Scheme-kesäkurssi luento 1

Riku Saikkonen

24. 6. 2009

Sisältö

1. Kurssi
2. Scheme-kieli
3. SICP luku 1
4. SICP luku 2

Kurssijärjestelyt

- T-106.6200 Ohjelmistotekniikan erikoiskurssi, 6–8 op
- Kurssikirja: Abelson, H., Sussman, G. J., Sussman, J.: *Structure and Interpretation of Computer Programs*, 2nd edition, MIT Press tai McGraw-Hill 1996
- Kotitehtävät (0–5), tentti (0–5), vapaaehtoinen harjoitustyö (hyl/hyv/+1)
- 6–7 luentoa
- Kotisivu: `http://www.cs.hut.fi/~scheme/`

Kurssin sisältö

- perustuu entiseen TKK:n Scheme-kurssiin, mutta on laajempi
- kurssikirja loppuun asti, entinen kurssi loppui kohtaan 5.1
- ja lisäaiheita, mm. continuation passing style ja Common Lispin CLOS-oliojärjestelmä
- kotitehtäväkierrosten 1–4 tehtävät ovat entiseltä Scheme-kurssilta (mutta helpoimpia on jätetty pois), 5. kierrokseen tulee uusia tehtäviä
- kurssikirja kannattaa lukea läpi, esim. kotitehtäviä tehdessään

Kurssikirja SICP

- ohjelmoinnin perusoppikirja, oli pitkään käytössä MIT:n ensimmäisellä ohjelmointikurssilla
- esittelee myös lyhyesti tietotekniikan osa-alueita
- miksi ohjelmointikielet ovat sellaisia kuin ovat?
- kantava teema: abstraktioiden tekeminen eri tavoilla
- tulkkiosio (luku 4) opettaa:
  - pienen ohjelmointikielen tulkin tekeminen ei ole vaikeaa
  - kokeilemaan ja miettimään "mitä tapahtuisi jos kielessä olisi . . ."
  - ja antaa esimerkkejä harvinaisemmista ohjelmointiparadigmoista (esim. logiikkaohjelmointi)
- kääntäjäosio (luku 5): assembler-kieli, kääntäjän perustoiminnot, muistinhallintaa

Sisältö

1. Kurssi
2. Scheme-kieli
3. SICP luku 1
4. SICP luku 2

Scheme-kielen suunnitteluperiaate

"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today. "

– R^{5}RS (Scheme-kielen määrittely)

Scheme lyhyesti

- dynaamisesti ja vahvasti tyypitetty, leksikaalinen skooppi
- tulkattavissa oleva kieli
- automaattinen muistinhallinta (roskankeruu)
- prefix-syntaksi: (⟨*operaattori*⟩ ⟨*operandi*⟩ . . .)
- puolifunktionaalinen kieli: tukee funktionaalista ohjelmointityyliä muttei pakota siihen
- ei eroa lauseilla ja lausekkeilla (melkein mitä tahansa voi laittaa minkä tahansa sisälle)
- jokaisella lauseella on (paluu)arvo
- hyvä tuki abstraktioiden tekemiselle: mm. ensimmäisen luokan funktiot ja makrot
- "koodi on dataa": Schemellä on helppo lukea Scheme-koodin näköistä tekstiä listarakenteeksi

Esimerkkejä syntaksista

Tyypillinen kertomafunktio

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

cond-esimerkki

```
(define (fact n)
  (cond ((= n 0) 1)
        (else (* n (fact (- n 1))))))
```

and ja or ovat katkaisevia (epätyypillistä koodia)

```
(define (fact n)
  (or (and (= n 0) 1)
      (* n (fact (- n 1)))))
```

Esimerkki suunnitteluperiaatteesta

- ”ei eroa lauseilla ja lausekkeilla (melkein mitä tahansa voi laittaa minkä tahansa sisälle)”
- C:ssä ja Javassa on kaksi iffiä, lause ja lauseke:
  `if (x == 3) b = 5; else b = 8;`
  `b = (x == 3) ? 5 : 8;`
- Schemessä sama `if` käy molempiin:
  `(if (= x 3) (set! b 5) (set! b 8))`
  `(set! b (if (= x 3) 5 8))`
- Miksei Javassa tai C:ssäkin voisi toimia esim:
  `b = ({ if (x == 3) 5; else 8; });`
  (gcc:ssä melkein toimiikin, sen omalla C:n laajennuksella)

