Funktion sisäisten apumuuttujien määrittely muodoilla `define`, `let` ja `let*`

Muuttujia ja funktioita voi määritellä paitsi ylätasolla (toplevel), niin myös funktioiden sisällä. Esimerkiksi:

```
(define (arvaa-luku arvattava-luku)
    (define arvaus (read))
    (cond ((= arvaus arvattava-luku) (display "Arvasit oikein!"))
          (else (display "Väärin meni, yritä uudestaan.")
                (newline)
                (arvaa-luku arvattava-luku)
          )
    )
)
```

määrittelee funktion `arvaa-luku` sisäisen muuttujan `arvaus` ja asettaa sen arvoksi käyttäjältä `read`-funktiolla luetun asian. Funktiota `read` siis kutsutaan joka kerta heti sen jälkeen kun on saavuttu `arvaa-luku` funktioon.

Muuttujaan `arvaus` voidaan viitata vain funktion `arvaa-luku` sisällä (missä tahansa kohdassa kyseisen `define`-rivin jälkeen aina funktiomääritelmän päättävään loppusulkuun asti), mutta ei missään sen ulkopuolella.

Hivenen *siistimpi* tapa (makuasia...) määritellä funktion sisäisiä apumuuttujia on käyttää `let`-muotoa, jonka syntaksi on seuraava:

```
(let ((muuttuja1 jotain1)
      (muuttuja2 jotain2)
      ...
      (muuttujaX jotainX)
     )
   muita lausekkeita, joista viimeisen tulos palautetaan koko let-muodon tuloksena
)
```

missä *muuttuja1* – *muuttujaX* ovat määriteltävien sisäisten apumuuttujien nimiä, ja *jotain1* – *jotainX* ovat mitä tahansa lausekkeita (vakioita, muuttujan nimiä tai kutsuja toisiin funktioihin), joiden arvot tai niiden palauttamat tulokset asetetaan vastaavan muuttujan arvoksi.

Edellisen sivun esimerkki let-muotoa käyttäen on tämän näköinen:

```
(define (arvaa-luku arvattava-luku)
   (let ((arvaus (read)))
      (cond ((= arvaus arvattava-luku) (display "Arvasit oikein!"))
            (else (display "Väärin meni, yritä uudestaan.")
                  (newline)
                  (arvaa-luku arvattava-luku)
            )
      )
   )
)
```

Hakasulkuja käyttävällä tyylillä ylläoleva taas näyttäisi tältä:

```
(define (arvaa-luku arvattava-luku)
   (let ([arvaus (read)])
      (cond [(= arvaus arvattava-luku) (display "Arvasit oikein!")]
            [else (display "Väärin meni, yritä uudestaan.")
                  (newline)
                  (arvaa-luku arvattava-luku)
            ]
      )
   )
)
```

Huomaa, että vaikka *jotain1 – jotainX* voivat viitata sekä koko funktion argumentteihin, että mahdollisissa ulommissa `let`-muodoissa määriteltyihin muuttujiin, niin ne eivät voi kuitenkaan viitata samassa `let`-muodossa määriteltyihin toisiin muuttujiin. Seuraavassa piilee yleinen virhe:

```
(define (alkio-listan-keskeltä lista)
  (let ([listan-pituus (length lista)]
        [keskikohta (inexact->exact (floor (/ listan-pituus 2)))]
       )
    (list-ref lista keskikohta)
  )
)
```

josta Racket-tulkki antaa heti virheilmoituksen: ”`listan-pituus: unbound identifier in module in: listan-pituus`”, sen takia että muuttujalle `keskikohta` yritetään asettaa arvo käyttäen muotoa jossa viitataan muuttujaan `listan-pituus`, joka ei kuitenkaan ole määritelty (tai ainakaan se ei ole *näkyvissä*) vielä tässä vaiheessa.

Onneksi tähän ongelmaan on yksinkertainen ratkaisu: Käytetään muotoa `let*` tavallisen `let`-muodon sijasta:

```
(define (alkio-listan-keskeltä lista)
  (let* ([listan-pituus (length lista)]
         [keskikohta (inexact->exact (floor (/ listan-pituus 2)))]
        )
    (list-ref lista keskikohta)
  )
)
```

Voidaan ajatella myös että `let` ja `let*` eroavat toisistaan vain siten, että edellinen suorittaa kaikki muuttujien alustukset samanaikaisesti (”rinnakkain”) kun taas jälkimmäinen alustaa ensin ensimmäisen muuttujan, sitten toisen, sitten kolmannen, jne, ja lopulta viimeisen (”peräkkäin”). Yleiseksi käytännöksi on vakiintunut käyttää edellistä muotoa, jos vain mahdollista, ja `let*` -muotoa vain silloin kuin se on tarpeen.

