Magas szintű programozási nyelvek 3

Molnár Antal Albert

2019. december 17.

Tartalomjegyzék

1. I. zárthelyi dolgozat

1.1. define

```
(define (reply2 s)
  (if (and (string? s)
           (>= (string-length s) 5)
           (equal? "hello" (substring s 0 5)))
      "hi"
      "huh?"))

(reply2 "hello")
(reply2 "hel")
(reply2 1234)
```

1.2. three-positive

```
(define (three-positive? a b c)
  (and (> a 0)
           (> b 0)
           (> c 0)))
```

1.3. has-negative

```
(define (has-negative? a b c)
  (or (< a 0)
          (< b 0)
          (< c 0)))
```

1.4. lambda

```
(define n 10)
(define small? (lambda (x) (<= x n)))
(define sqr (lambda (x) (* x x)))
(small? (sqr 10))
(small? (sqr 3))
```

1.5. my-pow

```
(define my-pow
  (lambda (x pow)
        (if (< pow 0)
          (/ 1 (my-pow x (* -1 pow)))
          (if (= pow 0)
            1
            (* x (my-pow x (- pow 1)))))))
```

1.6. my-fact

```
(define my-fact
  (lambda (n)
    (if (= 0 n)
      1
      (* n (my-fact (- n 1))))))
```

1.7. my-fact 3

```
(define f
  (lambda (n prod)
    (if (= 0 n)
      prod
      (f (- n 1) (* n prod)))))

(define (fakt n)
  (define f
    (lambda (n prod)
      (if (= 0 n)
        prod
        (f (- n 1) (* n prod)))))
  (f n 1))
(trace f)
(f 6 1)
```

1.8. farokrekurzív my-fib

```
(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
    b
    (fib-iter (+ a b) a (- count 1))))
```

1.9. let

```
; lokalis kornyezetek

(let ((i 2) (j 3))
  (+ i j))

(let ((+ *) (* +)) ; szimbolumok ertelmenek felcserele
  (+ (* 3 4) (* 2 5)))

(let ((+ *))
  (let((* +))
      (+ (* 3 4) (* 2 5))))

(let* ((i 2) (j (+ 1 2)))
  (+ i j))

; Egy (let ((p1 v1) (p2 v2)) ...) expr1 expr2...
; egyenerteku a kovetkezovel:
; ((lambda (p1 p2 ...) expr1 expr2...) v1 v2 ...)
```

1.10. quadratic

```
(define (quadratic a b c)
  (let ((d ((- (* b b) (* 4 (* a c))))))
    (if (> d 0)
      (print "nincs megold"))))
```

1.11. collatz

```
(define (collatz a l)
  (if (= a 1)
      l
      (if (= (modulo a 2) 0)
          (collatz (/ a 2) (+ l 1))
          (collatz (+ (* a 3) 1) (+ l 1))
          )
      )
  )
```

1.12. collatz2

```
(define collatz2 (lambda (a l)
      (if (= a 1)
          l
          (if (= (modulo a 2) 0)
              (collatz2 (/ a 2) (+ l 1))
              (collatz2 (+ (* a 3) 1) (+ l 1))
              )
          )
      )
  )
```

1.13. collatz3

```
(letrec ([collatz3 (lambda (a l)
      (if (= a 1)
          l
          (if (= (modulo a 2) 0)
              (collatz3 (/ a 2) (+ l 1))
              (collatz3 (+ (* a 3) 1) (+ l 1))
              )
          )
      )])
(collatz3 23 1)
)
```

1.14. collatz4

```
(define (col a l) (let collatz4 ((a a)(l l))
      (cond
        [(= a 1) l]
        [(= (modulo a 2) 0) (collatz4 (/ a 2) (+ l 1))]
        [(= (modulo a 2) 1) (collatz4 (+ (* a 3) 1) (+ l 1))]
        )

))
```

1.15. sum

```
(define sum (lambda (a b)
   (if (< a b)
     (+ a (sum (+ a 1) b)) a)))
```

1.16. lcs

```
(define (lcs str1 str2)
  (define (lcs-inner lst1 lst2 l)
    (cond
      [(or (null? lst1) (null? lst2)) l]
      [(eq? (car lst1) (car lst2)) (lcs-inner (cdr lst1) (cdr lst2) (add1 l))]
      [else (max (lcs-inner (cdr lst1) lst2 l) (lcs-inner lst1 (cdr lst2) l))]
      )
    )
  (lcs-inner (string->list str1) (string->list str2) 0)
  )
```

