FUNKCIONALNO PROGRAMIRANJE

Oznaka predmeta: FPR

Predavanje broj: 11

Nastavna jedinica: LISP,

Nastavne teme:

Predikati. Predikati. Strategije upravljanja: uslovne forme (when, unless, if, cond, case). Logičke operacije. Strategije upravljanja. Rekurzije: obična i repna. Apstrakcija podataka: konstruktori, čitači, pisači. Prototipovi nekih rekurzivnih operacija: transform, filter, count, find. Strategije upravljanja: ponavljanje preslikavanjem (mapcar, remove-if, remove-if-not, count-if, find-if). Procedure kao argumenti procedura (funcall, apply, lambda). Strategije upravljanja: ponavljanje iteracijom (dotimes, dolist, return, do, do*, loop, prog1, prog2, progn).

Predavač: prof. dr Perica S. Štrbac, dipl. ing.

Literatura:

David S. Touretzky: "Common Lisp: A Gentle Introduction to Symbolic Computation", Dover Publications, 2013.

LISP, predikati

- Predikati:
  (equal, atom, number, simbolp, listp, endp, null, member, zerop, plusp, minusp, evenp, oddp, <, >)
- Predikat je procedura koja na osnovu rezultata argumenata kao svoj rezultat vraća vrednost istina (T) ili neistina (NIL).
  - Osnovna podela:
    - predikati jednakosti
    - predikati tipova podataka
    - predikati lista
    - numerički predikati

Naziv predikata : EQUAL

Primeri	: (equal 'a 2)	(equal 'ab 'ab)
Rezultati	: nil	t

Naziv predikata : ATOM

Primeri	: (atom ' a)	(atom '(a b))
Rezultati	: t	nil

LISP, predikati

Naziv predikata	: NUMBERP		
Primeri	: (numberp 'a)	(numberp 2)	
Rezultati	: nil	t	
Naziv predikata	: SYMB0LP		
Primeri	: (symbolp 'a)	(symbolp '2)	
Rezultati	: t	nil	
Naziv predikata	: *LISTP*		
Primeri	: (listp '(a b))	(listp 'a)	
Rezultati	: t	nil	
Naziv predikata	: *ENDP*		
Primeri	: (endp '())	(endp '(a b))	(endp 'a)
Rezultati	: t	nil	ERROR
Naziv predikata	: NULL		
Primeri	: (null '())	(null '(a b))	(null 'a)
Rezultati	: t	nil	nil
Naziv predikata	: MEMBER		
Primeri	: (member 'b '(a b c))	(member 'd '(a c))	
Rezultati	: (b c)	nil	

LISP, predikati

Naziv predikata	: ZEROP	
Primeri	: (zerop 0)	(zerop 23)
Rezultati	: t	nil
Naziv predikata	: PLUSP	
Primeri	: (plusp 23)	(plusp -23)
Rezultati	: t	nil
Naziv predikata	: MINUSP	
Primeri	: (minusp -23)	(minusp 23)
Rezultati	: t	nil
Naziv predikata	: EVENP	
Primeri	: (evenp 10)	(evenp 11)
Rezultati	: t	nil
Naziv predikata	: ODDP	
Primeri	: (oddp 11)	(oddp 12)
Rezultati	: t	nil
Naziv predikata	: <	
Primeri	: (< 1 2 3)	(< 1 3 2)
Rezultati	: t	nil

LISP, predikati

Naziv predikata	: >	
Primeri	: (> 3 2 1)	(> 3 1 2)
Rezultati	: t	nil

- Dati su rezultati izračunavanja zadanih formi:

```
(equal '(ovo je lista) (setf l (reverse '(lista je ovo))))     t
(atom nil)                                                     t
(atom ())                                                      t
(symbolp nil)                                                  t
(symbolp ())                                                   t
(listp nil)                                                    t
(listp ())                                                     t
```

- Napisati proceduru DELJIV-SA-3 kao predikat koji vraća informaciju o deljivosti argumenta (koji je celi broj) sa 3. Koristiti primitive REM koja uzima 2 argumenta i vraća ostatak deljenja prvog argumenta sa drugim argumentom.

```
(defun deljiv-sa-3 (argument)
    (zerop (rem argument 3)) )
```

- Napisati proceduru PALINDROMEP kao predikat koji vraća informaciju o tome da li je argument (koji je lista) jednak kada se čita sa leva na desno i suprotno.

```
(defun palindromep (lista) (equal lista (reverse lista) ))
```

LISP, strategije upravljanja: uslovne forme

- Napisati proceduru SPECIJALNIP kao predikat koji za argumente ima dužine stranica trougla. Predikat vraća T ako je suma kvadrata dve kraće stranice unutar 2% od kvadrata veće stranice, inače vraća NIL. Neka je dužina najduže stranice trougla data kao prvi argument.

```
(defun specijalnip (a b c)
  (< 0.98 (/ (+ (* b b) (* c c)) (* a a)) 1.02) )
```

