Languages and Algorithms for Artificial Intelligence Module 2: Homework

Guido Laudenzi 0001033343

April 19, 2022

Exercise 1 (1.4)

Observe that our model of evaluation allows for 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 A query example for studying the compound behaviour of the procedure:

```
(a-plus-abs-b (and 3 2) -4)
6
(a-plus-abs-b (or 3 2) -4)
7
```

The procedure `a-plus-abs-b` sums the expression a with the absolute value of b. In fact, if b is bigger than 0, it will be a sum, otherwise it will be a subtraction.
Anyway, it is possible to use the logical compound operations such as *and*, *or* and *not*. This is because when the interpreter checks a predicate's value, it interprets 0 as false and any other value is treated as true.
In particular, we have different behaviors depending on *or*, *and* and *not*:

- AND: interpreter evaluates the expressions one at a time, in left-to-right order.
  If any expression evaluates to false, the value of the and expression is false, and the rest of the expressions are not evaluated. If all expressions evaluate to true values, the value of the and expression is the value of the last one.
- OR: e interpreter evaluates the expressions one at a time, in left-to-right order.
  If any expression evaluates to a true value, that value is returned as the value of the or expression, and the rest of the expressions are not evaluated. If all expressions evaluate to false, the value of the or expression is false.
- NOT: the value of a not expression is true when the expression evaluates to false, and false otherwise. Here, we would end up with an error if using not with a number ($\#f$ cannot be used in the procedure).

Exercise 1 (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 The interpreter first evaluates the operator and operands and then applies the resulting procedure to the resulting arguments. This is not the only way to perform evaluation. An alternative evaluation model would not evaluate the operands until their values were needed.
The first one is the applicative-order evaluation, the second one is known as normal-order evaluation. The standard evaluation of the interpreter is the first one. Doing so, it firstly evaluates the operands then `test` procedure, so it ends up in a loop while evaluating (p), that is a function that evaluates to itself.
Using normal-order evaluation, we would get 0 as result: this is because operands are not needed for evaluating `test`, so if x is 0 (that is in our query) the interpreter would not need to evaluate (p) (that results in a loop) because of the *if* clause.

Exercise 2 (1.35)

Show that the golden ration ϕ is a fixed-point of the transformation $x \mapsto 1 + 1/x$, and use this fact to compute ϕ by means of the `fixed-point` procedure.

Solution This is the `fixed-point` procedure:

```
(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))
```

We know that $x \mapsto 1 + 1/x$ basically means $x = 1 + 1/x$. It can be transformed in a second order equation which has one solution: $x = 1 + \sqrt{5}/2$.
We know that the golden ratio is exactly that quantity and that is equal to 1.6180.
`fixed-point` procedure is given by the book and wants as inputs a function and an initial guess. We need to give $x = 1 + 1/x$ as function input, while the initial guess could be whatever number.

Example:

```
(fixed-point (lambda (x) (+ 1 (/ 1 x))) 3.0)
1.618035190615836
```

Exercise 2 (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. (Note that you cannot start `fixed-point` with a guess of 1, as this would cause division by $log(1) = 0$.)

Solution Firstly, we take the book's function for the average:

```
(define (average x y)
  (/ (+ x y) 2))
```

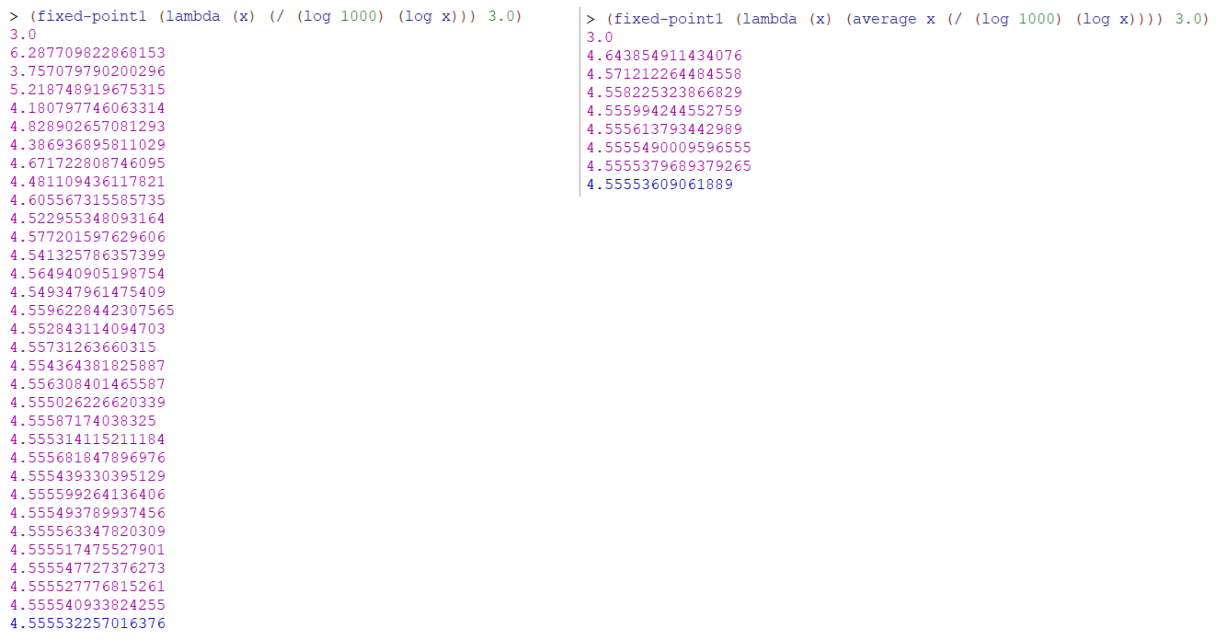
Then, we modify the fixed-point procedure so to print the guesses using `display` and `newline` procedures. The new procedure's name is `fixed-point1`:

```scheme
(define (fixed-point1 f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (display guess) ;here we print each guess
    (newline)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

Finally, we just need to use queries in order to compare the results with and without average damping. We must remember not to use 1.0 as initial guess, as written in the exercise, because we are using the primitive log. Input function would be $log(1000)/log(x)$; initial guess is the same as the previous exercise.

