SkeletonLisp
käyttöohjeet

Roni Kekkonen

Sisältö

1. Johdanto (ja vähän taustaa)

SkeletonLisp on LISP-kieliperheen typistetty ohjelmointikieli, sekä myös kyseisen kielen tulkki. Tulkki on rakennettu Java-kielellä Helsingin Yliopiston Tietojenkäsittelytieteen laitoksen kurssin Ohjelmoinnin harjoitustyö projektina.

Kaikkien LISP-kielten tapaan, SkeletonLispissä funktioilla on *first class citizen* -status: niitä voidaan antaa parametreina muille funktioille ja niitä voidaan jopa sijoittaa tietorakneteisiin, esimerkiksi parin tai listan sisälle.

Schemen tapaan SkeletonLispissä on vain yksi nimiavaruus. Siispä kaikki arvot kokonaisluvuista funktioihin ovat käytettävissä ja viitattavissa samalla tavalla.

Typistettynä kielenä SkeletonLisp tarjoaa vain oleellisimmat primitiivifunktiot, joiden avulla voidaan johtaa uusia funktioita tarvittaessa. Yhtenä filosofiana SkeletonLispissä onkin käyttäjän vapaus luoda omia määritelmiä ja uudelleenmääritellä lähes mitä tahansa: muutamaa avainasia (lähinnä cond- ja lambda-lauseet) lukuunottamatta kieli on käyttäjän vapaasti määriteltävissä.

Idea tulkin luomiseen lähti halusta selvittää kuinka haastavaa, ja kuinka työlästä LISP-tulkki on toteuttaa Javan kaltaisella imperatiivisella ohjelmointikielellä minimaalisilla ennakkotiedoilla siitä, minkälaisilla menetelmillä jo olemassaolevat toimivat Javalla koodatut Scheme/LISP-tulkit on luotu. Se on myös ensimmäinen LISP-tulkkini, jonka olen toteuttanut jollain muulla ohjelmointikielellä kuin Schemellä; kyseessä on siis varsin kokeilullinen leluprojekti.

Toisena perusfilosofiana SkeletonLispissä onkin ollut pyrkimys toteuttaa tulkki mahdollisimman Scheme-kielistä varianttia mukailevaksi sillä ajatuksella, että tällöin ongelman ymmärtäminen ja mallintaminen Javalle luonnistuisi helpommin kun voi tukeutua johonkin tuttuun ja turvalliseen.

Kaikille LISP-kielille ominainen roskienhallinta on jätetty Javan vastuulle, sillä Javassa on jo ennestään sisäänrakennettuna roskienhallintajärjestelmä.

Näin jälkeenpäin ajateltuna, olisi projektin varmasti voinut toteuttaa miljoonalla eri tavalla ja ainakin miljoonalla paremmallakin tavalla, ja olisikin mielenkiintoista suunnitella tulkki nyt uudestaan ja miettiä mitä asioita tekisin toisin.

Tämä käyttöohje ovat tarkoitettu nopeasti selattavaksi referenssiksi SkeletonLispin erityisominaisuuksista. Erinomainen Johdatus LISP-perheen kieliin, löytynee kirjoista "The Little Schemer" sekä "Structure and Interpretation of Computer Programs", joka on tunnettu myös englanninkielisellä termillä "wizard book"), tai vanhemmista erityisille LISP-koneille suunnitelluista manuaaleista.

2. Yleiskatsaus tulkin rakenteeseen

Itse tulkki koostuu kolmesta isommasta kokonaisuudesta: parserista, evaluaattorista, sekä REP-silmukasta ("ReadEvalPrintLoop"). Rep-silmukka pyörittää tulkkia ja hallinnoi sitä. Nimensä mukaan se koostuu kolmesta olennaisesta komponentista:

Read: lukee käyttäjän syötteen
Eval: evaluoi syötteen
Print: tulostaa syötteen evaluoidun arvon käyttäjälle.

Jotta Eval voisi evaluoida syötteen, on se ensin parseroitava SkeletonLisp-kieliseksi lauseeksi (L-lauseeksi). Tämän hoitaa tulkin parseri: se muodostaa käyttäjän syötteestä AINA jonkun L-lauseen, myös virhetapauksessa: tällöin tulkki muodostaa syötteestä virhelauseen (tämä mekaniikka on rakennettu Javalla käyttämällä Javan poikkeuksia virhetilanteissa joita Read-komponentti sitten tulkitsee virheiksi (LError) poikkeusten viestejä käyttämällä).

Huomionarvoista on myös se, että LError -lauseet ovat varattu tulkkia varten, eikä käyttäjä siis pääse niitä suoraan käyttämään.

Jotta parseri tietäisi, mitä sen tulee parseroida, täytyy REP-silmukan Read-komponentin osata tulkita sitä, milloin käyttäjä on kirjoittanut kokonaisen hyväksyttävän merkkijonon ja sen jälkeen tarjota tämä merkkijono parserille. Parseri palauttaa Read:lle merkkijonosta L-lauseen, jonka Read taas välittää eteenpäin Eval:lle.

