# Az Allegro Common Lisp 4.2 és használata

Készítette: Román Gyula MMT

dr. Tilly Károly MMT



## 1. Általános tudnivalók

Az Allegro Common Lisp 4.2 a szabványos Common Lisp egy teljes implementációja, kiegészítve néhény kiterjesztéssel. Ezek közül a legfontosabbak a multiprocessing és a top-level.

A programrendszer tartalmaz egy teljesen implementált objektum-orientált programozási környezetet, a CLOS-t, és egy, a debuggolást, statisztikákat megkönnyítõ grafikus felületet, a Composert.

(A grafikus csomag önállóan is programozható, gyakorlatilag a teljes XLib könyvtár hozzáférhetõ. Erre a felületre a Franz Inc. ráépített egy magasabb szintû protokollt, amit egy lisp csomagként - Common Windows - valósított meg, amelynek a felülete jól dokumentált.

A tanszéki gépeken a lispbõl négy image található meg, egy lecsupaszított, ami csak magát a lispet tartalmazza, egy olyan, amely a Common Windowst is tartalmazza, és egy legbõvebb, amely tartalmazza a fent leírt Composert.

Az Allegro CL magasabbszintû szolgáltatásai akkor érhetõk el, ha nem shellbõ, hanem emacs alól indítjuk. Ehhez a home katalogusban lévõ .emacs filenak a következõket kell tartalmaznia:

(setq load-path (cons "/usr/local/acl4.2/lib/emacs-fi" load-path))

(load "fi-site-ini")

(setq fi:common-lisp-directory "*~-lisp_directory*")

(setq fi:common-lisp-image-name "*image_name*")

(fi:common-lisp)

Az indítás úgy történik, hogy egy shell ablakban ki kell adni az

xhost kanga vagy az xhost graphy

xterm &

telnet kanga vagy telnet graphy

emacs -d *display-név* & (ahol a display-név a konzol display neve, setenv | grep DISPLAY paranccsal a display lekérdezhetõ.)

parancsokat, amely elindítja az emacs-et, és azon keresztül a lispet.

A fenti funkciókat sok más parancsszekvenciával is el lehet érni, pl. xrsh használata.

Az emacs editor a Unix világ egyik legelterjedtebb editorja, lispben nagyon jól alkalmazásra szabható. Az Allegro CL-hez is készült egy interface, amely lehetõvé teszi a kapcsolattartást a futó lisp process és az editor között.

Az alábbiakban táblázatosan leírjuk a legfontosabb emacs és emacs-lisp-interface parancsokat.

# 2. Emacs, az Allegro CL által használt editor

Az emacs parancsok általában valamilyen billentyûkombinációval aktivizálhatók. A leggyakoribb parancsok egy segédbillentyûvel érhetõk el, ami vagy a Ctrl, vagy a Meta karakter. (Ctrl-a - a kontrol és a karakterek együttes lenyomása, M-a - a meta karakter és az a billentyû együttes lenyomása.). Ha a terminálon nem található Meta karakter, akkor az az Escape billentyûvel kiváltható. Bevezetõ gyakorlatként célszerû végignézni az emacs oktató programját. (Üsd be a Ctrl-h t billentyû sorozatot.)

Legfontosabb parancsok:

(A parancsok egy bõvebb listája a parancs kártyán található.)

---

| | |
|---|---|
| `Ctrl-f` | `karakter elõre` |
| `Ctrl-b` | `karakter hátra` |
| `M-f` | `szó elõre` |
| `M-b` | `szó hátra` |
| `Ctrl-n` | `következõ sor` |
| `Ctrl-p` | `elõzõ sor` |

| | |
|---|---|
| Ctrl-a | sor eleje |
| Ctrl-e | sor vége |
| M-< | buffer eleje |
| M-> | buffer vége |
| Ctrl-v | scrollozás elõre |
| M-v | scrollozás hátra |
| Ctrl-x < | scrollozás balra |
| Ctrl-x > | scrollozás jobbra |
| Ctrl-d | karakter törlés elõre |
| DEL | karakter törlés hátra |
| M-d | szó törlés elõre |
| M-DEL | szó törlés hátra |
| Ctrl-k | sor törlés elõre |
| Ctrl-x 1 | összes ablak törlése, kivéve az aktívat |
| Ctrl-x 2 | ablak vízszintes kettéosztása |
| Ctrl-h a | apropók keresése |
| Ctrl-h f | függvény leírása |
| Ctrl-s | inkrementális keresés elõre |
| Ctrl-r | inkrementális keresés hátra |
| Ctrl-x Ctrl-f | file betöltése |
| Ctrl-x Ctrl-s | file elmentése |

---

## A legfontosabb lisp-emacs-interface parancsok:

---

| | |
|---|---|
| TAB | lisp formázás |
| ; | komment |
| Ctrl-c ] | szuperzárójel |
| Ctrl-^ | defun közepére állás |
| Ctrl-c Ctrl-e | defun végére állás |
| Ctrl-M-q | függvény formázása lisp stílusra |
| Ctrl-c l | lisp listener <> buffer csere |

| | |
|---|---|
| Ctrl-c Ctrl-c | megszakítás (SIGIO) |
| Ctrl-c Ctrl-q | megszakítás |
| Ctrl-c ? | apropók |
| Ctrl-c SPC | pop-up ablak megszüntetése |
| M-TAB | lehetséges kiegészítés |
| M-Sh-m | makró kifejtés |
| M-Sh-w | rekurzív makró kifejtés |
| M-Sh-f | függvény leírása |
| M-Sh-d | szimbólum leírása |
| M-Sh-a | argumentumok leírása |

---

Az emacs-lisp-interfacerõl bõvebb információ a program leírásában található.

# 3. Common Lisp bevezetõ

## 3.1. Lisp történet

A lisp egyike az elsõ magas szintû programozási nyelveknek. McCarthy az MIT Mesterséges Intelligencia (MI) laboratóriumában adta az ötletet egy lista feldolgozó nyelv kifejlesztésére, és az 50-es évek végén meg is indult a fejlesztés, amely a Lisp 1.5-öt eredményezte. A Lisp nevét a List Processing, vagyis lista feldolgozásról kapta. A fejlesztést nagyban motiválta, hogy szimbolikus integrálást, deriválást szerettek volna végezni, és ehhez próbáltak alkalmas nyelvet fejleszteni. A nyelv elsõ implementációi azt mutatták, hogy ez a megközelítés nagyon jól alakalmas a Mesterséges Intelligenciakutatás területén felvetõdõ problémák megoldására, és ez a felismerés újabb lendületet adott a fejlesztésnek. Sorra jelentek meg a különféle implementációk, amelyek alapjában azonosak voltak, de eléggé eltértek ahhoz, hogy a programok már ne legyenek portábilisek. Ez egyre inkább elõtérbe helyezte a standardizálás szükségességét, amelynek eredményeként megszületett az X3J13-as szabvány, ami a Common Lisp-et írja le. A szabványosított nyelv teljes leírása megtalálható []-ben.

## 3.2. Miért is olyan hasznos a Lisp megismerése?

A technika, tudomány fejlõdésével egyre nagyobb teret hódítanak az MI alkalmazások. Az elterjedt rendszerek két fõ nyelvet alkalmaznak: a Prolog-ot és a Lisp-et. A lisp nagy teret hódított a tudományos kutatóintézetekben, egyetemeken, számos szakértõi rendszert fejlesztettek ki benne. Fontos tulajdonsága a nyelvnek, hogy lehetõség van "tiszta" programozásra, hisz a lisp támogatja a funkcionális programozást. Másrészt egy interaktív programozási környezetet ad, amiben nagyon hatékonyan és gyorsan lehet rendszereket kifejleszteni, belõni. Egy másik fontos tulajdonsága a lisp rendszereknek, hogy támogatják az inkrementális programfejlesztést, ami nagyon lényeges nagy és összetett programrendszerek kifejlesztésénél. És talán a leglényegesebb újítás, amit a lisp vezetett be, hogy a program és adatszerkezet nem különül el, így programrészek feldolgozhatóak, átalakíthatóak, majd újra futtathatóak.