```scheme
(fixed-point1 (lambda (x) (/ (log 1000) (log x))) 3.0)
(fixed-point1 (lambda (x) (average x (/ (log 1000) (log x)))) 3.0)
```

Using average damping would let the procedure run faster, maintaining the same accuracy. Here are the results:



```
> (fixed-point1 (lambda (x) (/ (log 1000) (log x))) 3.0)
3.0
6.287709822868153
3.757079790200296
5.218748919675315
4.180797746063314
4.828902657081293
4.386936895811029
4.671722808746095
4.481109436117821
4.605567315585735
4.522955348093164
4.577201597629606
4.541325786357399
4.564940905198754
4.549347961475409
4.5596228442307565
4.552843114094703
4.55731263660315
4.554364381825887
4.556308401465587
4.555026226620339
4.55587174038325
4.555314115211184
4.555681847896976
4.555439330395129
4.555599264136406
4.555493789937456
4.555563347820309
4.555517475527901
4.555547727376273
4.555527776815261
4.555540933824255
4.555532257016376
```

```
> (fixed-point1 (lambda (x) (average x (/ (log 1000) (log x)))) 3.0)
3.0
4.643854911434076
4.571212264484558
4.558225323866829
4.555994244552759
4.555613793442989
4.5555490009596555
4.5555379689379265
4.55553609061889
```

Figure 1: Comparing results with and without average damping

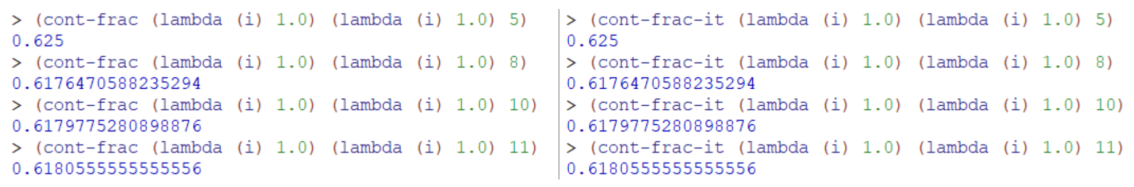
Exercise 2 (1.37)

Refer to exercise 1.37 from Structure and Interpretation of Computer Programs.

Solution a) This is the recursive procedure. In particular, it compiles the k-term finite continued fraction until the order k which is given by input. Recursion is given by the `if` clause that compares if the procedure has reached the desired order.

```scheme
(define (cont-frac n d k)
  (define (fraction i)
    (/ (n i) (+ (d i) (if (= k i) 0 (fraction (+ 1 i))))))
  (fraction 1))
```

The `cont-frac` procedure firstly defines the function `fraction`, then calls it with with starting value equal to 1. It will append to the denominator another fraction until the value of i get to the one of k. We can get an approximation that is accurate to 4 decimal places (0.6180) if we set k to 11 (or higher). Otherwise, the value would not be exact.



```
> (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 5)
0.625
> (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 8)
0.6176470588235294
> (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 10)
0.6179775280898876
> (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 11)
0.6180555555555556
```

```
> (cont-frac-it (lambda (i) 1.0) (lambda (i) 1.0) 5)
0.625
> (cont-frac-it (lambda (i) 1.0) (lambda (i) 1.0) 8)
0.6176470588235294
> (cont-frac-it (lambda (i) 1.0) (lambda (i) 1.0) 10)
0.6179775280898876
> (cont-frac-it (lambda (i) 1.0) (lambda (i) 1.0) 11)
0.6180555555555556
```

Figure 2: Recursive and Iterative procedures varying with k

b) This is the iterative procedure. The substantial difference is that the intermediate result will be carried as parameter at each step. Accuracy is the same as the recursive procedure.

```
(define (cont-frac-it n d k)
  (define (fraction k c)
    (if (= k 0) c (fraction (- k 1) (/ (n k) (+ (d k) c)))))
  (fraction k 0))
```

Exercise 2 (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 N_i are all 1, and the D_i 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 We know that D_i is a regular series except for the first index which has value 1.
All the values are 1, except for the indexes $2, 5, 8, ...$, where there are values multiples of 2 $(2, 4, 6, 8, 10$, etc.).
We need to define a procedure to establish the values of D_i based on the indexes. We find that $D_i = 2(i+1)/3$ where i is the index. However, this is true only when the remainder of $i+1/3$ is not 0.
Finally, all the N_i are equal to 1.

```
(define (d i)
  (if (not(= 0 (remainder (+ i 1) 3))) 1 (* 2 (/ (+ i 1) 3))))
(define (n i) 1.0)
(define (e k)
  (+ 2 (cont-frac n d k)))

(e 5)
2.71875
(e 10)
2.7182817182817183
```

Exercise 3 (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 The difference is that in the first case the interpreter binds the procedure `flip` to a variable called *foo*. Therefore, its value is not going to change afterwards and the list will display only the same values $(\#f, \#f, \#f)$ or $(\#t, \#t, \#t)$, that are repeated *foo*s.
The latter is defined as a procedure which recall the procedure `flip`. This means that everytime we construct a list of (*foo*), they will have different values because each one will be newly generated by the `flip` procedure.

Exercise 3 (5)

Refer to exercise 5 from Probabilistic Models of Cognition.

Solution a)

- **Expression 1**

