Scheme-kesäkurssi luento 3

Riku Saikkonen

6. 7. 2009

Sisältö

Miksi SICP-kirjassa on Scheme-tulkkeja?

- tulkin näkeminen auttaa ymmärtämään, miten ohjelmointikielet toimivat
  - ja miksi monissa kielissä on rajoituksia esim. ensimmäisen luokan funktioille tai sisäkkäisille määrittelyille
- useimmat tulkattavat ohjelmointikielet voi toteuttaa samalla tavalla kuin kirjassa
  - esim. Python-tulkin voisi tehdä kirjan tulkin pohjalta lisäämällä (paljon) yksityiskohtia ja jäsentimen
- uusien ohjelmointikieliominaisuuksien kokeilemiseksi (esim. SICP 4.2–4.4)
- isoissa ohjelmissa on usein pieni erikoistuneen kielen tulkki käsittelemässä esimerkiksi konfiguraatiotiedostoja

Nelilaskin: REPL

- tehdään aluksi laskinohjelma, joka osaa laskea esim. lausekkeen `(+ 1 (- 20 (* 4 3)))`
- tällainen ohjelma on hyvin yksinkertaisen kielen tulkki
- tulkkiproseduuri saa lausekkeen listana ja palauttaa arvon: `(calc-eval '(+ 1 (- 20 (* 4 3))))` $\Rightarrow$ `9`
- tälle voi tehdä tekstikäyttöliittymän, joka saa lausekkeen Schemen `read`-primitiiviltä

Käyttöliittymä tulkeille (read-eval-print loop, REPL)

```
(define (driver-loop)
  (display ";;; Eval input:") (newline)
  (let ((input (read)))
    (let ((output (calc-eval input)))
      (display ";;; Eval output:") (newline)
      (write output) (newline)))
  (driver-loop))
```

Muutama Scheme-primitiivi avuksi

- `(apply proc args)` kutsuu proseduuria `proc` antaen sille argumentteina kaikki listan `args` alkiot
  - `(apply (lambda (x y) (+ x y)) (list 1 2))` $\Rightarrow$ `3`
  - `(apply + (list 1 2 3 4))` $\Rightarrow$ `10`
- assosiaatiolista: lista, jonka alkiot ovat pareja, joiden `car` on "avain" ja `cdr` on "data"
  - esim. `((a 1) (b 2))` tai `((x 1 2) (y 3 4))` tai `(((1 2) 8) ((2 3) 15))`
  - Scheme-primitiivit `assq`, `assv` ja `assoc` etsivät annettua avainta tällaisesta listasta käyttäen avainvertailuun `eq?`:ta, `eqv?`:tä tai `equal`ia
  - esim. `(assq 'b '((a 1) (b 2)))` $\Rightarrow$ `(b 2)`
  - käytännössä: `assq` jos avain on symboli, `assoc` jos lista
  - tehottomia verrattuna esim. hajautustauluihin: käyvät listan läpi järjestyksessä

Nelilaskin: calc-eval

Ensimmäinen nelilaskin

```
(define (calc-eval exp)
  (cond ((number? exp) exp)
        ((pair? exp) (calc-apply (car exp) (cdr exp)))
        (else (error "Invalid input" exp))))
(define (calc-apply op-name args)
  (let ((arg-values (map calc-eval args)))
    (cond ((eq? op-name '+) (apply + arg-values))
          ((eq? op-name '-) (apply - arg-values))
          ((eq? op-name '*) (apply * arg-values))
          ((eq? op-name '/) (apply / arg-values))
          (else (error "Invalid operation" op-name)))))
```

Abstraktio: primitive-procedures

Abstraktimpi calc-apply

```
(define primitive-procedures
  (list (list '+ +)
        (list '- -)
        (list '* *)
        (list '/ /)))
(define (calc-apply op-name args)
  (let ((op (assq op-name primitive-procedures)))
    (if op
        (apply (cadr op) (map calc-eval args))
        (else (error "Invalid operation" op-name)))))
```

