

Scheme-kesäkurssi luento 2

Timo Lilja

1. 7. 2009

1 SICP luku 3

2 Makrot

3 Gambit

- olioilla on **paikallinen tila**, jota mallinnetaan **tilamuuttujilla**
- Scheme-kielessä on **sijoitusoperaattori** `set!`
  - muuttuja täytyy olla ensiksi määritelty **define**-lausekkeella
  - syntaksi: `((set!) <muuttuja> <lauseke>)`

set!-esimerkki

```
(define x 2)
(+ x 1)           ==> 3
(set! x 4)         ==> _unspecified_
(+ x 1)           ==> 5
```

Scheme-kielessä luodaan oliota yleensä yhdistämällä paikallinen tila ja sijoituslause

pankkitili

```
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
              balance)
      "Insufficient funds"))
```

Edellisessä esimerkissä on ongelmana se, että `balance`-muuttujaa ei ole **enkapsuloitu** vaan siihen pääsee globaalin ympäristön kautta käsiksi

pankkitili — paikallinen tila

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                  balance)
          "Insufficient funds"))))
```

Jos halutaan luoda useampia objekteja dynaamisesti, on tyypillistä tehdä **make**-proseduuri, joka palauttaa paikallisen tilan sisältävän proseduurin ja sitoa tämä paluuarvo omaan muuttujaan

pankkitili — objektien luonti dynaamisesti

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
                balance)
        "Insufficient funds"))))

(define W1 (make-withdraw 100))
```

Oliot voivat käsitellä myös useampia **viestejä**. Esimerkiksi seuraavasti voidaan luoda pankkitiliolio, johon voi laittaa ja josta voi nostaa rahaa

pankkitili — viestien välitys

```
(define (make-account balance)
  (define (withdraw amount)
    ...)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request--MAKE-ACCOUNT" m))))
  dispatch)
```

- + sijoituslauseke mahdollistaa modulaarisemman suunnittelun – puhtaasti funktionaalisessa koodissa ”tila” on kuljetettava mukana kaikessa laskennassa
- sijoituslauseiden järjestyksellä on yleensä väliä
- puhtaasti funktionaalinen proseduuri palauttaa aina samat arvot kun proseduuria kutsutaan samoilla argumenteilla, imperatiivinen koodi ei tätä takaa.
  - muuttujat eivät ole pelkästään nimiä **arvoille** vaan **viittauksia muistipaikkoihin**
  - samanlaisuus (equality) ei ole imperatiivisessa koodissa niin suoraviivaista kuin puhtaasti funktionaalisessa

## Sivuvaikutus ja listarakenteet

SICP 3.3.1

## Jono (1/4)

3.3.2

- Schemessä on primitiivit `set-car!` ja `set-cdr!` listarakenteiden muokkaamiseen
  - syntaksi: `(set-car! <pari> <lauseke>)`
  - syntaksi: `(set-cdr! <pari> <lauseke>)`
  - quotella tehtyjä listojen muuttamista ei ole määritelty

## set!-esimerkki

```
(define x (cons 1 2))
(set-car! x 3)           ==> _unspecified_
x                         ==> (3 . 2)
(set-car! (last-pair x) #f) ==> _unspecified_
(define y '(1 2))
(set-cdr! y 4)           ==> _error_
```

Huom! "error" on R5RS-terminologiaa ja tarkoittaa virheililannetta, josta ei tarvitse antaa virheilmoitusta

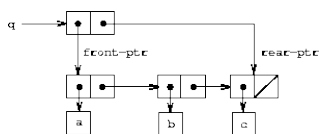
Toteutaan jono-tietorakenne Schemellä

- konstruktori
  - `(make-queue)` luo tyhjän jonon
- selektorit
  - `(empty-queue? <jono>)` testaa, onko jono tyhjä
  - `(front-queue <jono>)` palauttaa jonon ensimmäisen alkion tai virheilmoituksen jos jono on tyhjä
- mutaattorit
  - `(insert-queue! <jono> <alkio>)` lisää jonon alkuun alkion
  - `(delete-queue! <jono>)` poistaa jonon perästä alkion

## Jono (2/4)

3.3.2

Jono on osoittimien `front-ptr` ja `rear-ptr` muodostama pari.