## 3.3. A lisp alapvetõ adatszerkezetei

Ebben a fejezetben megismerkedünk a legalapvetõbb adatszerkezetekkel. Az adatszerkezetek belsõ

ábrázolásáról itt nem beszélünk, mert az implementáció függõ.

A Common Lisp által nyújtott legalapvetõbb adattípusok:

Számok:
- egészek,
- közönséges törtek,
- lebegõpontos számok,
- komplex számok.

Karakterek

Szimbólumok

Listák

Tömbök

Hash táblák

Olvasási táblák

Csomagok

Path nevek

Streamek

Struktúrák

Függvények

Osztályok: Használatukról a 4. fejezet ad képet.

Módszerek: Használatukról a 4. fejezet ad képet.

Generikus függvények: Használatukról a 4. fejezet ad képet.

Az alábbiakban definiálunk néhány kifejezést, amelyeknek megértése elengedhetetlen a továbbiakban.

**Atom:** Nem cons.

**Szimbólum:** Lisp adatszerkezet. Betûvel vagy aláhúzással kezdõdik, és betûvel, számmal illetve aláhúzás karakterrel folytatódik. (Nem térünk ki az escape szekvenciákra.). A szimbólumokat a lisp kiértékeli és az értéküket visszaadja. Amennyiben nincs értéke egy kiértékelt szimbólumnak, akkor hibaüzenetet kapunk. Léteznek speciális szimbólumok, amelyeknek önmaguk (nyomtatási képük) az értékük. Ilyenek pl. nil (ami írható () formában is), t, és a kettõsponttal kezdõdõ kulcsszavak.

**Lista:** Nyitó zárójellel kezdõdõ és záró zárójellel végzõdõ helyes lisp kifejezés.

**S-kifejezés:** Tetszõleges lisp adatszerkezet.

**Forma:** Olyan S-kifejezés, ami kiértékrlésre kerül.

## 3.4. Szimbólumok mint változók

Lispben a szimbólumok töltik be a változók szerepét, bár ez a dolgok erõs leegyszerûsítése. Szimbólumoknak létezik (létezhet) globális értéke, és egy adott szinten kötött értéke. A kettõ közti különbség a láthatósági szabályok megértésekor válik tisztává. Alapvetõen a Common Lispben lexikai láthatóság van, vagyis a kód textuális elhelyezkedése mondja meg, hogy egy adott szimbólum mely értéke érvényes. (Persze a nyelv lehetõséget nyújt dinamikus láthatóságra is a special deklaráció segítségével. ) Egy függvényen belül a függvény formális argumentumaira (lambda lista) kötések jönnek létre, vagyis az adott nevû szimbólum globális értéke a függvényen belül nem elérhetõ. Amennyiben egy függvény egy szimbólumot nem köt, akkor annak értéke az eggyel kijjebb lévõ kötés. Amennyiben a szimbólum egyáltalán nem kötött, akkor a szimbólum globális értéke az adott szimbólum értéke. Szimbólumokhoz értéket lehet rendelni a setq, set, setf függvényekkel, de kötni is lehet pl. a let makróval. A set, setq, setf mindig az adott szintû kötéseket módosítja. Egy példán keresztül megmutatjuk a különbséget. (A példán belül a user(szám) a lisp interpreter promptja. Az ezutáni kifejezést a felhasználó gépeli be, a következõ sor pedig az interpreter válasza.)

user(1): (setq x 5)

5

user(2): (setq z 3)

3

user(3): x

5

user(4): z

3

Ekkor mivel a lisp legfelsõ szintjén vagyunk, a globális értéket állítjuk.

user(5): (defun összead (x)

(print x)

(setq z 4)

(+ x z))

összead

Mint látjuk, az összead függvény lambda listáján szerepel az x szimbólum, így a függvény törzsén belül x-hez egy érték kötõdik, nem pedig a globális értékét látjuk. Meghívva összeadot:

user(6): (összead 3)

3

7

A függvény elõször kinyomtatja x összeadon belüli értékét, vagyis 3-at, majd a mûvelet eredményét.

user(7): x

5

user(8): z

4

A példából jól látható, hogy a függvényhíváskor a lisp x értékét elmentette, létrehozta a kötést, és a függvényből kilépve visszaállította x belépés elõtti értékét, viszont mivel z nem szerepel a formális argumentum listán, így z-nek a globális értékén végzünk mûveletet, így a függvényből kilépve is megmarad a változtatás hatása.

A let makró egy új kötést hoz létre a benne felsorolt változókra. Célja, hogy lokális változókat lehessen definiálni a let törzsén belüli kód számára. A let makrót használva biztosított, hogy a makrón belül felsorolt változók nem rontják el külsõ szimbólumok kötéseit.

## 3.5. Listák és lista kezelés

A lisp másik legfontosabb adatszerkezete a lista. A lista szolgál arra, hogy egyszûen összefoghassunk egy logikai egységbe egyébként különálló adatokat. Lispben egy zárójel pár határoz meg egy listát, de ezek tetszõleges mélységben egymásba ágyazhatóak. Minden lista két részbõl áll: a fejébõl és a farkából (az angol irodalomban CAR és CDR). A lista feje a nyitó zárójel után következõ S-kifejezés (tehát akár lista is lehet), a farka pedik a megmaradó rész a záró zárójelig. Szemléletesen egy lista fejét úgy kaphatjuk meg, hogy kivágjuk a lista elsõ S-kifejezését, és maga ez a kifejezés a fej, farkát pdig ugyanígy, de az a lista a farok, ami a kivágás után megmarad. Fontos!

A fentiekbõl következik, hogy míg egy lista feje bármilyen S-kifejezés lehet (szimbólum is, lista is), addig a lista farka mindig **LISTA**!

Például:

---

| Lista | Fej | Farok |
|---|---|---|
| (a b c) | a | (b c) |
| ((a) b c) | (a) | (b c) |
| ((a b c)) | (a b c) | () |
| () | () | () |

---

Egyszerû listának nevezzük azokat a listákat, amelyek nem tartalmaznak elemként újabb listákat. Azokat a listákat, amelyek tartalmaznak listákat is összetett listáknak nevezzük. Összetett listák bejárásakor egy adott S-kifejezés a listának annyadik szintjén helyezkedik el, ahány le nem zárt nyitó zárójel elõzi meg.

---

| Lista | S-kifejezés | Szint |
|---|---|---|
| (a b c) | a | 1 |
| ((a) b c) | a | 2 |

```
((a) b c)               (a)                    1

(((a) b ((c))))         c                      4()
```

---

Mielõtt rátérnénk a listakezelõ függvényekre szót kell ejtenünk egy fogalomról, a pontozott párról. A fentiekben láttuk, hogy a lista két részbõl áll: fejbõl és farokból, és a lista farka mindig lista. A pontozott pár is egy olyan adatszerkezet, amely fejbõl és farokból áll, azonban ennek farka vagy szimbólum, vagy pontozott pár. Itt a figyelmes olvasó felfedezheti, hogy tulajdonképpen a lista egy speciális pontozott pár, amelynek az utolsó farka nil. Pontozott párokat a listákhoz hasonlóan írunk le:

(a . b) egy pontozott pár. Fontos, hogy a pont elõtt és után is van egy egy szóköz karakter! A fentiekbõl következik, hogy

(a . nil) megegyezik (a)-val. És (a . (b . (c . nil))) megegyezik (a b c)-vel. A lisp amit lehet listaként nyomtat ki.

Most akkor térjünk rá a listaépítõ és listabontó függvényekre.

**cons:** Két S-kifejezésbõl egy pontozott párt hoz létre. Amennyiben a második elem egy lista, akkor az elsõ kifejezést a lista fejére fûzi.

(cons 'a 'b) -> (a . b)

Fontos! A fenti példában megjelent az aposztróf karakter. Ez egy beolvasási makró karakter, meggátolja a mögötte lévõ kifejezés kiértékelését. Itt a példában mi az a, b szimbólumokat akartuk összefûzni és nem az értéküket!

