# LAAI - M2 - Homework

Michele Milesi

March 2022

**Abstract**

## Exercise 1

### Exercise 1 (4)

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

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

**Solution** The function `a-plus-abs-b` defined above takes in input 2 parameters and returns the sum $a + |b|$. The formal parameters of the fuction are `a` and `b`, while the body of the fuction is composed by a a compound expression. The operator of the expression is a compound expression, indeed, it depends on the value of the parameter `b`, in particular, if the value of b is greater than zero, then it is performed a sum between the operands, while if it is less than or equal to zero, then it is performed a substraction.

When this procedure is called, the formal parameters are substituted by the actual parameters, e.g. when we call the procedure as follows: `(a-plus-abs-b 5 2)` all the instances of `a` in the body of the function are substituted by the value 5 and all the instances of `b` are substituted by 2 and then the body of the function is evaluated. Considering that the operator is a compound expression, the interpreter firts evaluates it. In particular it is a conditional expression, so the interpreter evaluates the predicate `(< b 0)` and if the condition is true, then it will evaluate the *consequent*, otherwise it evaluates the *alternative*. In the previous example, after evaluating the conditional expression the expression become `(+ 5 2)`. At this point the interpreter evaluates the *operator*: since it is a primitive procedure, the interpreter evaluates all the *operands* and it applies the operator to the *arguments* (i.e. the value of the operands). In the example above, it simplifies the expression with `7`. Now the interpreter cannot do any simplification of the expression, indeed, it has to handle a primitive expression, so it does not perform any computation step and it returns the computed value.

### 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 behaviour will Ben observe with an interpreter that uses applicative-order evaluation? What behaviour will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form `if` is the same whether the interpreter is using normal or applicative order: the predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression).

**Solution** In applicative-order evaluation, when a procedure is called, the arguments are evaluated first and then applied to the procedure; while in normal-order evaluation the arguments are not evaluated before the procedure call, but they are evaluated when the body of the function is evaluated. In this example, an interpreter which uses applicative-order evaluation first evaluats the arguments, so it evaluates `0` (nothing to do) and `(p)` which causes a loop, indeed, the procedure `p` calls itself and it does not terminate. On the other side, an interpreter which uses normal-order evaluation does not evaluate the arguments before the evaluation of the body of the function, but it passes them without any computation. In this case the procedure call `(test 0 (p))` returns the value `0`, indeed, the interpreter evaluates the conditional expression and, since the formal parameter `x` is substituted by the value `0`, the condition is true and the consequent (i.e. `0`) is returned.

We can observe that the applicative-order evaluation can be useful when the computation of the arguments is heavy and the arguments are used many times in the body of the function, but some arguments can be evaluated even if they are not used and this can cause, as in this particular case, some problems. Instead, normal-order evaluation can be useful when the are used few times or not used, but if they are used many times and their computation is heavy, then the performances are worse than applicative-order evaluation.

# Exercise 2

## Exercise 2 (1.35)

Show that the golden ratio $\varphi$ is a fixed point of the transformation $x \mapsto 1 + \frac{1}{x}$, and use this fact to compute $\varphi$ by means of 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))
```

**Solution** The *golden ratio* is defined as follows:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

It is the fixed point of the transformation $x \mapsto 1 + \frac{1}{x}$ indeed if we apply the transformation to $\varphi$ we obtain:

$$1 + \frac{1}{\varphi} = 1 + \frac{1}{\left(\frac{1+\sqrt{5}}{2}\right)} = 1 + \frac{2}{1+\sqrt{5}} = \frac{3+\sqrt{5}}{1+\sqrt{5}} = \frac{3+\sqrt{5}}{1+\sqrt{5}} \cdot \frac{1-\sqrt{5}}{1-\sqrt{5}} = \frac{-2-2\sqrt{5}}{-4} = \frac{1+\sqrt{5}}{2} = \varphi$$

The transformation can be defined in Racket as follows:

```
; definition of transformation x -> 1 + 1/x
(define (transformation x)
  (+ 1 (/ 1 x)))
```

```
; definition of golden ratio
(define phi (/ (+ 1 (sqrt 5)) 2))
```

And the procedure `fixed-point` is used to compute the *golden ratio*, as shown below, we can observe that the procedure `fixed-point` computes $\varphi$ with a good level of approximation.

```
> (fixed-point transformation 1.1)
1.6180364726455159
> phi
1.618033988749895
```

## 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 $x \mapsto \frac{\log(1000)}{\log(x)}$. (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** First of all the procedure `fixed-point` has been modified in order to print the sequence of approximations it generates.

```
(define (fixed-point-print-seq f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (display guess)
    (newline)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

Listing 1: Procedure `fixed-point` which prints the sequence of approximations it generates

Then the transformation $x \mapsto \frac{\log(1000)}{\log(x)}$ has been defined in racket as follows (both with and without average damping):

```
(define (log-transformation x)
  (/ (log 1000) (log x)))
```

Listing 2: Transformation without average-damping

```
; definition of procedure which computes the average
(define (average x y)
  (/ (+ x y) 2))

; definition of transformation x -> log(1000) / log(x)
; with average damping
(define (log-transformation-avg-dmp)
  (fixed-point-print-seq
   (lambda (x)
     (average x (log-transformation x)))
   1.1))
```

Listing 3: Transformation with average damping

After that the two procedures are used to make a comparison between the number of steps, in both cases the initial guess is 1.1 and it can be show how the procedure with average damping takes less time to converge to the solution. Below the approximation of the two methods are shown, the approximation without average damping takes 37 steps, while the approximation with average damping takes 13 steps.

```
; without average damping
> (fixed-point-print-seq
    log-transformation 1.1)
1.1
72.47657378429035
1.6127318474109593
14.45350138636525
2.5862669415385087
7.269672273367045
3.4822383620848467
5.536500810236703
4.036406406288111
4.95053682041456
4.318707390180805
4.721778787145103
4.450341068884912
4.626821434106115
4.509360945293209
4.586349500915509
4.535372639594589
4.568901484845316
4.546751100777536
4.561341971741742
4.551712230641226
4.558059671677587
4.55387226495538
4.556633177654167
4.554812144696459
4.556012967736543
4.555220997683307
4.555743265552239
4.555398830243649
4.555625974816275
4.555476175432173
4.555574964557791
4.555509814636753
4.555552779647764
4.555524444961165
4.555543131130589
4.555530807938518
4.555538934848503
```

```
; with average damping
> (log-transformation-avg-dmp)

1.1
36.78828689214517
19.352175531882512
10.84183367957568
6.870048352141772
5.227224961967156
4.701960195159289
4.582196773201124
4.560134229703681
4.5563204194309606
4.555669361784037
4.555558462975639
4.55553957996306
4.555536364911781
```

## Exercise 2 (1.37)

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

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

As an example, one can show that the infinite continued fraction expansion with the $N_i$ and the $D_i$ all equal to 1 produces $\frac{1}{\varphi}$, where $\varphi$ is the golden ratio. 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 $\frac{1}{\varphi}$ using

