LAAI homework

Andrea Pinto

Id: 0001002015

November, 2022

Contents

Exercise 1

Exercise 1.4

Observe that our model of evaluation allowsfor combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

```
    (define (a-plus-abs-b a b)
      ((if (> b 0) + -) a b))
```

Solution

This procedure takes in input a and b as parameters. It's body contains a combination that has a compound expression as first element/operator of the expression. When the procedure is called, the first element of the expression is evaluated; in this case we evaluate the *if* expression: it will return the operator '+' if the parameter b is greater than zero or the operator '-' if it isn't. Hence the result of the procedure is

$$\begin{aligned} a+b \quad & if\ b>0 \\ a-b \quad & if\ b\leq 0 \end{aligned}$$

Exercise 1.5

Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

```
    (define (p) (p))
    (define (test x y)
      (if (= x 0) 0 y))
```

Then he evaluates the expression

```
    (test 0 (p))
```

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer.

Solution

With the applicative-order evaluation the interpreter first evaluates the sub-expressions contained in a combination, then applies the operator/procedure to the arguments. In this case, since all elements of the expression are evaluated, when we evaluate p we'll have a non-terminating program since p calls itself. With normal-order evaluation the interpreter doesn't evaluate the operands of the expression until they're needed, so it first substitute operand expressions until there are only primitive operators. In this case, we'll have

```
    (test 0 p)
    -> (if (= x 0) 0 y)
    -> (if (= 0 0) 0 p)
    -> 0
```

Since the operands are evaluated only at the end, the if expression won't call p because we have that $x = 0$ and for this reason the program will terminate.

Exercise 2

Exercise 1.35

Show that the golden ratio (Section 1.2.2) is a fixed point of the transformation $x \to 1 + 1/x$, and use this fact to compute φ by means of the fixed-point procedure.

Solution

The definition of a fixed point says that a number x is a fixed point of a function f if it satisfies the equation $f(x) = x$. The exercise ask to prove that φ is a solution of the following equation

$$1 + 1/x = x$$

By imposing $x \neq 0$ we have:

$$x^2 - x - 1 = 0 \Rightarrow$$
$$\Rightarrow x_1 = \frac{1+\sqrt{5}}{2}, \quad x_2 = \frac{1-\sqrt{5}}{2}$$

Since φ is equal to x_1 this proves that it is a fixed point of our f.

Code

```
#lang racket

(define tolerance 0.00001)

(define (fixed-point f first-guess)

  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))

  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

(define (transform x) (+ 1 (/ 1 x)))
(fixed-point transform 1.0)
```

The following code utilizes the function fixed-point given in the book ”Structure and Interpretation of Computer Programs”. This procedure computes an estimate of a fixed point in the following way: it takes a function and an initial guess and applies the function to the initial guess, if the distance between the result and initial guess is below a certain tolerance it stops since it has found a fixed point, otherwise it will recursively search for a solution. The transform procedure is simply the implementation of the function given in the exercise.

Exercise 1.36

Modify fixed-point so that it prints the sequence of approximations it generates, using the newline and display primitives shown in Exercise 1.22. Then find a solution to $x^x = 1000$ by finding a fixed point of $x \to log(1000)/log(x)$. (Use Scheme's primitive log procedure, which computes natural logarithms.) Compare the number of steps this takes with and without average damping.

Solution

The first request of the exercise is simply, this can be obtained simply by adding the *displayln* procedure that displays the value and add a new line at the same time. In order to find the fixed point described in the exercise, the procedure transform has been changed to represent the function

$$f(x) = \frac{log(1000}{log(x)}$$

Average damping is used to help the convergence towards the fixed point, it is based on the fact that the fixed point will be found between x and $f(x)$ so we can avoid 'bounces' around the solution by moving toward the average of these values. Average damping basically means to search the following fixed point

$$x \mapsto \frac{1}{2}(x + f(x))$$

This has been implemented in the code by adding an 'average' procedure that computes the average of two numbers and by passing to the procedure 'fixed point' a lambda function that computes the average between x and $f(x)$.

It's possible to notice that average damping actually reduces the number of steps needed for convergence from 34 (no average damping) to just 9 steps.