(cons 'a '(b)) -> (a b)

**append:** Tetszõleges számú lista elemeit összefûzi egy listává.

(append '(a) '(b) '(c)) -> (a b c)

**list:** Tetszõleges számú paraméterbõl listát készít.

(list 'a) -> (a)

(list 'a '(b) '((c))) -> (a (b) ((c)))

**car:** Lista fejét adja vissza.

(car '(a b)) -> a

**cdr:** Lista farkát adja vissza.

(cdr '(a b)) -> (b)

## 3.6. Kiértékelés lispben

A lisp rendszerek általában kétféle módban mûködnek: interpreterként és compilerként. Általában programfejlesztés alatt a lisp interpretert használják, mert könnyû formákat újra kiértékelni, módosítani.

Amikor már a proogram megfelelõen mûködik, akkor a compiler segítségével hatékony kódot lehet generálni.

Az interpreter ciklikus mûködést mutat. Az interpreter beolvas egy S-kifejezést, kiértékeli, majd kiírja a kiértékelés eredményét. Ezt a mûködést lispben a következõképpen fogalamazhatnánk meg:

(loop (terpri) (princ *prompt*) (print (eval (read))))

Nézzük meg, hogyan is néz ki a kiértékelés. Az interpreter megkísérel beolvasni egy formát. Amenyiben a forma nem nyitó zárójellel kezdõdik, akkor az egy atom, és az interpreter visszaadja az értékét. Amennyiben az elsõ karakter egy nyitó zárójel, akkor a beolvasandó forma egy függvény. A függvény paramétereit kiértékeli, majd ezeket átadja a függvénynek, és a függvényhívás eredményét adja vissza. Tehát a lisp amennyiben egy nyitó zárójellel találkozik, akkor a nyitó zárójelet követõ szimbóumot függvénynévnek tekinti, és az utána következõ formákat pedig a függvény paramétereinek. Elõször a paraméterek értékelõdnek ki, majd csak ezekután történik meg a függvényhívás. (A paraméterek maguk szintén lehetnek újabb függvényhívások.)

## 3.7. Legfontosabb függvények

Bár a fejezet címe függvényekrõl beszél, itt szó esik néhány speciális formáról és makróról is. Az ezek közti kükönbséget [] részletesen tárgyalja.

**Értékadás:**

setq {változó forma}* [Speciális forma]
Egyszerû értékadás Lispben. A változó nem értékelõdik ki. (Megegyezik a (set (quote változó) forma) formával.
Visszatérési érték: a forma értéke.

set {szimbólum forma}* [Függvény]
A szimbólum által megnevezett változó értékét beállítja a forma értékére.
Visszatérési érték: a forma értéke.

setf {hely új-érték}* [Makró]
Hely bármilyen olyan kifejezés lehet, amelynek értéket lehet adni.
Visszatérési érték: a forma értéke.

**Új függvény definiálása:**

defun név paraméter-lista függvény-törzs [Makró]
Névvel bíró függvény definiálása. A defun paraméterei nem értékelõdnek ki.
Visszatérési érték: a függvény neve.

**Adatspecifikus predikátumok:**

(Predikátum egy olyan logikai függvény, amely az argumentumáról eldönti, hogy az adott tulajdonság teljesül-e. Általában a predikátumok neve p-re végzõdik.)

Visszatérési érték: a vizsgálat eredménye.

null objektum [Függvény]

symbolp objektum [Függvény]

atom objektum [Függvény]

consp objektum [Függvény]

listp objektum [Függvény]

numberp objektum [Függvény]

**Feltételes elágazás:**

cond (feltétel-1 törzs-1) [Makró]
(feltétel-2 törzs-2)
...
(feltétel-n törzs-n)
Kiértékeli a feltétel részt, és amennyiben az értéke igaz, akkor végrehajtja az utána következõ törzset és kilép, egyébként veszi a következõ feltétel részt. Amennyiben egyetlen feltétel sem teljesül, akkor "keresztülesik" a cond-on.
Visszatérési érték: ha volt igaz feltétel rész, akkor az azt követõ törzsben utoljára kiértékelt forma értéke, egyébként nil.

if [feltétel kifejezés-ha-igaz kifejezés-ha-hamis]

when [feltétel kifejezések]

unless [feltétel kifejezések]

case [kulcs-forma

(kulcs-lista1 kifejezések)

(kulcs-lista2 kifejezések)

...

(kulcs-listan kifejezések)]

Logikai operátorok:

and [logikai-kifejezések]

or [logikai-kifejezések]

not [logikai-kifejezés]

Blokképzés:

prog1 [kifejezések]

prog2 [kifejezések]

progn [kifejezések]

Iteráció:

loop [kifejezések]

do [((változó-1 kezdõérték-1 léptetés-1)

(változó-2 kezdõérték-2 léptetés-2)

...

(változó-n kezdõérték-n léptetés-n))

(vég-teszt . visszatérési-érték)

törzs]

Kilépés:

return-from [blokk-név visszatérési-érték]

return [visszatérési-érték]

Összehasonlítás:

eq [szimbólum1 szimbólum2]

eql [szám1 szám2]

equal [lista1 lista2]

Új változó kötések létrehozása:

let [((változó1 érték1)

(változó2 érték2)

...

(változón értékn))

törzs)]

Aritmetikai mûveletek:

+ [szám ...]

- [szám ...]

* [szám ...]

- [szám ...]

Aritmetikai relációk:

< [szám ...]

<= [szám ...]

> [szám ...]

>= [szám ...]

= [szám ...]

/= [szám szám]

Ki- / bevitel:

print [kifejezés]

princ [kifejezés]

terpri [kifejezés]

format [stream vezérlõ-string paraméterek]

read [stream]

read-char [stream]

read-line [stream]

# 4. Programozás CLOS-ban

## 4.1. Osztályok definiálása

Osztályt a defclass makróval lehet:

(DEFCLASS osztály-név (szülõosztály-név*)

(slot-leírás*)

osztály-opció*)

Egyszerû osztályoknál nincs szükség az osztály opciókra, ezekre majd csak a késõbbiekben térünk vissza.

A slot-leírás a kövtkezõ formát ölti: (slot-név slot-opció*), ahol mindenegyes opció egy kulcsszó, amelyet egy név vagy kifejezés követ. A leghasznosabb slot opviók a következõk:

:ACCESSOR függvénynév

:INITFORM kifejezés

:INITARG szimbólum

Az inicializálási argumentumok (initarg) általában kulcsszavak (kettõsponttal kezdõdnek), amelyek megfelelnek a slot neveknek. Például a név nevû slot inicializálási argumentuma :név lenne.

A defclass makró hasonló a defstructhoz, de a szintaxisa egy kicsit eltérõ, és nagyobb a szabadság a dolgok elnevezésében. Például tekintsük a következõ definíciót:

(defstruct személy

(név 'Gábor)

(kor 10))

A defstruct automatikusan olyan slotokat definiálna, amelyekhez hozzárendelõdnek a default kezdeti értékeket kiszámító kifejezések, hivatkozó függvények, amelyekkel le lehet hivni, illetve be lehet állítani slotok értékeit, mint pl. személy-név, személy-kor, é s a make-személy, ami elfogadna inicializálási argumentumokat, mint pl.

(make-személy :név 'gábor :kor 10)

A defclass, ami hasonló hivatkozó függvényeket nyújtona, a következõ lenne:

(defclass személy ()

((név :accessor személy-név

:initform 'gábor

:initarg :név)

(kor :accessor személy-kor

:initform 10

:initarg :kor)))

Vegyük észre, hogy a defclass lehetõséget nyújt arra, hogy mi döntsük el, hogy hogyan hívjuk a dolgainkat. Például, nem kell a név slotra hivatkozó függvényt személy-névnek hívni, lehet egyszerûen csak név.