Sisältö

Globaalit muuttujat

Lisätään nelilaskimeen muuttujat, joiden arvoja voi käyttää laskuissa:

Ajoesimerkki

```
;;; Eval input:
(define a 4)
;;; Eval output:
#!void
;;; Eval input:
(define b (+ 5 3))
;;; Eval output:
#!void
;;; Eval input:
(+ a (* b b))
;;; Eval output:
68
```

```
;;; Eval input:
(set! b (+ 1 2))
;;; Eval output:
#!void
;;; Eval input:
(+ a (* b b))
;;; Eval output:
13
```

Globaalien muuttujien toteutus

Toteutus ja muokattu tulkki

```
(define env '())
(define (lookup-variable-value var) (cdr (assq var env)))
(define (set-variable-value! var val)
  (set-cdr! (assq var env) val))
(define (define-variable! var val)
  (set! env (cons (cons var val) env)))
(define (calc-eval exp)
  (cond ((number? exp) exp)
        ((symbol? exp) (lookup-variable-value exp))
        ((and (pair? exp) (eq? (car exp) 'define))
         (define-variable! (cadr exp)
                           (calc-eval (caddr exp))))
        ((and (pair? exp) (eq? (car exp) 'set!))
         (set-variable-value! (cadr exp)
                              (calc-eval (caddr exp))))
        ((pair? exp) (calc-apply (car exp) (cdr exp)))
        (else (error "Invalid input" exp))))
```

Primitiivifunktioista globaaleja muuttujia

- `calc-apply` haki operaattoreiden nimiä `primitive-procedures`-listasta
- entä jos `+`, `-`, `*` ja `/` olisivatkin vain muuttujia, joiden arvo on operaattorin toteutuuttava Scheme-proseduuri?

Tämän toteutus (muutokset tulkkiin)

```
(define primitive-procedures
  (list (list '+ +) (list '- -) (list '* *) (list '/ /)))
(define (setup-environment!)
  (set! env '())
  (for-each (lambda (x)
              (define-variable! (car x) (cadr x)))
            primitive-procedures))
(define (calc-apply op-name args)
  (let ((op (calc-eval op-name))
        (arg-values (map calc-eval args)))
    (apply op arg-values)))
```

Sisältö

Paikalliset muuttujat: ympäristöt (SICP 3.2)

Esimerkkikoodi

```
(let ((x 1) (y 2))
  (let ((a (+ x y)) (b (+ y y)))
    (* y a)))
```

- lauseketta `(* y a)` evaluoidessa sen muuttujat haetaan ympäristöstä, jossa on (ainakin) kolme kehystä:
  sisin kehys: $\texttt{a} = 3$, $\texttt{b} = 4$
  toinen kehys: $\texttt{x} = 1$, $\texttt{y} = 2$
  globaali ympäristö: $* =$ kertolaskuprimitiivi, ...
- tämä ympäristön käsite sopii useimpiin ohjelmointikieliin

Ympäristöt kirjan tulkeissa (SICP 4.1.3)

- kirjassa määritellään proseduurit ympäristöjen käsittelyyn:
  - `(lookup-variable-value var env)` hakee muuttujan `var` arvon ympäristöstä `env`
  - `(set-variable-value! var val env)` muuttaa muuttujan `var` arvon arvoksi `val`
  - `(define-variable! var val env)` lisää uuden muuttujan ympäristön ensimmäiseen (sisimpään) kehykseen
  - `(extend-environment vars vals base-env)` laajentaa (ei muuta) `base-env`iä uudella kehyksellä
- nämä vain käsittelevät ympäristö-tietorakennetta:

```
(((a b) . (3 4))      ; sisin kehys
 ((x y) . (1 2))      ; toinen kehys
 ((* ...) . (<primitiivi> ...)))      ; globaali ympäristö
```

Proseduurit ja ympäristöt (SICP 3.2)