STRATEGIJE UPRAVLJANJA 1 - Uslovne forme (when, unless, if, cond)

- WHEN izvršava sve <akcije> ako je rezultat izvršavanja <uslova> različit od NIL i tada je rezultat WHENa rezultat poslednje <akcije>, a u suprotnom je rezultat NIL. Opšti oblik:

```
(WHEN <uslov> <akcijal> ...)
```

- UNLESS izvršava sve <akcije> ako je rezultat izvršavanja <uslova> NIL i tada je rezultat UNLESSa rezultat poslednje <akcije>, a u suprotnom je rezultat NIL. Opšti oblik:

```
(UNLESS <uslov> <akcijal> ...)
```

- IF izvršava <akciju1> ako je rezultat izvršavanja <uslova> različit od NIL i tada je rezultat IFa rezultat <akcije1>. U suprotnom se izvršava <akcija2> i tada je rezultat IFa rezultat <akcije2>. Opšti oblik:

```
(IF <uslov> <akcija1> <akcija2>)
```

LISP, strategije upravljanja: uslovne forme

- COND izvršava <uslove> sve dok neki ne vrati rezultat T i tada se izvršavaju sve njegove pripadne <akcije> a rezultat CONDa je rezultat poslednje izvršene akcije.
  - Ako niti jedan uslov nije bio različit od NIL, tada je NIL ujedno i rezultat CONDa.
  - Preporučuje se da je kao poslednja navedena sekvenca sa uslovom T koja će se izvesti ako se ne izvede ništa pre toga. Opšti oblik:

```
(COND (<uslovl> <akcija11> ...)
      (<uslov2> <akcija21> ...)
      • • •
      (T        <akcijatl> <akcijat2> ...) )
```

- CASE upoređuje <iznose> sa rezultatom <ispitnog izraza>.
  - Sve akcije koje pripadaju prvom iznosu za koji je rezultat poređenja T biće izvršene. Ako se niti jednim poređenjem ne dobije T, rezultat čitave forme je NIL. Ako se nijednim poređenjem ne dobije T, a poslednji iznos je T ili OTHERWISE izvršiće se njemu pripadne akcije.
  - Ako je <iznos> lista onda se za poređenje ne koristi EQUAL nego MEMBER.

```
(CASE <ispitniizraz>
    (<iznos1> <akcija11> ...)
      (<iznos2> <akcija21> ...)
      ...)
```

LISP, strategije upravljanja: uslovne forme

- Dati su rezultati izračunavanja datih nizova formi:

```
(setf visoka 98 temperatura 102)                          102
   (when (> temperatura visoka)
  (setf visoka temperatura))                              102
   ('nova-temperatura)                                    nova-temperatura
   visoka                                                         102
(setf n 1)
   (when (> n 0) 'pozitivno)                              pozitivno
   (when (< n 0) 'negativno)                              nil
   (unless (> n 0) n)                                     nil
   (unless (< n 0) n)                                     1
   (setf rezultat 'da)                                    da
   (when (> 3 2 1) rezultat)                              da
   (when (member rezultat '(da a mozda i ne)) rezultat)   da
(setf poduprto 'kocka)                                    kocka
   (when poduprto 'poduprto)                              poduprto
   (when poduprto poduprto)                               kocka
   (unless poduprto poduprto)                             nil
   (setf l '(pre mnogo i mnogo godina u carstvu))
      pre...carstvu
   (length l)                                             7
   (if (endp l) 0 (length l))                             7
   (if (endp l) 0 (length (rest l)))                      6
```

LISP, strategije upravljanja: uslovne forme

```
(if (endp l) 0 (+ 1 (length (rest l))))                    7
(setf x 'xyz)                                               xyz
(when (equal x 'xyz) (setf x 'old))                         old
x                                                           old
(when (equal x 'abc) (setf x 'new))                         nil
x                                                           old
```

- Date su IF forme koje su ekvivalentne sa:

```
(abs x )                    (if (< x 0) (* -1 x) x)
(min a b)                   (if (< a b) a b)
(max a b)                   (if (> a b) a b)
```

- Napisati proceduru PROVERA-TEMPERATURE koja na temelju jednog argumenta vraća JAKO-VRUĆE ako je njegov iznos bio veći od 100, JAKO-HLADNO ako je njegov iznos bio manji od 0, a inače NORMALNO.

```
(defun provera-temperature (tem)
       (cond  ( (> tem 100) 'jako_vruce)
              ( (< tem 0)   'jako_hladno)
              ( t           'normalno) ) )
```

- Veliki broj primitiva LISP-a se može implementirati malim skupom osnovnih primitiva. Pretpostavivši da ne postoje primitive 1+ i 1-, napisati procedure P1 i M1, koje će vratiti za 1 veću odnosno manju vrednost u odnosu na argument.