Általánosságban, olyan neveket kell választani, amelyek értelemmel bírnak egymással összefüggõ osztályok egy csoportjában, és nem kell mereven ragaszkodni a defstructnál megszokott konvenciókhoz.

Nem kell megadni mindenegyes slothoz minden opciót. Lehet, hogy nem akarsz lehetõséget nyújtani, hogy egy slotot inicializálni lehessen egyed generáláskor (make-instance hívásakor, amit majd a késõbbiekben tárgyalunk). Ebben az esetben ne adj meg :initarg-ot. Vagy esetleg nem létezik értelmes default érték. (Lehet, hogy az értelmes default értékeket majd minden esetben egy alosztály fogja megadni.) Ebben az esetben ne adj meg :initform-ot.

Ne feledd, hogy az osztályok objektumok. Az osztály objektum név alapjáni elérésére használd a

(FIND-CLASS név)

Általában erre nem lesz szükség.

## 4.2. Egyedek

Egy adott osztály egy egyedét a make-instance függvényhivással lehet generálni. Hasonló a defstruct által definiált make-x függvényekhez, de lehetõséget nyújt, hogy az osztályt, amelybõl példnyt szeretnél generálni argumentumként átadd.

(MAKE-INSTANCE osztály {initarg érték}*)

Az osztály objektum helyett használható az osztály neve is. Például:

(make-instance 'személy :kor 100)

Ennek a személy objektumnak a kor slotja 100-at, a név slotja pedig gábort tartalmazna. (Ez utóbbi a default értéke a slotnak.)

Gyakran hasznos, ha saját konstruktor függvényeket definiálsz, mintsem csak simán közvetlenül meghívnád a make-instance függvényt, mert így elrejtheted a megvalósítás részleteit, és nem kell mindenhez kulcsszó paramétert használnod. Például, ha azt szeretnéd, hogy a név es kor szükséges, pozicionális paraméterek legyenek ne pedig kulcsszó paraméterek, akkor a következõképpen definiálhatnád a konstruktorodat:

(defun make-person (name age)

(make-instance 'person :name name :age age))

A hivatkozó függvényekkel lehet slotok értékeit lekérdezni illetve beállítani.

user(1): (setq p1 (make-instance 'személy :név 'ági :kor 100))

#<person @ #x7bf826>

user(2): (személy-név p1)

ági

user(3): (személy-kor p1)

100

user(4): (setf (személy-kor p1) 101)

101

user(5): (személy-kor p1)

101

Vedd észre, hogy amikor defclass-t használsz, akkor az egyedeket a rendszer a következõképpen nyomtatja: #<...>, nem pedig #s(személy :név ági :kor 100). De ha akarod, megváltoztathatod az egyedek kinyomtatását, ha módszert definiálsz az adott osztályhoz a print-object generikus függvény alá.

Slotok értéke lekérdezhetõ a a nevükkel is, ha az alábbi konstrukciót használod:

(SLOT-VALUE egyed slot-név)

Amint a fenti példából kitûnik, a slot-value használata elárulja, hogy az adott érték egy slotban tárolódik. A hivatkozó függvények absztraktabbak, mert kevesebbet árulnak el a tárolás módjáról, és nem kell magukra a slotokra hivatkozni. Mivel a hivatkozó függvények függvények (valójában generikus függvények, mint azt majd a késõbbiekben láthatjuk), (újra) lehet definiálni õket, hogy értéküket más módon lehessen elérni. Amennyiben úgy döntesz, hogy egy értéket a továbbiakban már nem egy slotban kellene tárolni, hanem pl. valami mas módon számítani, megváltoztathatod a hivatkozó függvényt anélkül, hogy az õt hívó kódot is meg kellene változtatnod. the

Íme egy példa a slot-value-ra:

user(6): (slot-value p1 'név)

ági

user(7): (setf (slot-value p1 'név) 'judit)

judit

user(8): (személy-név p1)

judit

A describe hívásával számos fontos információt nyerhetsz egy függvényrõl:

user(9): (describe p1)

#<személy @ #x7bf826> is an instance of class

#<clos:standard-class személy @ #x7ad8ae>:

The following slots have :INSTANCE allocation:

kor 101

név judit

### 4.3 Slot opciók öröklõdése

A fenti osztálynak nincs szülõosztálya. Ezért állt a "()" a "defclass person" után. Valójában ez azt jelenti, hogy csak egyetlen szülõosztálya van: a standard-object.

Amikor vannak szülõosztályok, akkor az alosztályban lehetnek definiálva olyan slotok, amelyeket már definiáltunk a szülõosztályban.Amikor ez történik, akkor a slot opciókban rendelkezésre álló információt össze kell kombinálni. A fent említett opcióknál vagy az alosztályban lévõ opció jut érvényre (felülírja a szülõosztályban lévõ opciót), vagy unió képzés történik.

:ACCESSOR -- unió képzés

:INITARG -- unió képzés

:INITFORM -- felülírás

Ezt is várod el. Az alosztály az :initform felülírásával megváltoztathatja a default inicializálási értéket, és hozzáadhat az inicializálási argumentumokhoz illetve a hivatkozó függvényekhez.

Azonban az :accessor unió képzése csak annak következménye, ahogyan a generkus függvények mûködnek. Ha generikus függvények alkalmazhatóak egy C osztály egyedeire, akkor alkalmazhatóak a C osztály alosztályainak az egyedeire is. ( A hivatkozó függvények generikusak. )

Íme a személy osztály néhány alosztálya:

tanár - a személy osztály közvetlen alosztálya;

matek-tanár - a tanár osztály közvetlen alosztálya és a személy osztály indirekt alosztálya (a tanár osztályon keresztül).

```
user(10): (defclass tanár (személy)

((tárgy :accessor tanár-tárgy

:initarg :tárgy)))

#<clos:standard-class tanár @ #x7cf796>

user(11): (defclass matek-tanár (tanár)

((tárgy :initform "Matematika")))

#<clos:standard-class matek-tanár @ #x7d94be>

user(12): (setq p2 (make-instance 'matek-tanár

:név 'János

:kor 34))

#<matek-tanár @ #x7dcc66>

user(13): (describe p2)

#<matek-tanár @ #x7dcc66> is an instance of

class #<clos:standard-class matek-tanár @ #x7d94be>:

The following slots have :INSTANCE allocation:

kor 34

név János

tárgy "Matematika"
```

Amint láthatod, az osztályokat a lisp a következõképpen nyomtatja:

#<clos:standard-class maths-teacher @ #x7d94be>.

A #<...> jelölés általában az alábbi formájú:

#<az-objektum-osztálya ... more information ...>

Így a matek-tanár egy egyedét a rendszer #<matek-tanár ...>-ként nyomtatja. A fönti jelölésbõl kitûnik, hogy az osztályok a standard-class osztály egyedei. Defclass standard osztályokat, defstruct pedig stuktúra osztályokat definiál.

Mivel az osztályok objektumok, ezért maguk is osztályok egyedei. Azokat az osztályokat, amelyeknek osztályok az egyedei - mint pl. standard-class - metaosztályoknak nevezzük.

## 4.4 Többszös öröklõdés

Egy adott osztálynak több szülõosztálya is lehet. Egyszerû öröklõdésnél (egy szülõosztály) egyszerû a szülõosztályokat rendezni a legsecifikusabbtól a legáltalánosabbig. Ez esetben a következõ a szabály:

Mindenegyes osztály specifikusabb a szülõosztályánál.

Többszörös öröklõdés esetén a rendezés nem ilyen egyszerû. Tegyük fel, hogy az alábbi osztályt definiáltuk:

(defclass a (b c) ...)

Az A osztály specifikusabb mind a B mind pedig a C osztálynál (értsd: a megfelelõ osztályok egyedei), de mitévõk legyünk, ha valami (pl. egy :initform vagy egy módszer) definiálva van mind B-ben, mind pedig C-ben? Melyik bírálja felül a másikat?