Huomaa että seuraavanlainen koodinpätkä:

```
(define (foo bar)
  (let* ([muuttuja1 jotain1]
         [muuttuja2 jotain2]
         [muuttuja3 jotain3]
        )
   muita lausekkeita, joista viimeisen tulos palautetaan koko let-muodon tuloksena
  )
)
```

vastaa suunnilleen seuraavaa , jossa on käytetty funktion sisäisiä `define`-määrittelyjä:

```
(define (foo bar)
  (define muuttuja1 jotain1)
  (define muuttuja2 jotain2)
  (define muuttuja3 jotain3)
   muita lausekkeita, joista viimeisen tulos palautetaan koko foo-funktion tuloksena
)
```

Mietintäharjoitus!

Sinun pitää määritellä useampia apumuuttujia funktion alussa, niin että myöhemmät viittaavat aikaisempiin, mutta et saa käyttää sisäisiä `define`:jä etkä `let*` -muotoa. Voitko tehdä sen pelkästään tavallista `let`-muotoa käyttäen?

Lambadasta enemmän

Aikaisemmassa harjoituksessa (Huom: harjoitus vasta huomenna sunnuntaina!), jossa käänsimme listassa `otukset` olevat kuvat ylösalaisin ja 45 astetta vastapäivään selitin, että lambda-muodot ovat eräänlaisia nimettömiä ”inline”-funktioita.

Ehkä parempi kuvaus olisi: lambda-muodot ovat funktioita, joille ei *vielä* ole annettu nimeä, tai ehkä *ei koskaan* annetakaan!

Olemme tähän mennessä nähneet kahdenlaisia `define`-määrityksiä, muuttujia määritteleviä:

```
(define piistä-puolet 1.5707963)
```

ja funktioita määritteleviä:

```
(define (foo b a r) (list a b r))
```

Nyt on aika paljastaa salaisuus. Jälkimmäinen muoto on itse asiassa aivan sama asia kuin jos sanoisimme:

```
(define foo (lambda (b a r) (list a b r)))
```

Toisin sanoen, yllä `foo` asetetaan tuon kolmiargumenttisen lambda-muodon nimeksi (tai toisinpäin ajatellen: kyseinen lambda-muoto asetetaan `foo`-muuttujan arvoksi). Eli oikeasti on olemassa vain yhdenlaisia `define`-määrityksiä, koska itse asissa `(define (foo b a r) (list a b r))` on vain Schemessä käytetty lyhennysmerkintä jälkimmäiselle muodolle, joksi se automaattisesti muutetaan jo koodin lukuvaiheessa (”syntaktista sokeria ohjelmoijille”).

Tästä näemme myös, etteivät funktiot Schemessä mitenkään oleellisesti poikkea muista asioista (vaikkapa merkkijonoista, listoista, jne.), vaan niitä voidaan samalla tavalla asettaa muuttujien arvoksi ja välittää muille funktoille argumentteina. Myöhemmin, klosuurien (closures) yhteydessä näemme, että funktiot voivat myös *palauttaa* arvonaan uusia lambda-muotoja (siis funktioita joilla ei ole vielä nimeä). Tätä tarkoitetaan kun sanotaan, että funktiot ovat ”ensimmäisen luokan kansalaisia” Schemen kaltaisissa ohjelmointikielissä.

Let lambdana

Paljastan vielä toisenkin salaisuuden. Itse asiassa

```
(let ((muuttuja1 jotain1)
      (muuttuja2 jotain2)
      ...
      (muuttujaX jotainX)
     )
   yksi tai useampia lausekkeita, joista viimeisen tulos palautetaan koko let-muodon tuloksena
)
```

on vain syntaktista sokeria seuraavalle muodolle:

```
((lambda (muuttuja1 muuttuja2 ... muuttujaX)
   yksi tai useampia lausekkeita, joista viimeisen tulos palautetaan koko lambda-muodon tuloksena
 )
  jotain1 jotain2 ... jotainX
)
```

Toisin sanoen, jossain vaiheessa ohjelmoijan sitä huomaamatta, Scheme-parserin lukiessa koodia sisään `let`-muoto muutetaan ”inline funktiokutsuksi”, jossa sen apumuuttujat *muuttuja1 – muuttujaX* muuttuvat lambda-muodon muuttujannimiksi, ja niitä vastaavat ”alustuslausekkeet” *jotain1 – jotainX* muuttuvat ko. lambda-muodon kutsuargumenteiksi, jonka jälkeen tuo ”nimetön inline-funktio” suoritetaan normaalilla tavalla, niin että viimeisen lausekkeen tulos palautuu koko homman tuloksena.