Koodiesimerkki

```
(define (make-counter val)
  (lambda (add)
    (set! val (+ val add))
    val))
```

```
(define f (make-counter 1))
(define g (make-counter 2))
(f 3) ⇒ 4
(g 4) ⇒ 6
```

- mitä muuttujan `f` arvossa pitää olla tallessa?
  - proseduurin koodi ja ympäristö, jossa se on määritelty
  - `f`:ssä on ympäristö, jossa on kaksi kehystä:
    sisin kehys F_1: `val` $= 1$
    globaali ympäristö F_0: `+`, `make-counter` jne.
  - `g`:ssä on viittaus samaan F_0:aan, mutta eri sisin kehys:
    sisin kehys F_2: `val` $= 2$
    globaali ympäristö F_0: `+`, `make-counter` jne.
- proseduuriolio (ns. closure) siis tarvitsee omaa muistia
  - tästä syystä esim. C-kieli ei tue tällaisia funktioita

Tapa määritellä paikallisia muuttujia

- halutaan nelilaskimeen paikalliset muuttujat
- proseduurien argumentit ovat paikallisia muuttujia
- toteutetaan siis paikallisten muuttujien ohessa proseduurit (jotta nelilaskin lähestyisi SICP-kirjan tulkkia)
- tavallisemmat muuttujat voisi tehdä tämän jälkeen syntaksimuunnoksella: `(let ((x 1)) (+ x 2))` $\Rightarrow$ `((lambda (x) (+ x 2)) 1)`

Proseduurien toteutus nelilaskimeen 1/2

Uusi calc-eval

```
(define (calc-eval exp env)
  (cond ((number? exp) exp)
        ((symbol? exp) (lookup-variable-value exp env))
        ((and (pair? exp) (eq? (car exp) 'define))
         (define-variable! (cadr exp)
                           (calc-eval (caddr exp) env)
                           env))
        ((and (pair? exp) (eq? (car exp) 'set!))
         (set-variable-value! (cadr exp)
                              (calc-eval (caddr exp) env)
                              env))
        ((and (pair? exp) (eq? (car exp) 'lambda))
         (make-procedure (cadr exp) (caddr exp) env))
        ((pair? exp)
         (calc-apply (calc-eval (car exp) env)
                     (map (lambda (e) (calc-eval e env))
                          (cdr exp))))
        (else (error "Invalid input" exp))))
```

Proseduurien toteutus nelilaskimeen 2/2 (SICP 4.1.3)

Proseduurioliot ja uusi calc-apply

```
(define (make-procedure arg-names body env)
  (list 'procedure arg-names body env))
(define (compound-procedure? v)
  (and (pair? v) (eq? (car v) 'procedure)))
(define (primitive? v)
  (and (pair? v) (eq? (car v) 'primitive)))
(define (calc-apply proc arg-vals)
  (cond ((compound-procedure? proc)
         (calc-eval (caddr proc)
                    (extend-environment (cadr proc)
                                        arg-vals
                                        (cadddr proc))))
        ((primitive? proc)
         (apply (cadr proc) arg-vals))
        (else (error "Not a procedure -- APPLY" proc))))
```

Sisältö

Nelilaskintulkki kirjan tyylillä 1/2 (SICP 4.1.2)

Abstrakti syntaksi (muutoksia tulkkiin)

```
(define (tagged-list? exp tag)
  (if (pair? exp) (eq? (car exp) tag) false))

(define (self-evaluating? exp) (number? exp))
(define (variable? exp) (symbol? exp))
(define (definition? exp) (tagged-list? exp 'define))
(define (assignment? exp) (tagged-list? exp 'set!))

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (caddr exp))    ; kirjassa cddr

(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

Nelilaskintulkki kirjan tyylillä 2/2 (SICP 4.1.1)

Abstraktimpi calc-eval