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

for succesive values ok `k`. How large must you make `k` in order to get an approximation that is accurate to 4 decimals 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**

a. The procedure `cont-frac` has been defined with a recursive process. The function has 3 parameters: *(i)* `n` which is the function that returns the element $N_i$ of the continued fraction; *(ii)* `d` which is the function that returns the element $D_i$ of the continued fraction; *(iii)* `k` which is the number of iterations to be performed. In the body of the function is defined the local recursive procedure `cont-frac-rec` which is responsable for computing recursivly the continued fraction. It has two formal parameters: `k` and `counter` which is the counter of the performed iterations; when the `counter` reaches `k` (i.e. the number of iterations to be performed) the base case is reached and the returned value is computed. The procedure `cont-frac` calls the procedure `cont-frac-rec` with initial parameters `k` and 0, in this way we are sure that the number of iterations performed will be k (with `counter` from 1 to `k`). The code is shown below:

```
(define (cont-frac n d k)
  (define (cont-frac-rec k counter)
    (if (= counter k)
        (/ (n counter) (d counter))
        (/ (n counter)
           (+ (d counter)
              (cont-frac-rec k (+ counter 1))))))
  (cont-frac-rec k 1))
```

Listing 4: Recursive procedure `cont-frac`

Then it has been tested by approximating $1/\varphi \sim 0.6180$. When $k = 11$ or grater, the approximation is accurate to 4 decimals places.

b. Since before the procedure generates a recursive process, it has been rewritten in order to generate an iterative process. The procedure defines the local procedure `iter` which takes in input the number of iterations to be performed (`k`) and partial result computed so far. The idea is to begin from the last fraction (i.e. $N_k/D_k$) and then proceding backward to compute all the other fractions. In particular, the idea is to compute the quantity $D_{i-1} + \frac{N_i}{Q_i}$ where $Q_i$ is the quantity computed so far. Indeed, the procedure `iter` is called with initial parameters `k` and $Q_k = D_k$, after the first iteration, the computed value $Q_{k-1} = D_{k-1} + \frac{N_k}{D_k}$. In the last step (i.e. $k = 1$) the procedure returns the ration $N_1/Q_1$ that is exaclty the continued fraction to be computed.

```
(define (cont-frac-iter n d k)
  (define (iter k res)
    (if (= k 1)
        (/ (n k) res)
        (iter (- k 1)
              (+ (d (- k 1))
                 (/ (n k) res)))))
  (iter k (d k)))
```

### 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 $1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, \ldots$. Write a program that uses your `cont-frac` procedure from Exercise 1.37 to approximate $e$, based on Euler's expansion.

**Solution** The two procedures `cont-frac` and `cont-frac-iter` were be used to approximate $e$ in order to verify that the results matched. The procedure `euler-number` uses the procedure `cont-frac` (which generates a recursive process), while the procedure `euler-number-iter` uses the procedure `cont-frac-iter`. To get the values of $N_k$ and $D_k$, two procedures have been defined: `get-n` and `get-d`. The first one is trivial and returns 1 at each iteration step, while the latter is more complex. To get the element of the sequence $1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, \ldots$ it is possible to use the function $seq : \mathbb{N}^+ \to \mathbb{N}^+$ described below:

$$seq(i) = \begin{cases} i & \text{if } 1 \leq i \leq 2 \\ \left(\frac{i-2}{3} + 1\right) \cdot 2 & \text{if } i - 2 \ mod \ 3 = 0 \\ 1 & \text{otherwise} \end{cases}$$

So we defined the procedures `get-n` and `get-d` as follows:

```
(define (get-n i) 1)
(define (get-d i)
  (if (= (remainder (- i 2) 3) 0)
      (* (+ (quotient
             (- i 2)
             3)
            1)
         2)
      (if (<= i 2) i 1)))
```

Listing 5: Definition of the procedures `get-n` and `get-d`

Since the sequence of $D_i$ is used to approximate $e-2$, we can approximate $e$ by approximating $e-2$ and than by adding 2 to the computed quantity. This is the way the procedures `euler-number` and `euler-number-iter` approximate $e$.

```
(define (euler-number k)
  (+ (cont-frac get-n get-d k) 2))

(define (euler-number-iter k)
  (+ (cont-frac-iter get-n get-d k) 2))
```

Both the procedures return the value $23225/8544 \sim 2.7182 \approx e$.

## Exercise 3

### 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** In the first program we are defining `foo` as a variable and we are assigning it the value of the evaluation of the expression `(flip)`, indeed, the value of `foo` is either `#t` or `#f`. After that we create a list which contains three time the value of `foo`, so if `foo` has value `#t`, then we are defining a the list: `'(#t #t #t)`; otherwise we are creating the list: `'(#f #f #f)`.

In the second program, we are defining a new procedure `foo` which is a wrapper for the procedure `flip`. This means that each time the procedure `foo` is called, also the procedure `flip` is called. For this reason, when we define the list, the expression `flip` is evaluated three times and, since it is a non-deterministic procedure, the three elements of the list can be different.

## Exercise 3 (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. First of all, the procedures are defined and the interpreter associates the name of the procedures with their definition in the *global environment*. Then the interpreter evaluates the four expressions in the order in which they are written:

**Expression 1** `(winner '(alice) '(bob))`

The first step is to retrieve the body of the procedure `winner` and then the formal parameters are substituted by the actual parameters `'(alice)` and `'(bob)`. Now the interpreter has to evaluate the following expression:

```
(if (< (total-pulling '(alice)) (total-pulling '(bob)))
    '(bob)
    '(alice)))
```

Listing 6: Body of procedure `winner` with actual parameters

The following step is to evaluate the conditional expression, so it starts by evaluating the *predicate*: `(< (total-pulling '(alice)) (total-pulling '(bob)))`. The interpreter has to deal with the primitive predicate `<`, so it needs to evaluate the arguments and than apply the predicate to the evaluated arguments. The first argument to evaluate is `(total-pulling '(alice))`: the body of the procedure `total-pulling` is retrieved in the *global environment* and it is substituted by the expression, then the actual parameter is applied to the body of the procedure. The expression `(total-pulling '(alice))` becomes as follows:

```
(sum
 (map (lambda (person) (if (lazy person)
                            (/ (strength person) 2)
                            (strength person)))
      '(alice)))
```

Listing 7: Body of procedure `total-pulling` with actual parameters

The procedure `sum` takes as argument a list of numbers and returns the sum over the elements of the list. In this case the argument of the procedure is another expression, so the interpreter has to evaluate the argument and then apply it to the procedure `sum`. Now the expression to be evaluated is the following:

```
(map (lambda (person) (if (lazy person)
                           (/ (strength person) 2)
                           (strength person)))
     '(alice))
```

Listing 8: Argument of procedure `sum` in Listing 7

The procedure `map` takes as arguments a *lambda* expression and a list. The procedure applies the *lambda* expression to each element of the list and returns the list of the results of the *lambda* expression on the elements of the list in input. So the first step is to evaluate the *lambda* expression with argument `'alice`. The resulting expression is the body of the *lambda* expression with actual parameter `'alice`:

```
(if (lazy 'alice) (/ (strength 'alice) 2) (strength 'alice))
```

Listing 9: Application of `map` prcedure of Listing 8

Since the interpreter has to evaluate a conditional expression, the interpreter firsts evaluate the *predicate*. To evaluate the call `(lazy 'alice)`, the interpreter first takes the body of the procedure and then substitutes all the occurences of formal parameter by the value of the actual parameter. The resulting expression is:

```
(flip (/ 1 3))
```

