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)>
```

COND-lauseet

COND-lauseet ovat aina muotoa

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

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

```
>> (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 aaltosulkeet.

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)
1Ø
```

```
>> (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.

4. Semantiikka

Kun kun evaluaattori saa tulkattavaksi jonkin korrektin L-lauseen, se evaluoidaan tiettyjä semanttisia sääntöjä noudattaen.

Arvolauseet

Jos syöte on jokin arvolause: numero, atomi, NIL, pari/lista, LError, LString (joista Error ja String ovat varattu tulkin omaan käyttöön). Palautetaan lause sellaisenaan; arvolauseen arvo on siis arvolause itse.

Esimerkiksi:
>> 3
3
>> NIL
NIL

Huomioitavaa on se, että NIL-lauseen arvo vastaa aina totuusarvoa epätosi. NIL myös toimii listan päättävänä elementtinä (Listoista ja pareista lisää jäljempänä primitiivien CONS ja LIST yhteydessä).

ID

Jos syöte on ID, niin kyseisen ID:n arvo etsitään ensin tulkin globaalista ympäristöstä. Globaali ympäristö pitää sisällään kaikki käyttäjän "DEFINE"-primitiivillä määritellyt ID - ARVO -parit. Tällöin voidaan sanoa, että D on sidottu arvoon ARVO. ARVO voi taas olla mikä tahansa korrekti Lexp-lause. Siispä jopa COND-lauseita voidaan sitoa ID:een.

Jos nyt ID on sidottu globaalissa ympäristössä ARVOon, on evaluaation tulos ARVO. Muussa tapauksessa selvitetään, vastaako ID jotain SkeletonLisp-primitiiviä. Jos vastaa, ID:n arvo on ID itse.

Jos taas ID:tä ei ole sidottu globaalissa ympäristössä, eikä se ole mikään SkeletonLisp-primitiivi, palautetaan virhe:

>> UNBOUND-ID
<ERROR>: ID UNBOUND : <UNBOUND-ID (ID)>
>>

>> ADD1
<ADD1 (ID)>

COND

COND-lauseiden arvo on ensimmäinen RESULT, jonka CASE-parin arvo on jotain muuta kuin NIL

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) 'NOLLA)
      ((NUMBER? 5) (LIST 'LUKU))
      (ELSE NIL))
(LUKU)
```

Edellisen lauseen ensimmäinen CASE-RESULT -pari on:
((ZERO? 5) '#T).
Lauseen CASE-osa on (ZERO? 5) ja lauseen RESULT-osa '#T.
Mikäli olisi totta, että 5 = Ø, olisi COND-lauseen tulos ensimmäinen RESULT, eli '#T.

Koska luku 5 ei kuitenkaan ole ekvivalentti luvun Ø kanssa, siirrytään COND-lauseen seuraavaan CASE-RESULT -pariin, eli:
((NUMBER? 5) (LIST LUKU))
Sen CASE-osa, eli (NUMBER? 5) pitää paikkansa, sillä 5 on todellakin luku. Siispä COND-lauseen arvo on tämän CASE-RESULT -parin RESULT-osa, joka on (LIST LUKU) – tämä evaluoidaan siten ja tulokseksi saadaan seuraava lista:
(LUKU)

Loput lauseen CASE-RESULT -parit ohitetaan täten kokonaan, sillä on jo löytynyt jokin CASE-osa, jonka arvo on jotain muuta kuin NIL, eli epätosi (tässä tapauksessa (ELSE NIL) jätetään huomiotta).

APPLIKAATIO (APPLICATION)

Applikaatiot, eli funktion (tai proseduurin) kutsut, evaluoidaan siten että ensin evaluoidaan vasemmalta oikealle jokainen funktion parametreinaan saaduista argumenteista. Erikseen, ja sen jälkeen nämä argumentit "syötetään" kyseiselle funktiolle parametreina. Tämän jälkeen applikaation funktio-osio (applikaatiolistan ensimmäinen alkio) evaluoidaan, ja lopuksi funkti-osion evaluoinnista saatu arvo evaluoidaan annetuilla parametreilla. Siispä esimerkiksi seuraavassa applikaatiossa:

```
>> (LIST 12 (COND
            ((ZERO? 5) 'NOLLA)
            ((NUMBER? 5) (LIST 'LUKU))
            (ELSE NIL))
         4)
```

ensin evaluoidaan argumentti 12, joka on arvolause. Siispä sen arvo on luku 12. Näin ollen primitiivifunktio List tulee saamaan ensimmäisenä parametrinaan lukuarvon 12.

Lauseen 12 evaluoimisen jälkeen evaluoidaan lause

```
(COND
      ((ZERO? 5) 'NOLLA)
      ((NUMBER? 5) (LIST 'LUKU))
      (ELSE NIL))
```

Edellisessä osiossa lauseen evaluoinnista saatiin arvoksi "(LUKU)", siispä LIST tulee toiseksi parametrikseen saamaan listan "(LUKU)".

Tämän jälkeen täytyy vielä evaluoida arvolause 4, jonka LIST saa kolmantena parametrinaan.

Näin lause

```
>> (LIST 12 (COND
            ((ZERO? 5) 'NOLLA)
            ((NUMBER? 5) (LIST 'LUKU))
            (ELSE NIL))
         4)
```

Voidaan muuntaa **kuviteltuun** muotoon (LIST 12 (LUKU) 4), jossa 12 (LUKU) ja 4 ovat siis jo evaluoituja arvoja. Nyt vielä LIST täytyy evaluoida: koska LIST on primitiivi, sen evaluoinnin tulos on LIST. Nyt voidaan siis LIST evaluoida annetuilla parametreilla 12, (LUKU) sekä 4. Lauseen arvo on siis (12 (LUKU) 4). (LIST-primitiivin tarkempi selostus löytyy käyttöohjeiden luvusta Primitiivit).

Applikaatiot vaativat aina, että funktio-osio evaluoituu lopulta joko käyttäjän määrittelemäksi funktioksi (LAMBDA-lause), tai sitten primitiiviksi. Siispä lause

```
>> ((COND
      ((ZERO? Ø) ADD1)
      ('#T 8))
 5)
6
```

On täysin korrekti SkeletonLisp-lause, sillä applikaation funktio-osion COND-lause evaluoidaan arvoksi ADD1, joka on siis primitiivi. Mutta kuten huomataan, lause:

```
>> ((COND
      ((ZERO? 1) ADD1)
      ('#T 8))
 5)
<ERROR>: NOT A PROPER  PROCEDURE: 8
```

Palauttaa virheen, sillä applikaation funktio-osion COND-lause evaluoidaankin nyt arvoksi 8, sillä koska 1 ≠ Ø, ei ensimmäinen CASE-RESULT-osio päde. Toinen osio pätee, sillä sen CASE on '#T, joka on tosi. Koska arvolause 8 ei ole käyttäjän määrittelemä funktio eikä primitiivi, ei se voi toimia applikaation funktiona.

LAMBDA

Lambda-lauseilla määritellään uusia proseduureja, ja niitä usein sidotaan "DEFINE"-primitiivillä johonkin ID:een. Kuitenkin on täysin sallittua käyttää lambda-lausetta suoraan, ilman mitään ID-sidontaa. Tällöin puhutaan anonyymeista funktioista.

Esimerkiksi applikaatiolauseessa:

```
>> ((LAMBDA(x) (ADD1 x)) 5)
6
```

käytetään anonyymia funktiota luomaan uusi luku, joka on yhtä suurempi kuin funktion (LAMBDA (x) (ADD1 x)) parametrikseen saamansa argumentti 5, eli 6.

Lambda-lauseiden muuttujaosio kertoo sen kuinka monta argumenttia lause voi, ja lauseen täytyy ottaa argumentteina, silloin kun sitä käytetään applikaation funktiona. Edellinen lause sisältää siis yhden muuttujan, joten se ottaa yhden argumentin tällöin parametrinaan.

Mikäli LAMBDA-lauseelle annetaan liian vähän, tai vastaavasti liian monta argumenttia, ilmoittaa SkeletonLisp virheen tapahtuneen:

```
>> ((LAMBDA (X) (ADD1 X)) 5 9)
<ERROR>: EVALUATING A LAMBDA EXPRESSION: WRONG AMOUNT OF
ARGUMENTS GIVEN
```

eikä applikaatiota voida evaluoida.

LAMBDA-lauseen muuttujiin pätee myös seuraava kriteeri: LAMBDA-lauseessa ei voi olla useampaa kuin yksi samanniminen muuttuja. Siispä seuraava lause tuottaa syntaksivirheen:

```
>> (lambda (x y x) (list x y x))
<ERROR>: ILLEGAL LAMBDA VARIABLE DECLARATION - X
```

Muita rajoituksia muuttujien nimeämiseen ei ole. On siis täysin hyväksyttyä antaa LAMBDA-lauseen muuttujalle nimeksi esimerkiksi COND:

```
>> (LAMBDA (COND) (NOT COND))
```

Tällöin, minkä argumentin tahansa lause saakaan, lambda bodyn sisällä kaikki viittaukset ID:een COND viittaavat nyt tähän kyseiseen argumenttiin. Siispä esimerkiksi seuraava lause olisi virheellinen:

```
>> ((LAMBDA (COND)
      (COND ((ZERO? COND) Ø)
            ('#T 1)))
    8)
```

Sillä se muuttuisi kuvitteelliseksi lauseeksi:

```
(8 ((ZERO? 8) Ø)
   ('#T 1)))
```

Joka on sekä semanttisesti täysin mieletön lause.

Lambda-applikaatiot siis evaluoidaan, kuten edellisestä virhe-esimerkistä huomataan niin, että kaikki LAMBDA-lauseen rungon sisällä olevat **vapaat** viittaukse lambda-lauseen muuttujiin korvataan vastaavilla (vasemmalta-oikealle järjestyksiin nähden) applikaation argumenttien arvoilla:

```
>> ((LAMBDA (X Y) (+ X Y)) 5 6)
11
```

Eli, lauseen rungon: (+X Y) X korvataan luvulla 5 ja Y korvataan luvulla 6. Tämän jälkeen lauseen uusi runko (+ 5 6) evaluoidaan.

On huomioitava, että seuraavassa lauseessa:

```
>> ((LAMBDA (X)
    ((LAMBDA (X Y)
       (LIST X Y))
     (ADD1 X) X))
   5)
(6 5)
```

Sisemmän lambda-lauseen rungon muuttujaa X ei suinkaan voi korvata luvulla 5; (LIST 5 Y) olisi virhe. Sisin X on siis sidottu sisemmän lambda-lauseen muuttujaan, eikä se näin ollen ole sisemmän lauseen vapaa rungossa (LIST X Y).

5. Primitiivit

6. Huomautuksia

SkeletonLisp ei ole täydellinen Lisp-kieli, vaan pikemminkin leluprojekti, sillä siitä puuttuu muutama LISP-kielille oleellinen ominaisuus.

Ensimmäinen kriittinen puute on se, että SkeletonLispissä ei ole todellista quote-primitiiviä, jolla LISP-kielissä voidaan estää syötetyn lauseen evaluointi. Wuote mahdollistaa hyvinkin monipuolisten laskennallisten, sekä tietorakenteellisten työkalujen luomisen ja käyttämisen, jotka on mahdotonta toteuttaa ilman quote-primitiiviä. Vaikka SkeletonLisp tunnistaakin merkin ' käyttämisen atomien yhteydessä, ei tämä merkki vastaa toiminnallisuudeltaan quote-primitiiviä juuri lainkaan. Tätä merkkiä käytetään vain todellisista LISP-kielistä tutulla tavalla erottamaan atomeja muuttujista.

Toinen merkittävä puute on makrot. LISP-kielten eräs ehdoton vahvuus on LISP-makrot, joiden avulla käyttäjä pystyy itse luomaan suhteellisen helposti esimerkiksi tavanomaisesta evaluoinnista poikkeavia evaluointimenetelmiä - vaikkapa AND-primitiivin.

Kolmas puute on Scheme-kielestä tuttu Call-with-current-continuation, joka on erittäin tehokas työkalu esimerkiksi rekursiivisten funktioiden optimoimiseen. Javan Exceptionit ovat käytännössä poikkeustapaus call-with-current-continuationista. Mutta Scheme-kielen call/cc on paljon monipuolisempi mekanismi, kuin Javan Exceptionit, ja se tarjoaa kokeneelle ohjelmoijalle mahdollisuuden lopettaa jonkin ohjelman (osan) suoritus ennenaikaisesti. Tämän ohjelmoiminen olisi kuitenkin ollut sen verran haastava ja vaativa prosessi, että se olisi pitkittänyt ja vaikeuttanut projektia kohtuuttoman paljon. Siispä SkeletonLisp ei tue call-with-current-continuationeja.