Code

```racket
#lang racket

(define tolerance 0.00001)

(define (fixed-point f first-guess)

  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))

  (define (try guess)
    (let ((next (f guess)))

      (displayln next)

      (if (close-enough? guess next)
          next
          (try next))))

  (try first-guess))

(define (transform x) (/ (log 1000) (log x)))

(displayln "No average damping")
; It takes 34 steps to converge to the solution
(fixed-point transform 2.0)
(newline)

(displayln "Average damping")
; It takes 9 steps to converge to the solution
(define (average x y) (/ (+ x y) 2))
(fixed-point (lambda (x) (average x (/ (log 1000) (log x)))) 2.0)
```

Exercise 1.37

a. An infinite continued fraction is an expression of the form

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \cdots}}}$$

As an example, one can show that the infinite continued fraction expansion with the Ni and the Di all equal to 1 produces $1/\varphi$, where φ is the golden ratio (described in Section 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation— a so-called k-term finite continued fraction—has the form

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_k}{D_k}}}$$

Suppose that n and d are procedures of one argument (the term index i) that return the N_i and D_i of the terms of the continued fraction. Define a procedure cont-frac such that evaluating (cont-frac n d k) computes the value of the k-term finite continued fraction. Check your procedure by approximating $1/\varphi$ using

```
    (cont-frac (lambda (i) 1.0)
               (lambda (i) 1.0)
               k)
```

for successive values of k. How large must you make k in order to get an approximation that is accurate to 4 decimal places?

b. If your cont-frac procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

Solution

The first solution implemented for this problem is recursive. The idea is to define a procedure inside 'cont-fract' called 'recursion' that take one parameter 'i', this procedure is called inside 'cont-frac' with $i = 1$ because we want to start from the 'top' of the k-term finite continued fraction. At every step the procedure will compute

$$\frac{n_i}{d_i + recursion_{i+1}}$$

The stopping condition, implemented using an if statement, is that when we reach $i = k$ we want the result of this step to be just

$$\frac{n_k}{d_k}$$

without calling again the 'recursion' procedure. Implemented in this way, the procedure takes 10 steps to approximate $1/\varphi$ accurate to 4 decimal places.

The second solution implemented is iterative. In this case we start from the 'bottom' of the fraction. The procedure inside 'cont-frac' is called 'iter' and will take two parameters: counter (initialized with k) and result (initialized at 0). If counter is greater than zero, the 'iter' procedure is called with *counter-1* and the result updated as follows:

$$\frac{n_{counter}}{d_{counter} + result}$$

Also in this case, the number of steps needed to approximate $1/\varphi$ is 10.

Code

```
#lang racket

(define (cont-frac-rec n d k)

(define (recursion i)
    (if (= k i)
        ; if index k
        (/ (n i) (d i))
        ; if NOT index k
        (/ (n i) (+ (d i) (recursion (+ i 1))))
        )
    )
(recursion 1))

;(cont-frac-rec (   (i) 1.0)
;               (   (i) 1.0)
;               11)

(define (cont-frac-iter n d k)

(define (iter counter result)
    (if (> counter 0)
        (iter (- counter 1)(/ (n counter) (+ (d counter) result)))
        result
        )
    )
(iter k 0)
)

(cont-frac-iter (lambda (i) 1.0)
                (lambda (i) 1.0)
                11)
```

Exercise 1.38

In 1737, the Swiss mathematician Leonhard Euler published a memoir De Fractionibus Continuis, which included a continued fraction expansion for e 2, where e is the base of the natural logarithms. In this fraction, the Ni are all 1, and the Di are successively 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, ... Write a program that uses your cont-frac procedure from Exercise 1.37 to approximate e, based on Euler's expansion.

Solution

For this exercise has been used the procedure 'cont-frac-rec' shown in the previous exercise. The only thing that needs to be implemented is a procedure that returns the value of D_i based on the index that it's passed at it as parameter. The values of D_i are 1 and multiple of 2. The idea is this: the multiple of two can be found starting from index 2 and with steps of 3, so we'll have these multiples at index 2, 5, 8, ... we can add one to these indeces because now we can say that if the $index + 1$ is a multiple of 3, we'll have a multiple of two. Furthermore, diving the $index + 1$ by 3 gives us the number of times we have to multiply 2.