```
(winner '(alice) '(bob))
```

There is a call for the procedure `winner`, that decides which team is winner. `winner` procedure recalls `total-pulling` procedure for each team (in this case *Alice* and *Bob*). It, basically, sum the number referred to the strength, which is generated by the `strength` procedure, of each person in the team. Strength of each person is memorized as 5 if flip is $\#t$, 10 otherwise, with standard probability 0.5.
Then, strength depends on laziness of a person. Laziness is generated by the `lazy` procedure and is expressed as a unfair flip of a coin, where 1 times out of 3 is $\#t$.
If the person is lazy, then its strength is halved. Finally, the winner team is the one with the bigger total strength, or, in case of tie, the first team given as input.
Anyway, the strength of a person is kept in memory using the `mem` procedure, while the laziness is not.
Computing probabilities, it would give as results that the winner is Alice with probability 0.69, Bob with 0.31.

- **Expression 2**

```
(equal? '(alice) (winner '(alice) '(bob)))
```

This expression is similar to the first one, but there is another step of the process: using `equal?` we want to check if Alice is the winner between Alice and Bob.
In this case, the expression will not return the name of the winner but $\#t$ if Alice is the winner, $\#f$ otherwise.
If this expression is computed after the first one, Alice's strength would be kept in memory. So, the probability that Alice wins depends on the first expression: if Alice had strength 10 and Bob 2.5, the expression would always return $\#t$. Otherwise, the result of the expression would depend on the laziness probability of a person.

- **Expression 3**

```
(and (equal? '(alice) (winner '(alice) '(bob)))
     (equal? '(alice) (winner '(alice) '(fred))))
```

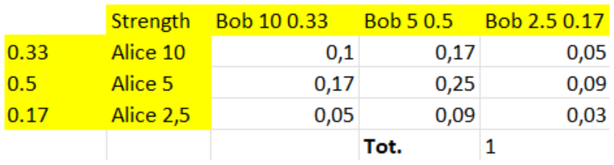
In this case, we want to check if Alice both wins against Bob and Fred. If so, the expression would return $\#t$; it would return $\#f$ if Alice is defeated by one between Bob and Fred, or both.
The probability that Alice wins is different than the previous expressions: while her strength and Bob's strength remain fixed, Fred's strength is still not memorized.
It is possible to notice though that if Alice would win against Bob, there is an increased probability that Alice wins against Fred too: this is because if Alice has strength equal to 10, then there is an higher probability that Fred has a lower strength overall.

- **Expression 4**

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

This expression is similar to the expression 3, but in this case we want to check if Alice wins against Bob and Jane wins against Fred.
The expression returns $\#t$ if both win, $\#f$ if one of them lose against their opponent.
Alice, Bob and Fred have their strength memorized since the previous expressions' evaluation; Jane has her strength to be calculated. This means that the probability that one wins is independent from the probability that the second one wins.

b) The probability that a person has a particular strength is showed in the picture. For each expression, we must calculate the probability that the person wins.



	Strength	Bob 10 0.33	Bob 5 0.5	Bob 2.5 0.17
0.33	Alice 10	0,1	0,17	0,05
0.5	Alice 5	0,17	0,25	0,09
0.17	Alice 2,5	0,05	0,09	0,03
			Tot.	1

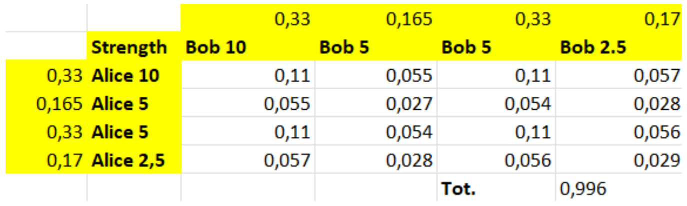
Figure 3: Probabilities of strength

- **Expression 1**: in this case, we want to know if Alice wins, so we must sum the upper right corner of the matrix, because if both have the same strength, the first team wins, as showed by the `if` clause of the `winner` procedure. The probability is:

$$P(Alice = \text{win}) = 0.1 + 0.17 + 0.05 + 0.25 + 0.09 + 0.03 = 0.69$$

- **Expression 2**: this expression is equal to the first one, except for the clause that compares the result of the `winner` procedure with Alice. So, the probability is the same as the first expression.

- **Expression 3**: here, we need to calculate the probability that Alice wins if she already won against Bob.
