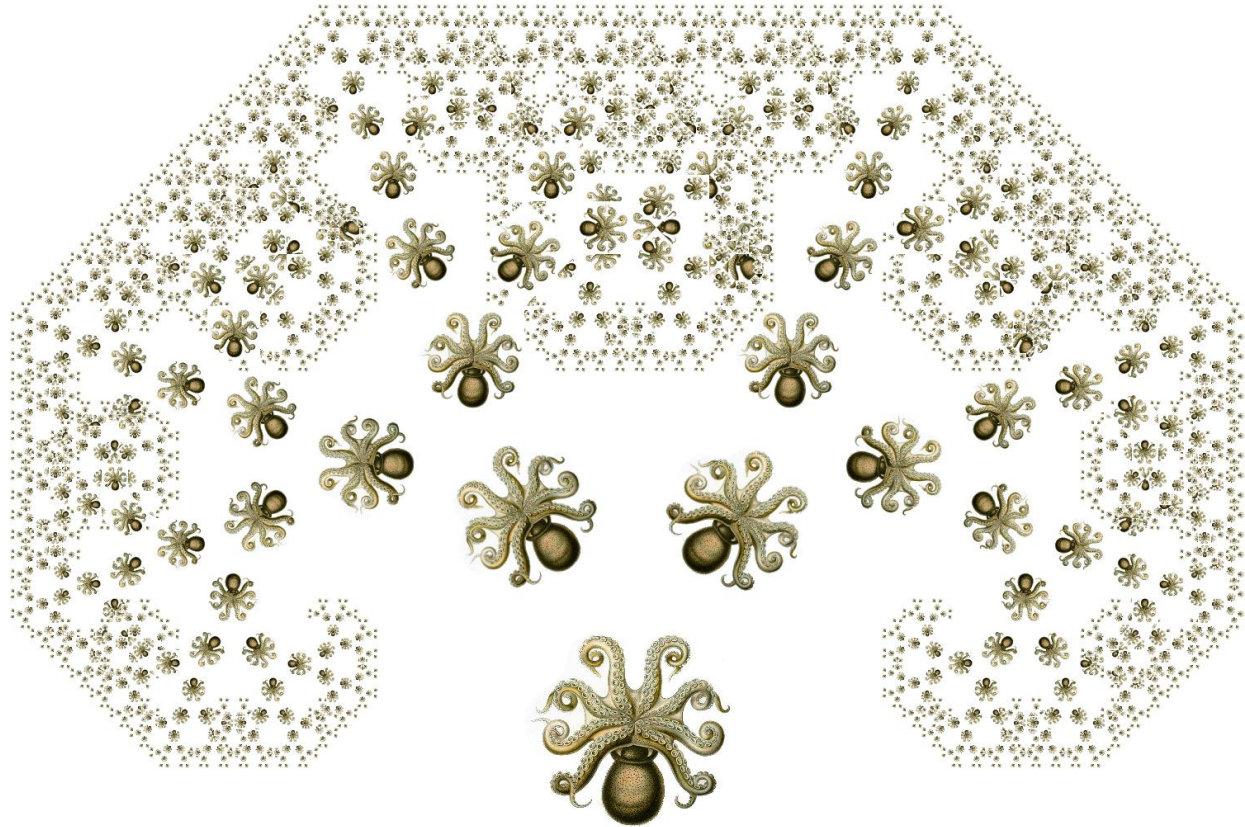


OHJELMOINNIN PERUSTEET RACKET-KIELELLÄ



LA 28.9.–6.10. KLO 10–15

OPISTOTALO, MIKKO-LUOKKA, HELSINGINKATU 26

KUKA OPETTAA?