Listing 10: Body of procedure `lazy` when evaluating Listing 9

The procedure `flip` is a probabilistic procedure, it can take as parameter the probability to return `#t`. In our case, the interpreter has to evaluate the expression `(/ 1 3)` and than apply the result to the procedure `flip`. We suppose that the result of the call `(flip 1/3)` is `#f`, so the result of the evaluation of the *predicate* in the conditional expression is `#f`. Since the *predicate* is false, the interpreter has to evaluate the *alternative* expression `(strength 'alice)`. It is a procedure call, so the interpreter has to retrieve the body of the procedure and then to substitute the formal parameter by the actual one. The result is that the interpreter now has to evaluate the following expression:

```
(if (flip) 5 10)
```

Listing 11: Body of procedure `strength` when evaluating Listing 9

The procedure `strength` is a particular type of procedure, indeed, it is memoized through the procedure `mem`. This means that the first time the procedure is called, the result of the call is memorized and each time the procedure is called with same the same argument, the interpreter does not evaluate the function, but it retireves the already computed result in the *global environment*. This is the first time that `(strength 'alice)` is evaluated, so the procedure has to be evaluated in a "standard" way. The body of the procedure is composed by a conditional expression, the procedure `flip` is evaluated and we suppose that the result is `#t`, so the procedure `strength` returns the value 5.

Thus, the expression `(strength 'alice)` is evaluated with 5; since the second argument of the procedure `map` is a list with only one argument (i.e. `'alice`) the result is a list with only one element (i.e. 5). This list is the argument of the procedure `sum` and the interpreter has to evaluate the following expression:

```
(sum '(5))
```

Listing 12: Expression to be evaluated after evaluating the expression in Listing 8

Which is evaluated with 5. This is the result of the evaluation of the call `(total-pulling '(alice))`, now the interpreter has to evaluate the second argument of the expression `(< 5 (total-pulling '(bob)))`, since 5 is a primitive expression and it cannot be reduced anymore. The evaluation of the expression `(total-pulling '(bob))` is the same as the expression `(total-pulling '(alice))`, but, since there are some probabilistic cases, the result can be different. In particular, we suppose that the evaluation of `(lazy 'bob)` returns the value `#t`, so the interpreter has to evaluate the expression `(/ (strength 'bob) 2)` instead of `(strength 'bob)`. The interpreter has to evaluate a primitive procedure, so it has to evaluate first the operands and then to apply the operator to them. We suppose that the evaluation of `(strength 'bob)` returns the value 5. It is important to notice that it is the first time that the procedure `strength` is called with actual parameter `'bob`, so the evaluation is "standard" and the result of the call is memorized in the *global environment*. So the expression `(/ (strength 'bob) 2)` is evaluated with 2.5.

Now the situation is represented in the following way:

```
(if (< 5 2.5)
    '(bob)
    '(alice)))
```

Listing 13: Expression to be evaluated after evaluating `(total-pulling 'bob)`

Since $5 > 2.5$, the *predicate* is false and the value `'(alice)` is returned. Thus the *expression 1* is evaluated with the value `'(alice)`.

**Expression 2** `(equal? '(alice) (winner '(alice) '(bob)))`

The predicate `equal?` returns `#t` when the two arguments are euqal, it return `#f` otherwise, so the interpreter has to evaluate the two *operands* and then to compare them. The first *operand* is a quoted data object, so it is a list which contains the symbol `'alice`. The evaluation of the second *operand* is identical to the evaluation of *expression 1*.

Since there are some probabilistic procedures which have to be evaluated, the result of the evaluation could be different from the previous one. In particular we suppose that in this case the call `(lazy 'alice)` returns `#f` and the call `(lazy 'bob)` returns `#f`. Since the procedure `strength` is memoized, the calls `(strength 'alice)` and `(strength 'bob)` return the same values as before, i.e. Both the calls return the value 5.

The two teams `'(alice)` and `'(bob)` have the same value (i.e 5) returned by the procedure `total-pulling`, so the evaluation of the procedure `winner` returns `'(alice)`. Since the two arguments of the predicate `equal?` are equal, the evaluation of *expression 2* is `#t`.

**Expression 3** The *expression 3* is the following one:

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

The logical composition **and** of the two predicates `equal?` is evaluated by first evaluating the first operand. If the evaluation returns `#t`, then the second operand is evaluated; if it is `#t`, then is evaluated the third one (if present) and so on and so forth. If all the values of the operands are `#t`, then the evaluation of the expression is `#t`, otherwise when one operand is evaluated as `#f`, then the expression is evaluated as `#f` and the follwing operands are not evaluated.

So the interpreter evaluates the first `equal?` expression and the evaluation is the same as the previous case. We suppose now that `'alice` is *lazy* and `'bob` is not *lazy*. In this case the procedure `winner` returns the value `'(bob)`, so the evaluation of the first operand returns the value `#f` since ('(alice)) is different from ('(bob)). Thus the evaluation of the **and** is stopped and the returned value is `#f`.

**Expression 4** The *expression 4* is the following one:

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

We have already dissussed in detail about the evaluation of the procedures and predicates in this expression, so the main argumentation concerns the results of the evaluation of the *expression 4*. Let us suppose that `'alice` is not *lazy* and `'bob` is *lazy*. The first argument of the **and** predicate is *true*, so the interpreter has to evaluate the second expression.

Since both the calls `(strength 'jane)` and `(strength 'fred)` have been evaluated, their evaluation is happens in a "standard" way and their values are memorized into the *global environment*. Let us suppose both `'jane` and `'fred` are strong, so the value returned by the procedure `strength` is 10 and we suppose that the evaluation of `(lazy 'jane)` is `#f` and the evaluation of `(lazy 'fred)` is *ture*.

Thus the evaluation of `(winner '(jane) '(fred))` returns `'(jane)` which is equal to the first argument of the predicate `equal?`. Since both the operands of the **and** predicate are *true*, the evaluation of the *expression 4* is `#t`.

b. To compute the probability for each possible return value from each expression it has been decided that the execution of the four expressions is sequential, thus it is necessary to make some assumption when the procedure `strength` is called, indeed, it is a memoized procedure, so the first evaluation influences the following calls.

**Expression 1** When the interpreter has to evaluate this first expression, the procedure `strength` has never called before for both `'alice` and `'bob`, so they can assume either the value 5 or 10 after evaluating the procedure `strength`. Furthermore they can result either *lazy* or *not lazy*, thus there are 16 combinations of values that the two teams can assume. In particular, it is possible to build a table whose rows and columns are composed by all the possible combinations of the outcomes of the probabilistic procedures `strength` and `lazy`, in particular the rows are all the possible outcomes of these procedures for one team (i.e. `('alice)`), while the columns are all the possible outcomes of these procedures for the other team (i.e. `'(bob)`). Each cell of the table contains the probability to get that particular case. The computed probability are shown in Table 1. The table shows all the possible combinations of values that `'alice` and `'bob` can get, the cells coloured in Violet are the cases where `'alice` wins over `'bob`, instead, the cells coloured in Orange are those where `'bob` wins.