We must build a more accurate table that divides the probabilities of Alice and Bob having strength equal to 5 depending on laziness (strength = 5 can happen if the person has strength 5 and not lazy, or with strength = 10 but lazy).



		0,33	0,165	0,33	0,17
	Strength	**Bob 10**	**Bob 5**	**Bob 5**	**Bob 2.5**
0,33	**Alice 10**	0,11	0,055	0,11	0,057
0,165	**Alice 5**	0,055	0,027	0,054	0,028
0,33	**Alice 5**	0,11	0,054	0,11	0,056
0,17	**Alice 2,5**	0,057	0,028	0,056	0,029
				Tot.	0,996

Figure 4: Probabilities of strength detailed

Now, we have to compute the probability that Alice has a certain strength given that she won the first match. Using Bayes' rule:

$$P(\text{strength} = n \mid \text{win}) = \frac{P(\text{win} \mid \text{strength} = n)P(\text{strength} = n)}{P(\text{win})}$$

From the table, we can obtain:

$$P(\text{win} \mid \text{strength} = 10) = 0.11 + 0.055 + 0.11 + 0.057 + 0.027 + 0.054 + 0.028 = 0.441$$

$$P(\text{win} \mid \text{strength} = 5) = 0.054 + 0.11 + 0.056 + 0.029 = 0.249$$

Let $P(\text{strength} = 10) = P(\text{strength} = 5) = 0.5$:

$$P(\text{strength} = 10 \mid \text{win}) = 0.221 \qquad P(\text{strength} = 5 \mid \text{win}) = 0.124$$

Normalized:

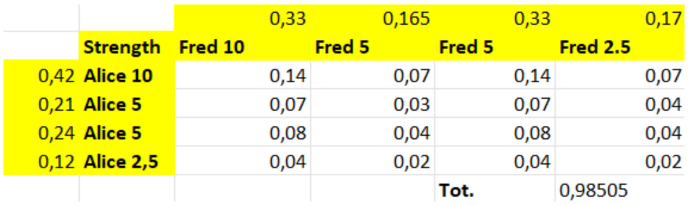
$$P(\text{strength} = 10 \mid \text{win}) = 0.64 \qquad P(\text{strength} = 5 \mid \text{win}) = 0.36$$

These results, multiplied with the probability of being lazy and not lazy (0.33 and 0.66), respectively return the new strength probabilities for Alice given that she won the first match:

$$P(\text{strength} = 10, \neg\text{lazy} \mid \text{win}) = 0.42 \qquad P(\text{strength} = 10, \text{lazy} \mid \text{win}) = 0.21$$

$$P(\text{strength} = 5, \neg\text{lazy} \mid \text{win}) = 0.24 \qquad P(\text{strength} = 5, \text{lazy} \mid \text{win}) = 0.12$$

We can now build the new table of Alice matching Fred given that she won against Bob:



		0,33	0,165	0,33	0,17
	Strength	**Fred 10**	**Fred 5**	**Fred 5**	**Fred 2.5**
0,42	**Alice 10**	0,14	0,07	0,14	0,07
0,21	**Alice 5**	0,07	0,03	0,07	0,04
0,24	**Alice 5**	0,08	0,04	0,08	0,04
0,12	**Alice 2,5**	0,04	0,02	0,04	0,02
				Tot.	0,98505

Figure 5: Probabilities given the first win

The probability of Alice winning against Fred:

$$P(Alice = \text{win}) = 0.14 + 0.07 + 0.14 + 0.07 + 0.03 + 0.07 + 0.04 + 0.04 + 0.08 + 0.04 + 0.02 = 0.74$$

The overall probability of winning is given by the multiplication of the probability that Alice wins the first match and the probability that she wins the second match:

$$P(Alice = \text{win both}) = 0.69 \cdot 0.74 = 0.51$$

- **Expression 4**: in order to get $\#t$ using the expression, we must calculate the probability that Alice and Jane wins against Bob and Fred. So, the probabilities are independent and we only need to multiply 0.69 and 0.69. Therefore:

$$P(Alice = \text{win}, Jane = \text{win}) = 0.4761$$

c) The probabilities of the last two expression are different because in the first one there is the same player (Alice) matching against two different players, while in the second there are different players (Alice and Jane) matching Bob and Fred. In the latter, the probabilities of winning in the two matches are independent.
As proof, we should consider that the `mem` procedure keeps in memory the strength of a person at each evaluation. When evaluating the match between Alice and Fred, the strength of Alice is already fixed. If Alice have already won the first match, there is an higher probability that her strength is equal to 10. This is the reason why the probability of getting $\#t$ is the expression 3 is higher than the expression 4.

Exercise 3 (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 `flip` p will return $\#t$ depending on the probability p, and $\#f$ otherwise (so, depending on $1-p$).
In order to obtain 5, the geometric distribution should give $\#f$, 5 times in a row and then a $\#t$. Each time, the result is independent from the previous one, so:

$$P((\texttt{geometric}\ p)=5)=(1-p)^5 p$$

Example, for $p=0.5$, the probability of getting 5 is $0.5^5 \cdot 0.5 = 0.015625$.