LISP, strategije upravljanja: uslovne forme

```
(defun P1(n) (+ n 1))
(defun M1(n) (- n 1))
```

- Velik broj primitiva LISP-a se moze implementirati malim skupom osnovnih primitiva. Pretpostavivši da primitiva NTHCDR ne postoji, napisati proceduru NTHCDR1, koja ce raditi isto što i NTHCDR.

```
(defun nthcdr1(n lista)
   (if (zerop n) lista
                 (rest ( nthcdr1 (- n 1) lista)) ))
```

- Pretpostavivsi da primitiva LAST ne postoji, napisati proceduru LAST1 koja će raditi isto sto i LAST.

```
(defun last1(lista)
   (if (endp (rest lista)) lista
                           (last1 (rest lista))))
```

LISP, logičke operacije

LOGIČKE OPERACIJE (and, or, not)

- AND ima proizvoljan broj argumenata. Argumenti se izračunavaju sa leva na desno. Ako se izračuna NIL, tada se preostali argumenti više ne izračunavaju, a AND vraća NIL. U suprotnom AND vraća iznos poslednjeg argumenta.
- OR ima proizvoljan broj argumenata. Argumenti se izračunavaju sa leva na desno. Ako se izračuna bilo kakva vrednost različita od NIL, tada se preostali argumenti vise *ne izračunavaju*, a OR vraća upravo izračunatu vrednost. U suprotnom OR vraća NIL.
- NOT ima jedan argument. Ako je njegova vrijednost bilo šta različito od NIL, NOT vraća NIL. Ako je njegova vrednost NIL, NOT vraća T.

•Dati su rezultati izračunavanja zadanih nizova formi:

```
(setf mypi 3.14)                                    3.14
(and (listp   mypi) (numberp mypi))                 nil
(and (list    mypi) (numberp mypi))                 t
(or  (listp   mypi) (numberp mypi))                 t
(or  (list    mypi) (numberp mypi))                 (3.14)
(or  (numberp mypi) (listp mypi))                   t
(or  (listp   mypi) (< mypi 0))                     nil
(and (listp   mypi) (setf rezultat 'pre))           nil
rezultat                                            ERROR
```

LISP, logičke operacije

```
(and (numberp pi) (setf rezultat 'posle))                    posle
rezultat                                                     posle
```

- Date su COND forme koje su ekvivalentne sa:

```
(not u)        (COND ((equal u nil)  t)
                     ( t            nil) )

(or x y z)     (COND ((equal (equal x nil) nil)   x)
                     ((equal (equal y nil) nil)   y)
                     ((equal (equal z nil) nil)   z)
                     ( t                          nil))

(and a b c)
      (COND ((equal a nil) nil)
            ((equal b nil) nil)
            ((equal c nil) nil)
            ( t              c))
```

- Data je CASE forma:

```
      (setq x 'b)                          B
      (case x
              ( a        5)
              ((d e)     7)
              ((b f)     3)                3
              (otherwise 9))
```

LISP, strategije upravljanja: ponavljanje rekurzijom

STRATEGIJE UPRAVLJANJA :

Ponavljanje rekurzijom (obična rekurzija, repna rekurzija)

- Da bi se mogla definisati rekurzivna procedura treba poznavati:
  1. Uslov okončanja rekurzivnih poziva;
  2. Način da se problem razbije na niz parcijalnih problema;
  3. Način obrade parcijalnog problema i formiranja konačnog rezultata na osnovu parcijalnih rezultata.
- Prototipovi rekurzivnih procedura:
  - Obična rekurzija (pri povratku iz najdublje procedure dobija se parcijalan rezultat):

```
(defun R (PROBLEM)
   (if <detektor kraja problema> PROBLEM)
      <parcijalni rezultat>
      (<dorada rezultata> (R (<smanjenje problema> PROBLEM)))) )
```

Primer:

```
(defun prebroj-elemente (lista)
   (if (endp lista)
      0
     (+ 1 (prebroj-elemente (rest lista)))) )
```