Thus to compute the probability that `'(alice)` wins it is sufficient to sum all the probabilities coloured in Violet, while to compute the probability that `'(bob)` wins it is sufficient to summ all the probabilities coloured in Orange. The result is the following:

$$P(winner = alice) = 25/36$$

$$P(winner = bob) = 11/36$$

| '(alice) \ '(bob) | strong lazy | strong ¬lazy | ¬strong lazy | ¬strong ¬lazy |
|---|---|---|---|---|
| **strong ∧ lazy** | 1/36 | 1/18 | 1/36 | 1/18 |
| **strong ∧ ¬lazy** | 1/18 | 1/9 | 1/18 | 1/9 |
| **¬strong ∧ lazy** | 1/36 | 1/18 | 1/36 | 1/18 |
| **¬strong ∧ ¬lazy** | 1/18 | 1/9 | 1/18 | 1/9 |

Table 1: Probabilities of all possible cases of the *expression 1*. The cells coloured in Violet are the ones where `'alice` wins against `'bob`.

**Expression 2** Since the procedure `strength` is memoized, the results of the evaluation of the previous expression influences the the probability for each possible outcome of this expression. In particular, the values of the strength of both `'alice` and `'bob` are fixed, so the probabilistic behaviour of the evaluation of this expression depends only on the result of the procedure `lazy`, so the table of probabilities has less entries.

Let us suppose that during the evaluation of the *expression 1* the strength assigned to both `'alice` and `'bob` is 5. Thus the table contains only four different cases: *(i)* Both `'alice` and `'bob` are *lazy*; *(ii)* `'alice` is *lazy* and `'bob` is *lazy*; *(iii)* `'alice` is *not lazy* and `'bob` is *lazy*; *(iv)* Both `'alice` and `'bob` are *not lazy*. The Table 2 represents the different probability of all possible cases: also in this case, the cells coloured in Violet are the ones where `'alice` wins. The result is the following:

$$P(winner = alice \mid strength(alice) = 5 \wedge strength(bob) = 5) = 7/9$$

Thus the probability that the *expression 2* returns the value `#t` is 7/9 and the probability that it returns the value `#f` is 2/9.

| '(alice) \ '(bob) | lazy | ¬lazy |
|---|---|---|
| **lazy** | 1/9 | 2/9 |
| **¬lazy** | 2/9 | 4/9 |

Table 2: Probabilities of all possible cases of the *expression 2*. The cells coloured in Violet are the ones where `'alice` wins against `'bob`. It is important to remeber that both `'alice` and `'bob` are *weak* (i.e. the value assigned to their strength is 5).

**Expression 3** This expression contains the logical operator `and`, so the value returned is `#t` if both the operands are *true*. The first operand is euqal to the previous expression, while the second operand is different: the first team (i.e. `'(alice)`) has already an assigned value to the *strength*, while the second team (i.e. `'(fred)`) has not any assigned value to the *strength*.

It is important to remeber that the second operand is not interpreted if the first one is evaluated as `#f`, but this fact does not influence the probabilities of the outcomes of this expression, it will influence the probabilities of the *expression 4*.

The expression can be divided in two independent parts: the first one is the match between `'(alice)` and `'(bob)` and the second part is the match between `'(alice)` and `'(fred)`. Thus it is possible to compute the probabilities of these parts and then multiply them to get the final probability.

The first probability to compute is euqal to the probability of `expression 2`, so we can refer to the Table 2 and the probability to get the value *true* is 7/9. Regarding the second operand, it is necessary to consider different cases, indeed, the interpreter has to evaluate only the laziness of `'alice` and it has to evaluate both the strength and laziness of `'fred`. For this reason it is possible to compute a table with two rows (i.e. the number possible values of the laziness of `'alice`) and four columns (i.e. the number of possible combinations of the strength and laziness of `'fred`). The Table 3 contains the probabilities for all possible cases of the second part of the *expression 3*.

| '(alice) \ '(fred) | strong lazy | strong ¬lazy | ¬strong lazy | ¬strong ¬lazy |
|---|---|---|---|---|
| (¬ strong) lazy | 1/18 | 1/9 | 1/18 | 1/9 |
| (¬ strong) ¬lazy | 1/9 | 2/9 | 1/9 | 2/9 |

Table 3: Probabilities of all possible cases of the second part of the *expression 3*. The cells coloured in Violet are the ones where `'alice` wins against `'fred`. It is important to remeber that `'alice` is *weak* (i.e. the value assigned to her strength is 5).

The result is that the probability that the second part of the expression is evaluated with *true* is 1/2, indeed, it is returned `#t` when `'(alice)` wins against `'(fred)`. Thus we can compute the final probability by multipling the probabilities of the two parts. Thus it results that the probability to get the value `#t` is $7/9 \cdot 1/2 = 7/18$. While the probability to get the value `#f` is given by $P(outcome = false \mid strength(alice) = strength(bob) = 5) = 1 - 7/18 = 11/18$.

**Expression 4** The computation of the probability for each possible result of this expression is strongly influenced by the result of the previous expression, indeed, if the second part of the *expression 3* is evaluated, then it is necessary to compute the probability conditioned by the value assigned to the *strength* of `'fred`. Otherwise the probabilities to compute are conditioned only by the values assigned to the *strength* of `'alice` and `'bob`.

Let us suppose that the resutl of the first part of the previous expression is `#f`, so the second part has not been evaluated and the value of the *strength* of `'fred` is not fixed. The computation of the probabilities for this expression is similar to the previous one, indeed, there are two independent parts logically connected by the `and` predicate. The first part is equal to the *expression 2* (Table 2), while the second part is equivalent to the *expression 1*, since both `'jane` and `'fred` have never been uesd as parameter of the procedure `strength`. The Table 4 represents all the possible cases of the second part of this expression and it is equal to the Table 1. The table is shown for greater clearly.

| '(jane) \ '(fred) | strong lazy | strong ¬lazy | ¬strong lazy | ¬strong ¬lazy |
|---|---|---|---|---|
| strong ∧ lazy | 1/36 | 1/18 | 1/36 | 1/18 |
| strong ∧ ¬lazy | 1/18 | 1/9 | 1/18 | 1/9 |
| ¬strong ∧ lazy | 1/36 | 1/18 | 1/36 | 1/18 |
| ¬strong ∧ ¬lazy | 1/18 | 1/9 | 1/18 | 1/9 |

Table 4: Probabilities of all possible cases of the second part of the *expression 4*. The cells coloured in Violet are the ones where `'jane` wins against `'fred`.

The result is that the *expression 4* returns the value `#t` with probability $7/9 \cdot 25/36 = 175/324 \approx 0.54$, while the probability to get the value `#f` is $1 - 175/324 = 149/324 \approx 0.46$.

c. The computed probabilities of *expression 3* and *expression 4* are different because in the first case the strength of three teams is certainly fixed, indeed, the strength of both `'alice` and `'bob` is assigned during the evaluation of the *expression 1*. Instead, in the second case, the strength is assigned certainly to two teams (i.e. `'(alice)` and `'(bob)`) and it can happen that it is assigned also to `'fred`. In this case the probability of *expression 4* might turn out equal to the probability computed for *expression 3*, indeed, if the strength of `'alice` is equal to the strength of `'fred`, then the two probabilities are equal. Quite the opposite, if the second part of the *expression 3* is not evaluated (i.e. as happened in points *a* and *b*) the two probabilities are different because in *expression 3* the second part is a conditional probability, while in *expression 4* the second part is not a conditional probability because the values of the strength of `'jane` and `'fred` are not evaluated.

In conclusion, the probabilities of *expression 3* and *expression 4* can be equal or different, they depend on the probabilistic behaviour of the procedures `lazy` and `strength`.