```
(define (eval-definition exp env)
  (define-variable! ...))
(define (eval-assignment exp env)
  (set-variable-value! ...))
(define (list-of-values exps env)     ; kirjassa kirjoitettu auki
  (map (lambda (e) (calc-eval e env)) exps))
(define (calc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((application? exp)
         (calc-apply (calc-eval (operator exp) env)
                     (list-of-values (operands exp) env)))
        (else (error "Invalid input" exp))))
```

Scheme-tulkki (SICP 4.1)

- nelilaskintulkista saa nyt tehtyä kokonaisen Scheme-tulkin vain lisäämällä siihen tukea uusille lauseille
- kirjan kohdan 4.1 tulkissa on lisätty erikoismuodot `quote`, `if`, `begin` ja `cond`
- ja `define`n toinen syntaksi `(define (f x) ...)`
- ja `primitive-procedures`-listaan on lisätty mm. `cons`, `car` ja `cdr`
- kirjan tulkki on metasirkulaarinen eli Schemellä tehty Scheme-tulkki
  - joten esimerkiksi primitiivit voi "lainata" alla olevasta Scheme-toteutuksesta

Esimerkki: if-lause (SICP 4.1.1)

if-lauseen toteutus kirjan tulkkiin (muutokset)

```
(define (true? x) (not (eq? x false)))
(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
(define (eval exp env)
  (cond ...
        ((if? exp) (eval-if exp env)) ...)
```

Esimerkki: cond-lause (SICP 4.1.2)

- `cond` toteutetaan kirjassa niin, että `cond`-lause muutetaan `if`-lauseeksi syntaksimuunnoksella
- kirjassa määritelty proseduuri `cond->if` (s. 373) tekee alla olevan muunnoksen
- ja sen kutsuminen on lisätty tulkin `eval`iin
- jos tulkki tukisi makroja, `cond`in voisi tehdä myös niillä:

```
(define-macro (cond . clauses)
  (cond->if (cons 'cond clauses)))
```

Esimerkki cond->if-muunnoksesta

```
(cond->if '(cond ((> x 0) x)
                 ((= x 0) (display 'zero) 0)
                 (else (- x)))) ⇒
(if (> x 0) x
    (if (= x 0) (begin (display 'zero) 0)
        (- x)))
```

Analysoiva Scheme-tulkki (SICP 4.1.7)

- optimointi koko tulkkiin: tehdään syntaktista analyysiä heti kun lauseke annetaan tulkille eikä vasta evaluoinnin aikana (esim. jokaisessa rekursiivisessa kutsussa)
- analyysimahdollisuuksia: syntaksimuunnokset, `eval`-proseduurin oikean haaran valinta ym. (kaikkea mitä voidaan tehdä ilman ajonaikaista ympäristöä)

Analysoivan tulkin runko