- Antti Karttunen, vm. 1968
- Ohjelmoinnin suhteen täysin itseoppinut, aloittanut noin vuonna 1985 (Commodore 64-kotitietokoneella.)
- Lisäksi mm. matematiikan, kieliteknologian (ent. tietokonelingvistiikka) ja kombinatoriikan opintoja, Helsingin Yliopistolta ja TKK:lta
- On ohjelmoinut sekä harrastuksenaan että ammatikseen
- Opettaa ohjelmointia myös Metropolia AMK:ssa, vuodesta 2011 –

MIKÄ OHJELMOINNISSA ITSEÄNI KIEHTOO? (EI PELKÄSTÄÄN HYÖDYLLISTÄ)

- Ohjelmointi luovuuden ilmaisukanavana: Perinteisistä taidemuodoista mm. kirjallisuus, (sävelletty) musiikki, mosaiikit, matonkudonta, jne. ovat perustuneet diskreettien ("digitaalisten") symbolien (kirjaimet, nuotit) asettamiseen peräkkäin tai ainakin vierekkäin ja lomittain. Ohjelmointi perustuu samaan (ohjelmat kirjoitetaan), mahdollisuudet vain ovat paljon laajemmat: "interaktiivinen fiktio" (seikkailupelit, jne.), "demoskene", "livekoodaus", "generatiivinen taide"
- Ohjelmat (pienet ja yksinkertaiset, sinänsä hyödyttömät ja vailla tarkoitusta) itsessään tutkimuksen kohteena, Cellular Automata, Simple Programs, "New Kind of Science"

KUINKA NOPEASTI OPIN OHJELMOIMAAN? (21 PÄIVÄÄ VAI 10 VUOTTA?)

- Peter Norvig, **Teach Yourself Programming in Ten Years**
<http://norvig.com/21-days.html>
- Pianoa oppii soittamaan... soittamalla pianoa! Ohjelmoimaan oppii ... ohjelmoimalla. Jos ohjelmoit itseäsi kiinnostavia asioita, ensin helpompia, sitten haastavampia, tuskin edes huomaat ajan kulumista!
- Ohjelmointikielistä C++ ja Java on tarkoitettu käytettäväksi suurissa tiimeissä -> paljon ”boilerplate”-koodia ja muuta turhauttavaa ”byrokratiaa”. Koodi pitää ensin kääntää, joten palaute ei ole välitöntä.
- Interaktiivista tulkkia käyttävät kielet kuten Lisp (Scheme, Racket) ja Python antavat välittömän palautteen ohjelmoijan onnistumisesta -> hyviä oppimiseen. Näissä kielissä ei ole myöskään turhaa kirjoitettavaa (esimerkiksi datatyyppejä ei tarvitse määritellä koska kieli päättelee ne puolestasi). Funktioiden toiminnan voi kokeilla heti.

LISP, HISTORIA

- Lisp-ohjelmointikieli keksitty jo vuonna 1958! ("toiseksi vanhin ohjelmointikieli", mikäli konekieliä ei lasketa), John McCarthy 1927-2011
- Perustui alun perin loogikkojen (mm. Alonzo Church) merkintöihin (ns. "lambda-laskenta", kynällä ja paperilla, jo 1930-luvulla)
- Vakiintui sittemmin (1960 – 1980 luvuilla) "tekoäly"-tutkimuksen kieleksi. Molemmilla sanoilla, "lambda-laskenta" sekä "tekoäly" on ollut pelottava kaiku perusjantus-ohjelmoijien keskuudessa. Ehkä sen vuoksi kieli ei ole luultavasti koskaan ollut top-5 käytetyimmän kielen joukossa
- Toisaalta kieli ei ole myöskään osoittanut koskaan mitään kuolemisen merkkejä, vaan päinvastoin, on neljännesvuosidatan välein poikinut uusia murteita, mm. Scheme, 1975 ja Clojure, 2007