Code

```
#lang racket

(define (cont-frac-rec n d k)

  (define (recursion i)
    (if (= k i)
        (/ (n i) (d i))
        (/ (n i) (+ (d i) (recursion (+ i 1))))
        )
    )
  (recursion 1))

(define (d k)
  (if (= (modulo (+ k 1) 3) 0)
      (* (/ (+ k 1) 3) 2)
      1.0)
  )

(+ 2 (cont-frac-rec (lambda (i) 1.0)
                    (lambda (i)
                      (if (= (modulo (+ i 1) 3) 0)
                          (* (/ (+ i 1) 3) 2)
                          1.0)
                      )
                    10))
```

Exercise 3

Exercise 2

Explain why (in terms of the evaluation process) these two programs give different answers (i.e. have different distributions on return values):

```
    (define foo (flip))
    (list foo foo foo)
```

```
    (define (foo) (flip))
    (list (foo) (foo) (foo))
```

Solution

In the first case 'foo' it's not a procedure, is a variable associated to the procedure flip. When it is called inside list for the first time, flip is evaluated and the result is stored into foo, when it's called the other two times the result will be the one linked inside the foo variable. The result will be a list containing all true or all false values. In the second case foo is defined as a procedure without arguments, basically a thunk, because it's contained inside parenthesis. Every time that '(foo)' is called inside the list the flip procedure will sample from it's distribution and return either true or false.

Exercise 5

Here is a modified version of the tug of war game. Instead of drawing strength from the continuous Gaussian distribution, strength is either 5 or 10 with equal probability. Also the probability of laziness is changed from 1/4 to 1/3. Here are four expressions you could evaluate using this modified model:

```

  (define strength (mem (lambda (person) (if (flip) 5 10))))
  (define lazy (lambda (person) (flip (/ 1 3))))
  (define (total-pulling team)
    (sum
     (map (lambda (person) (if (lazy person) (/ (strength person) 2) (strength person)))
          team)))
  (define (winner team1 team2) (if (< (total-pulling team1) (total-pulling team2)) team2 team1))

  (winner '(alice) '(bob))                         ;; expression 1

  (equal? '(alice) (winner '(alice) '(bob)))       ;; expression 2

  (and (equal? '(alice) (winner '(alice) '(bob))) ;; expression 3
       (equal? '(alice) (winner '(alice) '(fred))))

  (and (equal? '(alice) (winner '(alice) '(bob))) ;; expression 4
       (equal? '(jane) (winner '(jane) '(fred))))
```

a. Write down the sequence of expression evaluations and random choices that will be made in evaluating each expression.

b. Directly compute the probability for each possible return value from each expression.

c. Why are the probabilities different for the last two? Explain both in terms of the probability calculations you did and in terms of the “causal” process of evaluating and making random choices.

Solution

a. The first expression calls the 'write' procedure 'alice' and 'bob' as arguments. Inside this procedure will be called the 'total-pulling' procedure in the if expression on both arguments; this procedure will sum the elements in the list produced my 'map'. The map takes a function and a list and will apply the function to all elements of that list; in this case the map will call the 'lazy' procedure on all the elements of a team. The lazy procedure will decide randomly if a team member is lazy or not, this is done by sampling from the 'flip' distribution that has probability of 1/3 to return true. After deciding whether a person is lazy or not, the strength procedure will be called and, also in this case, the flip procedure is used to decide randomly the strength of a person, it can be 5 or 10 with 1/2 probability each. The second expression wants to determine if alice is the winner, if this is the case it will return t. Also in this case the winner procedure is called and will do what has been described above, except for the fact that when the 'strength' procedure will be called on alice and bob, since the procedure is memoized, we won't sample again their strength value but we'll simply read them from memory. The third expression will return true if alice is the winner both against bob and fred. The first 'winner' procedure will proceed as expression 2 while the second one will sample the strength value of fred because it is a new arguments, then its value will be also stored. In the last expression we'll have true if the first winner is alice and the second one is jane. The first 'winner' procedure will work as expression 2 while in the second one we'll sample the strngth value of jane.

b. For the first expression we'll compute the probability of the output 'alice', which is the same thing as saying that alice will win the competition against bob. In the following table will be shown the condition under which alice can win

Alice		Bob	
Strength	Lazy	Strength	Lazy
10	NL	//	//
10	L	$\neq 10$	$\neq$ NL
5	NL	$\neq 10$	$\neq$ NL
5	L	$= 5$	$=$ L