Scheme-toteutuksista

- Scheme (varsinkin R^{5}RS) on melko pieni kieli
  - $\Rightarrow$ helppo toteuttaa
    - mutta varsinkin standardikirjasto on pieni
- Scheme-toteutuksissa on yleensä paljon omia lisäyksiä erityisesti standardikirjastoon, usein myös tietotyyppeihin
  - ja rajapintoja muihin kieliin ja isoihin kirjastoihin kuten ikkunointiin, tietokantoihin ja OpenGL:ään
  - mutta toteutusten erilaisuus haittaa käytännössä
- useimmissa isoissa toteutuksissa on ”tulkissa” tavukoodikääntäjä ja lisäksi erillinen kääntäjä, joka usein kääntää Schemeä assemblerin näköiseksi C-koodiksi
- muutamia toteutuksia: Gambit-C (kurssilla), PLT Scheme (eli MzScheme ja DrScheme), MIT Scheme, Scheme48, Guile, Chicken, SCM, ...

Mihin Schemeä käytetään?

- laajennuskielenä tai skriptikielenä muulla kielellä kirjoitettuun ohjelmaan (esim. Gimp)
  - Guile ja SIOD yleisimpiä toteutuksia tässä käytössä
- pohjana omalle pienelle (tai isolle) ohjelmointikielelle, koska Schemessä on siisti ja helposti laajennettava semantiikka
  - esim. GNU R -tilasto-ohjelman ohjauskieli perustuu Schemeen
- ohjelmointikielitutkimuksessa uusien ideoiden kokeilussa (esim. PLT Scheme)
- opetuskäytössä
- ja yksittäisten Scheme-ohjelmien tekemiseen (mm. SSH on tehnyt tuotteita Schemellä)

Schemen tietotyypit

- perustietorakenne linkitetty lista `(1 2 3)`
  - tarkemmin pari `(1 . 2)`, joista voi rakentaa muutakin kuin varsinaisen listan
- vektori eli yksiulotteinen taulukko `#(1 2 3)`
- symboli `foo`
- totuusarvo `#t` tai `#f`
- numero (kokonaisluku `1`, rationaaliluku `1/3`, liukuluku `1.2`, kompleksiluku `1+3i`, epätarkka luku `#i1/3`; toteutuksessa ei tarvitse olla kaikkia)
- merkkijono `”foo”`
- merkki `#\a`
- proseduuri (primitiivi tai itsemääritelty)
- portti (esim. avatulle tiedostolle)

Koodi on dataa: sulkulausekkeet

- sulkulauseke (S-expression, symbolic expression): Lisp-syntaksin mukainen lauseke tai siitä tehty listarakenne
  - esim. `(+ (* 3 x) y)` tai `(if (= x y) 0 1)` voi tulkita joko Scheme-koodina tai linkitettynä listana, joka sisältää symboleita, numeroita ja alilistan
- `(read)` lukee käyttäjältä sulkulausekkeen ja tekee siitä listarakenteen
- `(write x)` tai `(display x)` tulostaa listarakenteen `x` sulkulausekkeena
  - `(newline)` tulostaa rivinvaihdon

Koodi on dataa: quote (SICP 2.3)

- Scheme-koodissa `(quote (+ (* x y) 3))` tai lyhyemmin `'(+ (* x y) 3)` tuottaa listarakenteen
  - `(if (= 2 1) 4 3)` on ehtolause (palauttaa kokonaisluvun `3`), `'(if (= 2 1) 4 3)` palauttaa linkitetyn listan, jonka ensimmäinen alkio on symboli `if`
  - koodissa `'x` palauttaa symbolin `x`, pelkkä `x` hakee muuttujan `x` arvon; esim. `(if (eq? x 'x) ...)`
- Schemessä on paljon listankäsittelyoperaatioita, joilla voi käsitellä mm. tällä tavalla tuotettuja listoja
  - joten Scheme-koodin näköisiä lausekkeita (sulkulausekkeita) on helppo käsitellä
- yksityiskohta: R^{5}RS ei välitä isoista tai pienistä kirjaimista (case-insensitive), mutta monissa toteutuksissa esim. `a` ja `A` ovat eri muuttujia (ja eri symboleita)