LISP, HISTORIA 2

- Nykyään Lispin (Schemen) ”symboliset” piirteet (esimerkiksi listat ja automaattinen muistinvapautus, eli ”garbage collection”), jotka mahdollistivat 1960-luvun ”tekoäly”-tutkimuksen, ovat vakiopiirteitä useimmissa ohjelmointikielissä, mm. Pythonissa. Samoin muita, ensin Lispissä keksittyjä asioita kopioidaan pikkuhiljaa muidenkin kielten repertuaariin -> muut kielet ”lispimäistyvät”, vaikka syntaksi ei olekaan yhtä yksinkertainen
- Kieltä ei ole enää pitkään aikaan mielletty erikoisalan kieleksi, vaan yleiskieleksi, jolla voidaan ohjelmoida hyvin monenlaisia asioita

LISP: TÄRKEIMMÄT KIELEN MURTEET (NYKYÄÄN)

- Common Lisp: vuonna 1994 standardoitu Lisp -> suuri, raskas toteutus, laajat kirjastot, käytetään enimmäkseen ”raskaassa ohjelmointiteollisuudessa”. Useita toteutuksia.
- Scheme, 1975, melko minimalistinen, kevyt versio Lispistä. Myös useita vapaasti ladattavia versioita.
 - Racket, aikaisemmalta nimeltään PLT Scheme, 1994, uskottelee olevansa uusi Lisp-ryhmän kieli, mutta on itseasiassa vain Schemen murre
- Clojure, 2007, Java-virtuaalikoneen päällä toimiva täysin uusi Lisp-murre. Vain yksi toteutus, mutta vapaata koodia. Lupaava tulokas!
- Erilaiset ohjelmien sisäiseen skriptaus-käyttöön tarkoitettut Lisp-murteet, usein melko epästandardeja: Guile, Emacs Lisp, Script-Fu (GIMP), eräät musiikinteko- ja livekoodaus-ohjelmat

SYNTAKSIVERTAILU:

KIELI 1 VS. KIELI 2

rosettacode.org/wiki/Count_the_coins

```
(define (countcoins cents coins)
  (cond
    ((null? coins) 0)
    ((negative? cents) 0)
    ((zero? cents) 1)
    (else
     (+ (countcoins cents (rest coins))
        (countcoins (- cents (first coins))
                     coins)
      )
    )
  )
)
```

```
function countcoins(t, o) {
  'use strict';
  var targetsLen = t + 1;
  var operandsLen = o.length;
  t = [1];

  for (var a = 0; a < operandsLen; a++)
  {
    for (var b=1; b < targetsLen; b++)
    {
      t[b] = t[b] ? t[b] : 0;
      t[b] += (b < o[a]) ? 0
                : t[b-o[a]];
    }
  }

  return t[targetsLen - 1];
}
```


LOT'S OF IRRITATING SILLY PARENTHESES? (SULUT JA SULUTUKSET)

- "Lisp" = “Lots of Irritating Silly Parentheses (paljon ärsyttäviä typeriä sulkumerkkejä) vaiko “Lisp Is Syntactically Pure” (Lispin syntaksi on pelkistetty) ?
- Sulkujen oikean käytön oivaltaminen ei ole vaikeaa. Niitä ei tarvitse ruveta laskemaan, vaan riittää vain kun kävelee portaita alas ja ylös. Alkusulku (vasen sulku): yksi porras alemmaksi. Loppusulku (oikea sulku): yksi porras ylemmäksi.

SULUT JOTKA VASTAAVAT TOISIAAN ("MATCHING PARENTHESES")

- Jos piirrämme sulutuksesta portaat alas ja ylös, ja vedämme luotinuoran jokaisesta laskevasta portaasta (alkusulusta) ensimmäiseen siitä oikealla olevaan, samalle tasolle nousevaan portaaseen, olemme silloin löytäneet vastaavan loppusulun. Kyseiset sulut siis ovat toistensa "vastinsulkuja", eli "mätsäävät" toisiinsa.