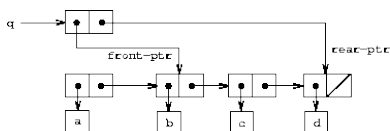
## perusoperaatiot

```
(define (front-ptr queue) (car queue))
(define (rear-ptr queue) (cdr queue))
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))
(define (empty-queue? queue) (null? (front-ptr queue)))
(define (make-queue) (cons '() '()))
```

## Jono (4/4)

3.3.2

Alkio poistetaan jonon edestä



## delete-operaatio

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
        (error "DELETE! called with an empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue))
         queue))))
```

## Virrat

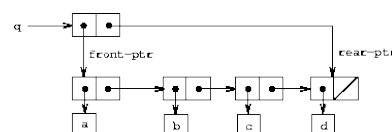
SICP 3.5

## Virrat viivästettyinä listoina

SICP 3.5.1

- sijoituksen avulla voidaan mallintaa ajan kanssa muuttuvaa tilaa, mutta sijoitus tuo mukanaan tiettyjä ongelmia
- voisiko ajan funktiona tapahtuvaa muutosta mallintaa muuten?
  - jos tarkastellaan funktiota  $x(t)$  hetkittäin, nähdään funktio muuttuvana suureena
  - jos tarkastellaan funktiota koko aikahistorian sisältävänä kokonaisuutena, muutosta ei tapahdu – funktio itsessään ei muutu

Alkion lisääminen jonoon tapahtuu jonon perään



## insert-operaatio

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
          (set-front-ptr! queue new-pair)
          (set-rear-ptr! queue new-pair)
          queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))
```

## Piirisimulaatio ja rajoitteet

SICP 3.3.4–3.3.5

- piirisimulaatio (SICP 3.3.4) ja rajoitteiden vyöryttämisyjärjestelmä (SICP 3.3.5) esimerkkejä monimutkaisemmista olioista jotka kommunikoivat **rajapintojen** läpi
- paljon paikallista tilaa ja simulaatio, jossa diskreetti aikainen askellus
- molemmat "lispmaisia" ongelmia
  - luodaan kohdealuekohtainen kieli (Domain Specific Language)
  - loppukäyttäjän ei tarvitse ymmärtää paljon lispiä/schemeä, jotta voi käyttää simulaattoreita

## Virrat

SICP 3.5

## Virrat viivästettyinä listoina

SICP 3.5.1

## iteratiivinen tyyli

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

## sekventiaalinen lista

```
(define (sum-primes a b)
  (accumulate +
    0
    (filter prime? (enumerate-interval a b))))
```

miten ohjelmat eroavat toisistaan muistinkulutukseltaan?

## Virtojen toteutus

SICP 3.5.1

- kursilla käytettävässä Schemessä on stream-tietotyyppille konstruktori `cons-stream` ja selektorit `stream-car` sekä `stream-cdr`
- nämä toteutetaan standardi-schemen primitiiveillä
  - `(delay <lauseke>)` on erikoismuoto joka ei evaluo argumenttiaan vaan palauttaa sen lupausena
  - `(force <lupaus>)` on proseduur, joka evaluo `delay`-primitiivillä viivästetyn lupauksen
- virtojen toteutus:
  - `(cons-stream <a> <b>)` on sama kuin `(cons <a> (delay <b>))`
  - Lisäksi voidaan määrittellä `(define (stream-car stream) (car stream))` ja `(define (stream-cdr stream) (force (cdr stream)))`

## Äärettömät virrat

SICP 3.5.2

Virtojen — toisin kuin listojen — ei tarvitse olla äärellisiä:

## Kokonaislukuluvirta

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))
(define integers (integers-starting-from 1))
```

Virtoja voi määrittellä myös implisiittisesti

## Implisiittinen kokonaislukuluvirta

```
(define ones (cons-stream 1 ones))
(define (add-streams s1 s2) (stream-map + s1 s2))
(define integers
  (cons-stream 1 (add-streams ones integers)))
```

## Funktionaalinen ohjelmointi

SICP 3.5.5

- I/O on imperatiivista, mutta sitä voidaan käsitellä äärettömänä syöte-virtana, jota prosessoidaan virta-operaatioilla ja tuloksena saadaan ääretön tulosvirta
  - puhtaasti funktionaalinen tapa hyötyineen/haittoineen (mitkä?)
  - Haskell-kielessä monadit tapa tehdä puhtaasti funktionaalista tilan enkapsulointia
- tulevaisuudessa moniydinprosessorit ja rinnakkaisohjelmointi entistä yleisempää
  - puhtaasti funktionaalisen ohjelman rinnakkaistaminen helpompaa (miksi?)
  - puhtaasti funktionaalisen ohjelman oikeellisuuden tarkastaminen yksinkertaisempaa
- funktionaalisten kielten huonoja puolia?

## Makrot

- funktiot abstrahoivat laskentaa, oliot dataa ja makrot ohjelmakoodin rakennetta
- makrojen avulla voidaan määrittellä uusia erikoismuotoja
  - funktioita joustavampi tapa määrittellä omia laajennuksia
  - yleensä tehokkaampia kuin funktiokutsut (inlining)
- keskeisimmät Scheme-kielessä käytetyt makrojärjestelmät: `define-macro` ja `syntax-rules`

## Virtojen käyttäminen

SICP 3.5.1

Lausekkeella haetaan toinen alkuluku väliltä [10000, 1000000], mutta lasketaan kaikki kyseisen välin luvut:

## Sekventiaalinen tyyli

```
(car (cdr (filter prime?
  (enumerate-interval 10000 1000000)))))
```

Lauseke laskee vain tarvittavat alkiot:

## Alkulukuluvirta

```
(stream-car
  (stream-cdr
    (stream-filter prime?
      (stream-enumerate-interval 10000
        1000000)))))
```

`stream-filter` ja `stream-enumerate-interval` toteutus hyvin analoginen vastaavien lista-proseduurien kanssa

## Iteratiivinen prosessi virroilla

SICP 3.5.3

Virtojen avulla tila voidaan esittää "ajattomana" virtana arvoja sen sijaan että prosessi päivittäväisi joukkoa muuttujia

## Alkuperäinen sqrt-proseduuri

SICP 1.1.7

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (sqrt-improve guess x) x)))
```

## sqrt-virta

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream 1.0
      (stream-map (lambda (guess)
        (sqrt-improve guess x))
        guesses)))
  guesses)
```

Mitä etuja ja haittoja virta-toteutuksella on?

- 1 SICP luku 3
- 2 Makrot
- 3 Gambit

## Sisältö

- `quasiquote` on erikoismuoto, jolla voidaan konstruoida lausekkeitä, joiden rakenne tunnetaan etukäteen vain osittain
  - `quasiquote`ja voi laittaa sisäkkäin (nested)
  - `'<qq-rakenne>` tai `(quasiquote <qq-rakenne>)`
  - `,<lauseke>` tai `(unquote <lauseke>)`
  - `,@<lauseke>` tai `(unquote-splicing <lauseke>)`

## esimerkki

```
'(a '(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
=> (a '(b ,(+ 1 2) ,(foo 4 d) e) f)

(quasiquote (list (unquote (+ 1 2)) 4))
=> (list 3 4)

'(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
=> (a 3 4 5 6 b)
```

## define-macro

- primitiivinen ja yksinkertainen rakenne
- syntaksi:  

```
(define-macro (<nimi> <parametri> <...>)  
  <runko>)
```
- määrittelyissä yleensä käytetään **quote**-erikoismuotoa
- mitä tapahtuu jos makron rungossa viitataan muuttujaan joka on sidottu makron kutsuympäristössä?

## esimerkki

