INFO020 - Langages de Programmation Evolués
Scheme Exercises 1

Roel Wuyts
roel.wuyts@ulb.ac.be

1 Evaluation of Expressions

Exercise 1: First guess the results of each of the following expressions. Then evaluate them and compare the results. Are they what you expected?

54 > *54*
(+ 23 55) > *78*
(+ 23 44 99) > *166*
(+ 23 (- 55 44 33)) > *1*
(* 2 (/ 8 4)) > *4*
(/ 66 43) > $1\frac{23}{43}$
(define a 3)
a > *3*
(/ 6 a) > *1*
(define b (+ a 1))
(+ a b (* a b)) > *19*
\+ > # *<primitive:+>*

Exercise 2: Continue with the following boolean expressions (so with a and b defined as in the previous exercise

(= 2 3) > *#f*
(= 3 3) > *#t*
(= a b) > *#f*
(not (or (= 3 4) (= 5 6))) > *#t*
(+ 2 (if (> a b) a b)) > *6*

Exercise 3: Continue with the following conditional expressions (so with a and b defined as in the previous exercise

(if (= 1 1) ”waaw” ”brrr”) > *”waaw”*
(if (> a b) a b) > *4*
(if (and (> b a) (< b (* a b))) b a) > *4*
(+ 2 (if (> a b) a b)) >*6*
((if (< a b) + -) a b) > *7*
(cond
    ((= 1 1) ”foo”)
    ((= 2 2) ”bar”)
    ((= 3 3) ”zork”)
    (else ”??que??”)) > *”foo”*
(* (cond ((> a b) a)

```
((< a b) b)
(else -1))
(+ a 1)) > 16
```

Exercise 4: Express the following expression in Scheme and evaluate it: $\frac{\frac{12}{19}+\frac{5+9}{2}}{(10+11)*\frac{20}{3}}$
The solution is the following expression: (/ (+ (/ 12 19) (/ (+ 5 9) 2)) (/ ((+ 10 11) 20) 3)), which evaluates to 29/532*

2 Procedures

Exercise 5: Write two functions celcius−>fahrenheit and fahrenheit−>celsius. The equation used is F = (C + 40) * 1, 8 - 40

```
(define celcius->fahrenheit
    (lambda (c) (- (* (+ c 40) 1.8) 40.0)))
(define fahrenheit->celcius
    (lambda (f) (- (/ (+ f 40) 1.8) 40.0)))
```

Exercise 6: Write a procedure to calculate the Fibonacci numbers using the following recursive definition: $f(1) = 1, f(2) = 1, \forall n > 2 : f(n) = f(n-1) + f(n-2)$

One possible solution (you can also define the procedure without using the lambda form, using a conditional, etc):

```
(define fib
    (lambda (n)
            (if (< n 3)
                    1
                    (+ (fib (- n 1)) (fib (- n 2))))))
```

Exercise 7: Write a procedure **exp** ***in Scheme for calculating*** b^n ***using the following recursive definition:*** $b^0 = 1, b^n = b * b^{n-1} if n \geq 1$

Solution:

```
(define (exp b n)
    (if (= n 0)
            1
            (* b (exp b (- n 1)))))
```

Exercise 8: ?Write a procedure **fast-exp** ***for calculating*** b^n ***that uses the following recursive definition:*** $b^0 = 1, b^n = b * b^{n-1}$ ***when n is odd,*** $b^n = (b^{n/2})^2$ ***when n is even. Then use the trace predicate to show that this algorithm is more efficient than the one from exercise 7. Optional: Can you make it even better?***

The implementation of fast-exp *using an auxiliary procedure* square. *Note that if you use fast-exp itself to calculate the squeare function you need to change the algorithm, as it cannot calculate 2 square (try it!). In that case change the implementation and add an extra case in the conditional for dealing with square (where n is 2).*

```
(require (lib "trace.ss"))

(define (even n)
    (= (modulo n 2) 0))
```

```
(define (square n)
    (* n n))
(define (fast-exp b n)
    (cond
            ((= n 0)
                  1)
            ((even n)
                  (square (fast-exp b (/ n 2))))
            (else
                  (* b (fast-exp b (- n 1))))))

(trace fast-exp)
```

To enable tracing, be sure to select debugging and profiling in the 'Choose Language...' dialog (click the show details button if you do not see the option) in the Language menu. When you trace exp *and* fast-exp*, you'll notice that the stack with the latter is much shorter than for the former (try it with* 3^{10}*, for example).*

Exercise 9: Write a procedure to calculate the greatest common divisor (gcd) using the algorithm of Euclides: $pgcd(a,b) = aifa = b, pgcd(a,b) = pgcd(a-b,b)ifa > b, pgcd(a,b) = pgcd(a,b-a)ifb > a$**.**

The implementation:

```
(define pgcd
    (lambda (a b)
            (cond ((= a b) a)
                  ((< a b) (pgcd a (- b a)))
                  ((> a b) (pgcd (- a b) b)))))
```

To enable tracing, be sure to select debugging and profiling in the 'Choose Language...' dialog (click the show details button if you do not see the option) in the Language menu. When you trace exp *and* fast-exp*, you'll notice that the stack with the latter is much shorter than for the former (try it with* 3^{10}*, for example).*