VÄÄRÄT SULUTUKSET, OSA 1

LOPPUSULKU PUUTTUU

- Jos ammunne lasersäteen laskevasta portaasta (alkusulusta) oikealle, eikä säde törmää mihinkään siitä oikealla olevaan, samalle tasolle nousevaan portaaseen, silloin kyseisellä alkusululla ei ole vastaavaa loppusulkua parinaan, ja kyseinen sulutus on siis virheellinen muodoltaan, jota tulkki ei pysty lukemaan loppuun asti.

VÄÄRÄT SULUTUKSET, OSA 2

LOPPUSULKUJA LIIKAA

- Jos yksikään nouseva porras nousee ”katutason yläpuolelle” (jolta alun perin lähdimme laskeutumaan), niin myös silloin kyseisessä sulutuksessa on virhe, sillä silloin kyseinen loppusulku ei ole minkään alkusulun vastinpari.

SULKULAUSEKKEET

- Jos saamme alku- ja loppusulut vastaamaan toisiaan, voimme kommunikoida Racket-tulkin kanssa. Kommunikointi tapahtuu muotoa
 - (tee-jotain)
 - (laske-tai-tee-jotain jollekin1)
 - (laske-tai-tee-jotain jollekin1 jollekin2)
 - (laske-tai-tee-jotain jollekin1 jollekin2 jollekin3)
 - jne.


olevilla lausekkeilla. (Viimeisen sulun jälkeen on painettava enteriä)

- Lausekkeen ensimmäistä jäsentä, "laske-tai-tee-jotain" voi ajatella käskylauseen verbinä. Sen seuraavia jäseniä, joita voi siis olla nolla tai useampia, kutsutaan sen argumenteiksi. Niitä voi ajatella käskylauseen objekteina tai muina määreinä, tähän tyyliin:

(vie roskat roskeen)

VAKIOT ("LITERAALIT")

Tietyntyypiset tietoalkiot *evaluoituvat* (siis saavat arvokseen) aina itsensä, toisin sanoen, jos syötät mitään seuraavan tyyppisistä asioista tulkille, sinun pitäisi saada tulokseksi tismalleen sama asia takaisin:


- Kokonaisluvut, esimerkiksi 3, -2 ja 0
- Desimaaliluvut, esimerkiksi 3.141592653589793
- Merkkijonot, "esimerkiksi tämä on merkkijono"
- Kuvat, esimerkiksi  tulkki-ikkunaan copy-and-paste tekniikalla tuotaessa

Ylläoleva kuva löytyy osoitteella:

<http://commons.wikimedia.org/w/index.php?search=haeckel+erythraea>

MUUTTUJAT

Muuttuja on periaatteessa vain jonkin asian nimi. Jos muuttujannimen syöttää tulkille, niin tuloksena on se asia, jonka nimi se on. Muuttujia voi määritellä (define muuttujannimi *jokin-asia*) lausekkeilla. Esimerkkejä:

- (define pii-karkea-likiarvo 3)
- (define pii 3.141592653589793)
- (define aika-ja-paikka
"La 28.9.-6.10. Klo 10-15, Opistotalo, Mikko-
luokka, Helsinginkatu 26")
- (define tyhjä-merkkijono "")
- (define sammakko )

MUUTTUJAT VOIVAT MYÖS MUUTTUA

Muuttujan arvo voi muuttua:

- `(define haukia 2) ;;` Määritellään muuttuja `haukia`, alkuarvonaan kokonaisluku `2`.
- `haukia -> 2` (Muuttujan `haukia` arvo on nyt kaksi)
- `(define haukia (+ haukia 1)) ;;` Kasvatetaan muuttujan `haukia` arvoa yhdellä.
- `haukia -> 3` (Muuttujan `haukia` arvo on nyt kolme)
- `lohia` (Muuttuja pitää olla määritelty ennen kuin siihen voidaan viitata, muuten tulee virheilmoitus):
 - *lohia: undefined; cannot reference an identifier before its definition*