## 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** The procedure computes the number of consecutive *false* (`#f`) results. Since each coin toss (i.e. `flip`) is independent, the probability of getting five consecutive *false* results (and the sixth one *true*) is given by:

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

The formula comes from the fact that the probability of getting a *true* value from the procedure `flip` is given by $p$, so the probability of getting a *false* value from `flip` is $(1-p)$. So the procedure `geometric` computes the number of trials needed to get the first occurence of success (i.e. `#t`). Each trial has the same probability of success $p$. For this reason the computed probability is equivalent to the geometric distribution with success probability $p$ and with the first occurence of success at the sixth trial.

To check the formula, some samples have been generated in order to approximate the probability to get five consecutive *false* results. The experiment consists of generating 100000 samples with probability $P(true) = P(false) = 0.5$. The following procedures are defined in order to implement the experiment: *(i)* `model` is a wrapper for the procedure `geometric`; *(ii)* `count-5` takes in input the list of samples and returns the number of occurences which have value 5. Then the samples are generated and the statistics are computed.

```
; number of samples we want to generate
(define n-samples 100000)

; model used to generate the samples
(define (model)
  (define p 0.5)
  (geometric p))

; procedure which counts the number of samples with value 5
(define (count-5 l)
  (if (null? l)
      0
      (if (= (car l) 5)
          (+ 1 (count-5 (cdr l)))
          (count-5 (cdr l)))))

; sampling
(define experiment (repeat model n-samples))

; ratio between the number of samples with value 5
; and the total number of samples
(/ (count-5 experiment) n-samples)

; histogram of the results
(hist experiment)
```

Listing 14: Experiment to approximate the probability of getting the first occurence of success at the sixth trial

The probability computed by hand is $P(geometric = 5) = 0.5^5 \cdot 0.5 = 0.5^6 = 0.015625$, while the probability computed by the program is $P_{program}(geometric) = 1569/100000 = 0.01569$. The two probabilities are very similar, so we can conclude that the calculation of the probability is correct. The histogram of the generated samples is shown in Figure 1.

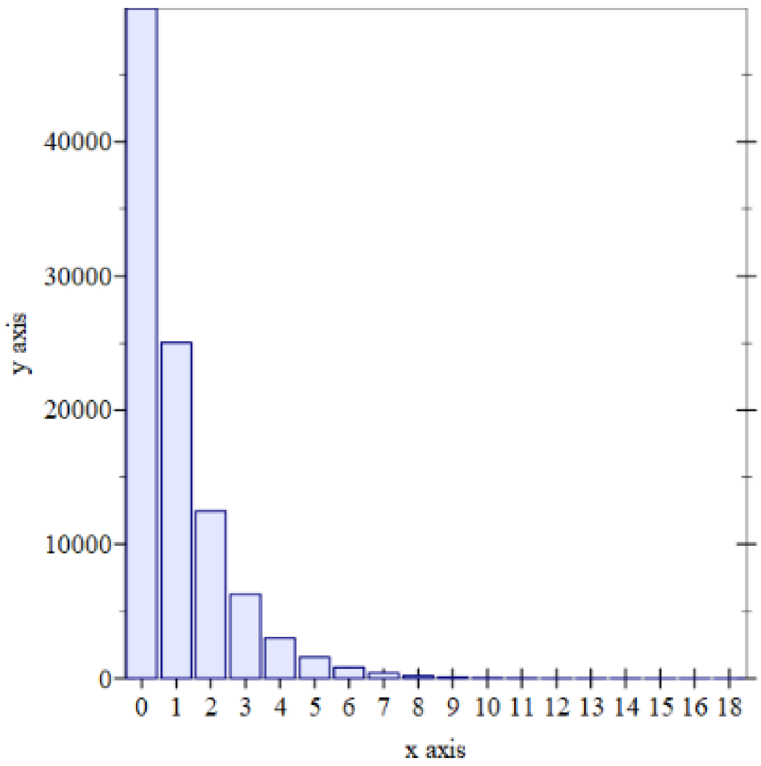


Figure 1: Histogram of geometric experiment: the *x-axis* represents the values generated during the sampling phase; the *y-axis* represents the number of samples which have a specific value.

## Exercise 3 (7)

Convert Table 5 to a compact Racket program.

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

Table 5: Probabilities to be computed with a Racket program

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.

```
(define a ...)
(define b ...)
(list a b)
```

**Solution** The `a-b-model` has been defined as follows:

```
(define (a-b-model)
  (define a
    (flip 0.8))
  (define b
```

```
    (if a
        (flip 0.5)
        (flip 0.3)))
  (list a b))
```

Listing 15: Model to compute the probabilities of A and B

The model does not contain the `rejection-sampler` because we do not need to compute a conditional probability. The `a-b-model` defines first the variable `a` which has probability 0.8 to be *true*: this probability can be computed by adding the last two rows of the Table 5, indeed, the value of `A` in the first two rows is *false*, while in the last two is *true*. Then the probability of the variable `b` depends on the value of the variable `a`, indeed, if `A` is *false*, then the probability that `B` is *true* $\frac{0.06}{0.06+0.14} = 0.3$; while if `A` is *true*, then la probability that `B` is true is 0.5.

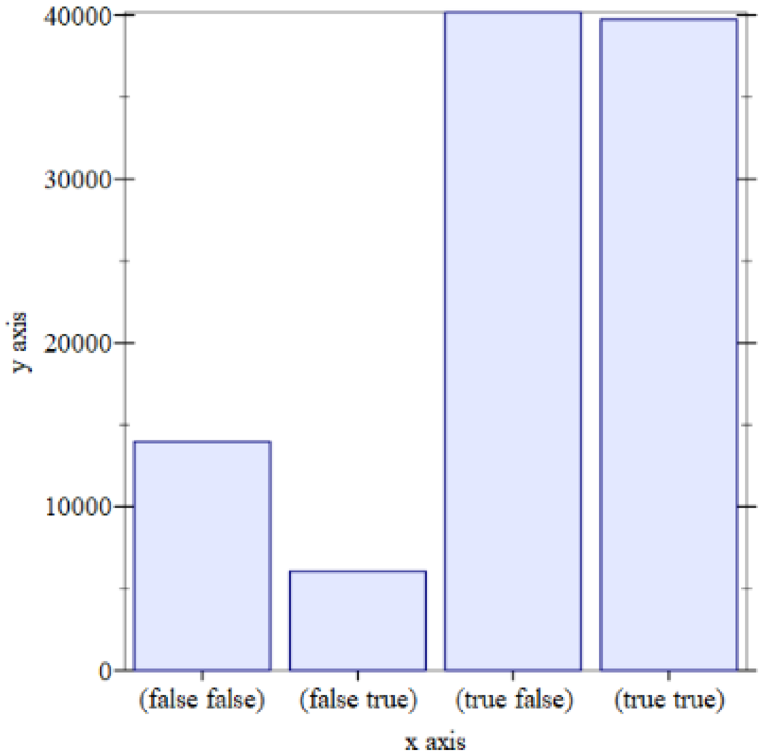


Figure 2: Histogram of A-B experiment: the *x-axis* represents the values generated during the sampling phase; the *y-axis* represents the number of samples which have a specific value.