LISP, strategije upravljanja: ponavljanje rekurzijom

2. "Repna" rekurzija (pri povratku iz najdublje procedure dobija se konačan rezultat, rezultat je na repu):

```
(defun R1 (PROBLEM REZULTAT)
   (if <detektor kraja problema> PROBLEM) <konacan rezultat>
      (R1 (<smanjenje problema> PROBLEM) <parcijalna obrada>
                                          REZULTAT))))
```

uz poziv tipa:

```
(R1 PROBLEM pocetni-rezultat)
```

Primer:

```
(defun prebroj-elemente1 (lista rezultat)
   (if (endp lista) rezultat
       (prebroj-elemente1 (rest lista) (+ 1 rezultat))) )
```

uz poziv tipa:

```
(prebroj-elementet1 nekalista 0)
```

- Napisati običnu rekurzivnu proceduru REC-EXP za računanje exponenta m**n. Pretpostaviti da je n nenegativan celi broj.

```
(defun rec-exp (m n)
   (if (zerop n) 1
       (* m (rec-exp m (- n 1)))) )
```

LISP, strategije upravljanja: ponavljanje rekurzijom

- Napisati repnu rekurzivnu proceduru REC-EXP1 za računanje exponents m * * n . Pretpostaviti da je n nenegativan celi broj.

```
(defun rec-exp1 (m n rezultat)
        (if (zerop n) rezultat
            (rec-exp1 m (- n 1) (* rezultat m))))
```

uz poziv tipa:

```
(rec-exp1 2 3 1)
```

- Napisati običnu rekurzivnu proceduru COUNT-ATOMS za prebrojavanje atoma u listi bez obzira na broj podlista.

```
(defun count-atoms (L)
        (cond
           ((null L) 0)
           ((atom L) 1)
           (t        (+ (count-atoms (car L))
                        (count-atoms (cdr L)) ) ) ) )
```

- Napisati običnu rekurzivnu proceduru za računanje Fibonačijeve funkcije f(n) = f(n-1)+f(n-2) za n>1 inače f(0)=f(1)=1.

```
(defun Fibonacci (N)
        (cond ((< N 2) 1)
              (t       (+ (Fibonacci (- N 1)) (Fibonacci (- N 2))))))
```

LISP, strategije upravljanja: ponavljanje rekurzijom

- Pod pretpostavkom da ne postoji primitiva NTHCDR napisati repnu rekurzivnu proceduru NTHCDR1 koja vraća ulaznu listu bez prvih n elemenata.

```
(defun nthcdr1 (n L)
        (if (zerop n) L
                        (nthcdr1 (- n 1) (rest L))) )
```

- Napisati običnu rekurzivnu proceduru ZADRZI-PRVIH-N koja vraća listu koju čini prvih n elemenata ulazne liste.

```
(defun zadrzi-prvih-n (n L)
     (if (zerop n) nil
                   (cons (first L) (zadrzi-prvih-n (- n 1) (rest L)))))
```

- Napisati repnu rekurzivnu proceduru ZADRZI-PRVIH-N1 koja vraca listu koju cini prvih n elemenata ulazne liste.

```
(defun zadrzi-prvih-n1 (n L R)
    (if (zerop n) R
          (zadrzi-prvih-n1 (- n 1) (rest L) (append R (list(first L)))
)))
```

LISP, strategije upravljanja: ponavljanje rekurzijom

- Postojeće primitive 1+ i 1- povećavaju, odnosno, umanjuju broj za 1.
  Napisati repnu rekurzivnu i običnu rekurzivnu proceduru SABERI za sabiranje dva broja. Pretpostaviti da primitiva + ne postoji, a brojevi koje treba sabrati su pozitivni.

```
(defun saberi2 (n1 n2)
      (if (zerop n2) n1 (saberi2 (1+ n1) (1- n2))) )
(defun saberi (n1 n2)
      (if (zerop n2) n1 (1+ (saberi n1 (1- n2)))))
```

- Napisati proceduru SAZMI koja vraća ulazni izraz, ali bez ikakvih unutrašnjih zagrada koje su pre postojale.

```
(defun sazmi (izr)
      (cond ((null izr) nil)
            ((atom izr) (list izr))
            (t      (append (sazmi(first izr)) (sazmi (rest izr))))))
```

LISP, apstrakcija podataka

APSTRAKCIJA PODATAKA:

konstruktori,

čitači,

pisači.

- Stvara se hijerarhija procedura, pri čemu niski nivo deluje neposredno nad strukturom podataka, a viši nivo ne može direktno dopreti do podataka, već se indirektno služi procedurama nižeg nivoa.
- Uz svaku strukturu podataka definišu se pripadne procedure nižeg nivoa - one služe za tri svrhe:
  - stvaranje podataka (constructor procedures)
  - dohvatanje podataka (reader procedures)
  - izmenu podataka (writer procedures)

Primer:

Neka je pojedina knjiga opisana listom sledećeg oblika:

```
( (naslov (<stvarni naslov>))
  (autor  (<stvarni autor> ))
  (klasa  (<stvarna klasa> )) )
```