Seuraavaksi Eval delegoi lauseen evaluoimisen Evaluaattorille, joka arvioi, mikä lauseen arvo on. Lauseen kirjoitushetkellä. Evaluaattori palauttaa arvioidun arvon takaisin Eval-komponentille, joka antaa sen edelleen Print-komponentille tulostettavaksi.

Tämän jälkeen siirrytään taas Read-vaiheeseen, jossa käyttäjän syötteen arvioiminenaloitetaan alusta – syotteenlukuvaiheesta – tulkin nykyisellä, mahdollisesti edellisen lauseen evaluoimisen johdosta muuttuneella tilalla.

Tulkin tila voikin muuttua aina kun käyttäjä määrittelee globaalisti DEFINE-primitiivillä muuttujille (Lid) arvoja, tai kun käyttäjä antaa komennon "(EXIT)", eli kutsuu EXIT-primitiiviä, jolloin REP-silmukka saa tiedon siitä, että käyttäjä haluaa lopettaa ohjelman suorituksen, ja osaa näin lopettaa ohjelman suorittamisen.

Parseri ja evaluaattori ovat taas omia kokonaisuuksiaan, joista huomionarvoista on se, että evaluaattori sisältää kaikki käyttäjän DEFINE-primitiiveillä määrittelemät globaalit Lid:t evaluaattorin hallinnoimassa GlobalEnvironment-tietueessa, sekä se, että parseroimisen jälkeen käyttäjän syöttämä merkkijono on muutettu puhtaaksi L-lauseeksi. Siispä parseria voisi pitää omana erillisenä komponenttinaan, ja sen voisi itse asiassa korvata koska vain täysin uudella parserilla, olettaen että tämä uusi parseri palauttaa REP-silmukan Read-komponentille samalla tavalla joko L-lauseita kuin nykyinenkin parseri, tai sitten poikkeuksia virhetilanteissa.

3. Syntaksi

Käymme ensin läpi nopeasti SkeletonLisp-kielen syntaksin, ja sen jälkeen pureudumme lauseiden semantiikkaan. Tätä syntaksiosiota voi sitten tarvittaessa käyttää referenssinä jatkossa. Käyttäjän syötteet, kyseisten lauseiden luomiseksi, on kursivoitu, ja niitä edeltää tulkin oma symboli ">>", joka kertoo sen, että tulkki on valmis ottamaan vastaan käyttäjältä uuden lauseen. Tätä seuraa Tulkin tuloste.

Vaikka kaikki syötteet on kirjoitettu tässä isoilla kirjaimilla, on sallittua kirjoittaa syötteet myös pienillä kirjaimilla, tai vaikkapa isoja ja pieniä kirjaimia yhdistämällä. Tulkki tosin aina muuntaa kaikki käyttäjän syötteet isoille kirjaimille perintesten Common Lisp-tulkkien tapaan.

SL-tulkki ymmärtää kaikkia syntaksin mukaisia käyttäjän syöttämiä L-lauseita. L-lauseita ovat:

NIL

NIL vastaa SkeletonLispissä sekä totuusarvoa epätosi, että listan päättävää alkiota (NIL ei kuitenkaan itse ole pari, eikä täten myöskään lista). Kaikki muut arvot ovat aina tosia.

```
>> NIL
NIL
```

kokonaisluvut

```
>> 4
4
>> -8
-8
>> 93
93
```

Huomionarvoista on, että SkeletonLisp ei tunne liukulukuja.

Atomit

atomit ovat merkeistä koostuvat yksittäiset sanat, jotka alkavat '-merkillä, jota seuraava merkki on jokin muu kuin numero. Lisäksi atomissa ei saa esiintyä mitään seuraavista merkeistä:
\ ' ([{) }] sekä ":

```
>> 'AUTO
AUTO
>> 'A
A
>> '$AY
$AY
>> 'HOP-1
HOP-1
```

ovat kaikki hyväksyttäviä atomeja.

Kuten huomataan, SkeletonLisp tulostaa käyttäjän syöttämät atomit kuitenkin ilman heittomerkkiä (').

muuttujat (ID)

Muuttujiin pätee täysin samat säännöt kuin atomeihin, paitsi, että ne eivät ala '-merkillä. Siispä '-merkkiä käytetään erottamaan atomit ja Idt toisistaan SkeletonLispissä:

```
>> F
<F : (ID)>
>> AUTO
<AUTO : (ID)>
>> MY-FUN
<MY-FUN : (ID)>
>> PI
<PI : (ID)>
```

ovat kaikki muuttujia. Kuitenkin on huomattava, että mikäli ID:tä ei ole määritelty mihinkään arvoon DEFINE-primitiivillä, tulostuu todellisuudessa seuraavaa:

```
>> MAX-ARVO
<ERROR>: ID UNBOUND: <MAX-ARVO : (ID)>
```

COND-lauseet

COND-lauseet ovat aina muotoa

```
(COND
 (CASE-1 RESULT-1)
 ...
 (CASE-N RESULT-N))
```

ja niiden arvo on ensimmäinen RESULT, jonka CASE-parin arvo on jotain muuta kuin NIL

CASE:t ja RESULT:t voivat olla mitä vain semanttisesti korrekteja L-lauseita.