In the first row of the table '//' means that for whatever value of strength and lazy bob will lose. We know that $P(10) = P(5) = 1/2, P(L) = 1/3$ and $P(NL) = 1 - P(L) = 2/3$. The probability of alice winning is

$$P(alice) = P(10) * P(NL) + (P(10) * P(L)) * (1 - P(10)P(NL)) + \\ P(5) * P(NL) * (1 - P(10) * P(NL)) + P(5)^2 + P(L)^2 = 0.694$$

And it follows that

$$P(bob) = 1 - P(alice) = 0.306$$

In the second expression we have the same procedure as before, so the probability will be the same. In this case we'll have that the probability of the output being t it's equal to the probablity of alice winning.

$$P(\#t) = P(alice) = 0.694$$
$$P(\#f) = P(bob) = 0.306$$

The probability of having t in the third expression is given by the probability of alice winning both matches. The strength of alice will be determined in the first match against bob, so in the second match only the lazyness can change. We can construct the following table with the cases in which alice wins

Alice			Bob		Fred	
Strength	Lazy1	Lazy2	Strength	Lazy	Strength	Lazy
10	NL	NL	//	//	//	//
10	NL	L	//	//	$\neq$ 10	$\neq$ NL
10	L	NL	$\neq$ 10	$\neq$ NL	//	//
10	L	L	$\neq$ 10	$\neq$ NL	$\neq$ 10	$\neq$ NL
5	NL	NL	$\neq$ 10	$\neq$ NL	$\neq$ 10	$\neq$ NL
5	NL	L	$\neq$ 10	$\neq$ NL	= 5	= L
5	L	NL	= 5	= L	$\neq$ 10	$\neq$ NL
5	L	L	= 5	= L	= 5	= L

From this we can obtain the probability

$$\begin{aligned}
P(t) = P(10) \cdot P(NL) \cdot [P(NL) + P(L) \cdot (1 - P(10) \cdot (PNL))] + \\
P(L) \cdot [P(NL) \cdot (1 - P(10) \cdot P(NL)) + (1 - P(10) \cdot P(NL))^2] + \\
P(5) \cdot P(NL) \cdot [P(NL) \cdot (1 - P(10) \cdot P(NL))^2 + P(L) \cdot (1 - P(10) \cdot P(NL)) \cdot P(5) \cdot P(L)] + \\
P(L) \cdot [P(NL) \cdot P(5) \cdot P(L) \cdot (1 - P(10) \cdot P(NL)) + P(L) \cdot P(5)^2 \cdot P(L)^2] = 0.52
\end{aligned}$$

$$P(\#f) = 1 - P(\#t) = 0.48$$

In the last expression the probability of having true is based on alice winning the first match and jane winning the second one. In this case the matches are independent, since nobody plays in both of them. The probability of the first match is the same one already computed in the expression 1 and 2. We can observe that the second match will also have the same probability the other match, since we can easily swap alice with jane and fred with bob. Hence we have that

$$P(\#t) = P(alice) \cdot P(jane) = P(alice)^2 = 0.482$$
$$P(\#f) = 1 - P(alice)^2 = 0.518$$

c. For the last two expressions the probabilities are different because in expression 3 the matches are not independent, this happens because the strength of alice is determined in the match with bob and it will influence the result of the match with fred. This means that in this case (expression 3) we have one less random sample compared to the expression 4, where both matches are independent from each other.

Exercise 6

Use the rules of probability, described above, to compute the probability that the geometric distribution defined by the following stochastic recursion returns the number 5.

```
    (define (geometric p)
      (if (flip p)
          0
          (+ 1 (geometric p))))
```

Solution

The probability of having 5 as output can be seen as the probability of having a ’true’ sampled from the ’flip’ procedure after five ’false’. The sample will be performed with a weight p, which is the probability of having true as output. For this reasons, the probability of getting 5 as output can be computed as such

$$P(5) = (1 - p)^5 \cdot p$$

Exercise 7

Convert the following probability table to a compact Church program:

A	B	P(A, B)
F	F	0.14
F	T	0.06
T	F	0.4
T	T	0.4

Hint: fix the probability of A and then define the probability of B to depend on whether A is true or not. Run your Church program and build a histogram to check that you get the correct distribution.

