

LISP

PROGRAMAZIO-LENGOAIA



J.R. Bastarrika
K. Sarasola

LISP
PROGRAMAZIO-LENGOAIA

Juan Ramon Bastarrika
Kepa Sarasola

HEZKUNTZA, UNIBERTSITATE ETA IKERKETA SAILAK ONETSIA
(1991-V)

***Juan Ramon Bastarrika Eusko Jaurlaritza eta UPV/EHUREN Postgraduatu
euskaldunentzako beka bat zuela parte hartu du lan honetan.***

© U.E.U.ko Informatika Saila
General Concha 25, 6. 48010 BILBO

I.S.B.N.: 84-86967-32-5
Lege-gordailua: BI-1214-91

Inprimategia: BOAN S.A.
Particular de Costa 12, 48010 BILBO
Azalaren disainua: Iñaki Madariaga

AURKIBIDEA

AURKIBIDEA	V
HITZAURREA	IX
1. SARRERA.....	1
1.1. <i>Sorrera eta bilakaera historikoa</i>	<i>1</i>
1.2. <i>Lengoiaren ezaugarriak</i>	<i>3</i>
1.3. <i>Aldeko eta kontrako eritziak. Balantzea</i>	<i>6</i>
1.4. <i>LISPeK ebazten dituen problema-motak</i>	<i>9</i>
2. LENGOSAIAREN OINARRIZKO ELEMENTUAK.....	13
2.1. <i>Objektu atomikoak</i>	<i>13</i>
2.1.1. <i>Zenbakiak (Zenbakizko atomoak).....</i>	<i>13</i>
2.1.2. <i>Sinboloak (Atomo sinbolikoak).....</i>	<i>14</i>
2.1.3. <i>Karaktere-kateak</i>	<i>15</i>
2.2. <i>Objektu konposatuak: S-espresioa. Listak</i>	<i>16</i>
2.3. <i>Memoriako adierazpidea</i>	<i>18</i>
ARIKETAK.....	22
3. S-ESPRESIOEN EBALUAZIOA. LISP INTERPRETATZAILEA.	25
3.1. <i>Objektu atomikoen ebaluazioa</i>	<i>26</i>
3.2. <i>S-espresio konposatuen ebaluazioa</i>	<i>26</i>
3.3. <i>Forma bereziak</i>	<i>27</i>
4. FUNTZIO ETA PREDIKATU PRIMITIBOAK	31
4.1. <i>S-espresioak sortu eta erabiltzeko funtzioak.....</i>	<i>32</i>
4.1.1. <i>CONS oinarritzko funtzio eraikitzailea</i>	<i>32</i>
4.1.2. <i>FIRST eta REST. Oinarritzko funtzio</i> <i>deskonposatzaileak</i>	<i>33</i>
4.1.3. <i>Listak erabiltzeko funtzioak</i>	<i>34</i>

4.1.4. Funtzio suntsitzaileak	38
4.2. Zenbakizko funtzioak	41
4.2.1. Funtzio aritmetikoak	41
4.2.2. Funtzio traszendenteak	45
4.2.3. Funtzio trigonometrikoak	45
4.3. Karaktere-kateei buruzko funtzioak	45
4.4. Predikatu-funtzioak	47
4.4.1. Baldintza bakunak	48
4.4.1.1. Oinarrizko predikatuak	48
4.4.1.2. Zenbakizko predikatuak	53
4.4.2. Baldintza konposatuak	54
4.5. Hautapen-funtzioak	56
4.6. Ebaluazio-funtzioak	59
4.6.1. Ebaluazio-funtzioa. EVAL	59
4.6.2. Lista baten gaineko funtzio baten aplikazioa. APPLY	59
4.6.3. Funtzio-aplikazioa funtzioaren izena ebaluatuz. FUNCALL	60
ARIKETAK	62
 5. FUNTZIO BERRIEN DEFINIZIOA	67
5.1. DEFUN funtzioa	67
5.2. LAMBDA funtzioa	69
5.3. Parametro-motak	70
5.4. Funtzioak ala prozedurak?	73
ARIKETAK	74
 6. PROGRAMAZIO-FUNTZIONALA LISPEZ	77
6.1. Ezaugarriak	77
6.2. Egitura errekurtsiboak	79
6.3. Errekurtsibitate-motak	88
6.4. Errekurtsibitatea eta eraginkortasuna	89
6.5. Sasialdagaiak	90
ARIKETAK	94

7. PROGRAMAZIO AGINTZAILEA LISP-EZ	103
7.1. Aldagaiak	103
7.2. Asignazioa	109
7.3. Iteraziozko kontrol-egiturak	111
7.3.1. DO, DOTIMES eta DOLIST	111
7.3.2. LOOP eta PROG	115
ARIKETAK.....	117
8. PROGRAMAZIO-METODOLOGIA	119
8.1. Bilketa progresiboa	119
8.2. Problema-erredukzioa	121
8.3. Funtzio-abstrakzioa	121
8.4. Datu-abstrakzioa	123
8.5. Paketeak	129
8.6. Programazio-eskemak	132
8.6.1. MAPCAR (listen transformazioa)	134
8.6.2. Listen iragazketa, elementuen kontaketa eta bilaketa	135
8.7. Programazio onerako arauak	136
8.8. Errore araketa (Debugging)	140
ARIKETAK	144
9. INFORMAZIO-EGITURAK	149
9.1. Elkartze-listak	149
9.2. Propietate-listak	150
9.3. Arrayak.....	152
9.4. Egiturak	155
9.4.1. Egituren definizioa eta oinarrizko erabilpena.....	156
9.4.2. Egituren beste erabilpen batzu	159
9.4.3. Egitura moten garrantzia	161
ARIKETAK	162

10. SARRERA/IRTEERAKO FUNTZIOAK	167
10.1. <i>Sarrera/irteerako oinarrizko funtzioak.</i>	
READ eta PRINT	167
10.2. <i>Sarrera/irteerako erabilpen bitxiagoak</i>	<i>169</i>
10.2.1. FORMAT	169
10.2.2. WITH-OPEN-FILE	171
10.2.3. <i>Fitxategi-bukaerako tratamendua</i>	<i>175</i>
10.2.4. <i>Karaktere-kateen irakurketarako funtzioa</i>	<i>176</i>
ARIKETAK	179
 11. MAKROAK	181
11.1. <i>Makroen helburuak</i>	<i>181</i>
11.2. <i>Makroen definizioa eta ebaluazioa</i>	<i>182</i>
11.3. <i>Makroak definitzeko laguntza sintaktikoak komatxo</i>	<i>184</i>
ARIKETAK	186
 ARIKETEN EBAZPENAK	189
<i>Bigarren kapitulua</i>	<i>189</i>
<i>Laugarren kapitulua</i>	<i>196</i>
<i>Bostgarren kapitulua</i>	<i>201</i>
<i>Seigarren kapitulua</i>	<i>204</i>
<i>Zazpigarren kapitulua</i>	<i>214</i>
<i>Zortzigarren kapitulua</i>	<i>215</i>
<i>Bederatzigarren kapitulua</i>	<i>218</i>
<i>Hamargarren kapitulua</i>	<i>223</i>
<i>Hamaikagarren kapitulua</i>	<i>223</i>
 BIBLIOGRAFIA	229
 KONTZEPTUEN AURKIBIDEA	231

HITZAURREA

LISP Adimen Artifizialerako lengoaia nagusia da. Funtzioen definizioa eta aplikazioa dira beraren tresnak. Liburua lengoaiaren oinarritzko aurkezpena egiten du, baina ez da erreferentzi eskuliburu bat. LISP lengoaia ikasi nahi dutenentzat zuzenduta dago, bai bere kabuz egin nahi duen autodidaktarentzat, bai ikasgelan testu-liburu gisa erabiliko duenarentzat. Lengoiaren deskribapenaz gain ariketa ebatzi ugari ere aurkituko ditu irakurleak.

Testua sortzeko izan dugun iturburu eta erreferentzia nagusia [Winston, 89] izan da, bertatik hartu baititugu zenbait definizio eta ariketa. Hala ere, bost urteetan jasandako irakasle-eskarmentuaren ondorioz edukinerako egitura berri bat sortu dugu eta hainbat ariketa zein azalpen berri sartu ere.

LISPen oinarritzko azalpena dela eta, zenbait kontzeptu interesgarri baina aurreratu samarrak alde batera utzi ditugu. Hauen artean objektuei zuzendutako programazioaren kontzeptuak geratu dira (Classes, Generic Functions, Flavors, ...). Tresna hauek COMMON LISP Object System (CLOS) lengoaia hedatuan bildu ohi dira.

Sarrera hau bukatzeko, eskerrak eman nahi dizkiegu liburu hau gauzatzen lagundu digutenei. Beraz, eskerrak alde batetik Julián Gutierrez, Paki Lucio eta Blanka Cases irakasleei LISPi buruzko aholkuengatik, beste alde batetik Ana Fernandez, Javi San Juan eta Nekane Intxaurtza konputagailuaren aurrean testua idazten eta egokitzen eman duten denborarengatik, eta bukatzeko Iñaki Madariagari azalaren diseinuarengatik.

1. SARRERA.

1.1. SORRERA ETA BILAKAERA HISTORIKOA.

LISP programazio-lengoaia 1950.eko hamarkadaren bukaeran jaio zen, Estatu Batuetako oso ikertoki ospetsua den MITean (Massachussets Institute of Techonology). John McCarthy irakasleak sortu zuen gaur egun hilik dagoen IPL lengoaia oinarritzat hartuz. Lehenengo deskribapen formala "LISP 1.5. Programmer's Manual" liburuan azaldu zen 1962. urtean.

FORTRAN edo COBOL lengoaiak bezain zaharra da LISP, baina urteekin izan duen bilakaera oso bestelakoa izan da. 15 urte bete arte ez zuen arrakasta handirik lortu, ordurarte "kutsu teorikoegia" zuten giro unibertsitari eta sasimatematikotik ia ez zen atera. Baina 1975. urte inguruan gertatu zen Adimen Artifizialaren ez tandarekin bat eginik zabalkuntza ikaragarria lortu zuen.

Izena "Listen prozesamendua" (List Processing) terminoaren akronimoa da. Lengoiaren datu eta programen funtsezko datu-egiturari erreferentzia egiten dio.

Programatzailak lortu nahi duen emaitza definitu behar du. Betebehar horretan funtzio-definizioak, funtzio-konposaketa eta funtzio-aplikazioak erabiltzen ditu. Funtzio-aplikazioak burutzeko ordena ez da zehatz-mehatz definitu behar.

Adibidez: $y(x) = \frac{2x^2 - 3}{x - 25}$ funtzioaren LISPezko definizioa honako hau da:

```
(DEFUN Y (X)
  (/ (- (* 2 X X) 3) (- X 25)))
```

(Y 3) funtzio-aplikazioan, y(3) balioa lortuko duena, kalkulatzeko prozesuan ez du axolarik zein funtzio-aplikazio ebaluatuko den lehenago: (- (* 2 X X) 3) edo (- X 25). Arruntago diren PASCAL, BASIC edo COBOL lengoaia agintzaileetan berriz, nahitaezko ordena bat definitzen da ekintzak eta kalkuluak egikaritzeko.

Funtzio-aplikazioetan oinarritzen den diseinurako filosofia hau jarraituz, geroago programazio-lengoi familia oso bat sortu da. LISP, lengoaia agintzaileen ezaugarri nabarmenak badauzka ere, lengoaia funtzionalen aitzindaritzat hartu ohi da.

"LISP 1.5 Programmer's Manual" liburu hartan deskribatzen zen hasierako lengoaiak, urteak joan urteak etorri, ondoko hobekuntzak jaso ditu:

- 1) Memori hartze eta exekuzio-denboraren aldetik eraginkortasuna hobetzen duten implementazio-hobekuntzak.
- 2) Lengoiaren ahalmenaren zabalkuntza.
 - Datu-egitura konplexuak definitu, behatu eta inferentzi prozesuetan erabiltzeko funtzio-bildumak.
 - Sarrera/Irteerako eragiketen gestiorako sistema.
 - Sintaxia zabaltzeko laguntzak. Sarritan errepikatzen diren egitura sintaktikoen idazketa errazteko.
- 3) Programatze-lana arintzen eta errazten duten programazio-inguruneak.
 - Ingurune hauek ondoko tresnak edukitzen dituzte:
 - a) Editorea. Fitxategi bat baino gehiago aldiberean editatu ahal dute eta gehienetan lengoiaren sintaxiak berak gidatzen du editaketa.
 - b) Errore sintaktikoak bilatu eta arakatzeko laguntzeko sistema.
 - c) Interpretatzailea
 - d) Konpiladorea
 - Aldi berean lehen aipatutako tresna bat baino gehiago elkarrekin lanean iharduteko aukera. Adibidez, editorea eta interpretatzailea era integratu batez erabili ahal dira, hau da, biak aldi berean daude kargatuta memorian eta beraz editoretik interpretatzaileara (edo alderantziz) igarotzeko ez dago beharrik editorea deskargatzeko eta interpretatzailea kargatzeko.
 - Pantaila grafikoak. Prozesatzeko orduan iturri desberdinetako informazioak testu edo grafiko moduan erakutsi ahal dituzte. Aldiberean prozesu bat baino gehiago kontrolatu ahal da pantailan, adibidez edizioa eta interpretazioa

Era honetako INTERLISP eta MACLISP ingurune ahaltsuak 70etako hamarkadan sortu ziren. Gaur egunean, era horretako inguruneak eskuragarri daude edozein konputagailutarako, handi zein txikiatarako.

Hardware aldetik ere, lengoia implementatzeko hobekuntza iraultzaileak etorri dira. Azken hamarkada honetan "LISP makina" deritzon konputagailu-egitura berria sortu da. Ez dago oinarrituta Von Neuman-en arkitekturan, LISP funtzioen aplikazio eta definizioetarako bereziki asmatutako arkitektura berrietan baizik. Makina hauetan ezin daiteke egikaritu LISPen oinarrituta ez dagoen programarik (ez dira "asmo orokorreko" konputagailuak). Beste batzuen artean Symbolics, LISP Machine Company, Xerox eta

Maia (Baionako empresa) dira LISP makinak saltzen dituzten etxe komertzialak. Aurrerakuntza honi esker LISPen eraginkortasuna gorengo mailaraino heldu da.

1.2. LENGOAIAAREN EZAUGARRIAK.

1. EZAUGARRIA. LISP lengoia funtzionala da.

Programazio-lengoaien sailkapen orokorrenaren arauera hiru familia bereizten dira :

- Lengoia agintzaileak (edo prozeduralak)
- Lengoia funtzionalak (edo aplikatiboak)
- Lengoia erazagutzaileak

Lengoia agintzaileen ezaugarri nagusia hau da: programa baten exekuzioa sekuentzia ordenatu moduan agertzen diren ekintza edo tratamenduen pausoz-pausoko exekuzioa. Ekintza desberdinek batera erabili behar dituzten balioak aldagaien bitartez gordetzen dira. Aldagaiei balioak emateko asignazio-ekintza erabiltzen da. Ekintz sekuentziaren exekuzio-ordena aldatzeko edo errepikaketak adierazteko kontrol-ekintza bereziak definitzen dira. Datuen kontrola eta sekuentziaren kontrol hauek Von Neuman-en arkitekturaren ondorio gisa datoz. Lengoai mota honen diseinuan Von Neuman-en eredua hartu da eta ez besterik. Gaur egungo aplikazio informatiko gehienak lengoia agintzaileen bidez antolatzen dira. PASCAL, COBOL, FORTRAN eta BASIC familia honetan aurkitzen dira.

Lengoia funtzionalen programa batean bi atal bereizten dira: hasieran zenbait funtzioen definizioa, eta gero zenbait funtzio-aplikazio. Funtzio-aplikazioak argumentu konkretuekin funtzio bati egindako deiak dira, argumentuen balioak balio konstanteak edo beste funtzioen aplikazioaz lortutako emaitzak izan daitezkeelarik. Era honetako programazioaz ari garenean, egikaritzapena datuek gidatzen dutela esan ohi da, eta ez ekintzek, lengoia agintzaileetan gertatzen den bezala. Erreferentzien gardentasuna lortzen da funtzioen aplikazioetan albo-ondoriorik ez baita onartzen. Prozesu errepikakorrek errekurtsibitatearen bidez adierazi ohi dira. Lengoia hauek diseinatzeko ez da hartzen abiapuntutzat Von Neuman-en arkitektura, ezta beste arkitektura-motarik ere. Helburua problemak ebazteko tresna lagungarria diseinatzea da baina konputagailu-eredurik kontutan hartu gabe. Hurbilpen funtzionala oso egokia da xede hori lortzeko. Lengoia funtzionalen artean Hope, Miranda, ML, LISP eta FP aurkitzen ditugu.

Lengoaia erazagutzaileak Programazio Logikoan erabiltzen dira. Lehenengo mailako predikatuen logikan oinarritzen dira. Programa objektuen arteko erlazioak erazagutzen dituen erregela-multzoa da. Adierazitako ezagumenduaz galderak proposatzen ditu erabiltzaileak lehengo predikatuak eta objektuak erabiliz. Galdera baten exekuzioa zera ziurtatzean datza: ea galderako erlazioak bat datozen erregela-multzoan dagoen ezagumenduarekin. Programazio Logikoko lengoaia aipagarriena PROLOG da. Hona hemen adibidea:

<pre> seme (jon , patxi) → ; seme (jon , alberto) → ; alaba (koldo , itziar) → ; alaba (jon , arantza) → ; seme-alaba (X , Y) → seme (X , Y) ; seme-alaba (X , Y) → alaba (X , Y) ; </pre>		Erregela-multzoa
<pre> seme (jon , patxi) ; TRUE seme (jon , koldo) ; FALSE seme (jon , X) ; X = patxi X = alberto seme-alaba (jon , X) ; X = patxi X = alberto X = arantza </pre>	<pre> <-- <-- <-- <-- <-- </pre>	Galderak

LISP lengoaia funtzionala da, baina ez funtzional hutsa. Sortu zeneko urteetan lengoaia guztiak agintzaile motakoak zirenez, LISPeK ere zenbait ezaugarri agintzaile hartu zuen, asignazio-ekintza eta ekintz sekuentziak definitzeko zenbait egitura hain zuzen. Testu honetan programazio funtzionalerako sarrera bat egin nahi dugunez gero, LISPen alde funtzionala sakonago azalduko dugu.