1.17. lcs-hash

```
(define (lcs-hash str1 str2)
  (define ht (make-hash))
  (define (lcs-inner lst1 lst2)
    (define hash-key (list lst1 lst2))
    (cond
      [(hash-ref ht hash-key #f) (hash-ref ht (list lst1 lst2))]
      [(or (null? lst1) (null? lst2)) 0]
      [(eq? (car lst1) (car lst2))
          (hash-set! ht hash-key (+ 1 (lcs-inner (cdr lst1) (cdr lst2)))) (hash-ref ht (list lst1 lst2))]
      [else
          (hash-set! ht hash-key
              (max (lcs-inner (cdr lst1) lst2) (lcs-inner lst1 (cdr lst2)))) (hash-ref ht (list lst1 lst2))]
      )
    )
  (lcs-inner (string->list str1) (string->list str2))
  )
```

1.18. my-sort

```
(define (my-sort lst)
  (define (list-min lst)
    (define (find-min lst min)
      (cond
        [(null? lst) min]
        [else (if (< (car lst) min)
                  (find-min (cdr lst) (car lst))
                  (find-min (cdr lst) min)
                  )]
        )
      )
    (find-min lst (car lst))
  )
  (define sorted '())
  (define (inside-sort lst)
    (cond
      [(null? lst) sorted]
      [else (define min-val (list-min lst))
            (set! sorted (append sorted (list min-val)))
            (inside-sort (remove min-val lst))
            ]
      )
    )
  (if (list? lst)
      (inside-sort lst)
      (error "Expected argument type: List, Given: Whatever you gave it")
      )
  )
(my-sort '(1 4 2 -2 50 3000 -200 60 50))
```

1.19. myremove

```
;elso elofordulast torli
(define (myremove v lst)
  (cond
    [(null? lst) (append null null)]
    [(equal? v (car lst)) (append null (cdr lst))]
    [else (append (list (car lst)) (myremove v (cdr lst)))]
    )
  )
```

1.20. myremove2

```
;osszes elofordulast torli
(define (myremove2 v lst)
  (cond
    [(null? lst) (append null null)]
    [(equal? v (car lst)) (append null (myremove2 v (cdr lst)))]
    [else (append (list (car lst)) (myremove2 v (cdr lst)))]
    )
  )
```

2. II. zárthelyi dolgozat

2.1. topcollatz

```
|(define (collatz-max n)
  (define (topcollatz ao)
    (let loop ( (n ao) (l (list ao)))
      (if (= 1 n)
          l
          (if (even? n)
              (loop (/ n 2) (append l (list (/ n 2))))
              (loop (+ 1 (* 3 n )) (append l (list (+ 1 (* 3  n )))))
              )
          )
      )
    )

  (apply max (topcollatz n)))|
```

2.2. how-many-boxes

```
(define (how-many-boxes boxes capacity trees)
  (if (> (sum trees) (* boxes capacity))
      (println "Nem fer el!")
      (ceiling (/ (sum trees) capacity))))


(how-many-boxes 30 1 '(30 40 50))
(how-many-boxes 30 1 '(3 4 5))
```

2.3. stimm?

```
; 1 - hazak
(define (stimm? lst)
  (cond
    [(>= 2 (length lst))
     (= 1 (abs (- (list-ref lst 1) (list-ref lst 0))))]
    [(not (= 1 (abs (- (list-ref lst 1) (list-ref lst 0))))) #f]
    [else
     (stimm? (cdr lst))]
    )
  )

(stimm? '(5 4 3 2 1 2 3 4 5))
(stimm? '(5 7 3 2 1 2 3 4 5))
(stimm? '(5 4 3 2 1 2 3 9 5))
```

2.4. smallest-div

```
(define (smallest-div n)
  (apply lcm (build-list n add1)))
```

2.5. kulonbseg

```
(define (sq n)
  (expt n 2))
(define (kul n)
  (define lst (build-list n add1))
  (define a (sum (map sq lst)))
  (define b (sq (sum lst)))
  (- b a)
  )
```

2.6. is-decreasing

```
(define (is-decreasing? lst)
  (cond
    [(> 2 (length lst)) #t]
    [(equal? lst (sort lst >)) #t]
    [else #f]))

(is-decreasing? '(5 4 3 2 1))
```

2.7. collatz

```
(define (collatz a l)
  (if (= a 1)
      l
      (if (= (modulo a 2) 0)
          (collatz (/ a 2) (+ l 1))
          (collatz (+ (* a 3) 1) (+ l 1))
          )
      )
  )
(define (collatz-call a) (collatz a 1))
```