Solution

It's possible to compute the probability of A by performing marginalization. We have that

$$P(A = T) = \sum_B P(A, B) = 0.4 + 0.4 = 0.8$$

Knowing this, it's clear that in order to have

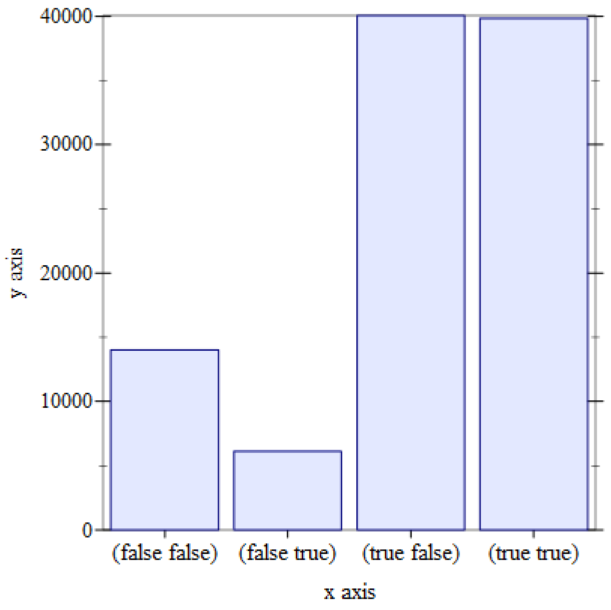
$$P(A = T, B) = P(A = T)P(B|A = T)) = 0.4$$

It must be that $P(B|A = T) = 0.2$. By performing the same reasoning for the case in which $A = F$ we obtain that $P(B|A = F) = 0.3$.

Code

The following code is used to encode what has been described above and print the histogram to check the distribution

```
(define (a) (flip 0.8))
(define (b) (flip (if a 0.5 0.3)))

(define (model)
  (set! a (a))
  (set! b (b))
  (list a b))

(hist (repeat model 100000))
```



Histogram of the distribution

Exercise 4

Exercise 1

What are (bernoulli-dist p), (normal-dist μ σ) exactly? Are they real numbers (produced in a random way)? We have seen that flip is a procudere with a probabilistic behaviour. Is, e.g., (normal-dist μ σ) something similar? Try to evaluate (normal-dist 0 1).

Solution

This are not real numbers nor procedures with a probabilistic beahavior, in fact if we evaluate (normal-dist 0 1) we get itself as output because it's not a procedure. They are distributions objects.

Exercise 2

Evaluate

```
        (dist? (normal-dist 0 1))
        (dist? (bernoulli-dist 0.5))
        (dist? flip)
```

What is the difference between flip and (bernoulli-dist 0.5)?

Solution

The evaluation of these lines of code returns t t f as expected, because dist? is used to check whether an object is a distribution or not. The first two are distribution objects while flip is a procedure. The difference between flip and (bernoulli-dist 0.5) is that flip is a procedure defined to sample from the bernoulli distribution with probability 0.5, while (bernoulli-dist 0.5) is simply a distribution object, if we want to sample we have to use the procedure 'sample'. In fact, flip is the same thing as (sample (bernoulli-dist 0.5)).

Exercise 5

Exercise 4

Casino game. Consider the following game. A machine randomly gives Bob a letter of the alphabet; it gives a, e, i, o, u, y (the vowels) with probability 0.01 each and the remaining letters (i.e., the consonants) with probability 0.047 each. The probability that Bob wins depends on which letter he got. Letting h denote the letter and letting Q(h) denote the numeric position of that letter (e.g., Q(a) = 1, Q(b) = 2, and so on), the probability of winning is $1/Q(h)^2$. Suppose that we observe Bob winning but we don't know what letter he got. How can we use the observation that he won to update our beliefs about which letter he got? Let's express this formally. Before we begin, a bit of terminology: the set of letters that Bob could have gotten, {a, b, c, d, ..., y, z}, is called the hypothesis space – it's our set of hypotheses about the letter.

A. In English, what does the posterior probability $p(h \mid win)$ represent?
B. Manually compute $p(h \mid win)$ for each hypothesis.
C. What does the my-list-index function do? What would happen if you ran (my-list-index 'mango '(apple banana) 1)?
D. What does the multinomial function do? Use multinomial to express this distribution:

x	**P(x)**
red	0.5
blue	0.05
green	0.4
black	0.05

E. Fill in the ...'s in the code to compute $p(h \mid win)$. Include a screenshot of the resulting graph. What letter has the highest posterior probability? In English, what does it mean that this letter has the highest posterior? Make sure that your Church answers and hand-computed answers agree – note that this demonstrates the equivalence between the program view of conditional probability and the distributional view.