The experiment consists of generating 100000 samples, the histogram of generated samples is shown in Figure 2. It is possible to observe that the samples are distributed as the probability distribution defined in Table 5, indeed, the number of samples with value false both for the variable `A` and `B` is about 14000; the number of samples with `A` *true* is about 6000 and the number of samples with value *(true, false)* and *(true, true)* is about 40000 each. The number of samples is not exact because we are approximating the probability distribution by sampling.

# Exercise 4

## Exercise 4 (1)

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

**Solution** `(bernoulli-dist p)` and `(normal-dist` $\mu$ $\sigma$`)` are two structures which represent two different distribution, the first one represents the *bernoulli* distribution with success probability $p$; instead, the latter object represents the *normal* distribution with parameters $\mu$ and $\sigma$ as mean and variance. They are not real numbers, they are structures which can be used to generate numbers according to the distribution they represent. To generate a sample from a distribution object it necessary to use the procedure `sample` which takes as argument a distribution object and returns the generated value.

The procedure `flip` has a probabilistic behaviour, but it is different from the call `(normal-dist 0 1)`, indeed, the first one is a procedure which returns the value `#t` with probability 0.5, while the second one is a structure, so the evaluation is different. Since it is a structure, so when the call is done, the interpreter returns a distribution object which has as parameters the arguments of the call, e.g. in our case it returns an object which represents a normal distribution with $\mu = 0$ and $\sigma = 1$.

The procedure `flip` has a similar behaviour to the call `(sample (bernoulli-dist 0.5))` that is equivalent to the call `(bernoulli 0.5)`. One difference is that `flip` returns a value which is either `#f` or `#t`, while `(bernoulli 0.5)` returns a value which is either 0 or 1.

### Exercise 4 (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

1. Evaluation of `(dist? (normal-dist 0 1))`: The interpreter first evaluate the procedure name `dist?`, then it evaluates it arguments. The only argument the interpreter has to evaluate is `(normal-dist 0 1)`. To evaluate this argument, it evaluates first element of the list and than it evaluates the arguments of expression. The expression `(normal-dist 0 1)` is a particular expression, indeed, it is a struct, so the interpreter creates a new instance of the struct `normal-dist` with parameters 0 and 1. The returned instance is the actual parameter of the procedure `dist?` which returns `#t` if the argument is a distribution object, `#f` otherwise. In this particular case the returned object is a distribution object, so the evaluation of the expression is `#t`.

2. Evaluation of `(dist? (bernoulli-dist 0.5))`: The evaluation of this expression is very similar to the previous one, indeed, the procedure `dist?` has the same behaviour as before and the evaluation of `(bernoulli-dist 0.5)` is similar to the evaluation of `(normal-dist 0 1)`. In both cases the interpreter has to deal with a struct, so it evaluates the *constructor* and returns an instance of the structure type. In this case it returns a distribution object which represents a bernoulli distribution with success probability of 0.5. Also in this case the final evaluation is `#t`.

3. Evaluation of `(dist? filp)`: In this case the result is different, indeed, the procedure `flip` is not a distribution object, but it is a procedure with probabilistic behaviour. For this reason, when the interpreter evaluates `(dist? flip)`, it returns `#f`.

The difference between `flip` and (bernoulli-dist 0.5) is that the first one is a procedure that can be called and its evaluation can return the value `#f` or `#t` both with probability 0.5. Instead, the second expression is the call to a constructor of the structure `bernoulli-dist` and the evaluation returns a distribution object of the structure type, i.e. it returns an instance with parameter 0.5. From this object it is possible to return some samples by the procedure according to the bernoulli distribution with success probability equal to 0.5.

## Exercise 5

### Exercise 5 (4)

Probabilistic Models of Cognition - Exercise 4

**Solution** The developed program is written in Church and it is memorized in the file *exercise_5.rkt*. This file is a Racket file, but the code is written in Church, so to be executed it is necessary to use a Church interpreter.

The answers to the questions are shown below:

A. The posterior probability $P(h \mid \text{win})$ is the conditional probability that the machine randomly gives to Bob a specific letter knowing that Bob has won. In particular, the fact that Bob has won is called *evidence* and it is all we kwnow. The posterior probability can be computed with the Bayes' theorem by knowing: *(i)* the conditional probability $P(win \mid \text{h})$ (or *likelihood*); *(ii)* The prior probability $P(h)$ and *(iii)* the marginal probability $P(win)$. In particular it is proven that:

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

B. The probability $P(h \mid \text{win})$ has been manually computed for each hypothesis with Excel. First of all, the *likelihood* has been computed by using the following formula: $P(win \mid \text{h}) = 1/Q(h)^2$ where $Q(h)$ is the position of the letter in the alphabet. Then the numerator of the formula of the Bayes' theorem is computed. To compute the marginal probability $P(win)$ the following formula has be used:

$$P(win) = \sum_h P(win \mid \text{h}) \cdot P(h)$$

Then the posterior probability has been computed for each hypothesis by using the Bayes' theorem and the results are shown in Table 6.

| **Letter** | $P(h \mid \text{win})$ |
|---|---|
| *a* | 0.2755 |
| *b* | 0.3237 |
| *c* | 0.1439 |
| *d* | 0.0809 |
| *e* | 0.0110 |
| *f* | 0.0360 |
| *g* | 0.0264 |
| *h* | 0.0202 |
| *i* | 0.0034 |
| *j* | 0.0130 |
| *k* | 0.0107 |
| *l* | 0.0090 |
| *m* | 0.0077 |

| **Letter** | $P(h \mid \text{win})$ |
|---|---|
| *n* | 0.0066 |
| *o* | 0.0012 |
| *p* | 0.0051 |
| *q* | 0.0045 |
| *r* | 0.0040 |
| *s* | 0.0036 |
| *t* | 0.0032 |
| *u* | 0.0006 |
| *v* | 0.0027 |
| *w* | 0.0025 |
| *x* | 0.0022 |
| *y* | 0.0005 |
| *z* | 0.0019 |

Table 6: Manually computed posterior probability $P(h \mid \text{win})$ for each hypothesis

C. The procedure `my-list-index` takes in input 3 parameters: *(i)* `needle` that is an object, which can be a number, a list, a quoted symbol or something else; *(ii)* `haystack` that is a list containing zero or more elements; *(iii)* `counter` that is a number. The procedure then returns `'error` if the first argument is not present in the second argument, otherwise it returns the third argument (i.e. the number) incremented by the position of the first argument in the list; the position is considered to be zero-based, so the first element of the list has index 0, thus if the `needle` is in the first position of the list `haystack`, then the third argument is not incremented. If the `counter` has value 0 when the procedure is called, then the procedure returns the position zero-based of the `needle` (if present), instead, if the value of the `counter` is 1, then the procedure returns the position one-based of the `needle` in the `haystack`.

D. The procedure `multinomial` takes in input two lists: the first one is a list of possible outcomes, instead the second one is the list of the weights associated to the outcomes in the first list. In particular, the higher is the weight of one element with respect to the weights of the other elements, the higher is the probability to get that particular element. Indeed, the probability to get the element a specific element is given by the following formula:

$$P(e_i) = \frac{weight(e_i)}{\sum_j weight(e_j)}$$

where $e_i$ and $e_j$ are the elements of the first list and the function *weight* returns the weight associated to the argument (i.e. the element of the second list which is in the same position of the element in the first list).

