Common Lisp Object System

Seminaarityö

Tomi Vihtari

Ohjelmointikielten periaatteet kevät 2004
Helsingin Yliopisto
Tietojenkäsittelytieteen laitos
Järvenpää 5. huhtikuuta 2004

Sisältö

1 Johdanto 1
2 Luokat ja perintä 1
3 Geneeriset funktiot 3
4 Metaluokat 5
5 Luokkien ja ilmentymien uudelleenmäärittely 5
6 Yhteenveto 6
Viitteet 8

1 Johdanto

Lisp-kieli kehitettiin 1950- ja 1960-lukujen vaihteessa aluksi listojen prosessointiin, mutta siitä kasvoi yleiskäyttöinen dynaamisesti tyypitetty funktionaalinen ohjelmointikieli [YuH86]. Common Lisp on eräs Lisp-kielen versio, jonka joukko tutkijoita kehitti halutessaan luoda Lispille standardisoidun kielen määrittelyn. 1980-luvulla oliokielet alkoivat vähitellen tehdä läpimurtoa ja myös Lispille alettiin kehitellä omaa olio laajennusta. Common Lispille luotiin 1980-luvun lopussa oliolaajennus CLOS (Common Lisp Object System) [Bob88]. CLOS perustuu geneerisiin funktioihin, moniperintään, deklaratiiviseen metodien yhdistelyyn (declarative method combination) ja meta-olio protokollaan (meta-object protocol). CLOSin perusrakenteita ovat luokat, ilmentymät, geneeriset funktiot ja metodit. Ohjelmointikielenä CLOS perii kaikki Common Lispin ominaisuudet. CLOS on dynaamisesti tyypitetty funktionaalinen kieli joka sisältää roskien keruun ja tehokkaat välineet luokkien, operaatioiden ja jopa kielen itsensä uudelleenmäärittelyyn.

2 Luokat ja perintä

Luokka määrittelee luokan ilmentymien rakenteen ja käyttäytymisen [Bob88]. Jokainen Common Lispin olio on jonkin luokan ilmentymä. Olion luokka määrittelee ne operaatiot, jotka voidaan oliolle suorittaa. Luokka määritellään CLOSissa defclass-makron avulla. Määrittely sisältää luokan nimen, listan luokan välittömistä yliluokista, luokan jäsenien määrittelyn (slot specifiers) ja luokan optiot (class options). CLOSissa luokan muuttujia kutsutaan englanninkielisellä termillä *slot*, joista käytän tässä tekstissä nimitystä *jäsen*. Jäsenistä jokainen voi sisältää yhden arvon. Luokalla voi olla nolla tai useampi jäsentä. Jäsenet voivat olla joko lokaaleja tai jaettuja. Lokaalit jäsenet näkyvät

vain yhdelle luokan ilmentymälle ja jaetut jäsenet ovat yhteisiä kaikkien luokan ilmentymien kesken.

Luokkia voidaan määritellä defclass-makron avulla seuraavanlaisesti:

```
> (defclass C1 ()
      ((S1 :initform 5.4 :type number)
       (S2 :allocation :class)))

> (defclass C2 (C1)
      ((S1 :initform 5 :type integer)
       (S2 :allocation :instance)
       (S3 :accessor C2-S3)))
```

Luokan C1 ilmentymillä on lokaali jäsen nimeltä S1, jonka oletusarvo on 5.4 ja jonka arvo pitää olla aina numeeristä tyyppiä. Luokalla C1 on myös jaettu jäsen S2. Luokka C2 on luokan C1 aliluokka. Luokan C2 ilmentymillä on lokaali jäsen S1, jonka oletusarvo on 5 ja jonka arvo on integer-tyyppiä. C2:n ilmentymillä on myös lokaalit jäsen S2 ja S3. C2:lla on metodi C2-S3, jolla luetaan jäsenen S3 arvo, kirjoittamista varten on metodi (setf C2-S3).

Luokan ilmentymä luodaan make-instance –makron avulla:

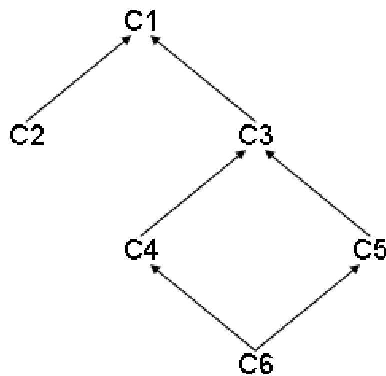
```
> (setf my_c2 (make-instance ’C2))
```

Tällä komennolla luokasta C2 luodaan ilmentymä my_c2, jonka jäsen S1 saa arvokseen 5, mutta jäseniä S2 ja S3 ei ole sidottu mihinkään arvoon.