LISP, apstrakcija podataka

dakle, konkretno:

```
'((naslov (artificial intelligence        ))
  (autor  (p h Winston                    ))
  (klasa  (tehnicka vestacka inteligencija)) )
```

Konstruktor bi bio:

```
(defun stvori-knjigu (naslov autor klasa)
   (list (list 'naslov naslov)
         (list 'autor autor  )
         (list 'klasa klasa  )) )
```

Čitači bi mogli biti:

```
(defun uzmi-naslov (knjiga)
   (second (assoc 'naslov knjiga)) )

(defun uzmi-autora (knjiga)
   (second (assoc 'autor knjiga)) )

(defun uzmi-klasu (knjiga)
   (second (assoc 'klasa knjiga)) )
```

Jedan od pisača bi mogao biti:

```
(defun zameni-autora (knjiga autor)
 (if (equal 'autor (first (first knjiga)))
   (cons (list 'autor autor) (rest knjiga))
   (cons (first knjiga) (zameni-autora (rest knjiga) autor))) )
```

LISP, apstrakcija podataka

- Napisati proceduru IN-LIST-P koja će davati informaciju da li je njen prvi argument element liste koja je njen drugi argument. Tražena procedura je primer tzv. TRAZECE procedure.

```
(defun in-list-p (a l)
        (cond ((null l)            nil)
              ((equal a (first l))   t)
              (t                   (in-list-p a (rest l) ) ) ) )
```

- Koristeći rezultat prethodnog zadatka i informacije iz ovog poglavlja napisati proceduru NADJI-KNJIGU-CIJI-JE-NASLOV koja će na temelju ključne reči izdvojiti iz liste knjiga onu traženog naslova. Tražena procedura je primer tzv. FILTRIRAJUCE procedure.

```
(defun nadji-knjigu-ciji-je-naslov (naslov knjigE)
 (cond
   ( (null knjigE)                                                    nil)
   ( (string-equal naslov (uzmi-naslov (first knjigE))) (first knjigE))
   (  t             (nadji-knjigu-ciji-je-naslov naslov (rest knjigE)))

 )

)
```

LISP, prototipovi

PROTOTIPOVI NEKIH REKURZIVNIH OPERACIJA (transform, filter, count, find)

- TRANSFORMISANJE liste u novu listu čini transformacija svih njenih elemenata:

```
(defun TRANSFORM (L)
  (if (endp L)
      nil
      (cons (TRANS-OP (first L)) (TRANSFORM (rest L)))) )
```

- FILTRIRANJEm liste u novu listu neki se njeni elementi izostavljaju:

```
(defun FILTER (L)
   (cond
        ((endp L)   nil)
        ((TEST-OP   (first L))   (cons (first L) (FILTER (rest L))))
        (t          (FILTER (rest L)) )  )  )
```

- BROJANJEm se prebrojava broj elemenata liste s nekim svojstvom:

```
(defun COUNTCC (L)
  (cond
       ((endp L)  0)
       ((TEST-UP (first L))              (+ 1 (COUNTCC (rest L)) ))
       (t         (COUNTCC (rest L)))  ) )
```

LISP, prototipovi

- PRETRAZIVANJEm se pronalazi prvi element liste s nekim svojstvom:

```
(defun FINDCC (L)
   (cond
        ((endp L)                          nil )
        ((TEST-OP (first L))            (first L))
        (t                     (FINDCC (rest L)))  ) )
```

- Brojeve u listi pomnožiti sa 10.

```
(defun puta-10 (L)
 (if (null L)
      nil
      (if(numberp (first L)) (cons (* 10 (first L)) (puta-10 (rest L)))
                             (cons (first L)        (puta-10 (rest L)))
)))
```

Napisete po jedan primer za svaki prototip.

LISP, strategije upravljanja: ponavljanje preslikavanjem

STRATEGIJE UPRAVLJANJA - Ponavljanje preslikavanjem (mapcar, remove-if, remove-if-not, count-if, find-if)

- MAPCAR primjenjuje proceduru definisanu rezultatom prvog argumenta nad svim elementima liste koja se pojavljuje kao rezultat drugog argumenta.
  - Kao rezultat vraća listu pojedinačnih rezultata. Ako procedura zahteva više argumenata tada definiciji procedure sledi više lista pa se pri pozivima iz svake liste koristi po jedan element. Opšti oblik:

    ```
    (MAPCAR <procedura> <lista argumenatal> ...)
    ```

- REMOVE-IF primenjuje predikat definisan rezultatom prvog argumenta nad svim elementima liste koja se pojavljuje kao rezultat drugog argumenta.
  - Kao rezultat vraća listu iz koje su uklonjeni elementi za koje je predikat dao rezultat različit od NIL. Opšti oblik:

    ```
    (REMOVE-IF <predikat> <lista>)
    ```

- REMOVE-IF-NOT primenjuje predikat definisan rezultatom prvog argumenta nad svim elementima liste koja se pojavljuje kao rezultat drugog argumenta.
  - Kao rezultat vraća listu iz koje su uklonjeni elementi za koje je predikat dao rezultat NIL. Opšti oblik:

    ```
    (REMOVE-IF-NOT <predikat> <lista>)
    ```

LISP, strategije upravljanja: ponavljanje preslikavanjem

- COUNT-IF prebrojava one elemente liste koja se pojavljuje kao rezultat drugog argumenta za koje predikat definisan rezultatom prvog argumenta daje rezultat različit od NIL.
  - Kao rezultat vraća broj takvih elemenata. Opšti oblik:

```
(COUNT-IF <predikat> <lista>)
```

- FIND-IF kao rezultat vraća prvi element liste koja se pojavljuje kao rezultat drugog argumenta za koji je predikat definisan rezultatom prvog argumenta dao rezultat različit od NIL. Opšti oblik:

```
(FIND-IF <predikat> <lista>)
```

- Dati su rezultati izračunavanja zadanih formi:

```
(mapcar 'first '((a b c) (x y z)))                (a x)
   (mapcar 'equal '(1 2 3) '(3 2 1))              (nil t nil)
   (mapcar 'rest  '((a b c) (1 2 3)))             ((b c) (2 3))
   (mapcar 'first '((ovo i ono) (ono i ovo)))     (ovo ono)
   (mapcar 'rest  '((ovo i ono) (ono i ovo)))     ((i ono) (i ovo))
   (mapcar 'zerop '(2 1 0 -1 -2))                 (nil nil t nil nil)
   (mapcar 'evenp '(2 1 0 -1 -2))                 (t nil t nil t)
   (mapcar 'numberp '(1 + 1 = 2))                 (t nil t nil t)
(remove-if 'numberp '(1 + 1 = 2))                 (+ =)
   (remove-if-not 'numberp '(1 + 1 = 2))                  (1 1 2)
```

LISP, strategije upravljanja: ponavljanje preslikavanjem

- Dati su rezultati izračunavanja datog niza formi:

```
(defun dodaj-jabuku (L)
        (cons 'jabuka L))
                                              dodaj-jabuku
    (dodaj-jabuku '(hleb sir))
                                              (jabuka hleb sir)
    (mapcar 'dodaj-jabuku '((kruška breskva) (trešnja grozdje)))
                          ((jabuka kruška breskva) (jabuka trešnja grozdje))
```

- Napisati proceduru UDVOSTRUČI koja će dvostruko povećati iznose svih brojeva u listi brojeva koju dobija kao argument. Lista može sadržavati liste.

```
(defun udvostruci (L)
  (cond
      ((null L)                                              nil)
      ((numberp L)              (first(mapcar '* '(2) (list L))))
      (t  (cons (udvostruci (first L)) (udvostruci (rest L))))
  )
)
```

LISP, procedure kao argumenti procedura

PROCEDURE KAO ARGUMENTI PROCEDURA (funcall, apply, lambda)

- Primitiva FUNCALL primenjuje rezultat prvog argumenta na rezultate preostalih argumenata.
  - Broj argumenata koji slede prvog treba odgovarati broju argumenata koje očekuje procedura dobijena izračunavanjem prvog argumenta. Opšti oblik:

    ```
    (FUNCALL <procedura> <argument1> ...)
    ```

- Primitiva APPLY primenjuje rezultat prvog argumenta *na elemente liste* dobijene izračunavanjem drugog argumenta. Poslednji argument mora biti lista.
  - Broj elemenata liste treba da odgovara broju argumenata koje očekuje procedura dobijena izračunavanjem prvog argumenta. Opšti oblik:

    ```
    (APPLY <procedura> <lista argumenata>)
    ```

- Specijalna forma LAMBDA definiše bezimenu proceduru koja se izvodi samo na onom mestu na kom je definisana.
  Naročito se koristi u kombinaciji sa MAPCAR, FUNCALL i APPLY. Opšti oblik:

  ```
  (lambda (<parametar1> ...) <forma1> ...)
  ```