To implement the distribution of Table 6 with the procedure `multinomial`, two lists have been defined: *(i)* `x` which contains all the possible outcomes and *(ii)* `x-weights` which contains the weight of the elements of the first list. An alternative definition of the list `x-weights` is provided in order to prove that the formula previously defined holds. Then a simple model is defined: it simply returns a value of the first list according to the probabilities defined by the second list.

```
; definition of the possible outcomes
(define x '(red␣blue␣green␣black))

; definition of the probability for each outcome
(define x-weights '(0.5␣0.05␣0.4␣0.05))

; equivalent definition of the x-weights
; (define x-weights '(5 0.5 4 0.5))

; definition of a model which represents
; the distribution given in the exercise
(define (model)
  (multinomial x x-weights))

; visualization of the results
(hist (repeat 10000 model))
```

Listing 16: Program which implements with `multinomial` the distribution of Table 6

Finally some samples are generated in order to show that the implemented distribution is equivalent to the distribution defined in Table 6. The results are shown in Figure 3.

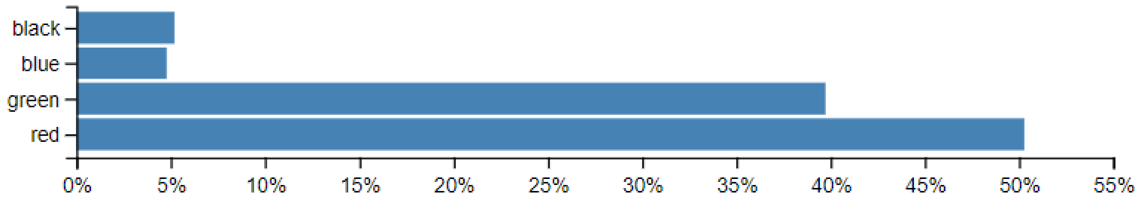


Figure 3: Distribution of the samples generated with the code in Listing 16.

E. The code is completed as shown in Listing 17, in particular it is reported only the definition of `distribution` because the other parts remain the same.

```
(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? (if (equal? my-letter 'a)
                    #t
                    (flip my-win-probability)))

   ; query to get the probability P(h | win)
   my-letter

   ; query to get the probabilities P(vowel | win)
```

```
    ; and P(consonant | win)
    ; (if (vowel? my-letter) 'vowel 'consonant)

    ;; condition
    win?
    ))
```

Listing 17: Definition of `distribution` in order to compute $P(h \mid \text{win})$.

The approximate distribution is shown in Figure 4. It is possible to observe that the computed probabilities are similar to the probabilities manually computed and that the letter with higher posterior probability it the letter $b$. This means that when we know that Bob has won, then it is more likely that he has received the letter $b$ by the machine.

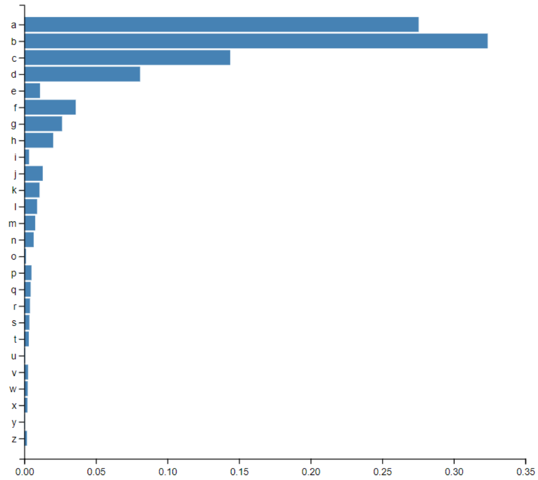


Figure 4: Approximate distribution of posterior probability $P(h \mid \text{win})$ through the code in Listing 17.

F. To compute the posterior probabilities $P(vowel \mid \text{win})$ and $P(consonant \mid \text{win})$ it is sufficient to replace the query part of the code in Listing 17 with the following piece of code:

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

Listing 18: Code to be put in place of `my-letter` in Listing 17

The result is that the program now computes the probability to get a *vowel* or a *consonant* given the evidence that Bob has won. We can observe in Figure 5 that it is more likely that Bob has received a consonant instead of a vowel.

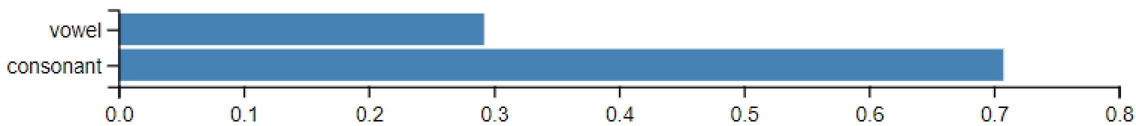


Figure 5: Approximate distribution of probabilities $P(vowel \mid \text{win})$ and $P(consonant \mid \text{win})$.

G. With the mathematical notation the computation is made without taking into account all the probability tree, but it is sufficient to apply the Bayes' theorem in order to compute the posterior probabilities. Instead, by using the Church program the computation is completely automatic and the only thing to do is to define the prior probabilities and then to define the query and the evidence. The disadvantage of the Church computation is that the interpreter has to evaluate all the paths in the probability tree, so it can be used only when the domain of interest is limited and when the query is not too complex.

# Exercise 6

## Exercise 6 (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 basesate. What happens if we set it to 0.01? And to 0.001?

**Solution** In order to assess the differences between different baserates, it has been developed a program which generates 100 samples for each baserate and computes the histogram of the results. The code is shown below:

```
(define (model baserate)
  (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))
  (take-sample))

; experiment with baserate = 0.1
(hist (repeat (model 0.1) 1000))

; experiment with baserate = 0.01
(hist (repeat (model 0.01) 1000))

; experiment with baserate = 0.001
(hist (repeat (model 0.001) 1000))
```

The procedure `model` has been developed in order to be able to pass the baserate as parameter of the procedure. So the procedure `model` is a wrapper for the procedure `take-sample`. In this way it is possible to call the procedure `take-sample` with different baserates by passsing a different parameter to the procedure `model`. Then three different experiments are run: *(i)* The baserate is set to 0.1; *(ii)* The baserate is set to 0.01; *(iii)* The baserate is set to 0.001. By observing the histograms of the results of the different experiments we can conclude that the computed probability is more or less the same in all three cases, but the main difference is that the time of execution is completely different. In particular the first example (i.e. $baserate = 0.1$) is faster than the other two cases. Furthermore the case with $baserate = 0.01$ takes much less time that the third case.

This behaviour is due to the probability to get at least two successful results, in particular the lower is the baserate the lower is the probability that the procedure `flip` returns `#t` so the lower is the probability that `A`, `B` and `C` are equal to one. Since we are approximating the posterior probability $P(A|D >= 2)$ by rejection sampling, all the samples which do not agree with the evidence (i.e. $D = 2$) are discarded. When the baserate is high the probability of getting $D >= 2$ increases, so it is less likely that the sample is discarded, instead if the baserate is low, then the probability to discard a sample increases because it is more probable that the variables `A`, `B` and `C` are equal to zero.

The results of the approximate probabilities are shown in Figure 6, Figure 7 and Figure 8. It is possible to observe that $P(A = 0|D >= 2) \approx 1/3$ and $P(A = 0|D >= 2) \approx 2/3$.