F. Which is higher, $p(vowel \mid win)$ or $p(consonant \mid win)$? Answer this using the Church code you wrote (hint: use the vowel? function)

G. What difference do you see between your code and the mathematical notation? What are the advantages and disadvantages of each? Which do you prefer?

Solution

A. Since h represent the letter that Bob got, $p(h \mid win)$ is the probability of Bob having letter h having observed that he won.

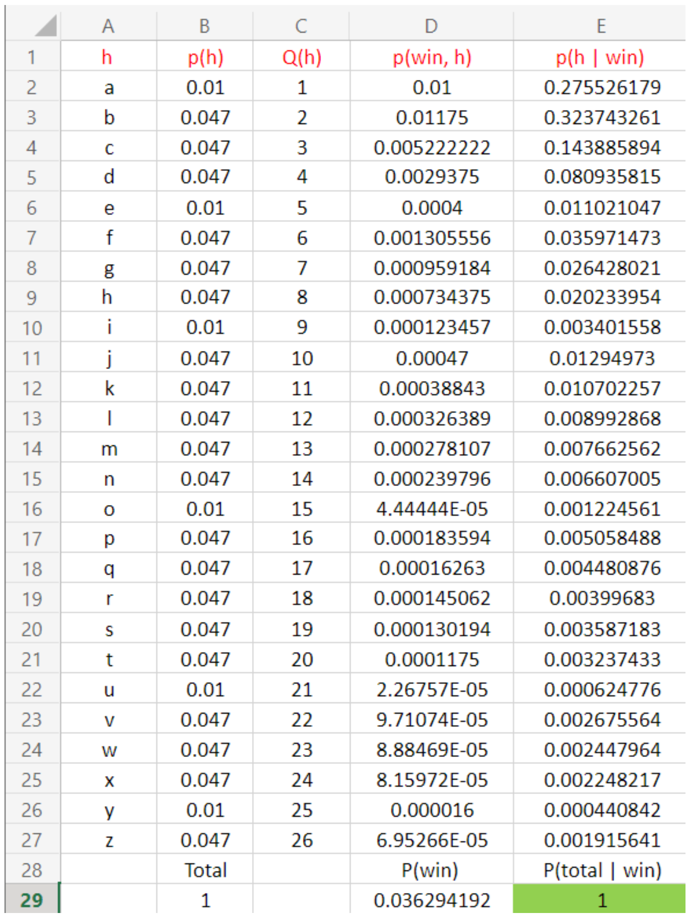
B. In order to compute $p(h \mid win)$ we must use the definition of conditional probability

$$P(h \mid win) = \frac{P(h, win)}{P(win)} = \frac{P(h) \cdot P(win \mid h)}{P(win)}$$

We know the probabilities P(h) for every h, we know that the probability of winning given h is $1/Q(h)^2$ and we can compute the probability of winning by computing the marginal probability

$$P(win) = \sum_h P(win,\ h)$$

In order to compute the value for each h, excel has been used. The results can be found in the image below. The values of $p(h \mid win)$ have been normalized by 0.999999995 such that their sum is equal to 1.



	A	B	C	D	E
1	h	p(h)	Q(h)	p(win, h)	p(h | win)
2	a	0.01	1	0.01	0.275526179
3	b	0.047	2	0.01175	0.323743261
4	c	0.047	3	0.005222222	0.143885894
5	d	0.047	4	0.0029375	0.080935815
6	e	0.01	5	0.0004	0.011021047
7	f	0.047	6	0.001305556	0.035971473
8	g	0.047	7	0.000959184	0.026428021
9	h	0.047	8	0.000734375	0.020233954
10	i	0.01	9	0.000123457	0.003401558
11	j	0.047	10	0.00047	0.01294973
12	k	0.047	11	0.00038843	0.010702257
13	l	0.047	12	0.000326389	0.008992868
14	m	0.047	13	0.000278107	0.007662562
15	n	0.047	14	0.000239796	0.006607005
16	o	0.01	15	4.44444E-05	0.001224561
17	p	0.047	16	0.000183594	0.005058488
18	q	0.047	17	0.00016263	0.004480876
19	r	0.047	18	0.000145062	0.00399683
20	s	0.047	19	0.000130194	0.003587183
21	t	0.047	20	0.0001175	0.003237433
22	u	0.01	21	2.26757E-05	0.000624776
23	v	0.047	22	9.71074E-05	0.002675564
24	w	0.047	23	8.88469E-05	0.002447964
25	x	0.047	24	8.15972E-05	0.002248217
26	y	0.01	25	0.000016	0.000440842
27	z	0.047	26	6.95266E-05	0.001915641
28		Total		P(win)	P(total | win)
29		1		0.036294192	1