Yksityiskohta: yhtäsuuruudet

- Schemessä on monta eri yhtäsuuruusvertailua:
  - `eq?` on tarkin ja nopein vertailufunktio (”pointteri samaan paikkaan?”)
  - `eqv?` toimii myös mm. numeroille (käytetään harvoin)
  - `equal?` käy listan, vektorin tai merkkijonon läpi rekursiivisesti ja käyttää `eqv?`:ta yksittäisiin alkioihin
    - käytetään kun halutaan tietää, onko listoilla tai vektoreilla sama sisältö (vaikka ne olisivat eri listoja)
  - `=` vain numeroille, `string=?` vain merkkijonoille ja `char=?` vain merkeille
    - myös esim. `<`, `>=`, `string<?`, `string-ci>=?`, `char<?`
- yleensä käytetään: `eq?` symboleille ja listoille (onko sama lista?), `equal?` listoille (onko samanlainen lista?), `=` numeroille, `string=?` merkkijonoille

Sisältö

1. Kurssi
2. Scheme-kieli
3. SICP luku 1
4. SICP luku 2

Häntärekursio (SICP 1.2)

- lausekielissä iteratiivista laskentaa tehdään usein silmukoilla (`for`, `while`, jne.)
- funktionaalisessa ohjelmoinnissa tyypillisempää on käyttää (häntä)rekursiota
- normaalisti pino kasvaa joka kerta kun kutsutaan funktiota uudelleen
  - mutta tätä ei tarvitse tehdä, jos kutsun jälkeen ei tehdä mitään muuta (eli kutsu on häntäpaikassa)
  - Scheme-toteutuksen täytyy tehdä tämä häntärekursio-optimointi, monissa muissa kielissä se on vapaaehtoista (esim. gcc tekee sitä joskus C-ohjelmissa)

Esimerkki häntärekursiosta (SICP 1.2)

Kertoma häntärekursiolla

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Silmukalla Pythonilla:

```
def factorial(n):
    p = 1
    c = 1
    while c <= n:
        p = c * p
        c = c + 1
    return p
```

Ja Javalla/C:llä:

```
int factorial(int n) {
    int p = 1, c = 1;
    while (c <= n) {
        p = c * p;
        c++;
    }
    return p; }
```

Proseduurit argumentteina (SICP 1.3–1.3.2)

Halutaan arvioida π:n arvoa kaavalla $\frac{\pi}{8} = \frac{1}{1\cdot3} + \frac{1}{5\cdot7} + \frac{1}{9\cdot11} + \cdots$.

Ratkaisu summa-abstraktiolla (sum: $\sum_{n=a}^{b} term(n)$)

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a) (sum term (next a) next b))))
(define (pi-sum a b)
  (define (pi-term x) (/ 1.0 (* x (+ x 2))))
  (define (pi-next x) (+ x 4))
  (sum pi-term a pi-next b))
(define (pi-sum a b)      ; vaihtoehtoinen ratkaisu
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b))
```

Testiajo: `(* 8 (pi-sum 1 1000))` ⇒ 3.139592655589783

Proseduurit paluuarvoina (SICP 1.3.4)

Yksinkertaista numeerista derivointia

```
(define dx 0.00001)
(define (deriv g)      ; Dg(x) = (g(x+dx) - g(x))/dx
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))
```

Ajoesimerkkejä

```
(define (cube x) (* x x x))
(define dcube (deriv cube))      ; tai seuraava rivi:
(define (dcube x) ((deriv cube) x))
(dcube 5) ⇒ 75.00014999664018
((deriv cube) 5) ⇒ 75.00014999664018
```

Sisältö

1. Kurssi
2. Scheme-kieli
3. SICP luku 1
4. SICP luku 2

Linkitetyt listat (SICP 2.2.1)

- funktionaalinen ohjelmointitapa perustuu usein laskemiseen linkitettyjen listojen käsittelyfunktioilla
- Schemessä `(cons a b)` tekee parin, jonka avulla muodostetaan listoja
  - `(car p)` palauttaa parin ensimmäisen alkion
  - `(cdr p)` toisen
- linkitetty lista on jono pareja, joka päättyy tyhjään listaan `nil` (tai `'()`):
  - `(cons 1 (cons 2 (cons 3 nil)))` ⇒ `(1 2 3)`
  - myös: `(list 1 2 3)` ja `'(1 2 3)`
- jono joka ei pääty listaan näytetään näin: `(cons 1 (cons 2 3))` ⇒ `(1 2 . 3)`

Listaoperaatioita (SICP 2.2.1, 2.2.3)

- `(length l)` alkioiden määrä
- `(list-ref l i)` tietty alkio
- `(map f l)` tekee `(a b c)`:sta `(f(a) f(b) f(c))`:n
- `(filter p l)` palauttaa listan, jossa on `l`:stä vain alkiot `x`, joille `(p x)` on tosi
- useimmissa funktionaalisissa kielissä on vastaavat
  - abstraktio näiden (ym.) päälle: list comprehensions, mm. Haskellissa ja Pythonissa (ei Schemessä)

Esimerkkejä