LISP, procedure kao argumenti procedura

- Dati su rezultati izračunavanja zadanih nizova formi:

```
(setf l '(sta sa ovom  listom))                    (sta sa ovom listom)
   (setf p1 'first)                                first
   (setf p2 'rest)                                 rest
   (funcall p1 l)                                  sta
   (funcall p2 l)                                  (sa ovom listom)
   (funcall p1 (funcall p2 l))                     sa
(funcall 'cons (funcall p1 l) (funcall p2 l))      (sta sa ovom listom)
   (setf l1 '(broj jedan))                         (broj jedan)
   (setf l2 '(broj dva))                           (broj dva)
   (setf l3 '((broj jedan) (broj dva)))            ((broj jedan) (broj dva))
   (setf p 'append)                                append
   (append l1 l2)                                  (broj jedan broj dva)
   (funcall p l1 l2)                               (broj jedan broj dva)
   (list l1 l2)                                    ((broj jedan) (broj dva))
   (funcall p (list l1 l2))                        ((broj jedan) (broj dva))
   (apply p (list l1 l2))                          (broj jedan broj dva)
   (apply      p l3)                               (broj jedan broj dva)
(apply p (list (apply  p l3) l1))          (broj jedan broj dva broj jedan)
   (defun izvrsi (p a)
     (funcall p a))                                izvrsi
   (izvrsi p1 l)                                   sta
```

LISP, procedure kao argumenti procedura

```
(izvrsi p2 l)                                   (sa ovom listom)
(izvrsi p1 (izvrsi p2  l))                      sa
(mapcar (lambda (n) (/ n 2)) '(16 8 4 2))       (8 4 2 1)
((lambda (n) (/ n 2)) 8)                        4
(setf l '(2))                                   (2)
(apply (lambda (n) (/ n 2)) l)                  1
(setf n 88)                                     88
((lambda (n) (/ n 2)) n)                        44
(setf l '((a b c) (x y z) (1 2 3)))             ((a b c) (x y z) (1 2 3))
(first l)                                       (a b c)
(mapcar 'first l)                               (a x 1)
(mapcar (lambda (l) (first l)) l)               (a x 1)
```

- Pretpostaviti da primitiva remove-if-not nije implementirana u LISP-u. Napisati proceduru RIN koristeći remove-if, lambda i funcall.

```
(defun rin (pre l)
   (remove-if (lambda (n) (member n (funcall 'remove-if pre l))) l))
```

ili:

```
(defun rin (p l)
   (setf a (remove-if p l))
   (remove-if (lambda (n) (member n a)) l) )
```

LISP, strategije upravljanja: ponavljanje iteracijom

- Pretpostaviti da primitiva count-if nije implementirana u LISP-u. Napisati proceduru CIF koristeći length, remove-if i funcall.

```
(defun cif (pre l)
     (funcall '- (length l) (length (remove-if pre l)) ) )
```

- Pretpostaviti da primitiva find-if nije implementirana u LISP-u. Napisati rekurzivnu proceduru FIF koristeći prototip procedure za traženje i funcall.

```
(defun fif (pre l)
      (cond ((null l)                                  nil)
            ((funcall pre (first l))                (first l))
            (t                              (fif pre (rest l)))  ) )
```

STRATEGIJE UPRAVLJANJA: Ponavljanje iteracijom (dotimes, dolist, return, do, do*, loop, prog1, prog2, progn)

- Specijalna forma DOTIMES izračuna <gornju granicu> i dobije broj prolaza n.
  - Celi brojevi od 0 do n-1 pridružuju se zatim <brojacu>. Za svaku vrednost brojača izvršava se telo. Nakon n-tog prolaza: vrednost <brojaca> postaje nedefinisana, izračuna se <rezultat>, što je ujedno i rezultat čitave forme. Opšti oblik:
    (DOTIMES (<brojac> <gornja granica> <rezultat>) <telo>)

LISP, strategije upravljanja: ponavljanje iteracijom

- Specijalna forma DOLIST izračuna <listu> čime se dobije lista elemenata.
  - Njeni se elementi zatim jedan po jedan pridružuju <elementu>.
  - Za svaku vrednost <elementa> izvršava se telo.
  - Nakon svih prolaza vrednost <elementa> postaje nedefinisana, izračuna se <rezultat>, što je ujedno i rezultat čitave forme. Opšti oblik:

```
(DOLIST ( <element> <lista> <rezultat> ) <telo> )
```

- RETURN bezuslovno prekida izvodjenje primitiva DOTIMES i DOLIST i vraća rezultat <izraz>. Opšti oblik:

```
(RETURN <izraz>)
```

- DO primitiva radi na sledeći način:
  1. U paraleli se izvedu sve <inicijalizacije> i <parametrima> se dodele tako izračunate početne vrednosti.
  2. Izračuna se <uslov završetka>.
     - Ako je rezultat različit od NIL izračunaju se <zavrsne forme> a DO vraća rezultat poslednje takve forme ili NIL ako nema završnih formi
     - Ako je rezultat uslova završetka jednak NIL izračunaju se sve <forme tela> .

LISP, strategije upravljanja: ponavljanje iteracijom

U paraleli se izračunaju sva <obnavljanja> i <parametrima> se dodele tako izračunate vrednosti.

Izvodjenje nastavlja tačkom 2.