CLOS-ban erre a következõ a szabály: az osztálydefinícióban korábban felsorolt szülõosztályok specifikusabbak a listán késõbb szereplõknél. Így:

Egy adott osztály esetén a korábban felsorolt osztályok specifikusabbak a késõbb felsoroltaknál.

ezeket a szabályokat alkalmazza az interpreter/compiler hogy egy adott osztályra és szülõosztályaira kiszámítson egy lineáris sorrendet, a legspecifikusabbtól a legáltalánosabbig. Ez a sorrend az osztály osztály precedencia listája.

Azonban a fenti két szabály nem mindig elégséges, hogy egy egyedi, egyértelmû sorrendet meghatározzunk, ezért a CLOS rendelkezik egy algoritmussal, amely kritikus esetekben feloldja a problémát. Ez biztosítja, hogy minden implementáció mindig azonos sorrendet produkál, de általában rossz programozási technikának tekinthetõ, ha a programozó kihasználja a pontos sorrendet. Ha némely szülõosztály sorrendje lényeges, az explicite kifejezhetõ az osztálydefinícióban.

## 4.5 Generikus függvények és módszerek

A CLOS generikus függvényei közel állnak az "üzenetekhez". Ahelyett, hogy a következõ formát írnánk:

(SEND egyed módszer-név argumentum*) ;nem CLOS

azt irod, hogy

(módszer-név egyed argumentum*) ;CLOS

A módszerek/üzenetek generikus függvények -- függvények, amelyeknek adott osztály egyedeivel szembeni viselkedését módszerek definiálásával lehet meghatározni.

(DEFGENERIC függvény-név lambda-lista)

használható generikus függvény definiálására. Azonben nem kell meghívnod a defgeneric-et, mert a defmethod automatikusan automatikusan definiálja a generikus függvényt, ha még az nem volt definiálva. Másrészrõl viszont hasznos lehet a defgeneric használata, mint egy deklaráció, hogy egy függvény létezik, és bizonyos paraméterei vannak.

Mindenesetre minden érdekes dolog a módszerekben történik. Egy módszert az alábbi formula definiál:

(DEFMETHOD generikus-függvény-név specializált-lambda-lista

forma*)

Ez felettébb titokzatosnak tûnhet, de hasonlítsd csak össze a defunnal!

(DEFUN függvény-név lambda-lista forma*)

A "lambda lista" a formális paraméterek listája, és még olyan dolgok, mint pl. &optional vagy &rest. Ez utóbbiak miatt mondjuk, hogy "lambda-lista" "(paraméter*)" helyett amikor a szintaxist leírjuk. (Itt nem beszélünk többet a módszerekben szereplõ &optional, &rest, ill. &key paraméterekrõl. A szabályok részletes leírását megtalálhatod []-ben.

Tehát egy egyszerû függvénynek (változó1 változó2 ...) alakú lambda listája van. A módszerek lambda listájában mindenegyes paramétert specializálni lehet egy adott osztályra a következõképpen:

((változó1 osztály1) (változó2 osztály2) ...)

A specializáló opcionális. Elhagyásakor a módszer alkalmazható lesz mindenféle osztály egyedeire, olyan osztályokat is beleértve, amelyek nem defclass-szal voltak definiálva. Például:

(defmethod tárgy-csere ((tanár tanár) új-tárgy)

(setf (tanár-tárgy tanár) új-tárgy))

Itt az új-tárgy bármilyen objektum lehet. Ha korlátozni szeretnéd, akkor valami hasonlót csinálhatnál:

(defmethod tárgy-csere ((tanár tanár) (string új-tárgy))

(setf (tanár-tárgy tanár) új-tárgy))

Vagy definiálhatnál tárgy osztályokat.

A "klasszikus" objektum-orientált programozásban a módszerek csak egy paramétert specializálnak. CLOS-ban többet is specializálhatsz. Ha eélsz ezzel a lehetõséggel, akor néha ezeket a módszereket többszörös módszereknek is nevezik.

Egy C osztálynak definiált módszer felülrja mindenegyes szülõosztályának definiált azonos nevû módszerét. C módszere "specifikusabb" a szülõosztályok módszereinél, mert C specifikusabb, mint azok az osztályok, amelyektõl örököl (pl. a kutya speciálisabb mint az állat).

Többszörös módszerek esetén annak eldöntéséhez, hogy melyik módszer specifikusabb, több paramétert kell megvizsgálnunk. A paramétereket balról jobbra lexikografikusan vizsgáljuk.

(defmethod teszt ((x szám) (y szám))

'(szám szám))

(defmethod teszt ((i egész) (y szám))

'(egész szám))

(defmethod teszt ((x szám) (j egész))

'(szám egész))

(teszt 1 1) => (egész szám), és nem (szám egész)

(teszt 1 1/2) => (egész szám)

(teszt 1/2 1) => (szám egész)

(teszt 1/2 1/2) => (szám szám)

## 4.6 Módszer kombináció

Amikor egynél több osztály definiál módszereket egy generikus függvényhez, és több módszer alkalmazható a paraméterek egy adott halmazához, akkor az alkalmazható módszereket a lisp egy effektív módszerbe kombinállja. Ekkor mindenegyes módszer definíciója csak része az effektív módszer definíciójának.

CLOS egy módszer kombinációt - a standard módszer kombinációt - mindig támogatja, de lehetõség van új módszer kombináció definiálására is. A standard módszer kombináció négyfajta módszert foglal magában:

* Primary módszerek az effetív módszer fõ törzsébõl. Csak a legspecifikusabb primary módszer hívódik le, de az meghívhatja a következõ legspecifikusabb primary módszert (call-next-method).

* :before módszerek a primary módszer(ek) elõtt hívódnak le, a legspecifikusabb :before módszer hívódik le elsõként.

* :after módszerek a primary módszer(ek) után hívódnak le, a legspecifikusabb :after módszer hívódik le utoljára.

* :around módszerek minden más módszer elõtt futnak. A primary módszerekhez hasonlóan:
csak a legspecifikusabb hívódik le, és a call-next-method hívhatja le a következõt. Amikor a legáltalánosabb :around módszer meghívja a call-next-method-ot, akkor a :before, :after, és primary módszerek kombinációja hívódik le.

:Before, :after, és :around módszereket úgy jelölünk, hogy a megfelelõ kulcsszavakat minõsítõkként betesszük a módszer definícióba. Az alábbi példa megmutatja, hogy hová is kell valójában tenni ezeket a minõsítõket.

(DEFMETHOD gf-név minõsítõ specializált-lambda-lista

forma*)

:Before és :after módszereket egyszerû használni, és egy egyszerû példán be is mutatjuk, hogy hogyan is mûködnek.

(defclass étel () ())

(defmethod süss :before ((f étel))

(print "Az étel nemsokára megsül."))

(defmethod süss :after ((f étel))

(print "Az étel megsült."))

(defclass pite (étel)

((töltelék :accessor pite-töltelék

:initarg :töltelék

:initform 'alma)))

(defmethod süss ((p pite))

(print "Pitét sütök")

(setf (pite-töltelék p) (list 'sult (pite-töltelék p))))

(defmethod süss :before ((p pite))

(print "A pite nemsokára kisül."))

(defmethod süss :after ((p pite))

(print "A pite kisült."))

(setq pite-1 (make-instance 'pite :töltelék 'alma))

És most:

user(): (süss pite-1)

"A pite nemsokára kisül."

"Az étel nemsokára megsül."

"Pitét sütök"

"Az étel megsült."

"A pite kisült."

(sült alma)

# 5. Az Allegro Common Lisp konkurens és valós idejû kiterjesztései

## 5.1. Processzek

### 5.1.1. A processz fogalma, implementációja és futtatása az ACL környezetben

A processz az ACL környezetben a stack-csoport (stack-group) fogalmára épül, mely lényegében korutinok, és közöttük vezérlérlésátadás megvalósítását illetve a félbemaradt mûködés folytatását teszi lehetõvé korutin transzfer segítségével. A stack-csoport fogalmával ezért a továbbiakban részletesen nem foglalkozunk, errõl az [] 12.1 fejezete tartalmaz bõvebb információkat.

