Magas szintű programozási nyelvek 3

Molnár Antal Albert

2019. november 4.

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