- Pri tome vredi: Ako izvodjenje <tela> naiđe na RETURN onda izvođenje DO primitive završava, a rezultat je rezultat izraza iza RETURNa.

- Opšti oblik:

```
(DO (<parametar1) <inicijalizacija1> <obnavljanje1>) • • • )
    (<uslov_zavrsetka> <zavrsna forma1) . . .)
      <forma1 tela)
         • • •
)
```

- DO* primitiva radi jednako kao i DO, osim što se rezultati <inicijalizacija> i <obnavljanja> dodeljuju sekvencijalno.
  Opšti oblik je jednak kao i kod primitive DO.
- LOOP izvodi <forme tela> beskonačno sve dok se ne izvrši primitiva RETURN koja mora činiti deo tela. Opšti oblik:

  ```
  (LOOP <forme tela>)
  ```

- PROG1 eksplicitno određuje niz formi kao celinu. Kao rezultat izračunavanja

LISP, strategije upravljanja: ponavljanje iteracijom

čitavog niza vraća se rezultat prve forme. Opšti oblik:

```
(PR0G1 <forme tela>)
```

- PR0G2 eksplicitno odredjuje niz formi kao celinu. Kao rezultat izračunavanja čitavog niza vraća se rezultat druge forme. Opšti oblik:

```
(PR0G2 <forme tela>)
```

- PROGN eksplicitno odredjuje niz formi kao celinu. Kao rezultat izračunavanja čitavog niza vraća se rezultat poslednje forme. Opšti oblik:

```
(PROGN <forme tela>)
```

- Napisati proceduru EX koja ce izračunati m**n koristeći primitivu DOTIMES .

```
(defun ex (m n)
    (  let ( (rezultat 1) )
       (dotimes (brojac n rezultat)
          (setf rezultat (* m rezultat)))) )
```

- Napisati proceduru FAC koja ce izračunati n! koristeći primitivu DOTIMES.

```
(defun fac (n)
    (  let ( (rezultat 1) )
       (dotimes (brojac n rezultat)
          (setf rezultat (* (+ brojac 1) rezultat)))) )
```

LISP, strategije upravljanja: ponavljanje iteracijom

- Napisati rekurzivnu proceduru FAC koja ce izračunati n! bez upotrebe primitive DOTIMES.

```
(defun fac (n)
    (if (zerop n)  1
                   (* n (fac (- n 1))) ) )
```

- Napisati proceduru PREBROJI-NETEKUCE koja ce prebrojiti koliko u nekoj listi temperatura vode ima iznosa za koje voda nije u tečnom agregatnom stanju. Koristiti primitivu DOLIST.

```
(defun prebroji-netekuce (L)
   (  let ((rez 0))
      (dolist (ell L rez) (cond ((> ell 100) (setf rez (1+ rez)))
                                ((< ell   0) (setf rez (1+ rez))) ))) )
```

- Napisati rekurzivnu proceduru PREBROJI-NETEKUCE koja ce prebrojiti koliko u nekoj listi temperatura vode ima iznosa za koje voda nije u tečnom agregatnom stanju. Koristiti prototip procedure za prebrojavanje.

```
(defun prebroji-netekuce (L)
   (cond ((null L)       0)
         ((> (first L) 100) (1+ (prebroji-netekuce (rest L))))
         ((< (first L)   0) (1+ (prebroji-netekuce (rest L))))
         (t                 (+ 0 (prebroji-netekuce (rest L)))) ))
```

LISP, strategije upravljanja: ponavljanje iteracijom

- Napisati proceduru PREBR0JI-NETEKUCE koja ce prebrojiti koliko u nekoj listi temperatura vode ima iznosa za koje voda nije u tečnom agregatnom stanju. Koristiti primitive COUNT-IF.

```
(defun prebroji-netekuce (L)
    (+ (count-if (lambda (n) (> n 100)) L)
       (count-if (lambda (n) (< n   0)) L) ))
```

- Napisati proceduru FACTORIEL koja će računati faktorijel prirodnog broja. Upotrebiti formu DO* s praznim telom.

```
(defun factoriel(n)
   (do* (
          (brojac n (1-  brojac   ))
          (fakt   n (* brojac fakt))
        )
        ((equal 1 brojac) fakt)
    )
)
```

LISP, strategije upravljanja: ponavljanje iteracijom

- Pretpostaviti da ne postoji primitiva MEMBER, te napisati iterativnu proceduru MEMBER1 koristeći DO. Kao i MEMBER, MEMBER1 treba da vrati deo liste koji počinje elementom koji se traži.

```
(defun member1(clan L)
   (do (
           (Lista L (rest Lista))
       )
       ( (or (equal clan (first Lista))
             (null Lista              ) ) Lista)
    )
)
```