Azken 10 urteetan programazio-lengoaia funtzional asko azaldu da. Egungo lengoaia funtzional horietan datu-mota eta funtzio-moten erabilpena sakonagoa da. Ebaluazio nagia ere erabili ohi dute, hau da, argumentu guztiak ez dira ebaluatzen emaitza kalkulatzeko hasi baino lehenago, argumentu bakoitza behar denean ebaluatzen da, eta

posiblea da funtzio baten aplikazioa bukatzea bere argumentu bat edo gehiago ebaluatu gabe.

2. EZAUGARRIA: Datuak eta programek adierazpide bera dute.
s-espresioak hain zuzen ere.

S-espresio kontzeptu berriak atomo eta listaren kontzeptuak biltzen ditu. Aurrerago azalduko da zehazki.

Datuak:

15

A

(Erregela I6

(BALDIN (ANIMALIAK DAUKA AGIN ZORROTZAK)

(ANIMALIAK DAUKA ATZAPARRAK)

(ANIMALIAK DAUKA AURRERANTZEKO BEGIAK)

(ORDUAN (ANIMALIA HARAGIZALEA DA)))

(UEU

(OBJEKTU-MOTA UNIBERTSITATEA)

(HELBIDEA ((KALEA GENERAL CONCHA

(ZBKIA 25))

(HERRIA DONOSTIA)))

(ZABALGUNEA (EUSKAL HERRIA))

(TELEFONOA 94 4317145)

(BURUA IÑAKI IRAZABALBEITIA))

(PRAKTIKAK II ASIGNATURA)

Programak (edo hobeto esanda: funtzio-aplikazioak):

A

15

(COND ((MEMBER X LISTA1) LISTA1)

((MEMBER X LISTA2) LISTA2)

(T (APPEND LISTA1 LISTA2)))

(FAKTORIALA (+ 2 A))

(SARTU-PILAN (FIRST LISTA1) LISTA2)

(Y 3)

3. EZAUGARRIA: Goimailako ordenako funtzioak erabili ahal ditu.

Hots, funtzio baten parametro gisa beste funtzio bat ageri ahal da. Hau posiblea da datuak eta programak adierazpide bera dutelako eta EVAL funtzio berezia erabili ahal delako. EVAL funtzioak edozein s-espresioaren ebaluazioa egiten du. Beraz, posiblea da LISP programa batek beste programa bat sortzea eta exekutatu gero. Erraztasun hau ez da batere normala programazio-lengoaiei artean, teorian ezinezko gertatzen ez bada ere, lan erraldi eta zaila suposatzen du eta. LISP lengoaiak (makina lengoaiak gertatzen den bezalaxe) erraz eta sinple izan daiteke programa baten itxura osoa duen lista bat eraikitzea, eta berau sortu duen programatik atera gabe geroxeago exekutatzea.

1.3. ALDEKO ETA KONTRAKO ERITZIAK. BALANTZEA.

Nahiz eta gaur egun, gaitututa egon LISPe urteetan zehar bere aurkako eritziak jaso ditu:

A.- LISP geldia da oro har eta eragiketa aritmetikoekin batik bat.

Hasieran horrelaxe gertatzen zen. Inplementazio berrietan interpretatzaile askoz arinagoak lortu ziren. LISPerako bereziki asmatutako arkitekturek zeharo baztertu dute eragozpen hau.

Adimen Artifizialean (LISPen aplikazio-eremu nagusietako batean) problemen anbiguitasuna oso handia denez, eraginkortasuna bigarren maila batera eramane daiteke, soluziobide bat aurkitu ahal bada behintzat.

B.- LISPe programak oso handiak dira.

Eragozpena da hau? Ala abantaila? Ez al da hau zailtasun handiko problemetarako soluzioak aurki daitezkeenaren frogak? Beste lengoia batzuekin azaldu ere ezin daitezke azaldu horrelako problemak.

C.- LISP irakurgaitza da eta erroreak arakatzeko prozesua neketsua da, parentesi nahaspilatsuengatik, datu-motarik ez dagoelako eta datuak programekin nahasten direlako. (LISP = List of Insipid and Stupid Parenthesis)

Paresien erabilpena eduki errazten da editore sintaktikoak erabiliz edo horrelako editoreen falta paresietarako idazkera metodikoa jarraituz.

Adibidea: Hona hemen funtzio bat modu irakurgaitz eta irakurterazaz idatzita:

```
( DEFUN FIBONACCI ( N ) ( COND (( ZEROP N ) (( EQUAL N 1 ) 1  
( T ( + ( FIBONACCI ( - N 1 ) ) ( FIBONACCI ( - N 2 ) ) ) ) ) ) ) )
```

```
( DEFUN FIBONACCI ( N )  
  ( COND (( ZEROP N ) 1 )  
          (( EQUAL N 1 ) 1 )  
          ( T ( + ( FIBONACCI ( - N 1 ) )  
                ( FIBONACCI ( - N 2 ) ) ) ) ) ) )
```

Interpretatzaile modernoek zenbait datu-mota egituratu onartzen dute.

D.- LISP ikasteko zaila da.

Lehenengo eskuliburuak ulergaitz gertatu ziren. Gaur egun testu-liburu asko dago eta batzu bikainak dira. Unibertsitate askotan LISP aukeratu dute programazioa irakasteko lehenengo lengoaia bezala. Gainera, LISPez ere erabili ahal dira programatzeko metodologiak.

E.- " LISP Batua" oraindik ez da sortu.

Dialekto ugari sortu da hasierako "LISP 1.5 Programmer's manual" eredutik abiatuta. Bakoitzean funtzio berezi berriak eranstzen ziren, beste lekutan egindako zabalkuntzak kontutan hartu barik.

Dialektoen arteko diferentzia nagusiak hauexek dira:

- Funtzio aurredefinitu gehiago edo gutxiago edukitzea edo/eta funtzio baliokideak identifikadore eta parametro-antolakuntza desberdinekin definitu behar izatea.
 - Sintaxi-aldakuntzak
- Adibidez:

(DEFUN XFUNTZIOA (X Y) ...)

(DE (XFUNTZIOA X Y) ...)

Identifikadore eta listak adierazteko erregela desberdinak. Adibidez:

a.- Zenbakiak ez diren atomoen luzera 16 baino txikiago izan ala ez.

b.- Zenbakiak ez diren atomoak digitoz hasteko aukera izan ala ez.

Dialektu ezagunenak honako hauek dira:

- EEBBetakoak: LISP 1.5 (1962), INTERLISP (1970),
MACLISP(1970), NIL (1980)
- Frantziakoa: LELISP (1982)
- Inglaterrakoa: XLISP (1980)
- Kataluniakoa: UPCLISP (1986)

Baina gaur egunean zalantzarik gabe esan daiteke LISP batua badagoela. COMMON LISP 1984. urtean sortu zen eta geroztik eredu standarda bihurtu da. Beraz, COMMON LISP da testu honetarako aukeratu duguna; are gehiago, aurrerantzean liburu honetan LISP izen hutsa erabiltzen dugunean eta kontrako oharrik ematen ez bada, COMMON LISPez aritzen garela suposatu beharko da.

LISPen aldeko eritzi nagusiak honako hauek izan dira:

A.- LISPen sintaxia eta semantika sinpleak dira eta ulertzeko errazak.

- Programa batean listak baino ez dira agertzen.
- Listak idazteko karaktere berezi bakarrak puntua, zurigunea eta parentesiak dira.
- Listek funtzio-aplikazioak edo funtzio-definizioak adierazten dituzte. Funtzio aplikazioa baldin bada funtzioaren izenak eta argumentuek osatzen dute lista. Funtzio-definizioa baldin bada funtzioaren izena, parametroak eta definizioaren gorputza agertzen dira.
(Parentesiaren erabilpenaren zailtasuna sintaxiaren sinpletasunaren ondorioa da)

B.- Informazio sinbolikoa (ez zenbakizkoa) tratatzeko aparteko gaitasuna.

Hitz-atomo bakoitzak bere esangura azaltzen duen balioa dauka. Balio beronek bestalde beste balio bat eduki lezake eta horrela segi liteke mugarik gabe.

C.- Implementazio erraza.

Bi arrazoiengatik: sintaxia eta semantika sinpleak direlako eta programak datutzat har daitezkeelako.

D.- Problemen analisi errekurtsiboa.

Gehienetan lengoaia agintzaileekin ez da posible eta posible denean orduan ez da berezko ebazpidea edo ez da oso eraginkorra.

Balantzea: Orain dela 20 urte LISPen kontra zeuden arrazoiak gaur egunean gaindituak izan dira. LISPen etorkizuna ezin hobea da, batez ere Adimen Artifizialaren eskutik

1.4. LISPeK EBAZTEN DITUEN PROBLEMA-MOTAK.

Oro har, LISP oso egokia da datuak egituraz eta tamainaz dinamikoki aldatzen direnean, edota datuen egitura edukina bezain garrantzitsua denean. LISPen aplikazioen eremu zabalenak ondokoak dira:

A.- Adimen Artifiziala.

Adimen artifiziala giza-adimena aztertzen duen eta jokaera adimentsuko sistemak eraikitzen dituen Informatikaren arloa da. Jokaera adimentsua diogunean hauetako ahalmen bat adierazi nahi dugu: Giza-hizkuntza bat ulertzea eta hitz egitea, argazki edo irudi batean objektuak edo pertsonak ezagutzea, helburu bat lortzeko plan bat asmatzea, inferentziak egitea, dedukzioak egiaztatzea, edo/eta dudazko egoeratan erabakitzea. Horrelako sistemetan erabili behar den jakintza ikaragarri handia da eta gainera mota askotakoa. Soluzioak aurkitzeko aukera posible asko aztertu behar da. Funtsezkoa da jakintza adierazteko bide egokia aukeratzea, bai egituraz aldakor diren datuak gordetzeko, bai beroriek modu eraginkorrez tratatzeko. Adimen artifizialaren barruan zenbait aplikazio konketuago bereizten da:

- Sistema Adituak.

Gai batean aditua den pertsona baten jakintza biltzen duten sistemak dira; baina ez da ezagumendu-pila hutsa baizik eta problema baten aurrean pertsona aditua bezain ondo erantzuteko gauza den sistema, bere erantzunaren arrazoiak eta azalpenak ematen dituelarik. Bestalde beti dago posibilitatea sistemaren jakintza erraz aldatu edo zabaltzeko. Sistema adituek azaltzen dituzten domeinuetan ez da esistitzen soluzioak aurkitzeko metodo sistematiko eta zehatzik, heuristiko, hurbilketa edota datu osagabeak erabili behar direlarik. Eritasun-diagnosian, bilaketa mineralogikoan, kimikako molekula egiturak aztertzen eta konputagailuekipoen konfigurazioan sistema adituak erabili dira arrakasta handiarekin.

- Lengoaia Naturalaren Prozesamendua.

Konputagailuekiko komunikazioa erraztea da aplikazio honen lehengo helburua. Konputagailu kontuan ezertxo ere ez dakitenez ere beren hizkuntzaz (euskara, txinoa, frantsesa, ...) erabili ahal ditzaten. Hala nola datu-baseetarako galdeketa-sistemak, sistema adituak erabiltzeko interfaceak etabar.

Beste alde batetik zenbait hizkuntz lan konplexu burutzen dituzten sistemak eraikitzen dira. Hala nola itzulpen automatikoa, testu-laburpenak edo informazio zehatzen bilaketa testu luzeetan.

- Ahotsaren analisisa eta sorkuntza.

Aurreko puntuko helburu berekin baina testu idatzietatik abiatu ordez, ahots-grabaketatik eginez.

- Konputagailuz Lagunduriko Irakaskuntza Inteligentea.

Konputagailu eta ikaslearen arteko komunikazioa atsegin gerta dadin konputagailuak jakin behar du zer dakien ikasleak eta horren arabera sortuko ditu ariketak eta azalpenak. Inork ez du azalpen luzeegirik nahi duda erraz bat argitzeko, ezta hitz bakarreko erantzuna ere galdera sakon bati erantzuteko.

- Konputagailu bidezko Ikusmena.

Irudi batetik ikuslearentzako deskribapen erabilkorra lortzea da helburua, balio gabeko informazioa baztertuz. Adibideak: Objektu-ezagupena, Kalitate-kontrola (zirkuitu inprimatuetan, altzairutan, beiratan, ...), Sateliteen irudien analisisa (mapak, meteorologia, ...), ebakitzeko eta soldatzeko makinaren kontrola, ...

B.- Sistemen programazioa.

LISP makinak LISPeZ daude programatuta goitik behera. Sistema eragilea, editoreak, konpilatzaileak, interpretatzaileak, programa lagungarriak,.... guztiak LISPeZ idatzita daude.

C.- Informatikaren irakaskuntza.

Unibertsitate askotan LISP da irakasten den lehenengo lengoaia, programazio funtzionala agintzailea baino naturalagoa eta errazagoa delakoan.

D.- Matematika sinbolikoa.

MACSYMA urte askotan oso sistema interesgarria izan da algebran.

2. LENGOAIAREN OINARRIZKO ELEMENTUAK.

2.1. OBJEKTU ATOMIKOAK.

LISPeko oinarri-oinarrizko elementua, beste objektuez zatiezina den elementua, "objektu atomikoa" da. Batzutan "atomo" izena erabiltzen da.

Atomo bat edozein luzeratakoa karaktere-sorta batez adierazten da. Karaktere-sorta horietan, LISPeke karaktere berezi ez den beste edozein karaktere sartu daiteke. Karaktere bereziak puntua ("."), zurigunea (" "), irekitzeko eta ixteko parentesiak ("(" eta ")"), komatxoa ("^"), apostrofoa ("'"), komatxoak ("\""), koma (",") eta puntu eta koma (";"). Letra minuskulak majuskulak bihurtuko ditu LISPeke interpretatzailea

Adibide gisa ondoren atomo batzuk aurkezten dira:

ABC

abc (abc eta ABC atomo bera adierazten dute)

17

ATOMO-HAU-LUZEA-DA

*A?->7

7A

47

3.141 (Kontuz hemen puntuaren erabilera berezia da!)

-1.E7 (Kontuz hemen puntuaren erabilera berezia da!)

Hiru atomo-mota desberdin daude:

Zenbakiak, sinboloak, eta karaktere-kateak.

2.1.1. Zenbakiak (Zenbakizko atomoak).

LISPek bi motatako zenbakiak erabiltzen ditu: osoak, errealak eta zatikiak.

Errealak koma higikorraz adierazten dira. Alde hamartar edo berretzaile duten zenbakiak errealak dira.

LISPen inplementazio desberdinek zenbakietarako zehaztasun desberdinak eskaintzen dituzte.

Adibideak :

4
2345
-546
+027
0.0
1.5
0.47
-0.3
+3.14159
6.03E23
1.E-9
+1.E7
10/3
5/6

COMMON LISPeK *zatiki* mota dauka, aurreko dialektuek ez

2.1.2. Sinboloak (*Atomo sinbolikoak*).

Atomo sinbolikoak, sinbolo izenarekin ere aipatuak, LISP programetako funtsezko elementuak dira. Sinbolo batek lau ezaugarri desberdin izan ahal ditu:

Inprimatzeko izena	("pname" edo "print name")
Atxekitako balioa	(binding)
Funtzio-definizioa	(function-definition)
Propietate-lista	(Property-list)

Inprimatzeko izena sinboloaren aipamena egiteko erabiltzen den karaktere-sorta da (batzutan "pname" izenarekin ezagutzen da). Sinboloaren aipamen idatzia da. LISP interpretatzaileak programa batetan atomo bat irakurtzen duenean, sinbolotzat hartuko du zenbakia edo karaktere-katea ez bada. Adibidea:

ABCD
17X

+25+

SINBOLO-LUZEA

A

Sinboloari balio bat atxekitzen zaio parametro gisa erabiltzen denean edo aldagai agintzaile moduan agertzen denean. Sinboloa ebaluatuz gero berari atxekitako balioa lortzen da.

Funtzioak ere sinboloen bidez aipatzen dira. Zenbait sinbolori funtzio-definizio bat lotzen zaio. Sinboloa funtzio-aipamen bezala erabiltzen denean, berari dagokion definizioa adierazten du.

Propietate-listak aurrerago (4. eta 9. kapitulan) azalduko dira. Oraingoz nahikoa da jakitea sinbolo bati beste balio batzu egokitu ahal zazkiola, bakoitza bere propietate-identifikadorearen bidez erabiliko delarik.

NIL eta T oso sinbolo bereziak dira. Beren balio atxekiak NIL eta T berberaiek dira. Ezin dute hartu propietate-listarik, ezta funtzio-definizioarik ere.

NIL sinboloa bi esangura diferentekin erabiltzen da: batetik, lista hutsa adierazten duena ("()" bezala ere adieraz daiteke), eta bestetik "faltsua" balio boolearra.

T sinboloa "egiazkoa" balio boolearra adierazteko balio du. Baina LISPeK eritzi orokorrago bat hartzen du "egiazkoa" adierazteko: NIL ez den beste edozein balio "egiazkoa"-tzat hartzen da balio bolear gisa erabiltzen bada.

2.1.3. *Karaktere-kateak.*

Sinboloak edo LISPeKo adierazpenak ez diren karaktere-sekuentziak adierazteko erabiltzen dira karaktere-katea motako atomoak. Komatxo-pareen artean idazten dira karaktere-kateak LISPeKo beste atomo eta adierazpenetatik bereizteko.

Adibideak:

"karaktere-katea bat"

"string = karaktere-katea"

"X"

Karaktere-kateak ez dira sinboloak. "X" karaktere-kateak ez dauka lotuta baliorik, propietate listarik edo funtzio-definizioarik. Kateak testu-zati hutsak dira. LISPeKo funtzioak aplikatuz katea batetik azpikatea bat atera ahal da edo bi katea bilduz beste berri

bat sortu ere bai. Kateetako minuzkulazko karaktereak errespetatu egiten ditu LISPeK, alegia, minuzkulaz idazten ditu .