```
(define-macro (unless test . body)  
  '(if ,test #f (begin ,@body)))  
  
(define-macro (push var #!optional val)  
  '(set! ,var (cons ,val ,var)))
```

## define-syntax ja syntax-rules

- R5RS-Schemessä syntax-rules-makrojärjestelmä,
  - **viiteläpinäkyvä** (referentially transparent): makron vapaat muuttujaviittaukset haetaan makron määrittely-ympäristöstä, ei makron kutsuympäristöstä
  - **hygieeninen**: jos makro määrittelee uusia muuttujia, nämä eivät peitä kutsuympäristössä leksikaalisesti ylempänä olevia sidoksia
- **define-syntaxin** lisäksi on olemassa myös **let-syntax** ja **letrec-syntax**:  

```
(define-syntax  
  (syntax-rules <literaalit>  
    (<hahmo> <template>)  
    (<hahmo> <template>)  
    <...>  
  ))
```

## Yksinkertaistettu cond

## cond-erikoismuodon toteutus

```
(define-syntax cond  
  (syntax-rules (else)  
    ((cond (else result1 result2 ...))  
      (begin result1 result2 ...))  
    ((cond (test)) test)  
    ((cond (test) clause1 clause2 ...)  
      (let ((temp test))  
        (if temp  
            temp  
            (cond clause1 clause2 ...))))  
    ((cond (test result1 result2 ...))  
      (if test (begin result1 result2 ...)))  
    ((cond (test result1 result2 ...)  
          clause1 clause2 ...)  
      (if test  
          (begin result1 result2 ...)  
          (cond clause1 clause2 ...)))))
```

## Lähteitä

- Gambit: define-macro ja define-syntax
  - <http://www.iro.umontreal.ca/~gambit/doc/>
  - 6.3 Miscellaneous extensions
- R5RS: define-syntax
  - <http://schemers.org/Documents/Standards/R5RS/>
  - 4.3 Macros
  - 5.3 Syntax definitions
  - 7.3 Derived expression types

## Sisältö

## Gambit C

- <http://www.iro.umontreal.ca/~gambit/doc/>
- sisältää tulkin **gsi** ja kääntäjän **gsc**
- R4RS, R5RS ja IEEE Scheme -standardit
- debuggeri
- paljon laajennuksia, mm. monta SRFI:tä
  - myös paljon dokumentoimattomia laajennuksia
- nimiavaruudet (ei dokumentoitu)
- unicode-merkistötuki
- record-rakenteet
- userland-säikeet
- poikkeukset
- C-rajapinta

## Puutteita

## Debuggeri

- ei GUI-kirjastoja
  - mutta helppo tehdä omat sidokset C-rajapinnan kautta
- dokumentaatiota puuttuu
- kääntäjälle annettavassa koodissa ei voi määritellä samaa muuttujaa useampaan kertaan **define**-avainsanalla
- apply-proseduurille voi antaa enintään 8192 argumenttia
- ks. Gambit-dokumentaatio: 20. System limitations

- breakpointin määrittely (**break** <proc>)
- kutsujäljen (trace) tulostus (**trace** <proc>)

| komento                                                 | selitys                                                                                                                                                                                                                |
|---------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ,? ,q ,t<br>,(c <expr>)                                 | help, quit, takaisin top-levelille<br>anna <b>expr</b> kontekstille joka odottaa nykyisen poikkeuksen aiheuttajan arvoa                                                                                                |
| ,s ,l<br>,(<n>),+ , -<br>,b ,be ,bed<br>,e , (e <expr>) | askellus (hyppien proseduurikutsujen yli)<br>liikuu kutsukehyksessä<br>näytä kutsupino (ja ympäristö)<br>näytä kehyksen muuttujat, evalui <expr><br>arvo nykyisessä kehyksessä<br>muuttujan arvo nykyisessä kehyksessä |