```
> (hist (repeat (lambda () (geometric 0.5)) 100))
```

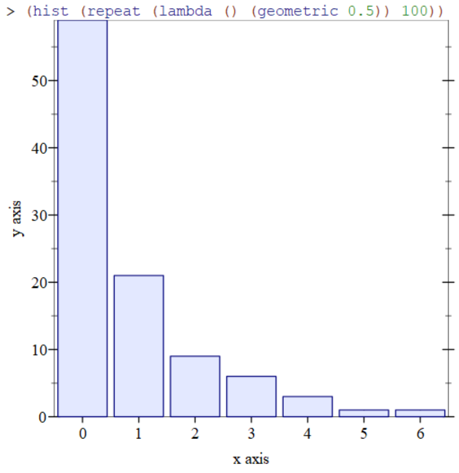


Figure 6: Probability distribution of `geometric` 0.5

While, using $p=0.2$, the probability of getting 5 is higher: $0.8^5 \cdot 0.2 = 0.065536$.

```
> (hist (repeat (lambda () (geometric 0.2)) 100))
```

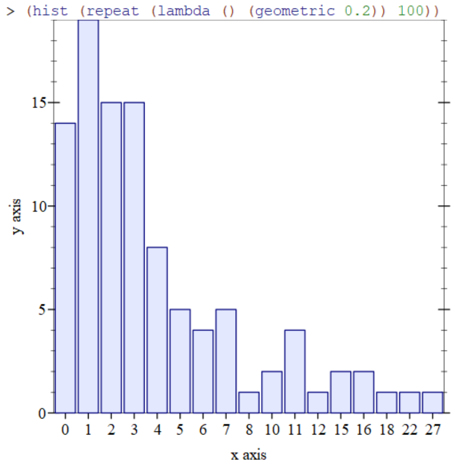


Figure 7: Probability distribution of `geometric` 0.2

Exercise 3 (7)

Convert the following probability table to a compact Church (Gamble) 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 is possible to find that probability of A being #t is fixed to 0.8, and B depends on A: 0.5 if A is #t, 0.3 otherwise.
This is given by the fact that $P(A = T, B = T) = P(A = T, B = F)$ and that we must find probabilities that sum to 1 for computing $P(A = F, B = T) = 0.06$ and $P(A = F, B = T) = 0.14$.
As proof:

$$P(A = T, B = T) = 0.8 \times 0.5 = 0.4$$
$$P(A = T, B = F) = 0.8 \times 0.5 = 0.4$$
$$P(A = F, B = T) = 0.2 \times 0.3 = 0.06$$
$$P(A = F, B = F) = 0.2 \times 0.7 = 0.14$$

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

Finally, the below histogram shows that the distribution is correct.

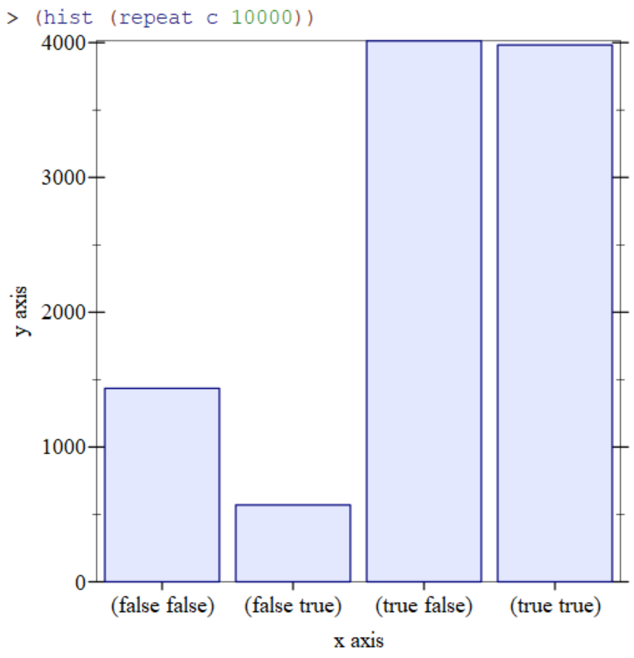


Figure 8: Distribution

Exercise 4 (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 The evaluation just returns the distribution as output (`normal-dist` 0.0 1.0), in fact we defined a new object that is a distribution object. `flip`, instead, is a sample of a probability distribution.
(`bernoulli-dist` p), (`normal-dist` μ σ) are distribution objects and not real numbers.

Exercise 4 (2)

Evaluate

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

Solution The first two are a distribution, because `dist?` returns #*t*. In fact, to check if something is a distribution-object, we use the procedure `dist?`.
(`dist? flip`) returns #*f*. This means that `flip` procedure is not a distribution. In fact, it is a sample of a probability distribution (Bernoulli distribution with probability 0.5).

Exercise 4 (3)

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

Solution The procedure flip is basically (λ () (`sample` (`bernoulli-dist` 0.5))), with 0 instead of #*f* and 1 instead of #*t*.
In fact, in order to produce probabilistic behaviours from a distribution, we use the procedure `sample`.

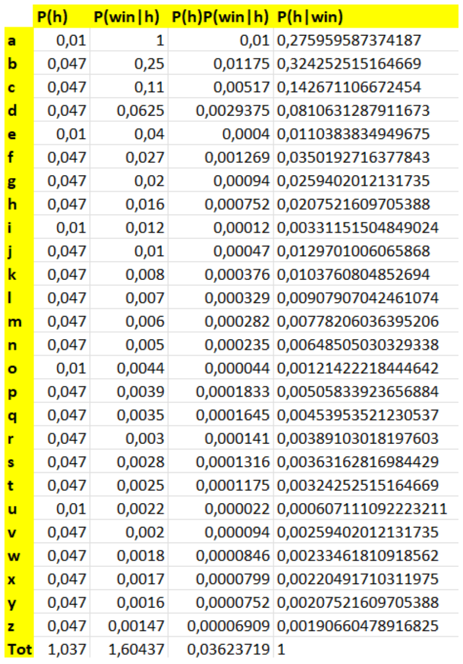
Exercise 5

Refer to exercise 4 from Probabilistic Models of Cognition.

Solution For the solution of this exercise, I will use the built-in **Church** script of the website.

a) In English, $p(h \mid \text{win})$ is explainable using the statement: given that Bob won, which letter did he probably have?

b) In the picture 9, $p(h \mid \text{win})$ for each hypothesis computed using Excel.



	P(h)	P(win|h)	P(h)P(win|h)	P(h|win)
a	0,01	1	0,01	0,275959587374187
b	0,047	0,25	0,01175	0,324252515164669
c	0,047	0,11	0,00517	0,142671106672454
d	0,047	0,0625	0,0029375	0,0810631287911673
e	0,01	0,04	0,0004	0,0110383834949675
f	0,047	0,027	0,001269	0,0350192716377843
g	0,047	0,02	0,00094	0,0259402012131735
h	0,047	0,016	0,000752	0,0207521609705388