Computation of $P(h)$ for every h

C. The 'my-list-index' function computes the index of the element 'needle' inside the list 'haystack', it returns 'error if the element is not present in the list or if haystack is empty. By running

```
(my-list-index 'mango '(apple banana) 1)
```

we simply get 'error as output since the element 'mango is not present in the list '(apple banana).

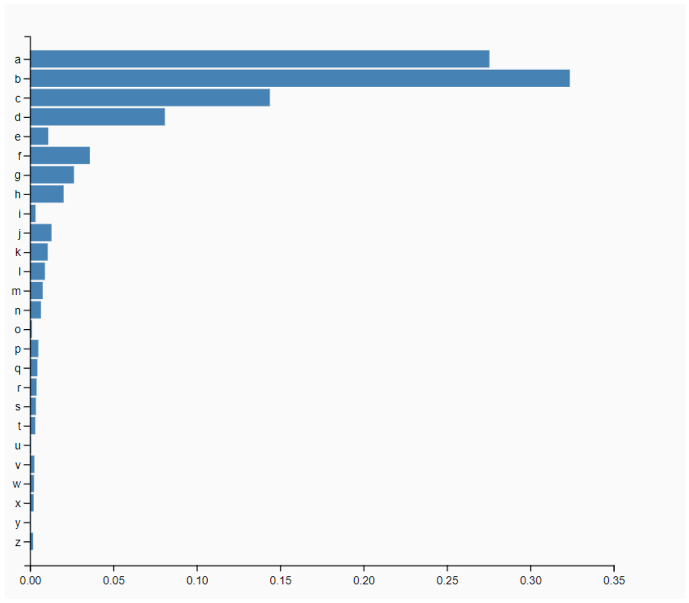
D. The multinomial function in Church takes a list of elements and a list of probabilities and samples from the list with the given probabilities. We can express the distribution P(x) and sample from it with the following code

```
(define x '(red blue green black))
(define Px '(0.5 0.05 0.4 0.05))
(multinomial x px)
```

E. The completed code is

```
 ;; define some variables and utility functions
 (define letters '(a b c d e f g h i j k l m n o p q r s t u v w x
y z) )
 (define (vowel? letter) (if (member letter '(a e i o u y)) #t #f))
 (define letter-probabilities (map (lambda (letter) (if (vowel?
letter) 0.01 0.047)) letters))

 (define (my-list-index needle haystack counter)
   (if (null? haystack)
       'error
     (if (equal? needle (first haystack))
         counter
       (my-list-index needle (rest haystack) (+ 1 counter)))))

 (define (get-position letter) (my-list-index letter letters 1))

 ;; actually compute p(h | win)
 (define distribution
   (enumeration-query
    (define my-letter (multinomial letters letter-probabilities))

    (define my-position (get-position my-letter))
 ; Q(h)
    (define my-win-probability (/ 1.0 (* my-position my-position)))
 ; 1/Q(h)^2
    (define win? (flip my-win-probability))
 ; P(win)

    ; We want to know the probability of 'my-letter' (i.e. h) when
conditioned with 'win'.
    ;; query
    my-letter

    ;; condition
    (condition win?)
    ))

 (barplot distribution)
```

The resulting graph is



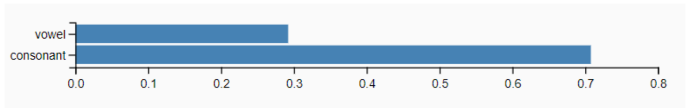
Probability distribution of $P(h \mid win)$

We can observe that the letter with the highest probability is 'b'. This matches the computations done before, since in the excel table we can observe that b has the highest value of $P(h \mid win)$. Saying that b has the highest posterior means that, when we observe that Bob won, the letter b is the most likely to have caused that win.