FUNKTIOITA: PERUSLASKUTOIMITUKSET

- $(+ \ 1 \ 2)$ laskee yhteen luvut yksi ja kaksi, ja antaa tuloksenaan niiden summan 3
- $(+ \ 1 \ 2 \ 3 \ 4)$ laskee yhteen luvut yksi, kaksi, kolme ja neljä, ja antaa tuloksenaan niiden summan 10
- $(- \ 5 \ 3)$ vähentää luvusta viisi luvun kolme, ja antaa tuloksenaan niiden erotuksen 2
- $(- \ 5)$ antaa tulokseksi luvun viisi vastaluvun, miinus viisi: -5
- $(* \ 3 \ 5)$ kertoo yhteen luvut kolme ja viisi, ja antaa tuloksenaan niiden tulon 15
- $(/ \ 10 \ 2)$ jakaa luvun kymmenen luvulla kaksi, ja antaa tuloksenaan niiden osamäärän 5
- $(/ \ 7 \ 2)$ jakaa luvun seitsemän luvulla kaksi, ja antaa tuloksenaan niiden osamäärän $3 \text{ ja } 1/2$

INPUT/OUTPUT-FUNKTIOITA, TULOSTUS

Seuraavat funktiot kommunikoivat käyttäjän tai ulkomaailman kanssa muutenkin kuin vain argumenttiensa tai palauttamansa tuloksen kautta. Ne ovat siis ensimmäiset esimerkit ns. ”epäpuhtaista”, sivuvaikutuksellisista funktioista:

- `(print "Hei maailma, onko siellä ketään?")` tulostaa argumenttinsa, tässä tapauksessa kyseisen merkkijonon.
- Jos syötät ylläolevan tulkki-ikkunaan, niin sen jälkeen pitäisi näkyä sama merkkijono: `"Hei maailma, onko siellä ketään?"` Huomaa että se ei ole kuitenkaan `print`-funktion palauttama arvo, vaan vain sen tulostama merkkijono. (Virallisesti `print`-funktio ei palauta mitään arvoa). Tulkki-ikkunassa näkyvästä tekstistä tätä ei voi kuitenkaan päätellä.
- `(newline)` tulostaa rivinvaihdon, eikä palauta mitään.

INPUT/OUTPUT-FUNKTIOITA, LUKU

Myös luku-funktiot ovat ”epäpuhtaita”, sivuvaikutuksellisia funktioita, sillä niissä tulos ei riipu ajattomasti argumenteista (tässä tapauksessa ei argumentteja lainkaan), vaan siitä mitä käyttäjä kulloinkin keksii syöttää kaoottisesta ”ulkomaailmasta”:

- (`read`) lukee käyttäjältä yhden tietoalkion, joka voi olla mitä tahansa tyyppiä (kokonaisluku, symboli, ”merkkijono lainausmerkkien sisällä”, jne.). Käytä tätä esimerkiksi kokonaislukujen lukemiseen käyttäjältä.
- (`read-line`) lukee käyttäjältä yhden rivin tekstiä, josta muodostetaan joka tapauksessa aina merkkijono, vaikka käyttäjä syöttäisi sisään kokonaisluvunkin. Käytä tätä merkkijonojen lukemiseen käyttäjältä.

Huom, molemmat funktiot avaavat tulkki-ikkunaan ruudun levyisen palkin (oikeassa laidassa keltainen nappi ”eof”) johon käyttäjän tulee syöttää lukunsa/tekstinsä ja painaa enteriä.