2.2. OBJEKTU KONPOSATUAK: S-ESPRESIOAK. LISTAK.

LISP lengoaiaren datu-mota nagusia espresio sinbolikoa da. *S-espresio* aipamena *espresio sinboliko*-aren laburdura da. Errekurtsiboki definitzen da, ondoan azaltzen den bezala:

- Atomoak s-espresioak dira.
- a eta b s-espresioak baldin badira, orduan (a . b) bikotea ere s-espresioa da.

Adibideak:

ATOMO
 (A . A)
 (A . (A . A)
 ((JON . PATXI) . BEGOÑA)
 ((A . B) . (A . (B . C)))

LISPeKo datu edo programa guztiak s-espresioak dira. Baina normalean LISPeZ erabiltzen diren datuak listak dira. Adibidez: (A), (B A), (), (C BA), ((C) B A), ... Listek s-espresioen azpimultzo bat osatzen dute. Listak s-espresio bezala ikusi ahal izateko honako bi puntuok hartu behar dira kontutan:

- 1) NIL atomo bereziak lista hutsa adierazten du.
- 2) L objektua lista baldin bada, orduan, (X . L) bikotea ere lista da., alegia, (X . L) lista L-ren osagai guztiak dauzka eta gainera X osagaia buruan sartuta.

Adibideak:

()	edo	NIL	lista hutsa
(A)	≡	(A . NIL)	A osagai bakarreko lista
(B A)	≡	(B . (A . NIL))	bi osagaitako lista
(C B A)	≡	(C . (B . (A . NIL)))	hiru osagaitako lista
((C) B A)	≡	((C . NIL) . (B . (A . NIL)))	hiru osagaitako lista, baina lehenengo osagaia ere lista da.

Ikus daitekeenez, listako buruan osagai berri bat eransteko, nahikoa da bikote berri bat sortzea; bikoteko eskuinaldea lehengo lista eta ezkerraldea eransteko osagaia direlarik.

Listak edo s-espresioak bi modu desberdinetaz aurkeztu ahal dira: bikote-idazkeraz edo lista-idazkeraz. Bikote-idazkera "puntu-idazkera" bezala ere ezagutzen da. Adibidez, $(C B A)$ lista-idazkeraz dago eta s-espresio bera puntu-idazkeraz $(C . (B . (A . NIL)))$ da.

Gehienetan erabiltzen diren s-espresioak listak direnez, eta lista-idazkera irakurterrazagoa denez, lista-idazkera bikotekoa baino erabiliagoa izaten da.

Ikus ditzagun idazkera batetik bestera itzultzeko metodoak:

1.- S-espresio bat lista-idazkeratik bikote-idazkerara itzultzeko metodoa.

Definitu dezagun X s-espresio batetarako bere bikote-idazkera lortzen duen J funtzio bat.

Baldin X atomoa edo lista hutsa bada

orduan $J(X) = X$

bestela

baldin X osagai bakarreko lista bada

orduan $\{ X=(X1) \text{ izanik} \} J(X) = J((X1)) = (J(X1) . NIL)$

bestela $\{ X = (X1 X2 XN) \}$

$$\begin{aligned} J(X) &= J((X1 X2XN)) = \\ &= (J(X1) . J((X2 ...XN))) \end{aligned}$$

Adibideak:

$$(A B C) = (A . (B C)) = (A . (B . (C))) = (A . (B (C . NIL)))$$

$$(A (B C) D) = (A . ((B C) D)) = (A . ((B . (C . NIL)) . (D . NIL)))$$

$$((A)) = ((A) . NIL) = ((A . NIL) . NIL)$$

Edozein lista bikote-idazkeraz adierazi ahal da, listak s-espresioen kasu partikularrak baino ez direlako.

2.- S-espresio bat bikote-idazkeratik lista-idazkerara itzultzeko metodoa.

Beti ez da posible s-espresio bat lista-idazkera hutsera itzultzea. Bikote bateko eskuinaldeko osagaia NIL ez den beste atomo bat baldin bada, orduan ezin da azaldu lista

bezala bikote hori. Hala ere, nahiz eta lista-idazkera hutsezko azalpena ezina izan, posible diren pauso guztiak emanez lista-idazkeretik gertuen dagoen idazkera lortuko dugu.

Defini dezagun X s-espresio batetarako beste lista-idazkera lortzen duen g funtzio bat:

baldin X atomoa edo lista hutsa bada

orduan $g(X) = X$

bestela $\{ X = (X1 . X2)$ izanik, itzuli lista-idazkerara ezker eta eskuinaldea

Bitez $XZ = g(X1)$ eta $XS = g(X2)$

Beraz, X honela idatzi ahal da : $X = (XZ . XS)$ }

baldin XS atomoa da eta $XS \pi \text{NIL}$

orduan $\{X$ s-espresioa ezin daiteke lista-idazkera hutsez azaldu.)

$g(X) = (XZ . XS)$

bestela

baldin XS atomoa bada eta $XS = \text{NIL}$

orduan $g(X) = g((XZ . \text{NIL})) = (XZ)$

bestela

baldin XS lista bada

orduan $\{XS = (X1 X2 \dots XN)$ izanik}

$g(X) = g((XZ . (X1 X2 \dots XN))) = (XZ X1 X2 \dots XN)$

bestela $\{XS$ bikotea da }

$g(X) = g((XZ . (XS1 . XS2))) = (XZ XS1 . XS2)$

Adibideak:

$(A . (B . (C . \text{NIL})))$

$(A B C)$

$(A . ((B . (C . \text{NIL})) . (D . \text{NIL})))$

$(A (B C) D)$

2.3. MEMORIAKO ADIERAZPIDEA.

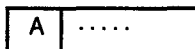
LISP-en datuak eta programak edozein luzera eta sakontasun-maila duten s-espresioak dira. Arbola bitarrak erabiltzen dira egitura aurrirakotezin hau adierazteko.

Objektu atomiko bakoitzari memoriako helbide bat egokitzen zaio bere errepresentazioa jartzeko. Objektu atomiko baten aipamen guztiak erakusle baten bidez adierazten dira, alegia, bere memoriako helbiderako erakuslea. Sinboloei dagozkien

informazioak (izena, balioa, propietate-lista eta funtzio-definizioa) memoriako helbide horretan gordetzen dira. Informazioa luzea baldin bada helbidean hasten da, baina memoriako gelazka gehiago ere bete ditzakeelarik.

Grafikoki ondoko erara adieraziko dugu:

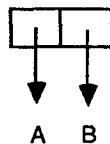
A atomoa:



Hau da, memoriako gelaska batean sinbolo baten izena, balioa, funtzio bat eta propietate-lista adieraziko ditugu. Hemendik aurrera, sinplifikatzeko, grafiko hauetan A soila erabiliko dugu.

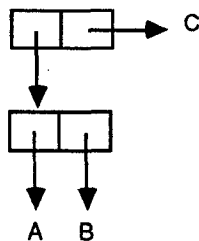
(A . B) bikotea memoriako gelazka baten bidez (edo biren bidez) adierazten da. Gelazkan bi erakusle daude, bikoteko osagai bioi erakusten dienak.

(A . B)



Oro har, s-espresio bat (atomo edo bikote) memoriako helbide baten bidez adierazten da. Helbide horretako gelazkan atomoari buruzko informazioa dago edo bikotea izatekotan partaide bien helbideetarako erakusleak.

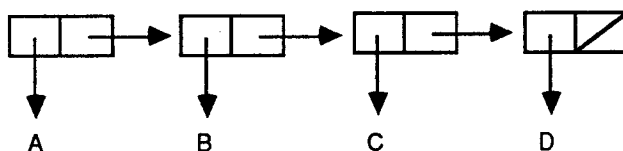
((A . B) . C)



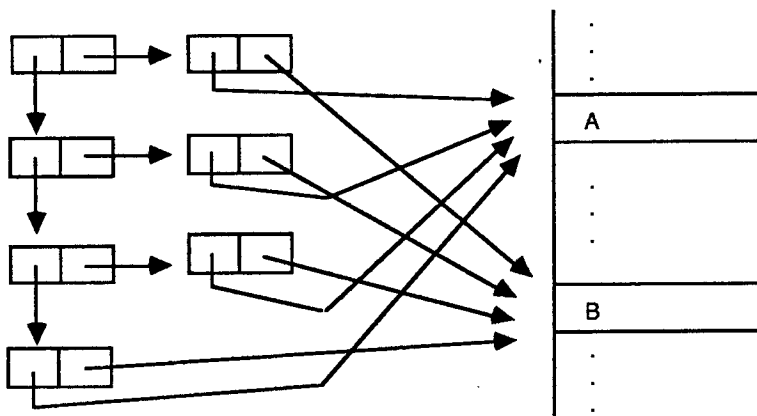
Lista hutsa (NIL) erakusle hutsaren bidez adieraten da:



S-espresio baten lista-idazkera jakinda erraz lortu ahal da bere memoriako adierazpidea:

$$(A\ B\ C\ D) \uparrow (A.(B.(C.(D.NIL))))$$


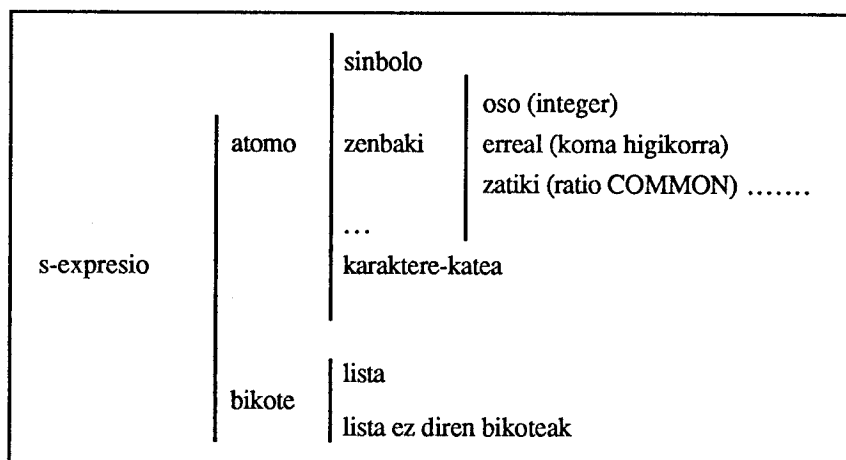
Atomo baten memoriako adierazpena bakarra da nahiz eta toki desberdinetan aipatua izan.

$$(((A.B).(A.B)).(A.B)).(A.B))$$


LISPeko interpretatzailea ondoko irudian azaltzen diren zatitan anatzen du memoria.

Sistema Eragilea
Interpretatzailea
Zenbakiak
Sinboloak
Karaktere-kateak
Listak (Erabilpen orokorrerako memoria)
Pilaren aldea
S. Eragileak erabiltzen duen aldea

LISP interpretatzaileak erabilpen orokorrerako memoria osoa lista estekatu bezala eratzen du. Espazio librearen lista estekatu horretatik memoriako gelazkak hartzen ditu beharrezko zaionean. Geroago espazio librea agortzeaz egotekotan, lehen erabili den baino aurrerago erabiliko ez den espazioa berreskuratzearen automatikoki pizten da *zabor-bilketa* deritzon prozesua (Garbage Collection). Prozesu honek memoriari bi pasada egiten dio. Lehenengoan aurrerantzean berriz erabiliko ez diren gelazkak markatzen ditu, eta bigarrenetan markatutako gelazkak espazio librearen lista estekatuari eransten dizkio.



2.1. Irudia.- LISPen oinarrizko datu-motak

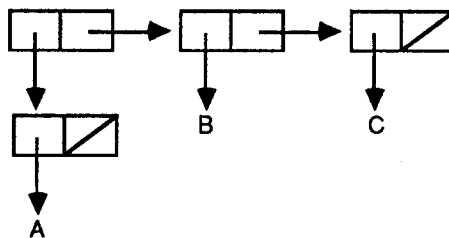
ARIKETAK

2.1. Puntu-idazkeraz adierazi ondoko espresioak eta memoriako errepresentazioa zehaztu.

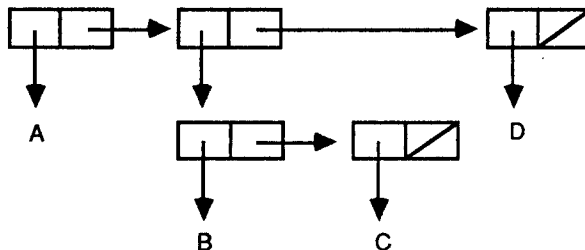
- a.- ((A) B C)
- b.- (A(B)C)
- c.- (A B(C))
- d.- ((A)(B)(C))
- e.- (((A B)))
- f.- ((A(B C))D(E.F))
- g.- (A.(B C D))
- h.- (((1 2)(3 4)).NIL)
- i.- ((A))
- j.- ((A B))
- k.- (((A(B))))
- l.- (A NIL (B(C D)))
- m.- (MEMBER X(QUOTE(C(B)(A.B)D)))
- n.- (ASSOC GILTZA(QUOTE((ADINA.25)(SEXUA.M))))

2.2. Zeintzu dira ondoko memoriako errepresentazioek adierazten dituzten s-espresioak? Aurkeztu itzazu puntu- eta lista-idazkeraz.

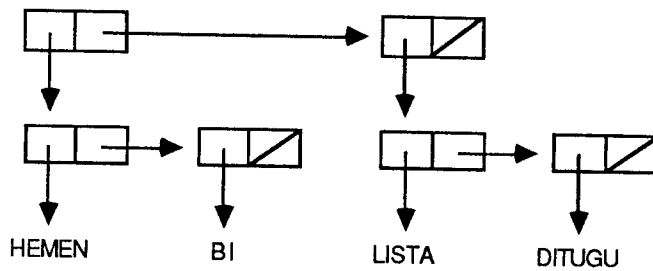
a.-



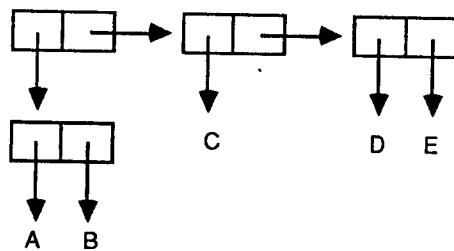
b.-



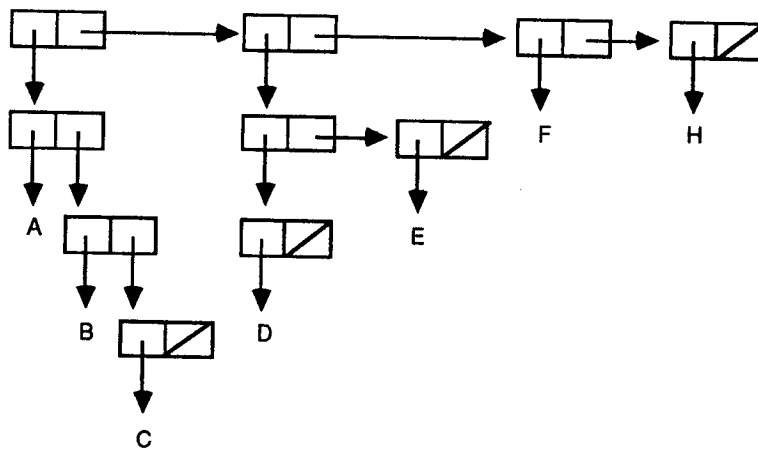
c.-



d.-



e.-



3. S-ESPRESIOEN EBALUAZIOA. LISP INTERPRETATZAILEA.

LISP programa bat s-espresio bat besterik ez da. Interpretatzailearen lana hiru pausotako ondoko bigizta etengabe exekutatzean datza:

- 1) S-espresio bat irakurri.
- 2) Ebaluatu s-espresioa.
- 3) Inprimatu ebaluazioaren emaitza.

S-espresio batzuk ebaluagarriak dira, beste batzuk ez. S-espresio ebaluagarriei *forma* deitzen zaie.

LISPe duen ebaluatzeko modua azaldu baino lehenago, azter dezagun matematikako espresio arrunt bat ebaluatzerakoan pertsona batek ematen dituen pausoak.

$\sin(n+\pi/4)$ espresioan sinbolo aldagai bakar bat dagoela dirudi, n sinboloa hain zuzen ere. Baina ikuspuntu sakonago batetik ikusita, espresioan agertzen ziren 8 sinbolo guztiok ebaluatu behar ditugu: $\sin$, $($, $)$, n , $+$, π , $/$ eta 4 . Kasu honetan n -rena ezik beste sinbolo guztien balioak errazak dira ezagutzeko, baina berauek ere ebaluatzen ditugu. " $\sin$ " sinboloak "zenbaki baten sinua" kalkulatzeko duen funtzioa adierazten du, "+" sinboloak "bi zenbakiren batura" kalkulatzeko duen funtzioa, "/" sinboloak "bi zenbakiren zatidura", " π "-k 3,141592653 zenbaki erreala, etabar.

$\sin(n+\pi/4)$ espresioa ebaluatzeko prozedura honi ekiten diogu: lehenengoz sinbolo guztiak ebaluatzen ditugu, eta gero, funtzio bezala ebaluatu direnak argumentu bezala ebaluatu direnei aplikatzen dizkiegu. Funtzio bakoitzaren argumentuak zeintzu diren jakiteko arau-mota desberdinak erabiltzen dira:

- 1) Sinboloarekiko posizioaren arauera.

"sinu" funtzioaren argumentua "sin" ikurrari darraio.

"batuketa" eta zatiketa funtzioen argumentuak "+" eta "/" ikurren aurreko eta atzeko posizioan dato.

- 2) Funtzioen lehentasun-mailaren arauera.

$n+\pi/4$ espresioan zatiketa batuketa baino lehenago kalkulatu behar da, zatiketa funtzioaren lehentasun-maila batuketarena baino handiagoa delako.

3) Parentesi edo kakoen arauera.

$$\sin(n+\pi/4) \neq \sin n+\pi/4$$

LISPen espresioen ebaluazioa erabat sinpleagoa da exekutatzeko. Exekutatzeko sinpletasun honen ondorio bezala zera aurkitzen dugu: espresioen idazkera luzeago eta metodikoago gertatzen dela.

3.1. OBJEKTU ATOMIKOEN EBALUAZIOA.

Ondoko erregelari darraio objektu atomiko baten ebaluazioa:

Zenbakia edo karaktere-katea baldin bada, bere emaitza balio bera izango da. Bestela (sinboloa da derrigorrez) bere ebaluazioaren emaitza sinboloak atxekirik duen balioa da. Sinboloak balio atxekirik ez badu errore bat azalduko da.