A folyamat az általános definícíónak megfelelõ tulajdonságokkal rendelkezik. Az ACL környezet beépített preemptív, prioritásos, idõosztásos ütemezõvel rendelkezik a folyamatok állapotátmeneteinek vezérlésére.

Minden folyamathoz definíció szerint tartozik két lista:

a *run reasons* listán azok az okok vannak felsorolva (mindegyik szabályos Lisp objektum), amelyek a folyamat futtathatóságát indokolják;

az *arrest reasons* listán azok az okok vannak felsorolva (mindegyik szabályos Lisp objektum), amelyek miatt a folyamat nem futtatható.

Az a processz futtatható, amelynek *run reason* listája legalább egyelemû, és *arrest reason* listája üres.

A folyamatok várakoztatásának legalapvetõbb eszköze az *mp:process-wait* függvény. Annek argumentumai egy *f* függvény és a hozzá tartozó *{<farg>}* argumentumok. Ha egy *mp:process-wait* hívás szerepel egy *P* processzben, a futtató lehívja az argumentumként megadott *f* függvényt a *{<farg>}* argumentum listával. *P* mindaddig várakozó állapotban marad, ameddig a hívás *nil* értékkel nem tér vissza. Ezt a tesztet a futtató megfelelõ idõpillanatokban automatikusan újra elvégzi minden várakozó processz esetén.

Az ütemezõ két várakozási sorban tartja nyilván, az egyikben a futásra kész, a másikban a várakozó folyamatokat. Az inaktív folyamatokat a futtató nem tartja nyilván. A folyamatoknak a futtató rendszer megfelelõ várakozási soraiba való behelyezése azoknak a függvényeknek a feladata, amelyek a folyamatok *run-* és *arrest reason* listáit módosítják.

Az ütemezõ feladata a futásra kész folyamatok közül a futó folyamat kiválasztása. Ezt a következõképpen hajtja végre. Sorban kiértékeli a várakozó folyamatok várakozó függvényeit, és az összes olyan folyamatot, amelynek a várakozó függvénye nem *nil* értéket ad vissza, áthelyezi a futásra kész sorba. Ha a futásra kész sor ezek után is üres, külsõ aszinkron I/O vagy timer eseményre kezd várakozni, és ilyenkor átadja a vezérlést az operációs rendszernek egy UNIX *select* ustasításon keresztül megadva azokat a file (socket stb.) leírókat, amelyekbõl bemeneti adatnak kell érkeznie. Ha ezek közül akár csak eggggyben is fel nem dolgozott adat található, a futtató (és az ACL környezet) visszakapja a vezérlést, és a megfelelõ folyamat futtatható. A futásra kész folyamatok küzül mindig a legnagyobb prioritású fut elõször, és legalább a folyamat definíciójában megadott idõszeleteig (tehát ennél korábban a folyamat futása csak a folyamaton belüli okból függesztõdhet föl, mint amilyen pl. a *process-wait* függvény hívása).

### 5.1.2. Processz változók és függvények

mp:*all-processes* [Változó]

Ennek a változónak az értéke az összes olyan processz listája, amelyeket létrehoztak, futásuk nem fejezõdött be, és nem lõtték le õket.

mp:*current-process* [Változó]

Értéke az éppen futó processz vagy nil, ha maga a futtató rendszer fut. Csak olvasható változóként kell kezelni.

mp:*default-process-quantum* [Változó]

A folyamatokhoz rendelt default idõszelet értéke. Legalább 0.1, legföljebb 20 (másodperc) lehet.

**mp:start-scheduler**

Az ütemezõ közvetlen elindítására szolgál. Az ütemezõt indirekt módon az *mp:process-reset* és az *mp:process-run-function* is elindítja.

**mp:make-process**

*Argumentumok* &key :name :quantum :priority :run-reasons :arrest-reasons :reset-action :initial-bindings

- Ez a függvény egy új processz objektumot generál, amelyet visszaad értékként, de a folyamat még nem futtatható (errõl külön kell gondoskodni az *mp:process-run-function* függvény segítségével).

:name a processz neve, értéke string, default értéke *Anonymous*.

:quantum a processz minimális futási idejét definiálja ms-okban, default értéke 2.

:priority a processz prioritása, default értéke 0 (nagyobb számok nagyobb prioritást jelentenek).

:run-reasons, :arrest-reasons

a processzhez rendelt megfelelõ (korábban leírt rendeltetésû) listák, default értékük *nil*.

:reset-action

ha értéke *t*, akkor a folyamat újraindul, ha reset-elõdik vagy végrehajtása befejezõdik. Különben a processz törlõdik. Default értéke *nil*.

**mp:process-run-function**

*Argumentumok* név-vagy-kulcsszavak függvény &rest argumentumok

név-vagy-kulcsszavak

a folyamat nevét jelölõ string vagy az *mp:make-process* függvény által elfogadott kulcsszó argumentumok listája.

függvény a folyamatot reprezentáló programarészt adja meg.

A függvény végrehajt egy *mp:make-process* hívást a megadott argumentumokkal, és az eredményül kapott folyamatot visszaadja.

**mp:process-enable**

*Argumentumok* processz

A processz-ben megadott folyamatot aktivizálja törölve minden *run*- és *arrest reason*-t, majd a folyamatnak egyetlen :enable *run-reason*-t adva.

**mp:process-disable** [Függvény]

*Argumentumok* processz

Az argumentumként megadott folyamat mûködését letiltja törölve a *run reasons* és *arrest reasons* lista minden elemét.

**mp:process-run-reasons** [Függvény]

*Argumentumok:* processz

Visszaadja az argumentumként megadott processz run-reason listáját.

**mp:process-arrest-reasons** [Függvény]

*Argumentumok: processz*

Visszaadja az argumentumként megadott processz arrest reason listáját.

**mp:process-add-run-reason** [Függvény]

*Argumentumok*: processz objektum

Objektumot hozzáadja processz run-reason listájához.

**mp:process-add-arrest-reason** [Függvény]

*Argumentumok*: processz objektum

Objektumot hozzáadja processz arrest reason listájához.

**mp:process-revoke-run-reason** [Függvény]

*Argumentumok*: processz objektum

**Objektumot eltávolítja processz run-reason listájáról.**

**mp:process-revoke-arrest-reason** [Függvény]

*Argumentumok*: processz objektum

**Objektumot eltávolítja processz arrest-reason listájáról.**

mp:process-runnable-p [Függvény]

Argumentumok: processz

Visszaadott értéke t, ha processz futássra kész.

mp:process-priority [Függvény]

Argumentumok: processz

Visszaadja processz prioritását. Default értéke 0, és setf-fel bármilyen integer értékre beállítható. Az ütemezõ mindig a legmagasabb prioritású folyamatot indítja, ha ezekbõl több is van, közöttük körforgó prioritás érvényes.

mp:process-quantum [Függvény]

Argumentumok: processz

Visszaadja processz minimális idõszeletének hosszát, melynek értékét setf-fel lehet 0.1-20 másodpercre beállítani. Default értéke 2.

mp:process-whostate [Függvény]

Argumentumok: processz

Visszaadja processz aktuális "who-line" stringjét, mely processz állapotáról ad tájékoztatást, és amelynek értéke setf-fel megváltoztatható.

**mp:without-scheduling** [Függvény]

*Argumentumok*: &body törzs

Segítségével megakadályozható processz preemptív (idõosztásos) átütemezése törzs végrehajtása során. Processz felfüggesztõdik, és átütemezés következik be, ha valamilyen ok miatt várakozó állapotba kerül vagy egy mp:process-allow-schedule hívást hajt végre. Kritikus szakaszok védelmére használható.

**mp:process-allow-schedule** [Függvény]

*Argumentumok*: &optional másik-processz

Hatására átütemezés történi. Ha másik-processz-t megadjuk, és futásra kész, akkor a scheduler a prioritásától függõen ezt fogja legalább egy idõszeletre elindítani.

mp:*disallow-scheduling* [Változó]

Csak olvasható változó, melyet felhasználói kódból állítani nem szabad. Értéke t, ha a konkurens processzek futtatása le van tiltva egy mp:without-scheduling forma törzsében.