CLOSin luokat voivat siis periä rakenteet ja käyttäytymisen toisesta luokasta. CLOS mahdollistaa moniperinnän, eli yksittäinen luokka voi periä ominaisuuksia monelta luokalta. Jokaisella luokalla on perintälista (class precedence list), joka sisältää luokan kaikki yliluokat läheisimmästä kaukaisimpaan [Bob88]. Läheisimmät luokat voivat

peittää tai ohittaa kaukaisempien luokkien ominaisuuksia. Perintälistan avulla järjestelmä päättää, mitkä ominaisuudet jäävät aliluokassa voimaan [LAM90].

Kuvassa 1 on esitetty eräs luokkahierarkia, jossa C1...C6 ovat luokkia ja aliluokista on vedetty nuolet yliluokkiin. Luokan C6 perintälista on (C6 C4 C5 C3 C1). Luokka C6 on määritelty seuraavasti: (defclass C6 (C4 C5) ()). Koska C4 on mainittu yliluokkien listassa ennen C5:ttä, on se perintälistassa myös ennen C5:ttä.



Kuva 1: Esimerkki luokkahierarkiasta

3 Geneeriset funktiot

Geneeristen funktiot ovat funktioita, joiden toiminta riippuu niille annettujen parametrien tyypistä [Bob88]. Geneerisen funktion määrittävä objekti sisältää joukon metodeja (sekä myös lambda-listan, metodiyhditelmätyypin (method combination type) sekä muuta tietoa). Metodit määrittävät luokkakohtaisen käyttäytymisen ja geneerisen funktion toiminnan. Kun geneeristä funktiota kutsutaan, se ajaa sille määritellyistä metodeista sen osajoukon, joka määräytyy funktiolle annettujen parametrien luokkien mukaan.

Geneeristä funktiota voidaan käyttää samalla tavoin, kuin tavallistakin Common Lispin funktiota. Se voidaan välittää parametrinä ja sille voidaan antaa globaali tai paikallinen nimi [GWB91].

Metodi-objektit sisältävät metodin toiminnan ja sarjan parametrien spesialisoijia (parameter specializers), jotka määrittävät ne tilanteet, joissa metodia voidaan käyttää [Bob88]. Metodi-objektit sisältävät myös sarjan määreitä (qualifiers), joilla metodiyhdistelmiä (method combination) tehtäessä voidaan eri metodit erottaa toisistaan. Jokaisella metodin vaatimalla formaalilla parametrilla on parametrispesialisoija (parameter spesializer) ja metodia kutsutaan ainoastaan, jos parametrit vastaavat parametrispesialisoijien vaatimuksia.

Metodien yhdistelijä (method combination facility) valitsee ajettavat metodit, metodien suoritusjärjestyksen ja geneerisen funktion palauttamat arvot. CLOS sisältää oletusarvoisen metodiyhditelmätyypin ja sallii uusien metodiyhdistelmätyyppien määrittelemisen.

Geneerinen funktio, joka laskee luvun kertoman voidaan luoda esimerkiksi näin:

```
> (defgeneric fact (n)
      (:method ((n integer)) (if (< n 2) 1 (* n (fact (1- n)))))
      (:method ((n string)) (concatenate 'string n "!"))
      (:documentation "Computes the factorial of a number"))
```

Kun funktiota kutsutaan numeroarvolla 4 se palauttaa 24:

```
> (fact 4)
24
```

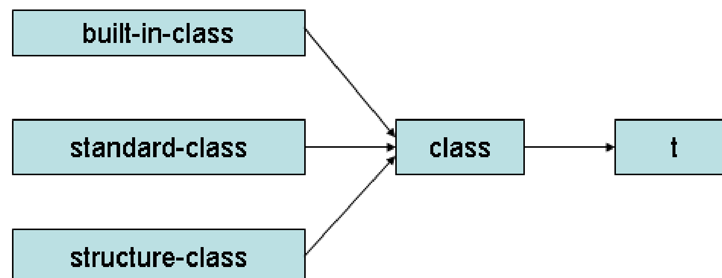
Kun funktiota kutsutaan merkkijonoarvolla “4” se palauttaa “4!”:

```
> (fact "4")
"4!"
```

4 Metaluokat

CLOSissa oliot määritellään luokkien avulla [LAM90]. Myös jokainen luokka on itsessään olio, sen metaluokan ilmentymä. Esimerkiksi kaikki defclass-makron avulla luodut luokat ovat standard-object luokan ilmentymiä. CLOSin toteutusta on mahdollista laajentaa tai muuttaa määrittelemällä uusia metaluokkia tai kirjoittamalla metodeja, jotka ohittavat metaluokkien oletusarvoisen käyttäytymisen.