TOTUUSARVOT, "TRUE" JA "FALSE"

Useimmat funktiot jotka tarkistavat tai vertailevat jotakin palauttavat arvonaan joko arvon $\#t$ ("true", siis tosi) tai $\#f$ ("false", siis "valetta" eli epätosi):

- (`equal?` "marsu" "mursu") vertailee merkkijonoja "marsu" ja "mursu", toteaa etteivät ne ole samat, ja palauttaa $\#f$:n, eli epätoden.
- (`zero?` `n`) tarkistaa onko muuttujan `n` arvo nolla. Jos on, palauttaa $\#t$:n muuten $\#f$:n. (Siis mikäli `n` mitä tahansa muuta kuin nolla.)
Huom! Tämä funktio olettaa että argumentti on aina jokin luku, joten esimerkiksi (`zero?` "nolla") tuottaa virheilmoituksen
 - *zero?: contract violation, expected: number? given: "nolla"*
- (`=` `luku1` `luku2`) vertailee lukuja `luku1` ja `luku2` jonka perusteella palauttaa joko $\#t$:n (jos yhtäsuuret) muuten $\#f$:n (jos erisuuret). Muuten melkein sama kuin `equal?`, mutta voidaan soveltaa vain lukuihin, muuten tulee virheilmoitus.

LUKUJEN VERTAILUFUNKTIOT

Seuraavia funktioita voidaan soveltaa vain lukuihin, muuten tulee virheilmoitus:

- `(< luku1 luku2)` vertailee lukuja `luku1` ja `luku2` jonka perusteella palauttaa joko `#t:n` (jos `luku1` pienempi kuin `luku2`) muuten `#f:n`.
- `(<= luku1 luku2)` vertailee lukuja `luku1` ja `luku2` jonka perusteella palauttaa joko `#t:n` (jos `luku1` pienempi tai yhtä suuri kuin `luku2`) muuten `#f:n`.
- `(> luku1 luku2)` vertailee lukuja `luku1` ja `luku2` jonka perusteella palauttaa joko `#t:n` (jos `luku1` suurempi kuin `luku2`) muuten `#f:n`.
- `(>= luku1 luku2)` vertailee lukuja `luku1` ja `luku2` jonka perusteella palauttaa joko `#t:n` (jos `luku1` suurempi tai yhtä suuri kuin `luku2`) muuten `#f:n`.

TYYPINTARKISTUSFUNKTIOT

Tyypintarkistusfunktiot palauttavat arvonaan `#t:n`, toden, vain mikäli annettu argumentti on juuri sitä tyyppiä:

- `(number? x)` tarkistaa onko `x` yleensä mikään luku, esimerkiksi kokonais- tai desimaaliluku.
- `(integer? x)` tarkistaa onko `x` nimenomaan kokonaisluku. Esimerkiksi `(integer? 3) -> #t` mutta `(integer? 3.14159) -> #f`
- `(string? x)` tarkistaa onko `x` merkkijono. Esimerkiksi `(string? "Joo, olen merkkijono") -> #t`.
- `(symbol? x)` tarkistaa onko `x` ns. symboli (Selitetään myöhemmillä tunneilla!). Esimerkiksi `(symbol? 'cembalo) -> #t`.
- `(list? x)` tarkistaa onko `x` lista (Selitetään myöhemmillä tunneilla!). Esimerkiksi `(list? '(olen lista)) -> #t`.

VERTAILUFUNKTIOIDEN KÄÄNTÖ

Vertailufunktioita tai niiden palauttamia totuusarvoja voi kääntää vastakohdakseen funktiolla `not` ("ei"):

- `(not jotakin)` tarkistaa onko argumentti `jotakin` epätosi, eli `#f`, jolloin palauttaa `#t:n` muuten `#f:n`. Esimerkkejä:
 - `(not (equal? asia1 asia2))` tarkistaa eroavatko `asia1` ja `asia2` jotenkin toisistaan, jolloin palauttaa `#t:n`, muuten `#f:n` (jos ne sisältävät samanlaiset arvot).
 - `(not (zero? n))` tarkistaa onko luku `n` jotain muuta kuin nolla. Jos on, palauttaa `#t:n` muuten `#f:n`, mikäli `n` on nolla.