i	0,01	0,012	0,00012	0,00331151504849024
j	0,047	0,01	0,00047	0,0129701006065868
k	0,047	0,008	0,000376	0,0103760804852694
l	0,047	0,007	0,000329	0,00907907042461074
m	0,047	0,006	0,000282	0,00778206036395206
n	0,047	0,005	0,000235	0,00648505030329338
o	0,01	0,0044	0,000044	0,00121422218444642
p	0,047	0,0039	0,0001833	0,00505833923656884
q	0,047	0,0035	0,0001645	0,00453953521230537
r	0,047	0,003	0,000141	0,00389103018197603
s	0,047	0,0028	0,0001316	0,00363162816984429
t	0,047	0,0025	0,0001175	0,00324252515164669
u	0,01	0,0022	0,000022	0,000607111092223211
v	0,047	0,002	0,000094	0,00259402012131735
w	0,047	0,0018	0,0000846	0,00233461810918562
x	0,047	0,0017	0,0000799	0,00220491710311975
y	0,047	0,0016	0,0000752	0,00207521609705388
z	0,047	0,00147	0,00006909	0,00190660478916825
Tot	1,037	1,60437	0,03623719	1

Figure 9: $p(h \mid \text{win})$ computed in Excel

Bayes' rule is applied using the sum of all the $p(h)p(\text{win} \mid h)$ as divider:

$$p(h \mid \text{win}) = \frac{p(h)p(\text{win} \mid h)}{\sum_k p(h)p(\text{win} \mid h)}$$

From the exercise, it is known that the probability of getting a vowel is different than the one of getting a consonant:

$$P(h = \text{vowel}) = 0.01 \qquad P(h = \text{consonant}) = 0.047$$

Furthermore, letting h denote the letter and letting $Q(h)Q(h)$ denote the numeric position of that letter (e.g., $Q(\text{a}) = 1$, $Q(\text{b}) = 2$, and so on), the probability of winning is:

$$P(\text{win} \mid h) = \frac{1}{Q(h)^2}$$

c) **`my-list-index`** is a procedure that returns the first index (*counter*) of a list (*haystack*) whose element is equal to an expression (*needle*).

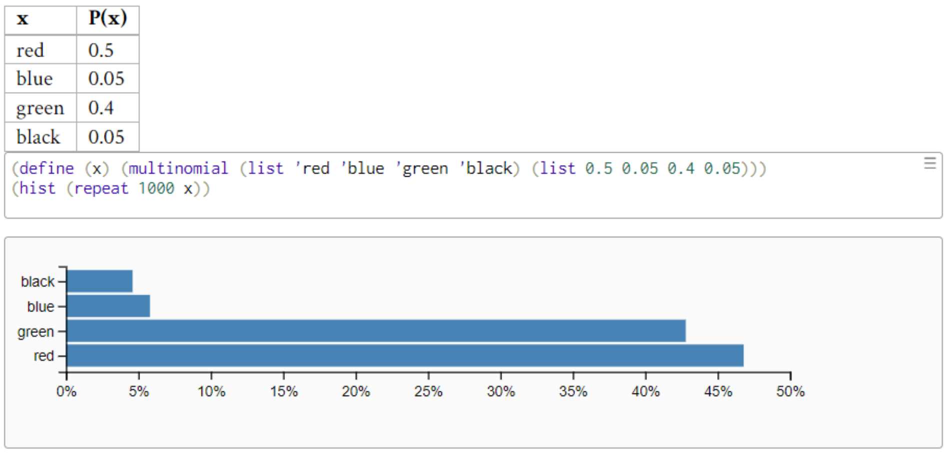
```
(define (my-list-index needle haystack counter)
  (if (null? haystack)
      'error
    (if (equal? needle (first haystack))
        counter
      (my-list-index needle (rest haystack) (+ 1 counter)))))
```

If we ran

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

the procedure would recursively search for an index starting from 1 of the list (apple banana) that is equal to mango. It checks if the list is empty, if so returns an error. Otherwise, checks if the first element of the list is equal to the *needle* argument. If so, returns the counter; otherwise the procedure is evaluated again using the rest of the list (without the first element) and with counter +1.
In this case, it returns the error of the procedure: it means the it checked all the list with no success.

d) Multinomial procedure samples an element from a list of categories with the respective probability. It requires as input a list of categories and a list of probabilities. It returns an element of the category using its probability.
For example, using 1000 repetition, the procedure got 47% of red (0.5 of probability), 42% of green (0.4 of probability), almost 6% of blue (0.05 of probability) and 5% of black (0.05 of probability). So the probability of getting a category are basically the ones given in input.

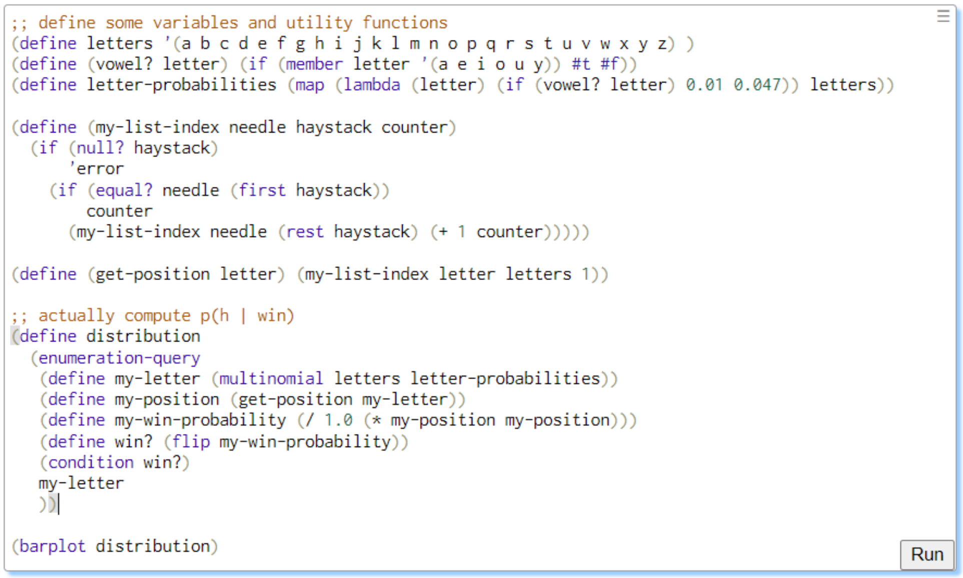


x	P(x)
red	0.5
blue	0.05
green	0.4
black	0.05

