-is 'duck frog-belief-post-boxes)) ;; frog's updated belief on box locations after seeing sample  
  
         ;; variable of interest  
         frog-belief-where-duck-is  
  
         ;; condition  
         (equal? imagined-sample observed-sample)))  
  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       ;; (b) PRIOR BELIEF MODEL  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       (('b)  
        (rejection-query  
         (define frog-belief-post-boxes post-boxes)  
  
         (define frog-belief-where-duck-is (where-is 'duck frog-belief-post-boxes))  
  
         ;; variable of interest  
         frog-belief-where-duck-is))  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       ;; (c) LOCATION MODEL  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       (('c)  
        (rejection-query  
         (define frog-switch-prior .159)  
         (define frog-thinks-switch? (flip frog-switch-prior))  
         (define frog-belief-post-boxes (if frog-thinks-switch? (switch-boxes pre-boxes) pre-boxes))  
  
         (define frog-belief-where-duck-is (where-is 'duck frog-belief-post-boxes))  
         (define imagined-sample (sample-n-from-box  
                                  (box-majority-object (box-on-side which-box-drawn-from frog-belief-post-boxes)) ;; which-box-drawn-from is directly observed  
                                  sampling-manner         ;; directly observed by frog  
                                  experimenter-preference ;; directly observed by frog  
                                  0)) ;; does not learn from sampling, modeled as didn't see samples.  
         (define frog-belief-where-duck-is (where-is 'duck frog-belief-post-boxes))  
  
         ;; variable of interest  
         frog-belief-where-duck-is))  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       ;; (d) SAMPLED DATA MODEL  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       (('d)  
        (case sampling-manner  
          (('random)  
           (rejection-query  
            (define frog-switch-prior .5)  
            (define frog-thinks-switch? (flip frog-switch-prior))  
  
            (define frog-belief-post-boxes (if frog-thinks-switch? (switch-boxes pre-boxes) pre-boxes))  
  
            (define imagined-sample (sample-n-from-box  
                                     (box-majority-object (box-on-side which-box-drawn-from frog-belief-post-boxes))  
                                     sampling-manner  
                                     experimenter-preference  
                                     (length observed-sample)))  
  
            (define frog-belief-where-duck-is (where-is 'duck frog-belief-post-boxes))  
  
            ;; variable of interest  
            frog-belief-where-duck-is  
  
            ;; condition  
            (equal? imagined-sample observed-sample)))  
  
          (('selective)  
           (rejection-query  
            (define frog-switch-prior .5)  
            (define frog-thinks-switch? (flip frog-switch-prior))  
  
            (define frog-belief-post-boxes (if frog-thinks-switch? (switch-boxes pre-boxes) pre-boxes))  
  
            (define imagined-sample (repeat (length observed-sample)  
                                            (lambda ()  
                                              (if (flip) 'duck 'ball))))  
            (define frog-belief-where-duck-is (where-is 'duck frog-belief-post-boxes))  
            ;; variable of interest  
            frog-belief-where-duck-is))))  
  
  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       ;; (e) RANDOM STAY, SELECTIVE SHIFT MODEL  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       (('e)  
          (cond ((equal? sampling-manner 'random) which-box-drawn-from)  
                ((equal? sampling-manner 'selective) (other-box which-box-drawn-from))))  
  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       ;; (f) CHANCE RESPONSE MODEL  
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
       (('f)  
        (rejection-query  
         (define frog-switch-prior .5)  
         (define frog-thinks-switch? (flip frog-switch-prior))  
  
         (define frog-belief-post-boxes (if frog-thinks-switch? (switch-boxes pre-boxes) pre-boxes))  
  
         (define imagined-sample (repeat (length observed-sample)  
                                         (lambda ()  
                                           (if (flip) 'duck 'ball))))  
         (define frog-belief-where-duck-is (where-is 'duck frog-belief-post-boxes))  
         frog-belief-where-duck-is))))  
  
  
   ;; END MODEL-SPECIFIC CODE  
  
   (define noise .25) ;; noisy response model: probability of child making a random response  
   (define response (if (flip noise)  
                        (uniform-draw '(left right))  
                        (frog-sample))) ;; child's response (possibly noisy)  
  
   response  
   true))  
  
  (define child-distribution (aggregate-samples (repeat 1000 child-sample)))  
  child-distribution