CLOSissa on joukko ennaltamääriteltyjä metaluokkia, joita kutsutaan standardeiksi metaluokiksi. Kuvassa 2 on osajoukko standardien metaluokkien hierarkiasta. Kaikki luokat ja samalla myös metaluokat perivät luokan t. Kaikki metaluokat ovat luokan class aliluokkia.



Kuva 2: Standardeja metaluokkia

5 Luokkien ja ilmentymien uudelleenmäärittely

Sellainen luokka, joka on standard-class -luokan ilmentymä, voidaan määritellä uudelleen, jos uusi luokka on myös standard-class -luokan ilmentymä [Bob88]. Luokan uudelleenmäärittely muuttaa jo olemassaolevaa luokka-oliota ja ei luo uutta luokka-oliota luokalle.

Kun luokka määritellään uudelleen, muutokset leviävät luokan ilmentymiin ja kaikkien luokan aliluokkien ilmentymiin. Ilmentymien päivitysajankohta on toteutuksesta riippuvainen, mutta päivitys tehdään kuitenkin ennen kuin ilmentymän jäseniä luetaan tai kirjoitetaan. Ilmentymän identiteetti ei myöskään muutu, kun se päivitetään. Luokan uudelleenmäärittely saattaa aiheuttaa sen, että luokkaan lisätään tai luokasta poistetaan jäseniä. Päivitysoperaatio muuttaa tällöin ilmentymän jäseniä, mutta se ei luo uutta ilmentymää.

CLOSissa on mahdollista muuttaa ilmentymän luokkaa geneerisen change-class -funktion avulla ohjelman kehityksen tai suorituksen aikana. Tämä operaatio toimii samoin, kuin luokan uudelleenmäärittelykin, mutta erona on siis se, että muutos tehdään vain yhteen ilmentymään. Jos vanhalla ja uudella luokalla on saman nimisiä ja tyyppisiä jäseniä, niiden arvo säilyy. Kaikki muut vanhan luokan jäsenet poistetaan ja uusille jäsenille tehdään normaalit alustusoperaatiot.

6 Yhteenveto

CLOSia voidaan kutsua funktionaaliseksi oliokieleksi, mutta Common Lispin tavoin CLOS on kaukana puhtaasti funktionaalisista kielistä, koska se sallii muun muassa funktioiden sivuvaikutukset. Myöskin olio-ohjelmointi on tyypillisesti luonteeltaan imperatiivista, koska siinä tavallisesti luodaan oliota, joiden tila muuttuu ohjelman suorituksen aikana. CLOSissa on oliokielenä myös joitain puutteita. Tiedon näkyvyyttä ei voi säädellä ja operaatiot määritellään luokan ulkopuolella. Dynaaminen tyypitys aiheuttaa sen, että tyyppivirheet huomataan vasta ohjelman suorituksen aikana, jos silloinkaan.

Puutteistaan huolimatta CLOS tarjoaa jotain, mitä monet muut olio-kielet eivät tarjoa. CLOS on erittäin dynaaminen kaiken suhteen ja kesken ohjelman suorituksen voi tehdä mitä vain. Luokan määritelmää voidaan muuttaa, jolloin muutos heijastuu kaikkiin

luokasta tehtyihin ilmentymiin. Myös ilmentymän luokkaa voidaan muuttaa, jolloin CLOS tekee parhaansa sovittaakseen ilmentymän uuteen luokkaan. CLOSilla on siis jopa mahdollista määritellä koko kieli uudelleen kesken ohjelman suorituksen.

Viitteet

[YuH86] Yuasa, T., Hagiya, M., *Introduction to Common Lisp*, Academic Press, Inc., 1986

[Bob88] Bobrow, D. et al., *Common Lisp Object System specification*, ACM SIGPLAN Notices, Volume 23 , Issue SI (September 1988), 1 - 142

[LAM90] Lawless, J., Miller, M., *Understanding CLOS The Common Lisp Object System*, Digital Press, 1991

[GWB91] Gabriel, R., White, J., Bobrow, D., *CLOS: integrating object-oriented and functional programming*, Communications of the ACM, Volume 34 Issue 9, September 1991, 29-38