```
(define (x) (multinomial (list 'red 'blue 'green 'black) (list 0.5 0.05 0.4 0.05)))
(hist (repeat 1000 x))
```

Figure 10: Multinomial procedure

e) The letter with highest posterior probability is B. In English, it means that, knowing that Bob won, B is the most likely letter he got.
It is easy to notice that the Church and the Excel results (Figure 9) are equal.



```
;; 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))
   (define my-win-probability (/ 1.0 (* my-position my-position)))
   (define win? (flip my-win-probability))
   (condition win?)
   my-letter
   ))

(barplot distribution)
```

Figure 11: Casino' game

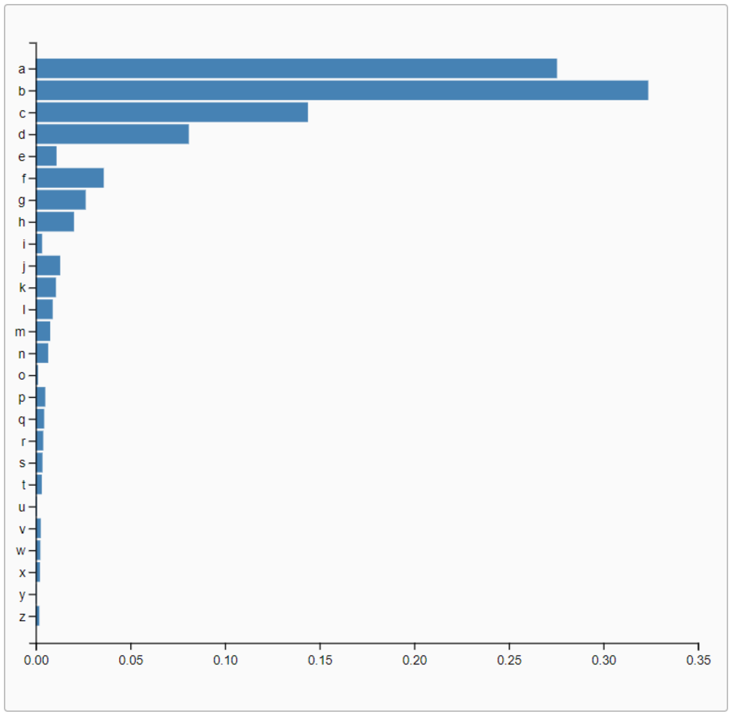
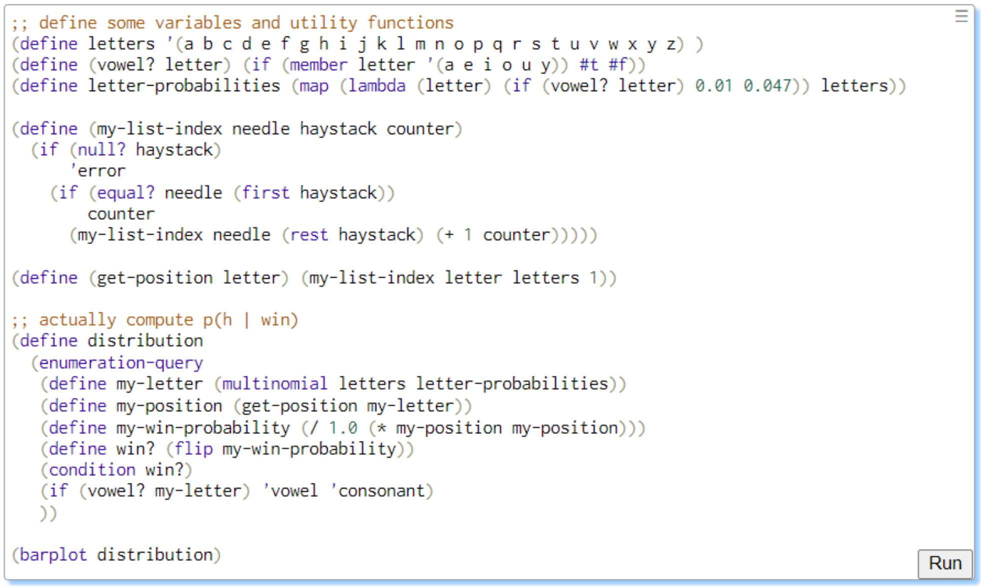


Figure 12: Barplot

f) It is possible to calculate the posterior probability of vowels and consonants adding an `if` clause to the query of the distribution using `vowel?` procedure. The `barplot` procedure shows that consonants have an higher posterior probability than vowels.


```
;; 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))
   (define my-win-probability (/ 1.0 (* my-position my-position)))
   (define win? (flip my-win-probability))
   (condition win?)
   (if (vowel? my-letter) 'vowel 'consonant)
   ))

(barplot distribution)
```

Figure 13: Casino' game vowels and consonants

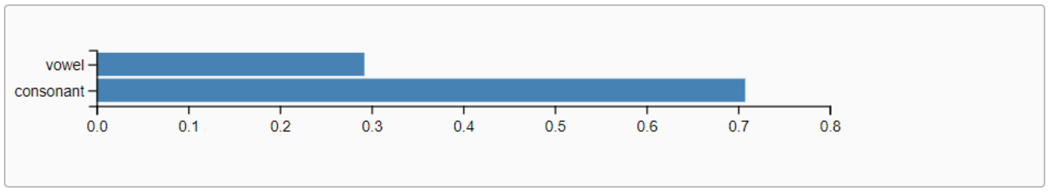


Figure 14: Barplot of vowels and consonants

g) Calculating the probability in Church or Racket is very efficient because it is possible to model big problems using just a few lines of code.
An alternative (which I personally used to manually calculate the probabilities) is Excel, but it would be a very long work if the problem does not have the appropriate size.
In general anyway, Excel is more understandable for a common user than Church/Racket, but I personally prefer Church because of my programming experience.

Exercise 6

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

```
(define baserate 0.001)

(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 The procedure `take-sample` uses rejection sampling to return samples that satisfy the condition of the `observe/fail` clause: it returns the value of A, given that D is greater than or equal to 2.