VERTAILUFUNKTIOIDEN YHDISTELY

Vertailufunktioita tai niiden palauttamia totuusarvoja voi yhdistellä funktioilla `and` ("ja") sekä `or` ("tai"):

- `(and arg1 arg2 ... argn)` tarkistaa onko kaikki annetut argumentit `arg1`:stä `argn`:ään tosia "laajassa merkityksessä", eli mitä tahansa muuta kuin epätosia (`#f`:iä), ja jos ovat, niin palauttaa viimeisen alilausekkeen `argn`:än arvon, muuten `#f`:n. Esimerkki:
 - `(and (integer? n) (> n 3))` tarkistaa onko luku `n` kokonaisluku (ei esimerkiksi desimaaliluku `3.5`) ja suurempi kuin kolme, jolloin palauttaa `#t`:n, muuten `#f`:n.
- `(or arg1 arg2 ... argn)` tarkistaa onko yksikään annetuista argumenteista `arg1`:stä `argn`:ään tosi "laajassa merkityksessä", eli mitä tahansa muuta kuin `#f`, ja jos on, niin palauttaa ensimmäisen sellaisen alilausekkeen arvon, muuten `#f`:n jos kaikki olivat epätosia.
 - `(or (< n 2) (> n 3))` tarkistaa onko luku `n` joko pienempi kuin kaksi, tai suurempi kuin kolme, jolloin palauttaa `#t`:n, muuten `#f`:n.

ERITYISMUODOT, BEGIN

Ns. erityismuodot (*”special forms”*), vaikka saattavat näyttää aivan tavallisilta funktioilta, evaluoivat alilausekkeensa/argumenttinsa jotenkin tavallisesta poikkeavasti.

- (`begin joku1 joku2 ... jokun`) suorittaa kaikki alilausekkeet `joku1:stä jokun:ään` peräkkäin, mutta palauttaa vain viimeisen (`jokun:n`) tuloksen tuloksenaan. Kaikkien edeltävien alilausekkeiden tulokset siis heitetään pois. (Vain niiden sivuvaikutus on tärkeä.)
- Esimerkki: (`begin (print x) (newline) x`) tulostaa muuttuja `x:n` sisällön, sen perään rivinvaihdon, ja lopuksi palauttaa juuri saman `x:n` sisällön tuloksenaan.
- Huom! `define`-käskyllä määritellyt funktiot sisältävät valmiiksi eräänlaisen ”näkymättömän” `begin`-muodon: määritellyn funktion kaikki alilausekkeet suoritetaan, mutta vain viimeisen arvo palautetaan.

ERITYISMUODOT, IF

- (if ehto then-haara else-haara) suorittaa ja antaa then-haaran tuloksenaan, mikäli ehto on tosi (*laajassa mielessä eli siis mitä tahansa muuta kuin #f*), muuten suorittaa ja antaa else-haaran tuloksenaan mikäli ehto on #f (epätosi).

- Esimerkiksi seuraava lauseke:

```
(if (odd? n) ;; Ehtolause, onko n pariton?  
    (print "n on pariton") ;; Then-haara  
    (print "n on parillinen") ;; Else-haara  
)
```

ei koskaan suorita molempia print-käskyjä samalla kertaa.

Huom! Jos kummassakaan haarassa haluaa suorittaa useampia käskyjä, niin ne on ryhmiteltävä begin-muodon sisään.