2.8. my-map

```
(define (my-map lst fun)
    (cond
    [(null? lst) '()]
    [else (append (list (fun (car lst))) (my-map (cdr lst) fun))]
    )
)
```

2.9. my-filter

```
(define (my-filter lst pred)
    (cond
    [(null? lst) '()]
    [(pred (car lst)) (append (list (car lst)) (my-filter (cdr lst) pred))]
    [else (my-filter (cdr lst) pred)]
    )
)
```

2.10. reverse-string

```
(define (reverse-string str)
    (define (reverse lst)
        (cond
        [(null? lst) '()]
        [else (append (reverse (cdr lst)) (list (car lst)))]
        )
    )
    (list->string (reverse (string->list str)))
)
```

2.11. dividers2

```
(define (dividers2 n)
    (define lst (range 1 (- n 1)))
    (filter (lambda (i) (= (modulo n i)) 0) lst)
)

(dividers2 20)
```

2.12. divisors-of

```
(define (divisors-of n)
    (filter (compose
        (curry = 0) ((curry modulo n))
    ) (range 1 n))

)

(divisors-of 5000)
```

2.13. pascal-n

```
(define (pascal-n n)
    (define (fact nn)
        (if (> nn 1)
            (* nn (fact (sub1 nn)))
            1
        )
    )
    (define (combination k)
        (/ (fact n) (* (fact k) (fact (- n k))))
    )

    (define (inner iteration)
        (cond
        [(> iteration n) '()]
        [else (append (list (combination iteration)) (inner (add1 iteration)))]
        )
    )
    (inner 0)
)

(pascal-n 3)
```

2.14. 0<n<10000; reverse(n): palindrom

```
(define (palindrom lst)
    (define (my-reverse lt)
        (cond
        [(null? lt) '()]
        [else (append (my-reverse (cdr lt)) (list (car lt)))]
        )
    )
    (equal? (my-reverse lst) lst)
)

(filter (lambda (num)
    (palindrom
      (number->list(+ num (list->number(reverse (number->list num))))))
) (build-list 10000 values))
```

2.15. reverse1

```
(define (reverse1 l)
  (if (null? l)
     nil
     (append (reverse1 (cdr l)) (list (car l)))))
```

2.16. lcs

```
(define (lcs str1 str2)
  (define (lcs-inner lst1 lst2 l)
    (cond
      [(or (null? lst1) (null? lst2)) l]
      [(eq? (car lst1) (car lst2)) (lcs-inner (cdr lst1) (cdr lst2) (add1 l))]
      [else (max (lcs-inner (cdr lst1) lst2 l) (lcs-inner lst1 (cdr lst2) l))]
      )
    )
  ;(trace lcs-inner)
  (lcs-inner (string->list str1) (string->list str2) 0)
  )
```

2.17. lcs-hash

```
(define (lcs-hash str1 str2)
  (define ht (make-hash))
  (define (lcs-inner lst1 lst2)
    (define hash-key (list lst1 lst2))
    (cond
      [(hash-ref ht hash-key #f) (hash-ref ht (list lst1 lst2))]
      [(or (null? lst1) (null? lst2)) 0]
      [(eq? (car lst1) (car lst2))
        (hash-set! ht hash-key
          (+ 1 (lcs-inner (cdr lst1) (cdr lst2))))
          (hash-ref ht (list lst1 lst2))]
      [else
        (hash-set! ht hash-key
          (max (lcs-inner (cdr lst1) lst2) (lcs-inner lst1 (cdr lst2))))
          (hash-ref ht (list lst1 lst2))]
      )
    )
  ;(trace lcs-inner)
  (lcs-inner (string->list str1) (string->list str2))
  )

(lcs "bcacbcabbaccbab" "bccabccbbabacbc")

(lcs-hash "bcacbcabbaccbab" "bccabccbbabacbc")
(lcs-hash "abcdefghijklmnopqrstuvwxyz"
          "a0b0c0d0e0f0g0h0i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0y0z0")
```

2.18. my-length

```
(define (my-length lst)
  (cond
    [(null? lst) 0]
    [else (add1 (my-length (cdr lst)))]
    )
)
```

2.19. modified-collatz

```
(define (modified-collatz n)
  (cond
    [(= n 1) 1]
    [(= (modulo n 2) 0) (+ 1 (modified-collatz (/ n 2)))]
    [(= (modulo n 3) 0) (+ 1 (modified-collatz (/ n 3)))]
    [else (+ 1 (modified-collatz (add1 (* n 3))))]
    )
  )
```