On helppo ymmärtää, että tässä tapauksessa syntaktinen sokeri on vain hyvästä, sillä lambdalla ilmaistuna koodi on yhtä hankalaa lukea kuin ns. top-down sähköposti: katse joutuu ensin etsimään alustusmuotoja sivun tai näytön alalaidasta, jonka jälkeen pitäisi siirtyä takaisin ylöspäin, kun taas `let`-muodossa koodin ajallinen suoritus ja sen järjestys tekstinä vastaavat paljon paremmin toisiaan, ja kunkin muuttujan nimi ja sitä vastaava alustusmuoto ovat lähellä toisiaan.

Nimetty let (”Named let”)

Niin sanottu “named let” –muoto on kätevä, jos haluaa välttää liian monen erillisen apufunktion kirjoittamista, varsinkin yksinkertaisia toistorakenteita kirjoitettaessa. Tässä muodossa avainsanan `let` ja *`(apumuuttuja alustumuoto)`* -pareja sisältävän listan väliin tulee symboli, jota ”kutsumalla” kyseisen let-muodon alkuun voi palata aina uudestaan. Tämän ymmärtää helpoiten ajattelemalla että ”named let” -muoto

```
(define (foo bar)
  (let label ((muuttuja1 jotain1)
              (muuttuja2 jotain2)
              ...
              (muuttujaX jotainX)
             )
   yksi tai useampia lausekkeita, joista viimeisen tulos palautetaan koko
let-muodon tuloksena
  )
)
```

voitaisiin yhtä hyvin kirjoittaa muodossa:

```
(define (foo bar)
  (define (label muuttuja1 muuttuja2 ... muuttujaX)
    yksi tai useampia lausekkeita, joista viimeisen tulos palautetaan
tämän sisäisen label-nimisen funktion tuloksena
  )
  (label jotain1 jotain2 ... jotainX)
)
```

Toisin sanoen, kyseessä on taas eräänlaisen ”inline-funktion” kutsumisesta, mutta toisin kuin tavallisen let-muodon tapauksessa, tällä kertaa kyseisellä funktiolla on oma nimi, jolla sitä voidaan kutsua, ei vain ensimmäisen kerran: `(label` *`jotain1 jotain2 ... jotainX`*`)` vaan myös kyseisen ”inline-funktion” sisältä, niin että saadaan aikaan toistoa.

Esimerkki valaiskoon asiaa.

Tässä luuppi nimeltä `loop` aloitetaan `n`:n arvolla nolla, ja niin kauan kun `n` ei ole kasvanut `yläraja`:n yli, sen arvo tulostetaan ja luupin alkuun palataan yhtä isommalla `n`:n arvolla:

```
(define (printtaa-lukuja-nollasta-ylöspäin yläraja)
   (let loop ([n 0])
      (cond [(> n yläraja) "valmis"]
            [else (display n)
                  (newline)
                  (loop (+ n 1))
            ]
      )
   )
)
```

tämä edellisen sivun selityksen mukaan vastaa seuraavaa määrittelyä:

```
(define (printtaa-lukuja-nollasta-ylöspäin yläraja)
   (define (loop n)
      (cond [(> n yläraja) "valmis"] ;; Palauta merkkijono ”valmis”
            [else (display n)
                  (newline)
                  (loop (+ n 1))
            ]
      )
   )
   (loop 0) ;; Kutsu sisäistä funktiota loop argumentti n:n arvolla 0.
)
```

eli asia on toteutettu siten, että sisäistä funktiota, jolle on annettu tässä nimi `loop`, kutsutaan ensin arvolla nolla, jonka jälkeen se kutsuu häntärekursiivisesti itseään, aina yhtä isommalla `n`:n arvolla, kunnes mennään `yläraja`:n yli.