F. By replacing the previous query with

```
    (if (vowel? my-letter)
        'vowel
        'consonant)
```

we can observe that clearly $P(consonant \mid win)$ is higher.



$P(vowel \mid win)$ *and* $P(consonant \mid win)$

Exercise 6

Exercise 1

To see the problems of rejection sampling, consider the following variation of the previous example:

```
    (define baserate 0.1)
    (define (take-sample)
      (rejection-sampler
       (define A (if (flip baserate) 1 0))
       (define B (if (flip baserate) 1 0))
       (define C (if (flip baserate) 1 0))
       (define D (+ A B C))
       (observe/fail (>= D 2))
       A))
```

Try to see what happens when you lower the baserate. What happens if we set it to 0.01? And to 0.001?

Solution

It can be observed that as we lower the baserate the execution time increases significantly. The rejection sampler produces a sample and, if it is coherent with the observed value we keep it, otherwise we reject it. In this case the observation is $D = A+B+C2$. When we lower the baserate, the likelihood of observing $D \geq 2$ is very low, so a lot of random samples will be rejected.

Exercise 7

Exercise 1

Complete the above proof. Prove, in particular that for any $x \in X$, $c_P(x)$ is indeed a distribution; that P_c is a stochastic matrix; and that $P_{c_P} = P$ and $c_{P_c} = c$.

Solution

Since the function c is defined as $c : X \to D(X)$, which means that it maps the finite set X to the set of distributions over X, we have that $\{c(x_1), ..., c(x_n)\}$ are probability distributions/row vectors of size n describing the probability of transitioning from state x_i to state x_j. Since P_c is defined as $P_c(i, j) = c(x_i)(x_j)$, this means that every row of P_c will be formed by $c(x_i)$, which is a distribution. Because we have n distributions $c(x_i)$, the matrix will be of size $n \times n$ with elements that belong in $[0, 1]$ and with row vectors such that $\sum_j P_c(i, j) = 1$; hence P_c is a stochastic matrix. Viceversa, having P stochastic matrix, we define $c_P(x_i)(x_j) = P(i)(j)$. Because we have $\sum_j P(i, j) = 1$ this implies that $\sum_j c_P(x_i)(x_j) = 1$, so , $c_P(x_i)$ is a distribution, in particular is a row vector of size n containing the probabilities of transitioning from state x_i to state x_j. Hence $c_P : X \to D(X)$.

Exercise 2

Prove that $c(x) = c^*(\delta_x)$.

Solution

We can apply the definition of c^* to a starting state x_i and a target state y.

$$c^*(\delta_{x_i})(y) = \sum_j \delta_x(x) \cdot c(x_j)(y) = \sum_j \delta_{x_i} \cdot c(x_j)(y) = c(x_i)(y)$$

This is due to the fact that $\delta_{x_i} \neq 0$ for $i \neq j$. This proves the thesis.

Exercise 3

Prove that $c^*(\psi) = \psi P_c$.

Solution

Consider the transition between state x_i to state x_j. We have that $\forall j$ of the resulting vector of ψP

$$(\psi P)(j) = \sum_i \psi(i) \cdot P_c(i,j) = \sum_i \psi(i) \cdot c(x_i)(x_j)$$

On the other hand, by using the definition of c^* on the target state x_j we have

$$c^*(\psi)(x_j) = \sum_i \psi(i) \cdot c(x_i)c(x_j)$$

Hence we have that $c^*(\psi) = \psi P$.

Exercise 4

Prove that if ψ satisfies DBC, then ψ is stationary for P.

Solution

Considering an initial state x_i and a target state x_j. The probability of being in x_i is described in $\psi(i)$, while the probability of transitioning from x_i to x_j is given by $(\psi P)(j)$, by using its definition we have

$$(\psi P)(j) = \sum_i \psi(i) \cdot P(i,j)$$

since ψ satisfies DBC we have

$$(\psi P)(j) = \sum_i \psi(i) \cdot P(i,j) = \sum_i \psi(j) \cdot P(j,i) = \psi(j) \sum_i P(i,j) = \psi(j)$$

because the sum of a row vector of P is 1. Hence $\psi P = \psi$.