NIL atomo berezia ebaluatuz gero NIL lortzen da. T atomo berezia ebaluatuz gero T lortzen da.

3.2. S-ESPRESIO KONPOSATUEN EBALUAZIOA.

Listak diren s-espresioen ebaluazioa interesatzen zaigu. Lista ez diren s-espresioak ez dira ebaluatzen.

(A1 A2 ... An) listaren ebaluazioa ondoko modura kalkulatzen da:

- A1 osagaia funtzio bezala ebaluatu. Funtzio baten definizioa lortzen ez bada errore bat azalduko da.
- A2 ... An espresioak ebaluatu.
- A2 ... An espresioen ebaluazioaren emaitzei aplikatu A1 ebaluatuz lortu den funtzioa.

Adibidea: "+" sinboloak batuketa funtzioa adierazten duela eta A sinboloak 2 zenbaki osoa adierazten duela suposatuz, ebaluatu (+ A 3) forma.

1) + sinboloak batuketa funtzioa adierazten du.

2) A → 2

3 → 3

3) Batuketa aplikatzen badiegu 2 eta 3 zenbakiei ... 5 lortzen dugu.

Beraz: (+ A 3) → 5

Beste adibide bat: ebaluatu $(+ (+ A A) 3)$ forma

- 1) $+$ sinboloak batuketa funtzioa adierazten du.
- 2) $(+ A A)$ s-espresioa ebaluatzeko hiru pausook bete behar ditugu:
 - 2.1) $+$ sinboloak batuketa funtzioa adierazten du
 - 2.2) $A \rightarrow 2$ eta berriro $A \rightarrow 2$
 - 2.3) $(+ A A) \rightarrow 4$
- 3) Batuketa aplikatzen badiegu 3 eta 4 zenbakiei ... 7 lortzen dugu.

Beraz: $(+ (+ A A) 3) \rightarrow 7$

3.3. FORMA BEREZIAK.

Zenbait kasutan ez da ebaluatzen azaldu dugun bezalaxe. Ikus dezagun LISPeK prozedura hori konplexukiago definitzeko duen beharra:

$$f(x) = \begin{cases} = 0 & \text{baldin } x = 0 \\ = 1/x & \text{baldin } x \neq 0 \end{cases}$$

Funtzio honen definizioa baldintzapeko ebaluazio baten bidez egin behar da.

Demagun LISPeKo IF funtzio aurredefinitu batek horrelako aukera eskaintzen digula. IF funtzio honek hiru argumentu hartuko lituzke. Lehenengo argumentu boolearraren ebaluazioaren emaitza NIL izanez gero, IF funtzioaren aplikazioaren emaitza hirugarren argumentuaren ebaluazioaren emaitza litzateke, bestela bigarren argumentuarena.

Orduan, lehen proposatutako $f(x)$ funtzioa honela idatz liteke LISPez:

$(\text{IF } (= X 0) 0 (/ 1 X))$

Deskribatutako ebaluazio-eskema jarraituz, $f(3)$ aplikazioa kalkulatzeko:

IF $\rightarrow$ IF funtzioa
 $(= X 0)$ $\rightarrow$ NIL
 0 $\rightarrow$ 0
 $(/ 1 X)$ $\rightarrow$ 0.333333

Beraz: $(\text{IF } (= X 0) 0 (/ 1 X)) \rightarrow 0.333333$ lortu nahi genuen bezelaxe.

Baina zer gertatzen da $f(0)$ aplikazioa kalkulatzekoan:

IF	→ IF funtzioa
(= X 0)	→ egiazkoa
0	→ 0
(/ 1 X)	→ errorea (zeroz zatitzen saiatu baita)

Argi dago problema: Errorea agertzen da argumentu guztiak ebaluatu behar direlako. IF funtzioak bigarren eta hirugarren argumentuen artean bakar bat ebaluatu beharko luke lehenengo argumentuaren ebaluazioaren emaitza jakin ondoren. Hau da:

$$(\text{IF } X \ Y \ Z) \equiv \begin{cases} \equiv Y\text{-ren emaitza} & X\text{-en ebaluazioaren emaitza "egiazkoa" bada} \\ \equiv Z\text{-ren emaitza} & X\text{-en ebaluazioaren emaitza "faltsua" bada} \end{cases}$$

Funtzio honen ebaluazioa ez da sartzen lehen definitu den ebaluazio-eskeman. Honelako beharrak bete ahal izatearren ebaluazio-modu desberdinak onartzen dira. **forma berezi** izenekin ezagutzen diren funtzioen ebaluazio-modua desberdina da. Forma berezi bakoitzak bere ebaluazio-erregela dauka ebaluatu behar ez diren argumentuak zehaztatzeko.

Forma berezien arteko erabiliena "QUOTE" da. Beronek argumentu bakar bat hartzen du eta ebaluatu gabe itzultzen du argumentua bera emaitza bezala.

$$\begin{aligned} (\text{QUOTE } A) &\rightarrow A \\ (\text{QUOTE } (/ \ 1 \ 0)) &\rightarrow (/ \ 1 \ 0) \end{aligned}$$

Maiz agertzen denez idazkera erraztu bat ere erabil daiteke. Apostrofo karakterea erabiltzen da idazkera honetan:

$$\begin{aligned} (\text{QUOTE } A) &\equiv 'A \\ (\text{QUOTE } (/ \ 1 \ 0)) &\equiv '(/ \ 1 \ 0) \end{aligned}$$

Laburpen gisa : S forma ebaluagarri baten ebaluazioa honela definitzen da:

baldin S atomoa bada

orduan baldin S zenbakia bada

orduan zenbakia itzuli

bestela baldin S karaktere-katea bada

orduan karaktere-katea itzuli

bestela {S sinboloa da} S-k atxekita duen balioa itzuli

bestela {S lista da}

baldin S-ren lehenengo osagaia QUOTE bada

orduan itzuli S-ren bigarren osagaia ebaluatu gabe

bestela

baldin lehenengo osagaia ebaluatuz forma berezi bat lortzen bada

orduan forma bereziari dagokion arauera ebaluatu

bestela {S-ren lehenengo osagaia funtzio bat adierazten du}

 S-ren beste osagai guztiak ebaluatu;

 S-ren lehenengo osagaia ebaluatuz lortzen den funtzioa
 aplikatu beste osagaiak ebaluatuz lortu diren emaitzak
 argumentutzat hartuz.

1. oharra: aurrerago definituko diren makroak ez dira kontutan hartu.
2. oharra: Sinbolo bat funtzioa adierazteko tokian agertzen bada (hain zuzen ere, listako lehenengo osagaia) ebaluatzen denean ez da itzultzen bere balio atxekia, sinboloari egokitu zaion funtzio-definizioa baizik.

4. FUNTZIO ETA PREDIKATU PRIMITIBOAK

LISPeko programatzaileak problemak ebazteko asmatu dituen funtzioak definitu behar ditu. Baina ez du zerotik hasten, LISP interpretatzaileak funtzio aurredefinitu ugari eskaintzen baitio oinarritzko tratamenduak burutzeko. Kapitulu honetan funtzio aurredefinitu arruntenak deskribatzen dira. Dialekto desberdinetan funtzio aurredefinituen multzoak desberdinak izaten dira, baina kapitulu honetan azaltzen diren funtzioak dialekto guztietan erabili ahal dira, agian beste identifikadore batez edo agian beste sintaxi desberdin batez baliatuz. Guzti-guztiak COMMON LISPez erabili ahal dira.

Funtzioen erabilpena azaltzeko honako idazkera hau aukeratu dugu:

- Funtzioaren izena letra majuskulaz
- Argumentuak letra minuskulaz

Adibidez: (FIRST <lista>)

Argumentu guztiak deietan agertu behar diren ala ez zehazteko "&optional" eta "&rest" hitz gakoak erabiltzen dira. Adibideen bidez azalduko dugu beren erabilpena:

(FUN <a> <c>)

FUN funtzioak hiru argumentu behar ditu.

(FUN <a> &optional <c>)

FUN funtzioa gutxienez bi argumentuekin erabiliko da "a" eta "b". Hirugarrena borondatezkoa da.

(FUN X Y) eta (FUN X Y Z) espresioak zilegiak dira.

(FUN &optional <a> <c>)

FUN funtzioa zero, bat, bi edo hiru argumentuekin erabil daiteke.

(FUN), (FUN 1), (FUN 1 2) eta (FUN 1 2 3) espresioak zilegiak dira

(FUN <a> &rest)

FUN funtzioak derrigorrez argumentu bat behar du ("a"), baina horrez gain nahi beste erabili ahal da. Adibidez:

(FUN 1 2 3 4 5) edo (FUN 1)

(FUN <a> &optional <c> <d> &rest <e>)

Bi argumentu derrigorrezkoak dira, hurrengo biak ("c" eta "d") borondatezkoak, baina gehiago ere etor liteke.

Ondoko espresioak zilegiak dira: (FUN 1 2), (FUN 1 2 3),

(FUN 1 2 3 4) eta (FUN 1 2 3 4 5 6 7)

(FUN 1) ez da zilegia, errorea gertatzen da argumentu gutxiegi dagoelako

"&optional" eta "&rest" hitz gakoak antzeko modura erabiliko dira funtzio berriak definitzeko, baina kontutan hartu behar da kontzeptu desberdinak direla.

4.1. S-ESPRESIOAK SORTU ETA ERABILTZEKO FUNTZIOAK.

4.1.1. CONS oinarrizko funtzio eraikitzailea.

(CONS <espr> <lista>)

Forma honek espresioa listako buruan erantsiz lortzen den lista berria itzuliko du.

A → GAUR

B → (ASTELEHENA DA)

C → (EDO BASORA GOAZ)

(CONS A B) → (GAUR ASTELEHENA DA)

(CONS A NIL) → (GAUR)

(CONS B NIL) → ((ASTELEHENA DA))

(CONS B C) → ((ASTELEHENA DA) EDO BASORA GOAZ)

CONS funtzioa zenbait aldiz aplikatuz gero edozein lista eraiki daiteke .

Oharra: CONS funtzioaren definizio orokorraren arauera (*CONS espr1 espr2*) espresioa ebaluatuz (*espr1 . espr2*) bikotea lortzen da. Adibidez:

A → GAUR

B → 25

(CONS A B) → (GAUR . 25)

4.1.2. FIRST eta REST. Oinarrizko funtzio deskonposatzaileak.

Bi funtzio hauek lista bat hartzen dute argumentutzat .

(FIRST lista)

listaren lehengo osagaia itzultzen du.

(REST lista)

Lehengo osagaia kenduz lortzen den listaren hondarra itzultzen du.

Adibideak:

$L \rightarrow (A\ B\ C)$

$X \rightarrow ((A)\ B\ C)$

$(FIRST\ L) \rightarrow A$

$(FIRST\ X) \rightarrow (A)$

$(REST\ L) \rightarrow (B\ C)$

$(REST\ X) \rightarrow (B\ C)$

$(FIRST\ (FIRST\ X)) \rightarrow A$

$(REST\ (FIRST\ X)) \rightarrow NIL$

FIRST eta REST funtzioak NIL balioari aplikatuz gero, NIL emaitza lortzen da.

$(FIRST\ NIL) \rightarrow NIL$

$(REST\ NIL) \rightarrow NIL$

Funtzio bi hauek maiz erabiltzen dira. CONS funtzioa edozein lista eraikitzeke erabil daiteke. FIRST eta REST funtzioak berriz, lista baten edozein atomo edo azpilista lortzeko balio dute. Adibidez, $(A\ B\ (C\ D))$ listatik B, D eta $(C\ D)$ espresioak eskuratzeko honako espresiook ebaluatu beharko ditugu:

$X \rightarrow (A\ B\ (C\ D))$

$(FIRST\ (REST\ X)) \rightarrow B$

$(FIRST\ (REST\ (REST\ X))) \rightarrow (C\ D)$

$(FIRST\ (REST\ (FIRST\ (REST\ (REST\ X)))) \rightarrow D$

"FIRST" eta "REST" funtzioak klasikoki "CAR" eta "CDR" identifikadoreekin erabili izan dira. Baina "CAR" eta "CDR" identifikadoreak oso arraroak dira, ez dute

inolako erlazorik beren funtzioek lortzen duten emaitzarekin. Akats honen arrazoia historikoa da: lehengo implementazioetan CAR funtzioa "Contest of Adress Register" zen eta CDR funtzioa "Context of Decrement Register". Geroko implementazioetan izen horiek ez ziren egokiak, baina "CAR" eta "CDR" laburpenen erabilpena guztiz finkatuta zegoen ordurako. Azken aldi honetan "FIRST" eta "REST" identifikadoreen erabilpena bultzatzen da. CAR eta CDR funtzioen aplikazioak elkarren segidan datozenean modu trinkoaz idatzi ahal dira. Adibidez:

$$\begin{aligned}(\text{CADR } X) &\equiv (\text{CAR } (\text{CDR } X)) \\(\text{CADDR } X) &\equiv (\text{CAR } (\text{CDR } (\text{CDR } X)))\end{aligned}$$

Dena dela, irakurgarritasun falta dela eta, ez da oso komenigarria funtzio hauen erabilpena. Hobe da erabiltzea geroago azalduko diren SECOND, THIRD, ..., TENTH, NTH edo NTHCDR funtzioak.

Oharra: FIRST eta REST funtzioen definizio orokorra bikote motako argumentuetarako egiten da. FIRSTek bikotearen ezkerraldea itzultzen du eta RESTek eskuinaldea.

$$\begin{aligned}A &\rightarrow (X . Y) \\(\text{FIRST } A) &\rightarrow X \\(\text{REST } Y) &\rightarrow Y\end{aligned}$$

Lehenago emandako definizioa azken definizio honen kasu partikularra da, baina gehienetan listak erabiliko ditugunez, eta ez bikoterik, definizio hura argiago gertatzen da.

4.1.3. Listak erabiltzeko funtzioak.

CONS, FIRST eta REST erabiliz edozein lista eraiki ahal da, baina neketsu samarra izan daiteke. LISPen ba dira listekin jolasteko funtzio erosoagoak. Adibidez, bi listaren elementu guztiak hirugarren lista bat osatzeko bildu nahi baditugu, CONS funtzioa erabiltzea bururatzen zaigu, baina emaitza ez da egokia:

$$\begin{aligned}A &\rightarrow (\text{BAT BI HIRU}) \\B &\rightarrow (\text{LAU BOST}) \\(\text{CONS } A B) &\rightarrow ((\text{BAT BI HIRU}) \text{LAU BOST})\end{aligned}$$

CONS funtzioaren erabilpen honek ez digu ematen (BAT BI HIRU LAU BOST) bilatzen dugun lista. Beraz banan banan sartu beharko ditugu B listan Aren elementu guztiak.

```
( FIRST A ) → BAT
( SECOND A ) → BI
( THIRD A ) → HIRU
(CONS (FIRST A )
      (CONS (SECOND A )
            (CONS (THIRD A )
                  B )
             )
      )
→ ( BAT BI HIRU LAU BOST )
```

Ondoan azaltzen den APPEND funtzioa erabiliz gero askoz errazago lortzen da emaitza hori.

```
( APPEND &rest <listak>)
```

Nahi beste lista argumentu onartzen du.

Listen kateadura itzultzen du .

```
A → ( BAT BI HIRU )
B → ( LAU BOST SEI )
( APPEND A B ) → ( BAT BI HIRU LAU BOST SEI )
( APPEND '( A ) ( B ) ) '((C) (D))) → ((A) (B) (C) (D))
( APPEND A B '(ZAZPI ZORTZI))
→ ( BAT BI HIRU LAU BOST SEI ZAZPI ZORTZI)
```

```
( LIST &rest <argumentuak>)
```

Nahi beste argumentu onartzen du.

Argumentuak osagaitzat hartuz, lista bat eraikitzen du.

```
A → (BAT BI )
B → HIRU
C → (LAU BOST )
( LIST A B C ) → ((BAT BI) HIRU (LAU BOST ))
( LIST '(A) (B) ) '((C) (D))) → (((A) (B)) ((C) (D)))
```

Oharra: ikus CONS, APPEND eta LIST funtzioen arteko diferentzia.

```
A → ( X )
B → ( Y )
( CONS A B )      → ((X) Y )
( APPEND A B )    → (X Y )
( LIST A B )      → ((X) (Y))
```

(SECOND <lista>)

(THIRD <lista>)

...

(TENTH <lista>)

Listako bigarren, hirugarren, ..., hamargarren osagaia itzultzen dute. CADR, CADDR,....., CADDDDDDDDDR funtzioen baliokideak dira, baina irakurgarriagoak dira.

(NTH <n> <lista>)

Listako n-garren osagaia itzultzen du, lehenengo osagaia "zerogarrena" delarik.

```
A → (BAT BI HIRU LAU )
( NTH 3 A ) → LAU
( NTH 0 A ) → BA
```

(NTHCDR <n> <lista>)

Listari REST funtzioa n aldiz aplikatuz lortzen duen lista itzultzen du. Beraz, itzultzen duena argumentu bezala emandako lista da baina lehenengo n elementuak kenduta. Lehenengo argumentuaren balioa listaren luzera baino handiagoa bada emaitza NIL izango da. Adibidez:

```
A → ( BAT BI HIRU LAU )
( NTHCDR 0 A ) → ( BAT BI HIRU LAU )
( NTHCDR 3 A ) → ( LAU )
( NTHCDR 30 A ) → ( LAU )
```

(BUTLAST <lista> <n>)

NTHCDR funtzioaren antzekoa da, baina elementuak bukaeratik kentzen ditu. Argumentuen ordena alderantzizkoa da: lehenengoa lista da eta bigarrena zenbakia. Beraz, itzultzen duena argumentu bezala emadako lista da baina azken n elementuak kenduta. Bigarren argumenturik ez badago, balio lehenetsizat 1 hartuko da. Adibideak:

A → (BAT BI HIRU LAU)
(BUTLAST A 0) → (BAT BI HIRU LAU)
(BUTLAST A 3) → (BAT)
(BUTLAST A) → (BAT BI HIRU)

BUTLAST funtzioa NTHCDR baino askoz motelagoa da exekuzio-denboraz.