**mp:process-sleep** [Függvény]

*Argumentumok*: másodpercek

A hívó folyamat legalább másodpercek-nek megfelelõ idõtartamra felfüggesztõdik. Másodpercek bármilyen nem negatív, nem komplex érték lehet.

**mp:process-wait** [Függvény]

*Argumentumok*: whostate függvény &rest argumentumok

Felfüggeszti az aktuálisan futó folyamatot, függvény-t argumentumok-kal ismételten meghívja, amíg az eredmény t nem lesz. A whostate paraméter string kell legyen, mely a várakozás idõtartamára beíródik a folyamat whostate változójába. a process-wait függvény visszadott értéke nil. A wait függvény az ütemezõ stack-jén (tehát az ütemezõ dinamikus változó terében) fut. Ezért azoknak a változóknak, amelyek a wait függvényben az állapotváltozás jelzésére szolgának, globális változóknak, vagy ezekre vonatkozó közvetlen referenciáknak kell lenniük, melyek a futtató stackjérõl is elérhetõk.

**mp:process-wait-with-timeout** [Függvény]

*Argumentumok*: whostate timeout függvény &rest argumentumok

Hatása és használata hasonló az mp:process-wait függvényéhez, de a várakozás legfeljebb timout másodpercig tarthat, azaz a várakozás megszakad, ha függvény timeout leteltével sem ad vissza t értéket. Timeout tetszõleges valós szám lehet, a negatív számokat 0-ként kezeli.

**mp:wait-for-input-available** [Függvény]

*Argumentumok*: stream-ek &key :wait-függvény :whostate :timeout

Ennek a függvénynek a segítségével lehet bemeneti adatokra várakozni a megadott stream-ekbõl.

A stream-ek argumentum lehet egyelemû vagy lista. Minden elem vagy stream objektum vagy egy, az operációs rendszerbõl származó file-leíró (tipikusan egy kicsi, nemnegatív integer).

Ezzel a függvénnyel inputra lehet várakozni a megadott streameken. Ha közülük bármelyiken van elérhetõ adat, a függvény visszaadja azoknak a stream-eknek (file leíróknak) a listáját, amelyeken új adat található. Ezután azok a függvények (pl. read-char), amelyek ezekbõl a stream-ekbõl olvasnak, várakozás nélkül visszatérnek (bár file-vége értékkel is visszatérhetnek).

A wait-függvény argumentum default értéke #'stream:stream-listen. Ez egy generikus függvény, amelynek megfelelõ módszerei vannak különbözõ stream-ekhez és file leírókhoz, mint pl. terminál, socket vagy más interaktív stream-ek. Olyan stream-ekre is mûködik, amelyeknek nincsen file leírójuk, mint pl. window stream-ek.

**mp:with-timeout** [Makró]

*Argumentumok*: (idõtartam . timeout-formák) &body törzs

A törzs kiértékelõdik, mint egy progn törzs. Ha a törzs kiértékelése nem fejezõdik be a megadott idõtartamon belül, akkor a törzs végrehajtása megszakad, és végrehajtódnak a timeout formák. Ha a törzs végrehajtása a megadott idõtartamon belül végetér, akkor a timeout formák nem hajtódnak végre.

**mp:make-process-lock** [Függvény]

*Argumentumok*

**mp:process-lock** [Függvény]

**mp:process-unlock** [Függvény]

*Argumentumok*

**mp:process-lock-locker** [Függvény]

*Argumentumok*

**mp:with-process-lock** [Függvény]

*Argumentumok*

# 6. Makrók

Lispben a makrók egy nagyon hatékony és rugalmas eszközt nyújtanak a Lisp szintaxis kiterjesztésére. Sajnos lényegesen nehezebb õket tervezni, és hibákat keresni.

A Lisp függvények lisp értékeket várnak bemenetként és lisp értékeket adnak vissza, és futásidõ alatt hajtódnak végre. Ezzel szemben a makrók Lisp kódot kapnak bemenetül és szintén Lisp kódot adnak vissza. Fordító elõfeldolgozási idõben hajtódnak végre, és majd az eredményül kapott kód hajtódik végre futási idõben. Majdnem minden a makrók használatából eredõ hiba ennek a mechanizmusnak a félreértéséhez vezethetõ vissza.

Azonban mielõtt rátérnénk a makrók tárgyalására nézzünk meg egy makrókbna gyakran használt eszközt, a backqote-t (A backquote nemcsak makrókban használatos, általános minták építésénél nagyon hasznos.).

**Sima Backquote.**

Önmagában a backquote megegyezik a quote-val, listák építésére kényelmesen alkalmazható.

*`(a b c) == '(a b c)*

*`alma == 'alma*

**Vesszõ.**

A vesszõ, amely csak backquote-olt lista belsejében használható, az "unquote" operátor, vagyis a vesszõ után álló kifejezés az egész listára érvényes quote ellenére (pontosan a vesszõ miatt) kiértékelõdik. Így ezen a szinten a backquote karakter a következõképpen is értelmezhetõ: "Vegyél mindent úgy ahogy le van írva, kivéve, ha vesszõ elõzi meg. A vesszõvel megelõzött kifejezés helyére helyettesítsd be az értékét".

*(setq foo '(a b))*

*`(foo ,foo) == (list 'foo foo) => (foo (a b))*

*(setq x 2)*

*`(x (+ x 3) ,x (+ ,x 3) ,(+ x 3)) => (x (+ x 3) 2 (+ 2 3) 5)*

**Vesszõ-At**

A comma-at, amely szintén csak backquote-olt lista belsejében használható, az "unquote-beszeletelõ" operátor. Vagyis kiértékeli a mögötte lévõ kifejezést, amelynek egy listára kell kiértékelõdnie, és a lista elemeit az õket körülvevõ zárójelek nélkül beteszi a quote-olt külsõ listába. (Gyakorlatilag ez az operátor a kiértékelt kifejezésrõl leszedi a zárójelet.)

*(setq foo '(a b))*

*`(foo ,foo ,@foo) => (foo (a b) a b)*

## 6.1. Az alapgondolat

A makrók kiértékeletlen Lisp kódot kapnak és egy lisp formát adnak vissza. Ez a forma egy olyan heleys kódrészlet kell hogy legyen, amely kiszámítja a megfelelõ értéket. Példa:

*(defmacro Square (x)*

*`(* ,x ,x))*

A fenti forma azt jelenti, hogy valahányszor az elõfeldolgozó talál egy *(Square XXX)*-szel akkor azt lecseréli *(* XXX XXX)*-re. A fordító már majd csak az eredményül kapott kódot fogja látni.

## 6.2. Hibakeresés: *macroexpand-1*

Amikor egy függvényt tervezünk, megthetjük, hogy a függvényt meghívjuk a Lisp Listenerben, és megnézzük, hogy a helyes értéket adja-e vissza. Azonban amikor a makrót hívjuk meg, két dolog történik: elõször a renszer kiterjeszti a makrót a megfelelõ kóddá, majd a kódot kiértékeli. Hibakereséskor sokkal hasznosabb, ha ezt a két fázist külön-külön, egymástól függetlenül tudjuk vizsgálni. A *macroexpand-1* függvény az elsõ fázis eredményét adja vissza:

*(macroexpand-1 '(Square 9)) => (* 9 9)*

## 6.3. Cél: Az argumentumok kiértékelésének kézbentartása

Mivel makrókat lényegesen nehezebb írni, mint függvényeket, ezért amikor nem indokolt, akkor nem érdemes makrót használni. Persze ez nem jelenti, hogy teljesern el kellene vetni a makrók használatát, viszont körültekintéssel kell használni. Például a Square esetében semmi sem indokolja a makró használatát, a defun tökéletesen megfelelt volna.