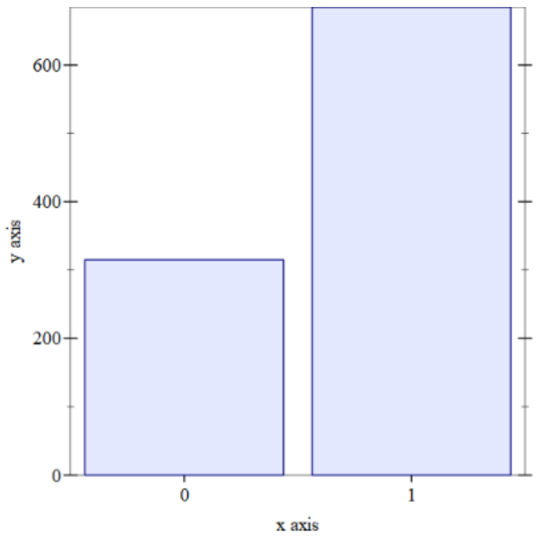


Figure 6: Approximate posterior probability $P(A|D >= 2)$ with $baserate = 0.1$

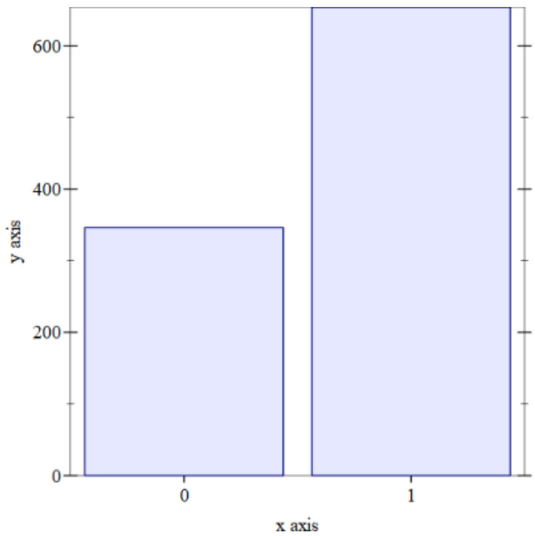


Figure 7: Approximate posterior probability $P(A|D >= 2)$ with $baserate = 0.01$

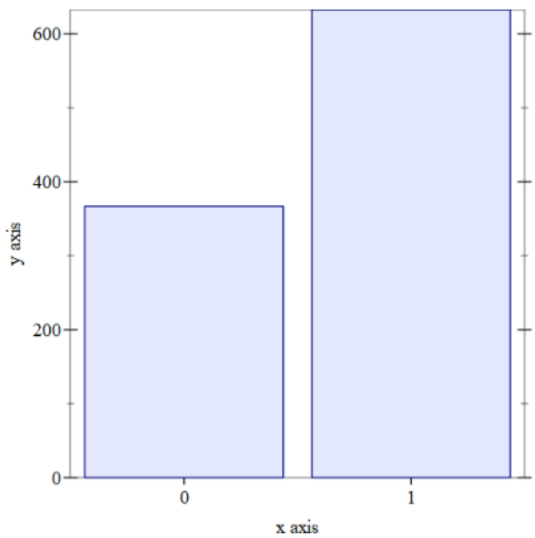


Figure 8: Approximate posterior probability $P(A|D >= 2)$ with $baserate = 0.001$

# Exercise 7

**Definition 1** (MC, functional view)**.** A MC over a state space $X$ is a function $c : X \to \mathcal{D}(X)$.

**Definition 2.** A *stochastic matrix* of dimension $n$ is a $n \times n$-matrix $P$ whose entries belong to $[0, 1]$ and such that each row vector gives a distribution, i.e.:

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

## Exercise 7 (1)

**Proposition.** *The functional and matrix-based definitions of a MC are equivalent.*

*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 $c_P(x_i)(x_j) = P(i, j)$. □

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

### Solution

*Proof.* To prove that the matrix $P_c$ is a stochastic matrix, we have to prove that the sum of the elements of each row of the matrix $P_c$ is equal to 1. Since we have defined the matrix $P_c$ as $P_c(i, j) = c(x_i)(x_j)$, we can write the follwing equality:

$$\forall i. \sum_j P_c(i, j) = \sum_j c(x_i)(x_j) = 1$$

because $c$ is a Markov Chain over the state space $X$.

On the other hand, we need to prove that $c_P$ is a distribution. We can procede as before: we have defined $c_P$ as $c_P(x_i)(x_j) = P(i, j)$, thus it is possible to write the follwing equality:

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

because $P$ is a stochastic matrix, so each row of the matrix gives a distribution, thus the sum of the elements of each row is equal to 1. □

## Exercise 7 (2)

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

**Solution** The Dirac distribution of a state $x_k$ is defined as follows:

$$\delta_{x_k} = \begin{cases} x_k = 1 \\ x_i = 0 & \text{if } i \neq k \end{cases}$$

We have defined the map $c^*(\phi)(y) = \sum_x \phi(x) \cdot c(x)(y)$, so we can write:

$$\forall y.\ c^*(\delta_{x_k})(y) = \sum_{x_i} \delta_{x_k}(x_i) \cdot c(x_i)(y) = 1 \cdot c(x_k)(y) = c(x_k)(y)$$

Since the Dirac distribution is always zero except for $x_k$ that is equal to 1. This equality holds for each state $y$, so have proved that:

$$c^*(\delta_{x_k}) = c(x_k)$$

The notation is slightly different from the one of the instructions, since I do not want to create confusion when the summation is written. Indeed, the goal is to make clear that the summation is defined over all the states of the *state space* and that the only addend that is not null is the one for which the value of $\delta_{x_x}(x_i) = 1$ (i.e. for $x_k$).

## Exercise 7 (3)

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

**Solution** The product $\psi P_c$ returns a vector of dimension $(1, n)$ where $n$ is the number of states. Furthermore it is important to remember that: *(i)* The matrix $P_c$ has been defined as $P_c(i, j) = c(x_i)(x_j)$; *(ii)* The map $c^*$ has been defined as follows: $c^*(\phi)(y) = \sum_x \phi(x)c(x)(y)$. Thus we can write the following equality:

$$\forall y.\ (\psi P_c)(y) = \sum_x \psi(x) \cdot P_c(x, y) \overset{(i)}{=} \sum_x \psi(x)c(x)(y) \overset{(ii)}{=} c^*(\psi)(y)$$

Since this holds forall $y$, then we can assert that $c^*(\psi) = \psi P_c$.

## Exercise 7 (4)

*Prove that if $\psi$ satisfied DBC, then $\psi$ is stationary for $P$.*

**Solution** Before starting with the demonstration, it is important to remember the definition of *DBC*.

**Definition 3.** Given a MC $P$, we say that a distribution $\psi$ satisfies *detailed balanced condition* (DBC) if

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

Since $\psi$ satisfies DBC, the equality above holds for each state $x, y$. Thus it is possible to write the following equation:

$$\forall y.\ (\psi P)(y) = \sum_x \psi(x) \cdot P(x, y) \overset{DBC}{=} \sum_x \psi(y) \cdot P(y, x) = \psi(y) \cdot \sum_x P(y, x) = \psi(y)$$

The last equality is due to the fact that is a stochastic matrix, so the sum of the elements of a row is equal to 1. Since the equation holds for each $y$, then it is possible to assert that $\psi P = \psi$.