(LENGTH <lista>)

Listaren osagai-kopurua itzultzen du. Adibideak:

A → (A (B C) D E)
B → (BAT (BI (HIRU)))
(LENGTH A) → 4
(LENGTH B) → 2
(LENGTH NIL) → 0

(LAST <lista>)

Listako azken elementua itzultzen du baina lista baten osagai bakarra bezala.

A → (W X Y Z)
(LAST A) → (Z)

Orokorrean, argumentua s-espresio konposatua baldin bada, bere azken bikotea itzultzen du.

B → (A . (B . C))
LAST B) → (B . C)

(REVERSE <lista>)

Beste lista bat itzultzen du, alegia, argumentu bezala emandako listaren osagaiak dituen baina alderantzizko ordenan .

$A \rightarrow ((A\ B)\ (C\ D)\ E)$
 $(\text{REVERSE } A) \rightarrow (E\ (C\ D)\ (A\ B))$
 kontuz!! ez da $(E\ (D\ C)\ (B\ A))$!!

4.1.4. Funtzio suntsitzaileak.

Orain arte deskribatu diren funtzioen artean bat ere ez zen suntsitzaile. Funtzio suntsitzaile batek bere argumenturen baten balioa aldarazten du. Suntsitzaile ez diren funtzioek argumentuen balioak behatzen dituzte, baina ez aldarazi. Funtzio suntsitzaileen erabilpena ez da komenigarria. Oso konplexu eta luze den balio batetan aldakuntza txiki bat egin behar denean erabiltzen dira memoriako espazioaren eskasiagatik edo eraginkortasun-arrazoiengatik. Hala ere, zehatz-mehatz ezagutu behar da datuen memoriako adierazpidea honelako tresnak errore-arriskurik gabe erabili ahal izateko. Ondoren azaltzen diren funtzioak suntsitu egiten dituzte beren argumentuen balioak.

(RPLACA <lista> <espresio>)

Listaren lehenengo osagaia espresioaren balioaz ordezkutzen du.
 Lista berria itzultzen du.

$A \rightarrow (\text{BAT BI } (\text{HIRU}))$
 $(\text{RPLACA } A\ \text{'BOST}) \rightarrow (\text{BOST BI } (\text{HIRU}))$
 $A \rightarrow (\text{BOST BI } (\text{HIRU}))$

(RPLACD <lista> <espresio>)

Listaren hondarra espresioaren balioaz ordezkutzen du.

$A \rightarrow (A\ B\ C)$
 $(\text{RPLACD } A\ \text{'D}) \rightarrow (A\ .\ D)$
 $A \rightarrow (A\ .\ D)$
 $(\text{RPLACD } A\ \text{'(D)}) \rightarrow (A\ D)$
 $A \rightarrow (A\ D)$

(NCONC &rest <listak>)

NCONC funtzioak APPEND funtzioaren emaitza bera itzultzen du, baina ez du sortzen lista berri bat emaitza eraikitzeko, fisikoki egiten baitu.

$X \rightarrow (A\ B\ C)$

$Y \rightarrow (D\ E\ F)$

$(APPEND\ X\ Y) \rightarrow (A\ B\ C\ D\ E\ F)$

$X \rightarrow (A\ B\ C)$

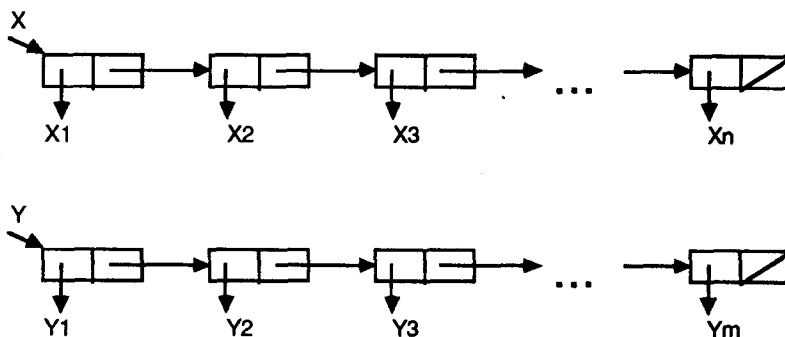
$Y \rightarrow (D\ E\ F)$

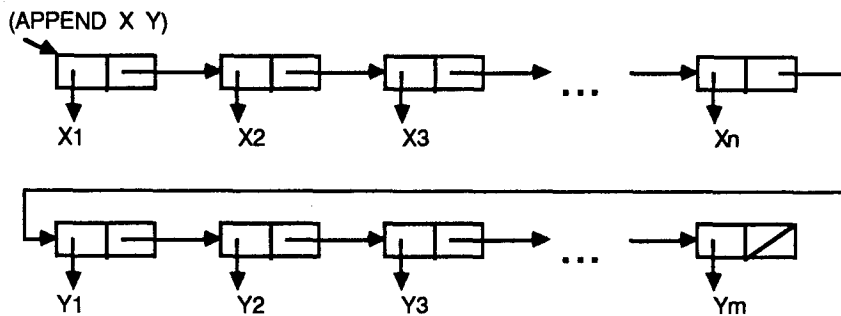
$(NCONC\ X\ Y) \rightarrow (A\ B\ C\ D\ E\ F)$

$X \rightarrow (A\ B\ C\ D\ E\ F)$

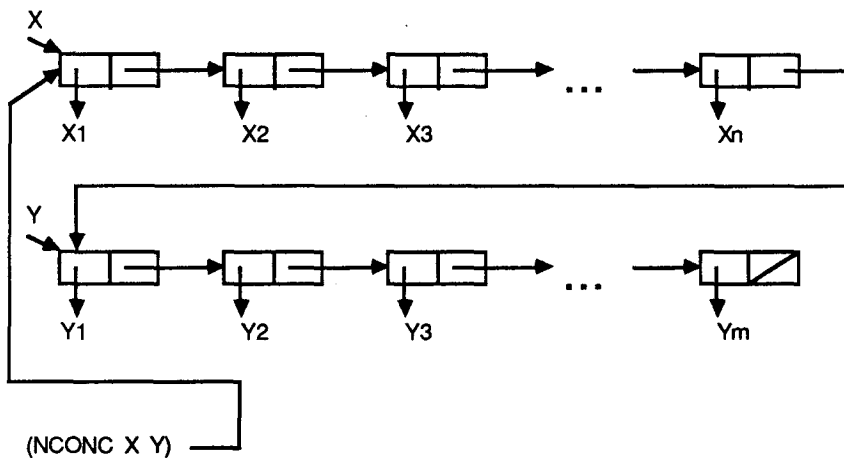
$Y \rightarrow (D\ E\ F)$

(APPEND X Y) forma ebaluatzen denean X eta Y listak kopiatu egiten dira emaitza eraikitzeko:





(NCONC X Y) ebaluatzen denean ez da kopiarik egiten, beraz X-en balioa aldarazi egiten da:



SETF funtzioa 7. eta 9. kapitulutan azalduko dugu. Bere erabilera arruntetan ez du eragin suntsitzailerik sortzen, baina horrelakoak ere lortu ahal dira bere lehenengo argumentua espresio konposatu baten osagai bat lortzeko forma bat denean. Kontu handiz ibili beharko da horrelako kasuetan. Adibidez:

$A \rightarrow (\text{BAT BI HIRU})$

$(\text{SETF (THIRD A) 'ZAZPI}) \rightarrow (\text{BAT BI ZAZPI})$

$A \rightarrow (\text{BAT BI ZAZPI})$

4.2. ZENBAKIZKO FUNTZIOAK.

Atal honetako funtzioen argumentuak zenbakizkoak izan behar dira. Horrela ez bada, erroreak agertuko dira.

4.2.1. Funtzio aritmetikoak.

(+ &rest <argk>)

Argumentuen batura itzultzen du. Argumenturik ez badago zero itzuliko du.
Adibideak:

(+ 3 4) → 7

(+ 3 4 8) → 15

(+) → 0

(+ 5) → 5

(- <arg1> &rest <argk>)

Lehenengo argumentua ken beste guztiak itzultzen du. Adibideak:

(- 3 4) → -1

(- 3 4 8) → -9

(-) → errorea , argumentu gutxiegi.

(- 5) → -5

(- -5) → 5

(ABS <x>)

x-en balio absolutoa. Adibideak:

(ABS 3) → 3

(ABS -3) → 3

(ABS -3.2) → 3.2

(MAX <zenb> &rest <zenbakiak>)

Argumentuen balioen arteko handiena itzultzen du.

(MAX 1 3 2) → 3

(MAX 3) → 3

(MIN <zenb> &rest <zenbakiak>)

Argumentuen balioen arteko txikiena itzultzen du.

(MIN 1 3 2) → 1

(1+ <x>)

x-en balioa gehi bat itzultzen du. (+ x 1) eta (1+ x) formak baliokideak dira

(1- <oso>)

oso-ren balioa ken bat itzultzen du. (- x 1) eta (1- x) formak baliokideak dira.

(* &rest <argk>)

Argumentuen balioen biderkadura itzultzen du.

Argumentu bat ere ez badago 1 itzultzen du.

(* 3 4 5) → 60

(*) → 1

(* 4) → 4

(GCD <oso1> <oso2>)

oso1 eta oso2-ren arteko zatitzaile komunetako handiena itzultzen du.

(GCD 12 16) → 4

Zenbait funtzioek zenbaki errealak eta zenbaki osoak bereizten dituzte, beren memoriako adierazpideak desberdinak dira eta. Mota batetik bestera itzultzeko honako funtzioak erabiltzen dira:

(FLOAT <oso>)

Zenbaki osoa erreal moduan adierazita itzultzen du.

(FLOAT 3) → 3.0

(ROUND <zenb>)

Zenbakitik gertuen dagoen zenbaki osoa itzultzen du.

(ROUND 2.9) → 3

Bigarren emaitza bat ere itzultzen du, alegia, zenbaki bien arteko distantzia. Baina oraingoz ez dugu erabiliko. Benetan ematen duen emaitza hau da:

(ROUND 2.9) → 3 ; -0.1

(TRUNCATE <zenb>)

Zenbakiaren digitu hamarrenak moztuz lortzen den zenbaki osoa itzultzen du.

(TRUNCATE 2.9) → 2

Bigarren emaitza bat ere itzultzen du, alegia, zenbaki bien arteko distantzia. Baina oraingoz ez dugu erabiliko. Benetan ematen duen emaitza hau da:

(TRUNCATE 2.9) → 2 ; 0.9

(TRUNCATE <arg1> <arg2>)

arg1 eta arg2 osoak izanik, beren zatidura osoa itzultzen du.

(TRUNCATE 14 4) → 3

Bigarren emaitza bat ere itzultzen du, alegia, zatiketa osoaren hondarra.
Baina oraingoz ez dugu erabiliko. Benetan ematen duen emaitza hau da:

(TRUNCATE 14 4) → 3 ; 2

Zatiketarako funtzioak:

(REM <arg1> <arg2>)

<arg1> eta <arg2> osoak izanik, zatiketa osoaren hondarra itzultzen du.

(REM 14 4) → 2

(/ <arg1> <arg2>)

Zatidura itzultzen du.

Argumentu bat erreala bada, emaitza koma higikorrez ageriko da.

Argumentu biak osoak badira eta zatidura ere zenbaki osoa bada, emaitza ere zenbaki osoa izango da, bestela emaitza zatiki bezala ageriko da.

(/ 5.0 2) → 2.5

(/ 5 2.0) → 2.5

(/ (FLOAT 5) 2) → 2.5

(* 6 (/ 5 2)) → 15

(* 5 10/3) → 50/3

Zatitzaile bat baino gehiago erabili ahal da :

(/ <arg1> &rest <argk>)

Lehenengoa zati beste argumentuak itzultzen du. Argumenturen bat koma higikorrez adierazten bada, koma higikorrez itzuliko du zatidura arrunta. Argumentu guztiak zenbaki osoak badira zatidura osoa itzulido du.

/ 27 9 3) → 1

(/ 27 10 3) → 9/10

(/ 27.0 10 3) → 0.9

4.2.2. Funtzio traszendenteak

Koma higikorrez erabiltzen dira. Argumentu osoak hartzen dituztenean koma higikorrezko zenbakia lortzen dute eragiketa burutu baino lehen.

(EXPT <berrekizun> <berretzaile>)

Zenbakien berredura itzultzen du.

(EXPT 2 3) $\rightarrow$ 8

(EXPT 2 3.0) $\rightarrow$ 8.0

(SQRT <zenb>)

Zenbakiaren erro karratua itzultzen du. Eraitza zenbaki erreala da.

(SQRT 4) $\rightarrow$ 2.0

(EXP <zenb>)

Funtzio esponentziala kalkulatzen du: ezbkia

(EXP 1) $\rightarrow$ 2.718282

4.2.3. Funtzio trigonometrikoak.

(SIN <X>) radianez adierazitako x-en sinua.

(SIN 3.14159) $\rightarrow$ 2.538186e-6 (hau da: ia ia zero)

(COS <X>) radianez adierazitako x-en cosinua.

(TAN <X>) radianez adierazitako x-en tangentea.

4.3. KARAKTERE-KATEEI BURUZKO FUNTZIOAK.

Lista eta karaktere-katea datu-motak elementu ordenatuzko multzoak adierazteko erabiltzen dira. Listetarako azaldu ditugun funtzio batzu karaktere-kateekin ere erabili ahal dira. Adibidez, azter ditzagun LENGHT eta REVERSE funtzioak:


```
(LENGTH '(A B C)) → 3
(LENGTH "ABC") → 3
(REVERSE '(A B C)) → (C B A)
(REVERSE "ABC") → "CBA"
```

ELT funtzioak ere (*element* hitzaren akronimoa) karaktere-katea eta listekin lan egin dezake. ELT forma batek bere lehen argumentua lista bat eta bigarrena N zenbaki naturala direnean listaren (N-1)garren elementua itzultzen du, zerok lehenengo elementua adierazten duelarik:

```
(ELT '(A B C) 0) → A
(ELT '(A B C) 2) → C
```

Bestalde, lehen argumentua karaktere-katea denean karaktere-kateako (N-1)garren karakterea itzultzen du. Karaktere motako objektuak adierazteko, aurretik #\ ikurrak jartzen ditu LISPe sinboloetatik bereiztearren:

```
(ELT "ABC" 0) → #A
(ELT "ABC" 2) → #C
```

Baina badaude karaktere-katea motarekin bakarrik lan egiten duten funtzioak ere. Horien artean STRING= eta STRING-EQUAL ditugu, bi karaktere-katea ea berdinak diren aztertzeko erabiltzen direnak. Lehenengoak majuskula eta minuskulak bereizten ditu, baina bigarrenak ez. Adibideak:

```
(STRING= "abc" "xyz") → NIL
(STRING= "abc" "abc") → T
(STRING= "abc" "ABC") → NIL
(STRING-EQUAL "abc" "xyz") → NIL
(STRING-EQUAL "abc" "abc") → T
(STRING-EQUAL "abc" "ABC") → T
```

Bi karaktere ea berdinak diren aztertzeko, CHAR= eta CHAR-EQUAL funtzio primitiboak dauzkagu. Lehenengoak majuskula eta minuskulak bereizten ditu, baina bigarrenak ez. Adibideak:

```

(CHAR= #A #A)    →    T
(CHAR= #A #B)    →    NIL
(CHAR= #a #A)    →    NIL
(CHAR-EQUAL #A #A) →    T
(CHAR-EQUAL #A #B) →    NIL
(CHAR-EQUAL #a #A) →    T

```

Karaktere-katea bat ea beste baten barruan dagoen aztertzeko SEARCH funtzioa erabiltzen da. Funtzio honen lehen argumentua bigarren argumentuaren barruan badago, parekaketa hasten den posizioa ematen digu. Bestela, NIL bueltatzen du:

```

(SEARCH "abc" "abcdefg") →    0
(SEARCH "def" "abcdefg") →    3
(SEARCH "hi" "abcdefg")  →    NIL

```

SEARCH funtzioa majuskula eta minuskulak bereiztu gabe erabiltzeko :TEST hitz gakoa CHAR-EQUALekin batera erabiliko dugu (bestela CHAR=ekin egiten du konparazioa):

```

(SEARCH "EFG" "abcdefg") →    NIL
(SEARCH "EFG" "abcdefg" :TEST #CHAR-EQUAL) →    4

```

LENGHT, REVERSE eta ELT funtzioek bezala, SEARCHek listekin ere lan egin dezake. Hona hemen adibide bat:

```

(SEARCH '(mikel) '(josu mikel)) →    1
(SEARCH '(josu) '(josu mikel)) →    0
(SEARCH '(patxi) '(josu mikel)) →    NIL
(SEARCH '(patxi josu) '(jon ane patxi josu koldo)) →    2
(SEARCH '(patxi josu mikel) '(patxi josu mikel)) →    0

```

4.4. PREDIKATUAK.

Predikatuen emaitza posibleak *egiazkoa* edo *faltsua* balioak dira. LISP lengoaian erabiltzen diren predikatuek modu orokorrago batez interpretatzen dituzte balio boolearrak, edozein s-espresio balio boolear gisa ikusi ahal izateko. NIL balioa "faltsua"

balio booleartzat hartzen da, eta beste edozein s-espresio "egiazkoa" balio booleartzat interpretatzen da. Beraz, LISPeo edozein funtzio predikatu gisa ere erabili ahal da. Hala ere ba dira funtzio batzuk bereziki predikatu gisa erabiltzen direnak. Atal honetan predikatu nagusiak azalduko dira.

NIL objektua oso berezia, hiru interpretazio onartzen baititu: lista hutsa, atomoa, eta "faltsua" balio boolearra.

Predikatuen bidez definitutako baldintzak ondoko modura sailka daitezke:

- Baldintza bakunak:
 - Oro har s-espresioei dagozkienak.
 - Bereziki zenbakiei dagozkienak.
- Baldintza konposatuak.

4.4.1. Baldintza bakunak.

4.4.1.1. Oinarrizko predikatuak.

Bi modutakoak dira:

- Espresioen mota egiaztatzen dutenak
(atomo, lista, lista hutsa ala ez hutsa, ...).
- Espresio biren arteko harremanak
(berdinak, bata bestean edukia,).