Lispben, a C-vel ellentétben, nincs szükség makrók használatára, hogy a függvényhívások miatti futási idõbeli overheadet elkerüljük, erre egy külön módszer van (az *"inline"* proclamáció), amely ezt lehetõvé teszi anélkül, hogy szintaxist kellene váltani. Makrókkal kézben lehet tartani az argumentumok kiértékelését (mikor értékelõdjönek ki és mikor ne), míg ezt függvényekkel nem lehet megtenni, mert a függvények minden argumentumukat kiértékelik még mielõtt belépnének a törzsükbe. A makrók hanem mondjuk meg nekik, hogy értékelje ki bizonyos argumentumait, akkor elõfeldolgozási idõ alatt egyetlen argumentumát sem értékeli ki, így egy olyan kóddá terjeszthetõk ki, amely adott esetben nem értékeli ki minden argumentumát. Példa kedvéért tegyük fel, hogy a *cond* része a nyelvnek, de az *if* nem, és meg szeretnénk írni az *if*-et a *cond* fellhasználásával.

```
(defun If-Wrong (Test Then &optional Else)
(cond
(Test Then)
(t Else)))
```

A fenti megvalósítással az a probléma, hogy az mindig kiértékeli az összes argumentumát, míg az *if* szemantikája szerint a Then-bõl, illetve az Else-bõl pontosan csak az egyik értékelõdik ki. Például:

```
(let ((Test 'A))
(If-Wrong (numberp Test)
(sqrt Test)
"Sorry, SQRT is only defined for numbers."))
```

A fenti forma hibát jelez, mert megpróbál gyököt vonni 'A-bol. A helyes változat, amely úgy viselkedik mint az *if* (eltekintve attól, hogy az *if*-nek csak egy kötelezõ argumentuma van) a következõ lehetne:

```
(defmacro If (Test Then &optional Else)
"IF helyettesítõ kód, 2 vagy három argumentumot fogad. Ha az elsõ nem Nil-re értékelõdik ki, akkor kiértékeli
és visszaadja a másodikat. Egyébként a harmadikat értékeli ki és adja vissza (amelynek a default értéke NIL)."
`(cond
(,Test ,Then)
(t ,Else)))
```

Egy hasonló példa a *NAND* ("Not And"), Amely igazat ad vissza, ha legalább egy argumentuma hamis, de a beépített and függvényhez hasonlóan "rövidített kiértékelés"t végez, vagyis mihelyst megvan a válasz rögtön visszaadja és nem értékeli ki a fennmaradó argumentumokat.

```
(defmacro Nand (&rest Args)

`(not (and ,@Args)))
```

## 6.4. Hibák

(A) Argumentumok futási idõ alatti kiértékelésének megkísérlése

(B) Argumentumok túl sokszor történõ kiértékelése

(C) Változónév ütközések

## (A) Argumentumok futási idõ alatti kiértékelésének megkísérlése

A makrók kiterjesztése compiler elõfeldolgozási idõ alatt történik, így az argumentumok értékei általában nem hozzáférhetõek, és az olyan kód, amely megpróbálja használni õket nem fog mûködni. Például tekintsük a Square alábbi definícióját, amely *(Square 4)*-et megpróbálja 16-tal helyettesíteni *(* 4 4)* helyett.

```
(defmacro Square (x)
(* x x))
```

Ez valóban mûködne *(Square 4)*-re, de minden bizonnyal problémához vezetne *(Square x)* esetében, mert x valószínûleg egy változó, amelynek az értéke majd csak futási idõ alatt lesz ismert. Mivel a makrók néha kiterjesztési idõben használnak változókat és függvényeket, és hogy a hibakeresést megkönnyítsük, ajánlatos minden makró definíciót és azokat a változókat, ill. függvényeket, amelyeket kiterjesztési idõben használnak (szembeállítva magával a kóddal, amivé kiterjesztõdnek) egy külön fille-ba helyezni, amelyet betöltünk, még mielõtt bármilyen file betöltõdne, amely ezekre hívatkozna.

### (B) Argumentumok túl sokszor történõ kiértékelése

Tekintsük át mégegyszer a *Square* makró elsõ definícióját.

```
(defmacro Square (x) `(* ,x ,x))
```

Elsõ pillantásra ez tökéletesnek tûnik. Azonban próbálj meg macroexpand-1-gyel kiterjeszteni egy formát, és észreveszed, hogy az argumentumot kétszer értékeli ki:

```
(macroexpand-1 '(Square (foo 2))) => (* (foo 2) (foo 2))
```

A foo kétszer hívódik meg, bár csak egyszer kellene. Ez nem csak hogy nem hatékony, de rossz választ is adhat vissza, ha pl. foo egy generátor függvény, és nem mindig ugyanazt a választ adja vissza. (Ilyen függvény lehet pl. a *Next-Integer*, amely mindig egyel nagyobb egészet ad vissza, vagy a *random* függvény, amely egy véletlen számot ad vissza.). Ezekben az esetekben már közel sem kapnánk négyzetszámot! Lispben már elõfeldolgozási idõben rendelkezésre áll a teljes Lisp, így tetszõleges konstrukció használható a fenti probléma megoldására. Például a let makró használható:

```
(defmacro Square2 (x)
`(let ((temp ,x))
(* temp temp)))

(macroexpand-1 '(Square2 (foo 2))) =>
(let ((temp (foo 2))
(* temp temp))
```

És ez az amit el szerettünk volna érni.

### (C) Változónév ütközések

Amikopr a *let*-et használjuk a többszörös kiértékelés elkerülése érdrkében, akkor biztosnak kell lennünk, hogy nincs ütközés a választott lokális változó és bármely más létezõ változónév között. A *Square2* fenti változata tökéletesen biztonságos, de tekintsük a következõ makrót, amely két számot kap, és az összegüket négyzetre emeli:

```
(defmacro Square-Sum (x y)
```

```
`(let* ((First ,x)
(Second ,y)
(Sum (+ First Second)))
(* Sum Sum)))
```

Ez jónak tûnik, még makrókiterjesztés után is:

```
(macroexpand-1 '(Square-Sum 3 4)) =>
(let* ((First 3)
(Second 4)
(Sum (+ First Second)))
(* Sum Sum))
```

ami a helyes eredményt adja. Azonban ennek a változatnak van egy "bujkáló' hibája. Az általunk választott lokális változó nevek ütközhetnek már meglévõ lokális változó nevekkel. Ebben az esetben pl. ha az egyik argumentum neve First lenne, akkor problémák jelentkeznének:

```
(macroexpand-1 '(Square-sum 1 First)) =>
(let* ((First 1)
(Second First)
(Sum (+ First Second)))
(* Sum Sum))
```

Ebben az esetben a problémát az jelenti, hogy a (Second First) az új First nevû lokális változó értékét kapja meg, és nem azt, amit masodik paraméterként a függvénynek megadtunk. Így a

```
(let ((First 9):
(Square-Sum 1 First))
```

4-et adna vissza 100 helyett! Ennek a problémának a megoldása eléggé bonyolult, a gensym függvény hívását igényli, amely segítségével egy olyan lokális változó nevet lehet generálni, amely garantáltan egyedi.

```
(defmacro Square-Sum (x y)
(let ((First (gensym "First-"))
(Second (gensym "Second-"))
(Sum (gensym "Sum-")))
(let* ((,First ,x)
(,Second ,y)
(,Sum (+ ,First ,Second)))
(* ,Sum ,Sum))))
```

Ezt a makró definíciót kiterjesztve:

```
(macroexpand-1 'Square-Sum2 1 First)) =>
(let* ((#:First-590 1)
(#:Second-591 First)
(#:Sum-592 (+ #:First-590 #:Second-591)))
(* :Sum-592 #:592))
```

Ea a definíció már nem függ egyetlen, a mekródefinícióban lokális változónévtõl, és mivel a generált szimbólumok garantáltan egyediek, így biztonságosan elkerülhetõk a névütközések.

Felhasznált irodalom: .Jeff Dalton

CLtL