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 1.35

Show that the golden ratio φ is a fixed point of the transformation $x \mapsto 1 + \frac{1}{x}$, and use this fact to compute φ 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 φ we obtain:

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

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 φ with a good level of approximation.

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

Exercise 1.36

Modify `fixed-point` so that it prints the sequence of approximations it generates, using the `newline` and `display` primitives shown in Exercise 1.22. Then find a solution to $x^x = 1000$ by finding a fixed point of $x \mapsto \frac{\log(1000)}{\log(x)}$. (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 1.37

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

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

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 φ 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 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` `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 (= (modulo (- 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

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

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

Exercise 3.7

Convert the following probability table to a compact Church program:

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

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

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

Solution

Exercise 4

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

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

Exercise 5

Exercise 5.4

Probabilistic Models of Cognition - Exercise 4

Solution

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

Exercise 7

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 $P_{cP} = P$ and $c_{Pc} = c$.*

Solution

Exercise 7.2

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

Solution

Exercise 7.3

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

Solution

Exercise 7.4

Prove that f ψ satisfied DBC, then ψ is stationary for P.

Solution