Espresioen mota egiaztatzen dutenen arteko garrantzitsuenak ondokoak dira:

(ATOM <x>)

x-en balioa atomoa bada T atomoa itzultzen du, bestela NIL.

$L \rightarrow (A \ B)$

$(\text{ATOM } L) \rightarrow \text{NIL}$

$(\text{ATOM } (\text{FIRST } L)) \rightarrow T$

(NULL <x>)

x-en balioa lista hutsa bada T atomoa itzultzen du, bestela NIL.

$L \rightarrow (A \ B)$

$(\text{NULL } L) \rightarrow \text{NIL}$

$(\text{NULL } (\text{REST } (\text{REST } L))) \rightarrow T$

$(\text{NULL } 'A) \rightarrow \text{NIL}$

(ENDP <x>)

x-en balioa lista hutsa bada T atomoa itzultzen du, bestela NIL.

Desberdintasun bat dauka NULL funtzioarekin: argumentuaren balioa lista ez bada errorea itzultzen du.

$L \rightarrow (A \ B)$

$(\text{ENDP } L) \rightarrow \text{NIL}$

$(\text{ENDP } (\text{REST } (\text{REST } L))) \rightarrow T$

$(\text{ENDP } 'A) \rightarrow \text{errorea !}$

(SYMBOLP <x>)

x-en balioa atomo sinbolikoa bada T atomoa itzultzen du, bestela NIL.

$(\text{SYMBOLP } 'A) \rightarrow T$

$(\text{SYMBOLP } '(A)) \rightarrow \text{NIL}$

$(\text{SYMBOLP } 3) \rightarrow \text{NIL}$

$(\text{SYMBOLP } \text{NIL}) \rightarrow T$

(CONSP <x>)

x-en balioa s-espresio konposatua bada (hau da, bikotea bada) T atomoa itzultzen du, bestela NIL.

$A \rightarrow (BAT\ BI)$
 $(CONSP\ A) \rightarrow T$
 $(CONSP\ '(A\ .\ B)) \rightarrow T$
 $(CONSP\ (FIRST\ A)) \rightarrow NIL$
 $(CONSP\ NIL) \rightarrow NIL$

$(LISTP\ <x>)$

x-en balioa s-espresio konposatua edo lista hutsa bada T atomoa itzultzen du, bestela NIL.

$A \rightarrow (BAT\ BI)$
 $(LISTP\ A) \rightarrow T$
 $(LISTP\ '(A\ .\ B)) \rightarrow T$
 $(LISTP\ (FIRST\ A)) \rightarrow NIL$
 $(LISTP\ NIL) \rightarrow T$

$(NUMBERP\ <x>)$

x-en balioa zenbakia bada, T atomoa itzultzen du, bestela NIL.

$(STRINGP\ <x>)$

x-en balioa karaktere-katea bada, T atomoa itzultzen du, bestela NIL.

$(BOUNDP\ <\text{sinbolo}>)$

Sinboloaren balioa definituta badago, T atomoa itzultzen du, bestela NIL.

Beste alde batetik, espresio biren arteko harremana aztertzeko funtzio garrantzitsuenak ondokoak dira:

$(EQUAL\ <x>\ <y>)$

x eta y argumentuen balioak berdinak badira T atomoa itzultzen du, bestela NIL. Memoriako errepresentazioa bakarra den ala ez kontutan hartu gabe. Bi zenbaki berdintzat hartu ahal izateko mota berekoak izan behar dira.

$A \rightarrow (BAT\ BI)$
 $B \rightarrow (HIRU\ BAT\ BI)$
 $(EQUAL\ A\ B) \rightarrow NIL$
 $(EQUAL\ A\ (REST\ B)) \rightarrow T$
 $(EQUAL\ 3\ 7) \rightarrow NIL$
 $(EQUAL\ 3\ 3) \rightarrow T$
 $(EQUAL\ 3.0\ 3) \rightarrow NIL$

Berdintasuna aztertzen duen funtzio honek oso garestia da konputazio-denboraz. EQ, EQL eta = funtzioak eraginkorragoak dira, baina beren erabilpenak mugatuagoak dira:

$(EQ\ <x>\ <y>)$

x eta y argumentuek sinbolo berbera adierazten badute, T atomoa itzultzen du bestela NIL.

$A \rightarrow ETXE$
 $L \rightarrow (ETXE\ HAU)$
 $(EQ\ A\ (FIRST\ L)) \rightarrow T$

Ikuspegi zabalago batetik ikusita, EQ funtzioak egiaztatzen du ea x eta y argumentuen balioak memoriako helbide berbera dauden. Baina kontuz erabili behar da funtzio hau, argumentuak atomoak ez badira LISpeko inplementazio desberdinetan emaitza desberdinak itzultzea posiblea baita. Alegia, bi lista "berdinak" memoriako toki desberdinetan adierazi daitezkeenez, ez dute beti emango T balioa EQ funtzioarekin aplikatzen direnean.

$(EQ\ L\ (ETXE\ HAU)) \rightarrow ?$ Normalean NIL izango da emaitza

$(EQL\ <x>\ <y>)$

x eta y argumentuek sinbolo edo zenbaki bera adierazten badute, T atomoa itzultzen du, bestela NIL. Baina kontuz: zenbakiak konparatzeko erabiltzen

bada eta biak mota desberdinekoak badira, orduan emaitza NIL izango da, nahiz eta zenbakiak balioak izan.

$A \rightarrow \text{ETXE}$
 $L \rightarrow (\text{ETXE HAU } 5)$
 $(\text{EQL } A (\text{FIRST } L)) \rightarrow T$
 $(\text{EQL } 5 (\text{NTH } 2 L)) \rightarrow T$
 $(\text{EQL } 5.0 (\text{NTH } 2 L)) \rightarrow \text{NIL}$

Konparatzeko funtzioak laburbilduz:

FUNTZIOA	HELBURUA
EQUAL	Argumentu bien balioak s-espresio berdinak al dira ?
EQL	Argumentu bien balioak sinbolo edo zenbaki berdinak al dira ?
EQ	Argumentu bien balioak sinbolo berdinak al dira ?
=	Argumentu bien balioak zenbaki berdinak al dira ?

(MEMBER <atomo> <lista>)

Atomoa listaren osagaia ez bada, NIL itzuliko du. Bestela, atomo bilatua baino lehenago dauden osagaiak kenduaz lortzen den azpilista izango da emaitza.

$A \rightarrow (x \ y \ z)$
 $(\text{MEMBER } 'x \ A) \rightarrow (x \ y \ z)$
 $(\text{MEMBER } 'y \ A) \rightarrow (y \ z)$
 $(\text{MEMBER } 'p \ A) \rightarrow \text{NIL}$

Bilaketa hau burutzeko, EQ funtzioaz baliatzen da interpretatzailea listako osagaiak banan banan atomo bilatuarekin alderatzeko, NIL itzuliko ez duen osagai bat aurkitu arte.

(MEMBER 'A '(X (A) Z)) → NIL

LISPeZ, lista bat bilatu nahi bada beste lista baten osagai gisa, ondoko deia egin beharko da.

(MEMBER <espresio> <lista> : TEST #EQUAL)

Adibidez:

(MEMBER '(A) '(X (A) Z') : TEST #EQUAL) → ((A) Z)

4.4.1.2. Zenbakizko predikatuak.

Predikatu hauek argumentu bezala zenbakiak hartzen dituzte. Argumentuen ebaluazioaren emaitza zenbakia ez bada, erroreak sortuko dituzte.

(ZEROP <x>)

x-en balioa zero baldin bada T itzuliko du, bestela NIL.

(PLUSP <x>)

x-en balioa zenbaki positibo zero baino handiagoa bada T itzuliko du, bestela NIL.

(PLUSP 2) → T

(PLUSP 2.5) → T

(PLUSP -2) → NIL

(PLUSP 0) → NIL

(MINUSP <x>)

PLUSP bezalakoa, baina zero baino txikiago den aztertuz.

(ODDP <x>)

x-en balioa zenbaki oso bakoitia bada T itzuliko du, bestela NIL.

(EVENP <x>)

x-en balioa zenbaki bikoitia bada T itzuliko du, bestela NIL.

Zenbakien arteko orden erlazioak aztertzen dituztenak:

(f <zbk1> <zbk2> &rest <zbkiak>)

f delakoa ondoko hauetako bat izanik: <, <=, =, >, >=

Adibidez:

(> 4 3) → T

(> 4 2 1 -2) → T

(> 4 2 3 -2) → NIL ; 2>3 faltsua baita

4.4.2. Baldintza konposatuak.

Hiru eragiketa logiko arruntei dagozkien ingelesezko atomo eragileak erabiltzen dira baldintza konposatuak adierazteko.

- AND Konjuntzioa.
- OR Disjuntzioa.
- NOT Ukapena

Balio boolearren kontzeptu orokorretik abiatu garenez, funtzio konposatzaileen definizioa hedatu beharrean aurkitzen gara.

(NOT <A>)

T atomoa itzuliko du Aren balioa NIL bada, bestelakoetan NIL itzuliko du.

Oharra: NOT eta aurrerago ikusiko dugun NULL funtzioa guztiz baliokideak dira. Idazkera irakurriago lortzearren definitzen dira biak eta ez bat bakarra.

Adibideak:

$$(\text{NOT } (\text{MEMBER 'URDIN ' (URDIN GORRIA))) \rightarrow \text{NIL}$$

$$\begin{array}{c} \text{[-----]} \\ (\text{URDIN GORRIA}) \\ \text{[-----]} \\ \text{NIL} \end{array}$$

$$(\text{NOT } (\text{MEMBER 'URDIN ' (GORRIA))) \rightarrow \text{T}$$

$$\begin{array}{c} \text{[-----]} \\ \text{NIL} \\ \text{[-----]} \\ \text{T} \end{array}$$

$$(\text{AND } \langle B1 \rangle \langle B2 \rangle \dots \langle Bn \rangle)$$

AND forma berezia da, beti ez ditu ebaluatzen bere argumentu guztiak. Ezkerretik eskuinera banan banan ebaluatzen dira baldintzak, berauetako baten emaitza NIL baldin bada, orduan ANDez hasten den formak NIL itzuliko du eta ondorengo baldintzak ebaluatu gabe utziko ditu. Bestela, baldintza guztien artean NIL itzultzen duen bat bera ere aurkitzen ez bada, orduan ANDez hasten den formak Bn azken baldintzaren emaitza itzuliko du.

$$(\text{OR } \langle B1 \rangle \langle B2 \rangle \langle B3 \rangle \dots \langle Bn \rangle)$$

OR ere forma berezia da. Ezkerretik eskuinera banan banan ebaluatzen dira baldintzak, berauetako baten emaitza NIL ez bada, orduan balio berori itzuliko da OReko formaren emaitza gisa. Bestela, ebaluazio guztiek NIL emaitza itzultzen badute, NIL izango da OReko formaren emaitza.

Adibideak:

$$(\text{AND 'URDIN (MEMBER 'URDIN ' (GORRI))) \rightarrow \text{NIL}$$

$$(\text{AND 'URDIN (MEMBER 'URDIN ' (URDIN GORRI))) \rightarrow (\text{URDIN GORRI})$$

$$(\text{OR 'URDIN (MEMBER 'URDIN ' (URDIN GORRI))) \rightarrow \text{URDIN}$$

$$(\text{OR (EQUAL 6 5) (MEMBER 'X (Y Z))) \rightarrow \text{NIL}$$

4.5. HAUTAPEN-FUNTZIOAK.

Hautapen-funtzioak forma bereziak dira. Baldintza logiko bat (edo gehiago) ebaluatuz lortzen den emaitzaren arauera beste argumentuen artean bat hautatuko dute bere emaitza itzultzeko.

Hautapen-funtzio garrantzitsuena COND funtzioa da. COND sinboloaz eta zenbait klausulaz erabiltzen da. Klausula bakoitzean baldintza bat eta zero edo gehiago s-espresio agertzen dira. Baldintzari "aurreko" deritzo, beste s-espresioei "ondoko".

```
( COND  ( <aurreko> <ondoko> <ondoko> ... )
          ( <aurreko> )
          ( <aurreko> <ondoko> ... )
          ... )
```

Hautatutako klausularen s-espresio ondokoak bakarrik ebaluatzen dira eta ez besteenak. Hautatzen den klausula bere baldintzaren emaitza NIL ez den lehenengo da.

COND forma bereziak banan banan aztertzen ditu bere klausulak. Lehenengo klausularen aurrekoa ebaluatuz hasten da. Emaita NIL baldin bada, klausula hau baztertu egiten da eta bigarrena aztertzerara doa, bestela lehenengo klausularen ondokoak sekuentzialki ebaluatzen dira. Hautatutako klausularen ondokoak ebaluatutakoan, CONDeK ez ditu aztertzen gainerako klausulak. Besterik gabe, ebaluatutako azken ondokoaren emaitza itzuliko da COND forma osoaren emaitza gisa. Aurreko guztien emaitza NIL bada, orduan ez da klausularik hautatzen eta CONDeK NIL emaitza itzultzen du.

Hautatutako klausulan ondokorik ez badago, baldintzaren emaitza itzuliko da.

Normalean ondoko bakarra erabiltzen da klausula bakoitzean.

Adibideak:

Demagun ZBKIA sinboloaren balioa kalifikazio bat dela 0-tik 10-ra bitartean dagoen zenbaki bat hain zuzen . Ikus dezagun kalifikazioari dagokion maila (gutxi, nahikoa, ongi ala bikain) lortzeko erabil dezakegun forma hau:

```
(COND (( < ZBKIA 5 ) 'GUTXI)
       (( < ZBKIA 7 ) 'NAHIKOA )
       (( < ZBKIA 9 ) 'ONGI )
       ( T  'BIKAIN ))
```

X eta Y sinboloen balioak zenbaki desberdinak dira. Bietako handiena idazteko:

```
(COND ((> X Y) X)
      ((> Y X) Y))
```

(IF <baldintza> <orduan-forma> &optional <bestela-forma>)

Baldintza ebaluatuz NIL emaitza jasotzen ez bada, "orduan-forma" ebaluatzen da eta bere balioa itzuli. Bestela "bestela-forma" ebaluatzen da eta bere balioa itzultzen da, "bestela-forma" argumentorik ez balego NIL itzuliko da.

$$(\text{IF } \langle \text{baldintza} \rangle \langle \text{orduan-forma} \rangle) \equiv (\text{COND } (\langle \text{baldintza} \rangle \langle \text{orduan-forma} \rangle))$$
$$\begin{aligned} (\text{IF } \langle \text{baldintza} \rangle \langle \text{orduan-forma} \rangle &\equiv (\text{COND } (\langle \text{baldintza} \rangle \langle \text{orduan-forma} \rangle) \\ &\langle \text{bestela-forma} \rangle) \quad (\text{T } \langle \text{bestela-forma} \rangle)) \end{aligned}$$

(WHEN <baldintza> <forma-1> <forma-2> ... <forma-n>)

Ondoko adierazpenaren baliokidea da:

(COND (<baldintza> <forma-1> <forma-2> ... <forma-n>)

(UNLESS <baldintza> <forma-1> <forma-2> ... <forma-n>)

Ondoko adierazpena baliokidea da:

(COND ((NOT <baldintza>) <forma-1> <forma-2> ... <forma-n>))

```
( CASE <forma-giltza>
  ( <giltza-1>    <ondoko-1-1> ...)
  ( <giltza-2>    <ondoko-2-1> ...)
  ...
  ( <giltza-n>    <ondoko-n-1> ...))
```

Forma-giltza ebaluatuz lortzen den balioa sekuentzialki konparatzen da ebaluatzen ez diren giltza-1, giltza-2 , ..., giltza-n balioekin. Konparaketa

batek arrakasta lortzen badu giltzari dagozkion ondoko guztiak ebaluatzen dira. Ebaluatutako azken ondokoaren emaitza itzuliko da CASE osoaren emaitza gisa. Konparaketetan EQL funtzioa erabiltzen da. Adibidez, ikusi ondoko bi forma baliokideak:

```
(COND ((EQL OBJEKTU 'ZIRKULU) (* PI R R))
      ((EQL OBJEKTU 'ESFERA) (* 4 PI R R)))
```

```
(CASE OBJEKTU
  (ZIRKULU (* PI R R))
  (ESFERA (* 4 PI R R)))
```

Konparaketa guztiak arrakastarik gabe bukatzen badira, NIL itzultzen da.

```
OBJEKTU → PUNTU
R → 1
(CASE OBJEKTU
  (ZIRKULU (* PI R R))
  (ESFERA (* 4 PI R R)))
→ NIL
```

OTHERWISE erabiliz badaezpadako balio bat definitu ahal da, horrelako kasuetarako:

```
OBJEKTU → PUNTU
R → 1
(CASE OBJEKTU
  (ZIRKULU (* PI R R))
  (ESFERA (* 4 PI R R))
  (OTHERWISE 0))
→ 0
```

Ebaluatzen ez den giltza baten ordez balio-lista bat jar daiteke. Horrelako kasuetan konparaketa ez da egiten EQL funtzioaz, MEMBERez baino.

OBJEKTU $\rightarrow$ PILOTA

R $\rightarrow$ 1

(CASE OBJEKTU

((ZIRKULU ZIRKUNFERENTZIA) (*PI R R))

((ESFERA PILOTA) (*4 PI R R)))

$\rightarrow$ 12.56637

4.6. EBALUAZIO-FUNTZIOAK

4.6.1. Ebaluazio-funtzio arrunta. EVAL.

(EVAL <forma>)

EVALek "forma" ebaluatuz lortzen den emaitza berrebaluatzen du eta bigarren ebaluazioaren emaitza itzultzen du.

X $\rightarrow$ 27

A $\rightarrow$ (X Y Z)

C $\rightarrow$ A

(FIRST A) $\rightarrow$ X

(EVAL (FIRST A) $\rightarrow$ 27

(EVAL C) $\rightarrow$ (X Y Z)

4.6.2. Lista baten gaineko funtzio baten aplikazioa. APPLY.

Demagun lista baten osagaiak zenbakiak direla eta beren batura kalkulatu nahi dugula.