If we decrease the *baserate*, the probability of satisfying the condition decreases (because D is the sum of A, B, C and their probability of being 1 is the *baserate*), so the interpreter needs way more time to find correct samples. This is one of the main problem of rejection sampling: even if we are sure that our model can satisfy the condition, it will often take a very long time to find evaluations that do so.

We can plot the histograms varying the *baserate* using:

```
(hist (repeat (take-sample) 100))
```

Using it, it is worth to notice that using $baserate = 0.001$ the results are way slower to be computed.

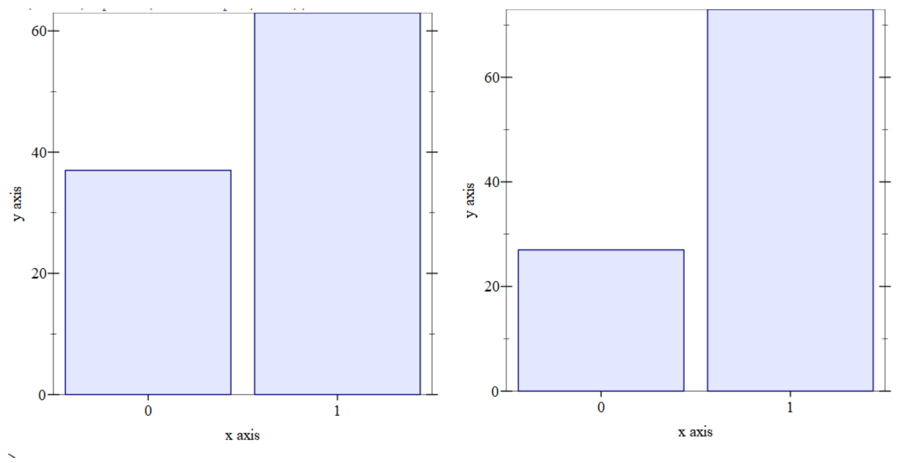


Figure 15: Left: *baserate* 0.01; right: *baserate* 0.001

Exercise 7 (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 *Proof sketch*: given $c : X \to D(X)$, with $X = \{x_1, ..., x_n\}$, we construct the matrix P_c as $P_c(i, j) = c(x_i)(x_j)$. Vice versa, given P, we define $c_P(x_i)(x_j) = P(i, j)$.

Given that $X = \{x_1, ..., x_n\}$ is a set of states, so P is a stochastic matrix and we know that:

$$c_P(x_i)(x_j) = P(i, j)$$

Using the Definition 2 of the stochastic matrix, each entry of P belongs to $[0, 1]$:

$$\forall i, j.\ \ P(i, j) \in [0, 1]$$

And we know that, using the same definition:

$$\forall i.\ \ \sum_j P(i, j) = 1$$

We already stated that $P(i, j) = c_P(x_i)(x_j)$, so:

$$\forall i.\ \ \sum_j c_P(x_i)(x_j) = 1$$

Finally, it means that:

$$c_P(x) \in D(X) \quad \forall x \in X$$

Given that $D(X)$ is the set of all distributions over X, $c_P(x)$ is a distribution.

On the other hand, if we consider $c : X \to D(X)$ a MC, $P_c(i,j) = c(x_i)(x_j)$ is a stochastic matrix. It follows that:

$$c_{P_c}(x_i)(x_j) = P_c(i,j) = c(x_i)(x_j) \quad \forall i,j$$
$$P(i,j) = c_P(x_i)(x_j) = P_{c_P}(i,j) \quad \forall i,j$$

And therefore:

$$c_{Pc} = c \qquad P_{cP} = P$$

Exercise 7 (2)

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

Solution Using the Definition 4, given a MC $c : X \to D(X)$, we define the map $c^* : D(X) \to D(X)$ by $c^*(\phi)(y) = \sum_x \phi(x) \cdot c(x)(y)$. Substituting we obtain:

$$c^*(\delta_x) = \sum_x \delta_x(x) \cdot c(x)(y) = \delta_x(x) \cdot c(x)(y) + \sum_{x \neq x} \delta_x(x) \cdot c(x)(y)$$

We know that $\delta_x(x)$ is equal to 1 for $x = x$, 0 otherwise. Therefore:

$$c^*(\delta_x) = \delta_x(x) \cdot c(x)(y) = c(x)(y)$$

That is equal to say:

$$c^*(\delta_x) = c(x)$$

Exercise 7 (3)

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

Solution Again, thanks to the Definition 4:

$$c^*(\psi) = \sum_x \psi(x) \cdot c(x)(y)$$

Or, given a set of states $X = \{x_1, ..., x_n\}$:

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

We know that $P_c(i,j) = c(x_i)(x_j)$ and therefore:

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

Or, equally:

$$c^*(\psi) = \psi P_c$$

Exercise 7 (4)

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

Solution For the Definition 5 if ψ satisfies DBC then:

$$\forall x, y. \quad \psi(x) \cdot P(x,y) = \psi(y) \cdot P(y,x)$$

It means that:

$$\sum_x \psi(x) \cdot P(x,y) = \sum_x \psi(y) \cdot P(y,x)$$
$$\sum_x \psi(x) \cdot P(x,y) = \psi(y) \sum_x P(y,x)$$
$$\sum_x \psi(x) \cdot P(x,y) = \psi(y)$$
$$\psi P = \psi \qquad \psi \text{ is stationary for } P$$