ERITYISMUODOT, COND

Useampia `if`-ehtolauseita voidaan ketjuttaa sisäkkäin, mutta koodi on pian hankalaa lukea:

```
(if (< n 1) ;; Ulompi ehtolause.  
    "n on pienempi kuin yksi" ;; Then-haara  
    (if (< n 2) ;; Sisempi ehtolause.  
        "n on >= 1 mutta pienempi kuin kaksi"  
        "n on >= 2."  
    )  
)
```







Tämä voidaan muuttaa seuraavaksi `cond`-lauseeksi (eng. "conditional"):

```
(cond  
  ((< n 1) "n on pienempi kuin yksi")  
  ((< n 2) "n on >= 1 mutta pienempi kuin kaksi")  
  (else "n on >= 2.")  
)
```

KUVAFUNKTIOITA: KUVIEN PYÖRITYS

Huom! kaikki kuvankäsittelyfunktiot tarvitsevat määritelmä-ikkunan alkuun seuraavan lisämääreen: `(require 2htdp/image)`

Lisäinfoa: <http://docs.racket-lang.org/teachpack/2htdpimage.html>

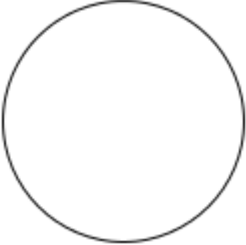


- `(flip-horizontal )` antaa kuvan peilikuvan: 
- `(flip-vertical )` antaa kuvan ylösalaisin: 
- `(rotate 45 )` kiertää kuvaa 45 astetta vastapäivään: 

KUVAFUNKTIOITA: KUVIEN YHDISTÄMINEN

<http://docs.racket-lang.org/teachpack/2htdpimage.html>

- (`beside` ) antaa kuvan jossa kuvat vierekkäin: 

- (`above` ) antaa kuvan jossa kuvat yllekkäin: 

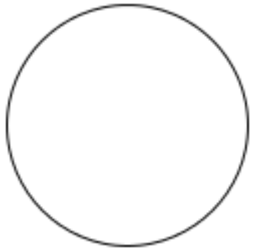
- (`overlay`  ) antaa kuvan jossa kuvat päällekkäin: 

- Funktio `underlay`, muuten sama, mutta argumentit toisinpäin. Katso myös funktiot `overlay/align`, `overlay/offset` ja `overlay/align/offset` otsikon URL-osoitteesta.

GEOMETRISTEN MUOTOJEN PIIRTÄMINEN






<http://docs.racket-lang.org/teachpack/2htdpimage.html>

- `(circle 60 "outline" "black")` antaa tuloksenaan kuvan jossa kuudenkymmenen pikselin säteinen musta ympyrä
- `(circle 40 "solid" "red")` antaa tuloksenaan kuvan jossa neljäkymmenen pikselin säteinen punainen kiekko
- `(square 80 "outline" "blue")` antaa tuloksenaan kuvan jossa sinisellä piirretty neliön ääriverrat, sivun pituus 80 pikseliä
- `(square 40 "solid" "red")` antaa tuloksenaan vihreän, 40 pikseliä leveän ja korkean neliön
- `(triangle 50 "solid" "red")` antaa tuloksenaan punaisen tasasivuisen kolmion, jonka jokaisen sivun pituus 50 pikseliä
- `(rectangle 89 55 "solid" "forestgreen")` antaa tummanvihreän 89 (lev) x 55 (kork) pikselin kokoisen suorakaiteen



MUITA KUVAFUNKTIOITA

<http://docs.racket-lang.org/teachpack/2htdpimage.html>

- (`frame` ) antaa tuloksenaan kuvan jossa alkuperäisen kuvan reunoihin on piirretty raamit (hyvä ”debuggaukseen”):
- (`scale 1.5` ) antaa tuloksenaan kuvan jossa alkuperäistä kuvaa skaalattu puolitoista kertaa suuremmaksi:
- (`scale 0.5` ) antaa tuloksenaan kuvan jossa alkuperäistä kuvaa on skaalattu kaksikertaa pienemmäksi
- (`image-width` ) antaa tuloksenaan kuvan leveyden pikseleissä, tässä tapauksessa luvun 89 (sama suorakaide kuin edellisen sivun viimeisessä esimerkissä)
- (`image-height` ) antaa tuloksenaan kuvan korkeuden pikseleissä, tässä tapauksessa luvun 55