L $\rightarrow$ (1 4 7 5 2)

(+ L) forma bururatzen zaigu berehala, baina errorea izango da lortuko dugun emaitza bakarra. + eragiketak edozein batugai-kopuru onartzen du, baina ez lista bat. Honela, ondokoa posiblea da:

(+ 1 4 7 5 2) $\rightarrow$ 19

baina ez:

$(+ \ '(1\ 4\ 7\ 5\ 2)) \rightarrow \text{errorea !}$

Problema hau gainditzeko APPLY funtzioa erabili beharko dugu.

$(\text{APPLY } \langle \text{funtzio} \rangle \langle \text{lista} \rangle)$

Lehenengo argumentuaren ebaluazioak funtzio bat eman behar du, eta bigarrenak lista bat. APPLYren emaitza lortzeko, funtzioa aplikatzen da listaren osagaiak (ebaluatu gabe) argumentutzat hartuz. Demagun:

ERAGIKETA $\rightarrow +$

$(\text{APPLY ERAGIKETA } \ '(1\ 4\ 7\ 5\ 2)) \rightarrow 19$

$(\text{APPLY } \#'\text{APPEND } \ '((+ \ 3\ 2)(A))) \rightarrow (+ \ 3\ 2\ A)$

Funtzio baten izena funtzio bat adierazten duen balio gisa erabili nahi bada hobe da $\#'$ karaktereekin idaztea eta ez QUOTE karaktere hutsarekin. Adibidez, idatzi: $\#'\text{FIRST}$, $\#'\text{APPEND}$ edo $\#'+$ baina ez: FIRST , APPEND edo $+$.

4.6.3. Funtzio-aplikazioa funtzioaren izena ebaluatuz. FUNCALL.

$(\text{FUNCALL } \langle \text{funtzio} \rangle \langle \text{arg1} \rangle \langle \text{arg2} \rangle \dots \langle \text{argn} \rangle)$

FUNCALLek "arg1", "arg2" "argn" argumentuen ebaluazio-emaitzei "funtzio" ebaluatuz lortzen den funtzioa aplikatzen die. FUNCALLen emaitza lortzeko:

$L \rightarrow (\text{FIRST } \text{REST } +)$

$(\text{FIRST } \ '(A\ B)) \rightarrow A$

$(\text{FUNCALL } (\text{FIRST } L) \ '(A\ B)) \rightarrow A$

$(\text{FUNCALL } (\text{SECOND } L) \ '(A\ B)) \rightarrow (B)$

$L1 \rightarrow (2\ 3\ 7\ 4)$

$L2 \rightarrow (+ \ - \ \text{FIRST } \text{REST})$

$(\text{FIRST } L1) \rightarrow 2$

$(\text{FUNCALL } \#'\text{APPEND } L1\ L2) \rightarrow (2\ 3\ 7\ 4\ + \ - \ \text{FIRST } \text{REST})$

```
(FUNCALL (FIRST L2) 3 5) → 8
(FUNCALL (THIRD L2) L1) → 2
(FUNCALL (THIRD L2) L2) → +
(FUNCALL (FIRST L1) L1) → errora
```

FUNCALL eta APPLY antzekoak dira, biek argumentu gisa hartzen duten funtzio bat aplikatzen baitute. Azter ditzagun beraien arteko diferentziak:

```
(APPLY <funtzio> lista)
(FUNCALL <funtzio> <arg1> <arg2> .... argn)
```

FUNCALLen argumentu-kopurua aplikatzeko funtzioaren argumentu-kopurua gehi bat izan behar du, funtzioa adierazteko argumentua geitu behar da eta. APPLYk aldiz, gehienetan argumentu pare bat erabiltzen du: aplikatzeko funtzioaren izena eta funtzioarentzako argumentuak lista batean. Lista honen elementu-kopurua eta funtzioak behar duen argumentu-kopurua berdinak dira. Gorago aurkeztu diren FUNCALL edo APPLYren hiru adibideren forma baliokideak ondoak dira :

```
(FUNCALL (FIRST L) '(A B)) → A
(APPLY (FIRST L) '((A B))) → A
```

```
(FUNCALL #'APPEND L1 L2) → (2 3 7 4 + - FIRST REST)
(APPLY #'APPEND '((EVAL L1) (EVAL L2)))
→ (2 3 7 4 + - FIRST REST)
```

```
(FUNCALL #'APPEND '(+ 3 2) '(A)) → (+ 3 2 A)
(APPLY #'APPEND '((+ 3 2) (A))) → (+ 3 2 A)
```


ARIKETAK

4.1.

- a) Ebaluatu ondoko s-espresioak

```
(FIRST '(P H W) )
(REST '(B K P H))
(FIRST '((A B) (C D)) )
(REST '((A B) (C D)))
(FIRST (REST '((A B) (C D))))
(REST (FIRST '((A B) (C D))))
(REST (FIRST (REST '((A B) (C D)))))
(FIRST (REST (FIRST '((A B) (C D)))))
(FIRST (REST (FIRST (REST '((A B) (C D) (E F))))))
(FIRST (FIRST (REST (REST '((A B) (C D) (E F))))))
(FIRST (FIRST (REST '(REST ((A B) (C D) (E F)))))
(FIRST (FIRST '(REST (REST ((A B) (C D) (E F)))))
(FIRST '(FIRST (REST (REST ((A B) (C D) (E F)))))
'(FIRST (FIRST (REST (REST ((A B) (C D) (E F)))))
```

- b) FIRST eta REST aplikazio-sekuentziak definitu ondoko s-espresioetatik
MADARIA harrapatzeko.

```
(SAGARRA LARANJA MADARIA MAHATSA )
((SAGARRA LARANJA) (MADARIA MAHATSA) )
(((SAGARRA) (LARANJA) (MADARIA) (MAHATSA)))
(SAGARRA (LARANJA (MADARIA (MAHATSA))))
((((SAGARRA))) ((LARANJA)) (MADARIA) MAHATSA )
((((SAGARRA) LARANJA) MADARIA) MAHATSA )
```

- 4.2. L sinboloari datxekion balioa (A B) dela jakinda, ebaluatu ondoko s-
espresioak:

```
( APPEND L L)
( APPEND L L L)
( APPEND '(A) '() '(B) '() )
( APPEND L L L)
```

(APPEND '(A) '(B) '((C) (D)))
 (APPEND '((AUTOA FORD) (EDARIA SAGARDOA)))
 (LIST L L)
 (LIST L L L)
 (LIST 'L L)
 (LIST '(A) '(B) '((A) (B)))
 (LIST '(A B) '(C D))
 (APPEND '(A B) '(C D))
 (CONS L L)
 (CONS 'L L)
 (LENGTH L)
 (LENGTH 'L)
 (LENGTH '(A B))
 (LENGTH '((A B) (C D)))
 (LENGTH (APPEND L L))
 (LENGTH '(PLATON SOCRATES ARISTOTELES))
 (LENGTH '((PLATON) (SOCRATES) (ARISTOTELES)))
 (LENGTH '((PLATON SOCRATES ARISTOTELES)))
 (REVERSE '(A B))
 (REVERSE '((A B) '(C D)))
 (REVERSE L)
 (REVERSE (APPEND L L))
 (REVERSE '(PLATON SOCRATES ARISTOTELES))
 (REVERSE '((PLATON) (SOCRATES) (ARISTOTELES)))
 (REVERSE '((PLATON SOCRATES ARISTOTELES)))
 (REST '(A B))
 (REST (LIST L L))
 (NTH 4 (APPEND L L))
 (NTHREST 3 (APPEND L L))
 (LAST L)
 (LAST '((A B) (C D)))
 (LAST 'A)
 (LAST '(A B C D E))
 (APPEND '(A B C) '())
 (LIST '(A B C) '())
 (CONS '(A B C) '())

4.3. Ondoko s-espresiozko segidak ebaluatu, X eta Yri balio atxekiak (MARITXU NORA ZOAZ) eta (1 2 3) izanik.

a)

```
X
Y
(NCONC X Y)
X
Y
(NCONC Y X)
X
Y
```

b)

```
X
Y
(APPEND X Y)
X
Y
(APPEND Y X)
X
Y
```

c)

```
X
Y
(RPLACA X Y)
X
Y
```

d)

```
X
Y
(RPLACD X Y)
X
Y
```

- 4.4. DIGITUAK sinboloari datxekion balioa (ZERO BAT BI HIRU LAU) dela jakinda, ebaluatu ondoko s-espresioak.

(ATOM DIGITUAK)
(ATOM HIRU)
(ATOM DIGITUAK)
(ATOM (ZERO BAT BI HIRU LAU))
(EQUAL DIGITUAK DIGITUAK)
(EQUAL DIGITUAK DIGITUAK)
(EQUAL DIGITUAK (ZERO BAT BI HIRU LAU))
(EQUAL DIGITUAK DIGITUAK)
(NULL '())
(NULL T)
(NULL ())
(NULL DIGITUAK)
(NULL DIGITUAK)
(NULL NIL)
(MEMBER HIRU DIGITUAK)
(MEMBER BOST DIGITUAK)
(MEMBER BI DIGITUAK)
(MEMBER BI ((BAT BI HIRU) (LAU BOST)))
(SYMBOLP DIGITUAK)
(SYMBOLP UNO)
(SYMBOLP 2)
(NUMBERP (FIRST DIGITUAK))
(NUMBERP 2)
(NOT NIL)
(NOT T)
(NOT TXAKURRA)
(NOT (MEMBER BAT DIGITUAK))
(NOT (MEMBER SEI DIGITUAK))
(AND TXAKURRA KATUA)
(OR TXAKURRA KATUA NIL)

4.5. Demagun ondoko balio atxekiez aparte besterik ez dela definitu.

```
L → ( 1 2 3 )  
A1 → 1  
A2 → 2  
A3 → 3  
ERL → ( + - FIRST REST )
```

Ebalua itzazu ondoko espresioak:

```
( FUNCALL + A1 A2 A3 )  
( FUNCALL #' + A1 A2 A3 )  
( FUNCALL #' + 'A1 'A2 'A3 )  
( FUNCALL + 1 2 3 )  
( FUNCALL #' + 1 2 3 )  
( FUNCALL #' + '1 '2 '3 )  
( FUNCALL ( FIRST ERL ) L )  
( APPLY + A1 A2 A3 ) →  
( APPLY ( FIRST ERL ) A1 A2 A3 )  
( APPLY ( FIRST ERL ) L )  
( + L )  
( + A1 A2 A3 )  
( EVAL ( CONS ( FIRST ERL ) '( A1 A2 A3 ) ) )  
( EVAL ( APPEND '( + 1 ) '( A2 A3 ) ) )
```

5. FUNTZIO BERRIEN DEFINIZIOA.

5.1. DEFUN FUNTZIOA.

LISP lengoaiak eskaintzen dituen funtzio aurredefinituetatik abiatuz, programatzaileak bere funtzioak definitu ditzake. Aukera hau erabat ezinbestekoa da Programazio Modularren ideiei jarraitu ahal izateko. Problema oso bat ebazteko idatzi behar den funtzioan erabili ahal dira bere azpiproblema ebartziko dituzten beste funtzio batzu. Zehatz-mehatz definitu den azpiproblema bakoitza geroxeago aztertuko da, batzutan berauek ere beste problema desberdin eta sinpleagotan banatuko direlarik.

Funtzio berriak definitzeko DEFUN funtzioa erabiltzen da, ondoko egitura sintaktikoari jarraituz:

(DEFUN <izena> <parametro-lista> <gorputza>)

DEFUN forma berezia da, ez ditu ebaluatzen bere argumentuak, bat ere ez. Bere helburua zera da: "izen" sinboloari funtzio berri baten definizioa lotzea. Hortik aurrera, "izen" sinboloa forma ebaluagarri baten lehenengo osagai gisa erabili ahal izango da. DEFUN forma batek beste funtzio-aplikazioek egiten duten bezala balio bat itzuliko du: definitu berri duen funtzioaren izena, hain zuzen.

Normalean funtzio-aplikazioek ondorio bakar bat dute: emaitzaren kalkulua. Normalean ere, emaitzaren kalkuluak ez dakar aldaketarik sinboloei datxekieten balio, funtzio edo propietate-listetan (programazio-inguruan alegia). Salbuespen gisa, funtzio baten aplikazioak bere emaitza kalkulatzear gain, inguru horretako aldeketarik sor dezake. Bere emaitza kalkulatu ondoren, funtzioaren aplikazioak sortu duen emaitza bera ez den beste edozein ekintzari *albo-ondorioa* deituko diogu.

"Izen" sinboloari egiten zaion funtzio-definizio baten atxekidura DEFUN forma bereziaren albo-ondorioa da.

"Parametro-lista" delakoan "gorputza" barruan ageri daitezkeen parametroen sinboloak azaltzen dira. Funtzioaren aplikazio konkretu batetan parametroek hartuko dituzten balioak funtzio-deiko argumentuak ebaluatuz lortuko dira. Parametro-listan ageri daitezkeen parametro-mota guztiak 5.3 atalean deskribatzen dira.

"Gorputz" delakoa forma bat (s-espresio ebaluagarri bat) izaten da. Funtzioaren aplikazio konkretu baten emaitza kalkulatzeko gorputzeko forma ebaluatzea da.

Gorputzean forma bat baino gehiago ere ageri ahal da. Kasu horretan funtzio-aplikazioaren emaitza gorputzeko azken formaren emaitza izango da. Hasierako formak albo-ondorioak lortzearen eransten dira.

Adibidea:

Bi zenbakien arteko maximoa kalkulatzeko funtzioa. Zenbakiak berdinak badira NIL itzuliko du.

```
(DEFUN MAXIMO (X Y)
  (COND ((> X Y) X)
        ((< X Y) Y)
        (T NIL))) ; azken klausula hau sobran egon liteke.
```

```
(MAXIMO 3 7) → 7
(MAXIMO 5 5) → NIL
```

Funtzio desberdinek sinbolo bera erabili ahal dute parametro bat izendatzeko.

Funtzio bat aplikatzerakoan parametro baten izena indarrean dagoen beste sinbolo batena baldin bada, azken honi zetxekion balioa gorde egiten da funtzioaren deiak bere balioa kalkulatu arte, eta ondoren sinboloak bere antzinako balioa berreskuratuko du. Funtzioa ebaluatzen den bitartean, sinboloari egokitu zaion argumentuaren balioa atxekitzen zaio. Beraz, LISPeke atomo sinbolikoen esparrua dinamikoa da. Exekuzioan erabakitzen da zein den atomo sinboliko baten erreferentzia.

Adibidez:

Demagun X sinboloari (A B) balioa datxekiola.

```
X → ( A B )
(DEFUN MAXIMO (X Y)
  (COND ((> X Y) X)
        ((< X Y) Y)
        (T NIL)))
→ MAXIMO
X → ( A B )
(MAXIMO 3 7)
```

$X \rightarrow 3$; maximoa kalkulatzeko
 $Y \rightarrow 7$; X-en balioa 3 da.
 $\rightarrow 7$
 $X \rightarrow (A \ B)$; X-en lehengo balioa berreskuratu da.

DEFUN-en bidez sinbolo bati funtzio baten definizioa lotzen zaio. Lotura honek indarrean iraungo du beste DEFUN baten bidez izen bera birdefinitua izan arte edo lan-saioa amaitu arte (LISP interpretatzailetik atera arte, hain zuzen).

"Puntu eta koma" karakterea (",") lerroko irazkinei hasiera emateko erabiltzen da programetako edozein lerrotan. Puntu eta koma karakteretik lerro-bukaeraraino dauden karaktereak ikustezinak dira interpretatzailearentzat.

Adibidea:

```

( DEFUN MAXIMO ( X Y )
  ; X eta Y zenbakiak dira
  ; MAXIMOk bietako maximoa itzultzen du
  ; X eta Y berdinak badira NIL itzuliko du.
  ( COND (( > X Y ) X ) ; X handiena bada X itzuli
          (( < X Y ) Y ) ; X txikiena bada Y itzuli
          ( T NIL ))) ; bestela ( X=Y ), NIL itzuli

```

5.2. LAMBDA FUNTZIOA.

LAMBDA funtzioa izenik gabeko funtzio berriak definitzeko erabiltzen da. Honelako definizioak egoki izan daitezke aldi bakar batean exekutatu nahi diren tratamenduetarako.

LAMBDA funtzioaren definizioen egitura ondokoa da:

```

( LAMBDA <parametro-lista> <gorputza> )

```

<parametro-lista> eta <gorputza> direlakoen esanahiak DEFUN funtzioan bezalakoak dira. LAMBDA espresioaren ebaluazio-emaizta izenik gabeko funtzioa da, bere argumentuei zuzenean aplikatu ahal zaiena.

Adibideak:

Aukerazko parametroak. Funtzioaren definizioko parametro-listan "&optional" hitz gakoari jarraituz adierazten dira. Funtzio-aplikazio batean era honetako parametro

batentzako argumentua ageri daiteke edo ez. Argumentua zehaztuz gero parametroak hartzen duen balioa haren ebaluazioaren emaitza izango da; bestela parametroari dagokion argumenturik zehazten ez bada, parametroaren balio lehenetsia hartuko du. Balio lehenetsia NIL izaten da, baina beste edozein balio definitu daiteke. Horretarako parametro-listan parametroaren sinboloa bakarrik azaldu ordez sinboloa eta balio lehenetsia lista bat osatuz idatzi behar dira. "&optional" hitz gako ondoan nahi beste aukerazko parametro definitu ahal da.

Adibidea:

Funtzioaren definizioa:

(DEFUN F2 (X &optional Y (Z 273)) ...)

Funtzioaren aplikazioak eta bakoitzean parametroek hartzen dituzten balioak:

(F2 3)

X: 3 Y: NIL Z: 273

(F2 3 '(A))

X: 3 Y: (A) Z: 273

(F2 3 '(A) 280)

X: 3 Y: (A) Z: 280

Gainontzeko argumentuak biltzen dituzten parametroak. Funtzioaren definizioan parametro-listan "&rest" hitz gakoari jarraituz azaldu behar du mota honetako definitu daitekeen parametro bakarra. Parametro erregular eta aukerazkoen argumentuak kendu ondoren geratzen diren argumentuak ebaluatu eta lista batean bilduz lortzen da azken parametro honentzako balioa. Funtzio-aplikazioan beste argumenturik geratzen ez bada, parametroaren balioa NIL lista hutsa izango da.