```
(define (eval exp env)
  ((analyze exp) env))
(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp)) ...))
(define (analyze-self-evaluating exp)
  (lambda (env) exp))
```

Sisältö

Mikä on kontinuaatio?

- kontinuaatio eli jatko = laskennan tila
  - kutsupino, muuttujien arvot
  - paikka koodissa eli lauseke, jota parhaillaan evaluoidaan
  - myös esim. jo evaluoidut alilausekkeet
- esimerkki: lausekkeen
  `(let ((a 1)) (- (+ a a) (* a 2)))`
  eräs kontinuaatio on kohdassa, jossa alilausekkeet `-` ja `(+ a a)` on jo evaluoitu ja lauseketta `(* a 2)` ollaan juuri evaluoimassa

Eksplisiittiset kontinuaatiot Schemessä

- Schemessä laskennan nykyisen kontinuaation voi ottaa talteen ja siihen voi myöhemmin hypätä takaisin, jopa useita kertoja
  - sama lauseke voi siis palata monta kertaa esim. eri paluuarvoilla
- kontinuaation talletusmuoto on proseduuri, jota kutsumalla kontinuaatioon hypätään
  - kutsu ei palaa sinne, mistä kutsu tehtiin
- kutsussa annetaan argumentti, josta tulee kontinuaation luoneen lausekkeen arvo (sitä oltiin evaluoimassa kun kontinuaatio luotiin)

Mihin kontinuaatioita käytetään?

- eksplisiittisillä kontinuaatioilla voi (ainakin periaatteessa) toteuttaa monia kontrollirakenteita:
  - `break` silmukasta ja `return` funktiosta
  - poikkeukset
  - korutiinit
  - generaattorit ja epädeterministinen laskenta (useita paluuarvoja, joita kokeillaan yksi kerrallaan)
- voi tallettaa laskennan tilan ja jatkaa siitä myöhemmin
  - tätä käytetään joissain WWW-sovelluskehyksissä
  - kun sovellus jää odottamaan käyttäjältä vastausta esim. lomakkeeseen, sen tila talletetaan kontinuaatioon
  - jos/kun vastaus tulee, kontinuaatiota kutsutaan käyttäjältä saadulla syötteellä
  - sovelluksen ohjelmoijalle tämä näyttää tavalliselta odottavalta (blocking, synchronous) I/O:lta

call-with-current-continuation

- Scheme-primitiivi `call-with-current-continuation` (lyh. `call/cc`) kutsuu argumenttinaan saamaansa proseduuria antaen kontinuaation sille argumentiksi
- käyttö yleensä: `(call/cc (lambda (k) ...))`
- tämän `call/cc`-lausekkeen arvo on normaalisti `...`-lausekkeen arvo
- jos `k`:ta kutsuu, koko `call/cc`-lausekkeen arvoksi tulee `k`:lle annettu argumentti, ja evaluointi etenee niin kuin `call/cc`-lauseke olisi juuri palannut
- `k`:ta voi kutsua joko lausekkeen sisältä (vrt. `break` tai poikkeukset muissa kielissä) tai myöhemmin (vrt. C:n `longjmp`)

Esimerkkejä

Kontinuaatioesimerkkejä

```
(call/cc (lambda (k) 1)) ⇒ 1
(call/cc (lambda (k) k)) ⇒ #<procedure>

(define (copy-list-if-positive l)
  (call/cc (lambda (k)
             (define (loop l)
               (cond ((null? l) '())
                     ((<= (car l) 0) (k '()))
                     (else (cons (car l)
                                 (loop (cdr l))))))
             (loop l))))
(copy-list-if-positive '(1 2 3 4 5)) ⇒ (1 2 3 4 5)
(copy-list-if-positive '(1 2 -3 4 5)) ⇒ ()
```

Sisältö

Continuation-passing style

- ohjelmointityyli, jolla kontinuaatiot saa käyttöön, vaikka ohjelmointikieli ei tukisi eksplisiittisiä kontinuaatioita
- tarvitsee kuitenkin ensimmäisen luokan funktiot
- kirjan luvun 4.3 tulkki on kirjoitettu (noin) CPS-tyyliin
- myös käännöstekniikka: jotkut funktionaalisten kielten kääntäjät muuttavat koodin CPS-muotoon
- CPS:ssä kaikilla proseduureilla on ylimääräinen kontinuaatioargumentti (yleensä `k`)
  - proseduurit palaavat aina kutsumalla `k`:ta paluuarvollaan (eivät siis koskaan palauta arvoa normaalisti)
  - tällainen kutsu on häntäpaikassa
  - `k` vastaa `call/cc`:n tuottamaa kontinuaatiota
- sovelluksesta riippuu, kuinka pitkälle CPS-muoto viedään

CPS-esimerkki

Tavallinen kertoma

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

Eräs CPS-kertoma

```
(define (=k a b k) (k (= a b)))      ; ''primitiivi'', ei CPS
(define (-k a b k) (k (- a b)))      ; ''primitiivi'', ei CPS
(define (*k a b k) (k (* a b)))      ; ''primitiivi'', ei CPS
(define (fact n k)
  (=k n 0 (lambda (pred)
   (if pred
       (k 1)
       (-k n 1 (lambda (arg)
        (fact arg (lambda (res)
         (*k n res k)))))))))
(fact 10 display) ⇒ 10
```