Exercise 10: Write a recursive procedure* displayn *that takes two arguments, a character and a number. Using the procedure display, show the given character as many times as given by the number argument

One possible solution:

```
(define (displayn c n)
    (display c)
    (if (> n 1)
            (displayn c (- n 1))))
```

Exercise 11: Write a recursive function parametrized by a number that shows 4 squares of size number in such a way that those squares form a larger square themselves. For example:

```
> (squares 3)          > (squares 5)
          ******               **********
          * ** *               *   **   *
          ******               *   **   *
          ******               *   **   *
          * ** *               **********
          ******               **********
                               *   **   *
                               *   **   *
                               *   **   *
                               **********
```

Hint: use **displayn** ***from exercise 10.*** *Solution:*

```
(define (squares n)
   (define (full-line c n)
         (displayn c (* n 2))
         (newline))
   (define (holes-line n)
         (display "*")
         (displayn " " (- n 2))
         (display "**")
         (displayn " " (- n 2))
         (display "*")
         (newline))
   (define (square-inner i)
         (holes-line n)
         (if (> i 3)
              (square-inner (- i 1))))
   (full-line "*" n)
   (square-inner n)
   (full-line "*" n)
   (square-inner n)
   (full-line "*" n))
```

3 Higher-order procedures

Exercise 11: Write a procedure (sum term a next b) ***that takes two numbers (a and b) and two functions (term and next). The procedure sums all (term i), where i lies between a and b. The next i is found by applying the procedure next on the previous i. For example, we can use this to calculate all the squares of the first 10 integers as follows:***

(sum (lambda (x) (* x x)) 1 (lambda (x) (+ x 1)) 10)

```
(define (sum term a next b)
   (if (¿ a b)
         0
         (+ (term a) (sum term (next a) next b))))
```

Exercise 12: Write a procedure* (product term a next b) *analogous to exercise 11. This can then be used to, for example, calculate the product of all odd numbers between 1 and 10 as follows:

(product (lambda (x) x) 1 (lambda (x) (+ x 2)) 10)

```
(define (product factor a next b)
   (if (¿ a b)
         1
         (* (factor a) (product factor (next a) next b))))
```

Exercise 13: Implementing the procedure* factorial *using the function* product *from exercise 12.

```
(define (fac n)
   (product (lambda (x) x) 1 (lambda (x) (+ x 1)) n))
```

Exercise 14: Write a procedure* (accumulate combiner null-value term a next b) *that abstracts away from the functions defined in exercises 11 and 12. Then rewrite* sum *and* product *in terms of* accumulate*.

```
(define (accumulate combiner null-value term a next b)
   (if (> a b)
         null-value
         (combiner
              (term a)
              (accumulate combiner null-value term (next a) next b))))

(define (sum2 term a next b)
   (accumulate + 0 term a next b))
(define (product2 term a next b)
   (accumulate * 1 term a next b))
```

4 Lists

Exercise 15: First guess the results of each of the following expressions. Then evaluate them and compare the results. Are they what you expected?

() > *()*
(cons 1 2) > *(1 . 2)*
((car (cons (cons 1 2) (cons 3 4))) > *(1 . 2)*
(cons (cons (cons (cons 1 2) 3) 4) 5) > *((((1 . 2) . 3) . 4) . 5)*
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 ()))))) > *(1 2 3 4 5)*
(list 1 2 3 4 5) > *(1 2 3 4 5)*
(car (list 1 2 3 4 5)) > *1*
(cdr (list 1 2 3 4 5)) > *(2 3 4 5)*
(cadr (list 1 2 3 4 5)) > *2*
(cadr (list 1 2 3 4 5)) > *3*

Exercise 16: Write a procedure* member? *that checks whether a certain element is a member of a list.

```
(define (member? el lst))
    (cond ((null? lst) #f)
            ((eq? (car lst) el) #t)
            (else (member? el (cdr lst)))))
```

Exercise 17: Write a procedure* prefix? *that checks whether a certain list is the prefix of another list.

```
(define (prefix list1 list2)
    (cond
            ((null? list1) #t)
            ((null? list2) #f)
            ((eq? (car list1) (car list2)) (prefix (cdr list1) (cdr list2)))
            (else #f)
    )
)
```

Exercise 18: Write a procedure* my-inverse *to calculate the inverse of a list.

Without using append:

```
(define (my-inverse l)
    (define (iter l ac)
            (if (null? l)
                    ac
                    (iter (cdr l) (cons (car l) ac))))
    (iter l ()))
```

Using append:

```
(define (my-reverse l)
    (cond
            ((null? l) ())
            (else (append (my-reverse (cdr l))
                            (list (car l))))))
```

Exercise 19: Write a procedure* my-length *to calculate the number of elements of a list.

```
(define (my-length l)
    (if (null? l)
            0
            (+ 1 (my-length (cdr l)))))
```

Exercise 20: Write a higher-order procedure* my-map *to construct a new list from a given list by applying a given procedure on each element. For example, we can construct the list of triples from a given list of integers as follows:

(my-map (lambda (x) (* x 3)) '(1 2 3))

```
(define (my-map proc L)
    (define (my-map-iter current)
            (if (null? current)
                 '()
                 (cons (proc (car current)) (my-map-iter (cdr current)))))
    (my-map-iter L))
```