On myös tärkeää huomioida, että mikäli kaikkien CASE-lauseiden arvo on lauseita evaluoitaessa NIL, on tämä virhe, ja SL-tulkki myös ilmoittaa siitä virheenä. Siksi on erittäin suositeltavaa, että COND-lauseille annetaan jokin DEFAULT-case, joka aina on tosi.

Common LISPistä tuttu '#T-arvo on mainio vaihtoehto tälle. Schemen ELSE-avainsanaa SkeletonLisp ei tunne, joten se täytyy määritellä eksplisiittisesti, mikäli sitä halutaan käyttää:

```
>> (DEFINE ELSE '#T)
'#T
>> (COND
      ((ZERO? 5) '#T)
      (ELSE NIL))
NIL
```

Cond-lauseet siis vaativat lauseen aloittavan "("-merkin ja päättävän ")"-merkin. Myös jokaista CondCase-paria (CASE-I RESULT-I) ympäröi sulkeet.

APPLIKAATIOT (APPLICATIONS)

Applikaatiot ovat aina muotoa

(PROC PARAMVAL-1 … PARAMVAL-N)

Jossa PROC voi olla joko LAMBDA-lause, COND-lause, ID, tai sitten toinen applikaatio (jonka täytyy tällöin palauttaa arvonaan joko hyväksyttävä LAMDA-, COND-, tai ID-lause). PARAMVAL-I voi olla mikä tahansa L-lause.

Applikaatiot siis vaativat aloittavan "("-merkin ja päättävän ")"-merkin.

Huomionarvoista on se, että funktiot ja proseduurit voivat saada arvoikseen myös toisia funktioita ja proseduureja, ja myöskin huomionarvoista on LISP-kielessä toisinaan käytetty tekniikka, jossa parametrin arvo määräytyy jonkin ehdon mukaan COND-lauseella.

Esimerkkejä Applikaatioista:

```
>> (DEFINE X 12)
12
>> (> X 7)
'#T
```
(olettaen, että funktio ">" on määritelty; ">" ei ole primitiivinen SkeletonLispissä, johtuen kielen minimalistisesta luonteesta. Funktio > voidaan toki määritellä SL-tulkilla).

```
>> (ADD1 9)
10
```

```
>> (LIST 'A 'B (CONS 'C (CONS 'D NIL)) (CONS 'E 'F) 'G)
(A B (C D) (E . F) G)
```

```
>> (PAIR? (LIST 'a 'b (CONS 'c 'd) 'e))
'#T
```

LAMBDA-lauseet

Lambda-lauseet ovat aina muotoa:
(LAMBDA (ARG-1 ARG-2 … ARG-N) RUNKO)
Näillä lauseilla luodaan uusia funktioita ja proseduureja.

LAMBDA-lauseiden syntaksi on seuraava:

Kuten COND-lauseet ja applikaatiotkin, LAMBDA-lause on eroteltu "(" ja ")"-merkkien väliin. Myös funktion argumenttien lista (ARG-1 ARG-2 … ARG-N) on eroteltu sulkeiden väliin.

Argumenttien ARG-1 … ARG-N tulee olla ID:itä; ne eivät voi olla mitään muuta L-lausetyyppiä.
RUNKO taas voi olla mikä tahansa syntaktisesti korrekti L-lause, vaikka jopa toinen LAMBDA-lause: funktiot siis voivat palauttaa paluuarvoinaan lambda-lauseita.

Esimerkiksi seuraavat lambda-lauseet ovat kaikki syntaktisesti järkeviä:

```
>> ((LAMBDA (X) X) 'IDENTITY)
IDENTITY.


>> (DEFINE >
      (LAMBDA (X Y)
       (NOT (NEGATIVE? (- X Y)))))
<PROCEDURE>


>> (DEFINE MAX
      (LAMBDA (X Y)
       (COND ((> X Y) X)
             ('#T Y))))
<PROCEDURE>


>> (DEFINE A-PLUS-ABS-B
      (LAMBDA (A B)
       ((COND
         ((NEGATIVE? B) -)
          ('#T +))
        A B)))
<PROCEDURE>
```

Huomioitavaa on, että kaikki edelliset LAMBDA-lauseet voidaan evaluoida, vaikka funktioita "-", "<" ja "+" ei olekaan määritelty, mutta tästä lisää seuraavassa osassa.

3. Semantiikka