```
(length (list 1 2 3)) ⇒ 3
(filter odd? (list 1 2 3 4 5)) ⇒ (1 3 5)
(map (lambda (x) (* x x)) '(1 5 10)) ⇒ (1 25 100)
```

Accumulate eli fold-right (SICP 2.2.3)

Accumulate-esimerkki

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
(define (pr-sq-odd s)
  (accumulate *
              1
              (map square (filter odd? s))))
```

Ajoesimerkkejä:

`(accumulate + 0 (list 1 2 3 4 5))` ⇒ 15

`(pr-sq-odd (list 1 2 3 4 5))` ⇒ 225

- ”yhdistää” listan alkiot halutulla tavalla
- yleisempi nimi tälle on `fold-right`

Symbolinen derivoija (SICP 2.3.2)

- Miten tekisit ohjelman, joka ottaa aritmeettisen lausekkeen (esim. $2x + 3$ eli `(+ (* 2 x) 3)`) ja muuttujan (`x`) ja palauttaa derivaatan (`2`)?
- Riittää osata summan ja tulon derivointisäännöt: $D(x + y) = Dx + Dy \quad Dxy = xy' + x'y$
- Eikä tarvitse sieventää
- Kuinka paljon työtä tällaisen ohjelman tekemisessä olisi?

Ratkaisu (kokonaan!)

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((symbol? exp) (if (eq? exp var) 1 0))
        ((and (pair? exp) (eq? (car exp) '+))
         (list '+ (deriv (cadr exp) var)
                  (deriv (caddr exp) var)))
        ((and (pair? exp) (eq? (car exp) '*))
         (list '+ (list '* (cadr exp)
                           (deriv (caddr exp) var))
                  (list '* (deriv (cadr exp) var)
                           (caddr exp))))
        (else
         (error "unknown expression type -- DERIV" exp))))
```

Ajoesimerkkejä:

```
(deriv 15 'x)       ⇒ 0
(deriv 'x 'x)       ⇒ 1
(deriv 'y 'x)       ⇒ 0
(deriv '(+ x 3) 'x) ⇒ (+ 1 0)
(deriv '(* x y) 'x) ⇒ (+ (* x 0) (* 1 y))
```


Abstrahoidumpi ratkaisu (kirjasta) (SICP 2.3.2)

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        ((sum? exp) (make-sum (deriv (addend exp) var)
                              (deriv (augend exp) var)))
        ((product? exp)
         (make-sum (make-product (multiplier exp)
                                 (deriv (multiplicand exp) var))
                   (make-product (deriv (multiplier exp) var)
                                 (multiplicand exp))))
        (else (error "unknown expression type -- DERIV" exp))))
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
(define (sum? x) (and (pair? x) (eq? (car x) '+)))
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (product? x) (and (pair? x) (eq? (car x) '*)))
(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))
```


Geneerinen aritmetiikka (SICP 2.4–2.5)

- isohko esimerkki, jossa
  - tehdään itse dynaaminen tyypitys ja tyyppijärjestelmä
  - tehdään oma toteutus metodin/funktion ylikuormittamiselle (overloading)
  - sivutaan olio-ohjelmointikäsitteitä kuten moniperintää
  - näiden avulla toteutetaan ”kirjasto”, jolla voi laskea mm. rationaaliluvuilla ja polynomeilla
- lukiessa kannattaa miettiä myös Javan tai C++:n ylikuormitusta (samannimisiä metodeja tai operaattoreita eri tyyppisillä argumenteilla)
- myös Common Lispin CLOS-oliojärjestelmä ja Haskellin tyyppiluokat ovat saman tapaisia kuin tämä esimerkkiä (tosin niissä on toki paljon muutakin)