Adibideak:

Funtzioaren definizioa:

(F3 X &rest Y)

Funtzioaren aplikazioak eta bakoitzean parametroek hartzen duten balioak:

(F3 3 '(A) '(B) 3)

X: 3 Y: ((A) (B) 3)

(F3 3)

X: 3 Y: NIL

Funtzioaren definizioa:

(F4 A B &optional C D &rest E)

Funtzioaren aplikazioak eta bakoitzean parametroek hartzen dituzten balioak:

(F4 1 2)

A: 1 B: 2 C: NIL D: NIL E: NIL

(F4 1 2 3)

A: 1 B: 2 C: 3 D: NIL E: NIL

(F4 1 2 3 4)

A: 1 B: 2 C: 3 D: 4 E: NIL

(F4 1 2 3 4 5 '(6) (+ 6 1))

A: 1 B: 2 C: 3 D: 4 E: (5 (6) 7)

Parametro giltzadunak. Parametro asko daudenean eta berauek aplikazio gehienetan balio bera hartzen dutenean erabiltzen dira era honetako parametroak. Horrelako parametro batek balio lehenetsia hartzen ez duen kasu bitxietan funtzioa aplikatzeko forman parametroari dagokion giltzaren ondoan zehazten da balioa. Giltza parametroaren sinboloa bera da baina ":" karaktereaz hasita. Funtzioaren definizioan "&key" hitz gakoari jarraituz azaldu behar dira parametro giltzadunak. Parametroaren sinbolo hutsa ageriko da NIL balio lehenetsia definitzeko edo parametroaren sinboloa eta balio bat biak lista bat osatuz beste balio lehenetsi bat ezartzeko.

Adibidea:

Funtzioaren definizioa:

(DEFUN F5 (X &key Y1 Y2 (Y3 Ø) ...)

Funtzioaren aplikazioak eta bakoitzean parametroek hartzen dituzten balioak:

(F5 3)

X: 3 Y1: NIL Y2: NIL Y3: Ø

(F5 3 :Y2 'A)

X: 3 Y1: NIL Y2: A Y3: Ø

(F5 3 :Y3 2 :Y1 '(A))

X: 3 Y1: (A) Y2: NIL Y3: 2

Parametro laguntzaileak. Funtzio-aplikazioetan parametro hauei ez zaie argumenturik egokitzen. Funtzioaren definizioa irakurgarriago idaztearren erabiltzen dira. Aplikazio batean hartuko duen balioa parametro-listan bertan bere ondoan azaltzen den espresioaren emaitza izango da. Funtzioaren definizioan "&aux" hitz gakoari jarraituz azaldu behar dira parametro laguntzaileak. Parametro hutsa ageri beharko da NIL balio lehenetsia definitzeko edo parametroaren sinboloa eta espresioa biak lista bat osatuz espresioaren emaitza har dezan.

Adibidea:

```
(DEFUN MUTURRAK (LISTA)
  (CONS (FIRST LISTA)
    (LAST LISTA)))
```

funtzio hau irakurgarriago azaltzen da parametro laguntzaileak erabiliz gero:

```
(DEFUN MUTURRAK (L &aux (LEHENENGOA (FIRST L))
  (AZKENEKO-LISTA (LAST L)))
  (CONS LEHENENGOA AZKENEKO-LISTA))
```

Parametro-mota guztien deskribapena eta erabilera bukatzeko esan beharko da parametro-lista batean mota guztietako parametroak bildu ahal direla, baina beren arteko ordena honako hau izan behar dela.

- erregularrak
- aukerazkoak (&optional)
- gainontzekoenak (&rest)
- giltzadunak (&key)
- laguntzaileak (&aux)

5.4. FUNTZIOAK ALA PROZEDURAK ?.

Lehenago 5.1 atalean azaldu denez, funtzio baten definizioko gorputzean s-espresio bat baino gehiago ageri ahal da. Programazio funtzional hutsaren aholkuak bete nahi izanez gero, ez dugu inoiz bat baino gehiago idatziko, bestela funtzioaren albo-ondorioekin topo egingo genuke eta, beraz, ezin genezakeen esan funtzio hutsak erabiltzen ditugula.

Liburu honetan LISPEko alde funtzionala landu dugu bereziki. Beti ekidin dugu albo-ondoriorik sartzeari. Hori dela eta, liburu osoan "funtzio" hitza erabiltzen dugu eta ez "prozedura".

Beste idazle batzuk "prozedura" hitza nahiago dute. Konkreterik WINSTON eta HORNEk bere azken liburuan albo-ondorioen erabilpena praktika arruntatzat hartzen dutenez "prozedura" eta ez "funtzioa" erabiltzen dute.

ARIKETAK

- 5.1. Definitu LEHEN, HONDAR, KATEATU eta LISTA funtzioak FIRST, REST, APPEND eta LIST funtzioak euskaraz deitu ahal izateko. KATEATU eta LISTA funtzioek bi parametro hartuko dituzte.
- 5.2. TRUKABI funtzioa definitu. Bi baino elementu gehiago daukan lista bat hartu eta bere lehenengo bi elementuak elkarren artean trukaturaz lortzen den lista itzuliko du.
- 5.3. PORTZENTAIA funtzioa definitu. Bi zenbaki hartu eta lehenengoak bigarrenarekiko definitzen duen portzentaia itzuliko du. Adibidea:
 $(\text{PORTZENTAIA } 3 \ 4) \rightarrow 75$
- 5.4. COTAN, SEC eta COSEC funtzio trigonometrikoak definitu TAN, COS eta COSEC funtzio primitiboetan oinarriturik.
- 5.5. EMAITZA funtzioa definitu. Lista bat hartu eta T itzuliko du listako osagaiak lau badira eta gainera, bigarren eta laugarren osagaiak zenbaki oso positiboak baldin badira. Bestela NIL itzuliko du.
- 5.6. Lista bat hartu eta lista bera baina lehenengo osagaia azken posiziora eramanda itzuliko duen EZKER-BIRA funtzioa definitu.
- 5.7. ESKUIN-BIRA funtzioa definitu, lista baten azken osagaia listako lehenengoa bihurtzen duena.
- 5.8. IZPILU funtzioa definitu. Atomo-lista bat hartu eta luzera bikoitzeko beste lista bat itzuliko du. Beronek hasieran lista argumentuaren osagai guztiak edukiko ditu eta ondoren berriz osagai guztiak baina alderantzizko ordenan.
- 5.9. Hiru zenbaki oso hartuta, barauk triangelu zuzen baten aldean luzera badira T atomoa edo bestela NIL itzuliko duen TRIANGELU funtzioa eraiki. Triangelu zuzentzat hartuko dugu alde laburren kuadratuaren batura eta alde luzearen karrutuaren arteko distantzia alde luzeraren karrutuaren %2 baino txikiagoa baldin bada.

- 5.10. A, B, eta C parametroetan zenbaki osoak hartzen dituen ERROAK funtzioa definitu. Funtzioak $Ax^2 + Bx + C = 0$ polinomioaren erro biek listan bat itzuliko du. Erro biak errealak direla eman.

$$x = (-b \pm (b^2 - 4ac)^{1/2}) / (2a)$$

- 5.11. Definitu KONPLEXUAK funtzioa. A, B eta C parametroak hartu eta T itzuliko du $b^2 - 4ac$ espresioa negatiboa baldin bada, bestela NIL itzuliko du.

- 5.12. Suposa dezagun FUN1 eta FUN2 funtzioak era honetara definitzen direla:

```
(DEFUN FUN1 (X &OPTIONAL L (M 10) N &REST R)
```

```
  GORPUTZA1 )
```

```
(DEFUN FUN2 (X &KEY L (M 10) N)
```

```
  GORPUTZA2 )
```

Funtzio hauen ondoko aplikazio bakoitzean parametroek hartzen dituzten balioak azaldu:

```
(FUN1 7)
```

```
(FUN1 7 4 3)
```

```
(FUN1 7 4 3 5)
```

```
(FUN1 7 4 3 5 '(A) '(B))
```

```
(FUN2 2 3)
```

```
(FUN2 2 3 :L 3)
```

```
(FUN2 2 3 :N 2 :M 3 :L 4)
```

- 5.13. Minutu-kopuru bat orduraz adierazten duen funtzio bat idatzi. Batzutan funtzio honek bigarren argumentu bat ere hartuko du segundu-kopuru bat gehitzeko.

- 5.14. Kapitulu honen hasieran MAXIMO funtzioa bi parametroekin definitu dugu. Birdefinitu funtzioa zenbaki bat edo gehiagoren maximoa itzul dezan. Adibidez:

```
(MAXIMO 3 7 8 5) → 8
```

```
(MAXIMO 3) → 3
```

5.15. Definitu ORDEZKATU funtzioa. Bere ohizko erabileraren arabera bi atomo eta lista bat hartu eta itzultzen du lista bera baina argumentu bezala eman zaion lehenengo atomoaren lehen adiera bigarren atomoaz ordezkaturik.

ORDEZKATU funtzioak erabilera bitxiagoak ere baditu. Alegia, ALDIAK parametro giltzadunaren balioa zenbaki natural bat bada listako n lehenengo agerpenekin burutuko du ordezkapena. ATZERA parametro giltzadunaren balioa NIL ez bada ordezkapenak ez dira hasiko listako burutik, buztanetik baino.

a) Idatz ezazu ORDEZKATU funtzioaren definizioko parametro-lista.

b) Idatz ezazu ORDEZKATU funtzioaren aplikazio bat L listako V atomoaren lehenengo hiru agerpenak B atomoaz ordezkatzeko.

c) Idatz ezazu ORDEZKATU funtzioaren aplikazio bat L listako V atomoaren azken agerpena B atomoaz ordezkatzeko.

6. PROGRAMAZIO FUNTZIONALA LISPeZ.

6.1. EZAUGARRIAK.

Lehenengo kapituluan azaldu zen bezala, LISP lengoaia funtzionalen aitzindaria izan zen. Kontzeptu hau ez zen sortu hogeiei urte geroago arte (Henderson 1980), ordurarteko programazio agintzailearen alternatiba gisa aurkeztua izan zenean. Programazio-lengoaia bat funtzionaltzat hartzen dugu baldin eta bere funtsezko baliabideak funtzioen definizioa, aplikazioa eta konposizioa badira. Batzutan "funtzional" kalifikatzailearen ordeztu "aplikatzaile" ematen zaie. Programazio funtzionalean programak funtzio baten aplikazio bat adierazten duten espresioak dira, espresio hauek definitzeko funtzio primitibo aurredefinitu eta programatzaileak berak definitutako funtzioak ere erabiltzen direlarik. Programaren exekuzioaren emaitza programa den espresio hori ebaluatuz lortzen den balioa da. Baina ESPRESIOA EBALUATZERAKOAN EZ DA ALDARAZTEN EZ BALIOA, EZ ESPRESIOA EZTA BESTE INONGO OBJEKTURIK ERE. Propietate honi **erreferentzi-gardentasuna** esaten zaio. Beraz espresio baten ebaluazio-emaitzak beti izango dira berdinak, eta ez da inoiz albo-ondoriorik gertatuko.

Programazio agintzailetik bereizteko bi ezaugarri nagusi dira:

Lehenengoa, aldagai kontzeptua ez da bitarteko emaitzak gordetzeko kutxa bat, bere balioa asignazioaren bidez aldarazi ahal dena. Matematikako aldagaia erabiltzen da hemen; aldagaia funtzioen definizioetan erabiltzen da argumentuaren balioa adierazteko. Funtzioa argumentu konkretuetarako kalkulatu denean aldagaiaren ordeztu bere balioa jarriko da. Balio hori ezin daiteke aldarazi funtzioaren aplikazioa kalkulatzeko denean.

Adibidez: $f(x) = x^2 + 2x + 1$

x aldagaia da, $f(3)$ kalkulatu nahi bada definizioa x-en aipamen guztiak 3 zenbakiaz ordezkatzeko dira:

$$f(3) = 3^2 + 2 \cdot 3 + 1$$

Programazio klasikoaren errore asko kutxa kontzeptu horren ondorio da. Sarritan gertatzen baita aldagai baten balioa okerra dela nahigabe programako beste puntu batetan

beste balio bat asignatu zaiolako. Programazio agintzailcan beti ez da betetzen *erreferentzi gardentasuna*.

Bigarren ezaugarria kontrol-egituren eza da. Programa ez da agindu-sekuentzia bat. Beraz jauzirik edo bigiztarik ez da behar exekuzio-ordena kontrolatzeko. Geratzen diren aztarna bakarrak IF eta COND forma bereziak dira, baina hauek ere funtzio gisa ikusi ahal dira, forma berezi gisa hain zuzen:

$$\text{IF (baldintza, orduan, bestela)} = \begin{cases} = \text{orduan} & \underline{\text{baldin}} \text{ baldintza} = \text{egiazkoa} \\ = \text{bestela} & \underline{\text{baldin}} \text{ baldintza} = \text{faltsua} \end{cases}$$

Kontrol-egiturarik ez dagoenez, programatzaileak ez du pentsatu behar prozesua burutzeko eman behar diren pausoetaz, bere indar guztiak emaitzaren deskribapenean bilduko dira. Ikuspuntu berri hau dela eta, programa baten sorkuntza erabat desberdina da.

Azter dezagun faktorialaren problema bi ikuspuntuetatik, ebazpide bien arteko aldea argi uzteko asmoz:

Modu agintzaileaz:

$$\text{Fact (n)} = 1 * 2 * 3 * \dots * (n - 1) * n$$

eta

$$\text{Fact (0)} = 1$$

Modu funtzionalaz:

$$\text{Fact (n)} = \begin{cases} = 1 & \text{baldin } n = 0 \\ = n * \text{Fact (n - 1)} & \text{bestelakoetan} \end{cases}$$

PASCALezko programa agintzailea:

```
function fakt ( n : integer ) : integer
var i, tartekoa : integer ;
begin tartekoa := 1 ;
  for i := 1 to n do
    tartekoa := tartekoa * i ;
  fakt := tartekoa
end
```

LISPeZko programa funtzionala:

```
(DEFUN FAKT ( N )  
  (COND ((= 0 N ) 1 )  
        ( T  (* N (FAKT (1- N ))))))
```

Bi ohar azalduko ditugu programa hauetaz:

1. Funtzionalan ez dago aldagairik, ezta kontrol-egiturarik ere, (COND funtzioari egindako deia baino ez dago definizioan).
2. Nren balioa 0 ez denean, definitzen ari den funtzioa bera erabili da emaitza zehaztatzeko. Definizio errekurtsiboak maiz agertzen dira programa funtzionaletan.

Kapitulu honetan programazio funtzionalaren definizio errekurtsiboaren definizio errekurtsiboen azalpena ematen dugu. Hurrengo kapituluan ohizko programazio agintzailearen definizio iteratiboak azalduko dira.

6.2. EGITURA ERREKURTSIBOAK.

Prozesu errepikakorrek funtzio-dei errekurtsiboen bidez definitu ahal dira, hau da, funtzio baten definizioan funtzioari berari deitzen zaio beste argumentu desberdinak erabiliz. Definizio-mota hau erabiliz gero, kontu handiz egiaztatu behar da dei errekurtsiboen kateak bukaera bat aurkituko duela. Ondoko bi erregela gorde beharko dira:

- a) Gutxienez behin funtzioari egindako deia modu ez-errekurtsiboz kalkulatzeko da (kasu nabaria).
- b) Funtzio-deia errekurtsiboki kalkulatzeko denean zera ziurtatu egin behar da: funtzioa behin edo gehiagotan aplikatuz, azkenean beti helduko da kasu nabari batetara, eta orduan emaitza ez da kalkulatu errekurtsiboki.

Problema bat azpiproblema txikiagotan banatzen duen ebazte-estrategia orokorraren kasu partikularra dugu planteamendu errekurtsiboa, alegia, azpiproblema bat problema nagusiaren mota berekoa baina argumentu sinpleagoekin azaltzen denean.

Funtzio errekurtsibo bat LISPez implementatzeko honako pausoak bete beharko dira:

- 1) Parametrizazioa
- 2) Kasu nabarien analisia
- 3) Kasu orokorren analisia
- 4) Funtzioaren eraikuntza
- 5) Dei-kopurua finitua dela ziurtatzea.

Ondoren 6 adibide desberdin aurkeztuko ditugu pauso guztiak ondo bereizten.

1. Adibidea.

Definitu BERREKETA funtzioa. X eta Y bi zenbaki positibo hartuta, X^Y berredura itzuliko du.

1. pausoa. Parametrizazioa.

Argi dago bi parametro hartu beharko direla, alegia, berrekizuna eta berretzailea jasoko dutenak. Demagun X eta Y direla bi parametro horiek.

2. pausoa. Kasu nabarien analisia.

Berreketa kalkulatzeko beste berreketa bat berrekizun txikiagoaz erabiliko dugunez, kasu limitea berrekizuna zero denean gertatzen da, eta orduan emaitza zuzenean kalkulatu ahal da errekurtsibitaterik gabe.

$$X^0 = 1$$

3. pausoa. Kasu orokorren analisia.

Beste kasu guztietarako honela definitzen da emaitza:

$$X^Y = X * X^{Y-1}$$

4. pausoa. Funtzioaren eraikuntza.

(DEFUN BERREKETA (X Y)

(IF (= 0 Y)

1

(* X (BERREKETA X (1- Y))))))

5. pausoa. Dei-kopurua finitua dela ziurtatzea.

Kasu orokorreko dei bakoitzean Y zenbaki oso positiboa 1ez txikiagotzen da. Beraz beti helduko da kasu nabarira.

Konputagailuan funtzioa implementatu ondoren komeniko litzateke ebaluatzea kasu nabaria eta kasu orokorren bat .

(BERREKETA 2 0) $\rightarrow$ 1

(BERREKETA 2 3) $\rightarrow$ 8

Ikus dezagun nola ebaluatzen den (BERREKETA 2 3). Asmo horrekin konputazio-arbola idatziko dugu. Arbolaaren adabegiak BERREKETA funtzioaren aplikazio-deiak dira; adabegiaren aplikazioa kalkulatzeko sortzen diren funtzioaren aplikazio-dei berriak bere umeak izango dira arbolan.

$$\begin{array}{c}
 (\text{BERREKETA } 2 \ 3) \rightarrow 8 \\
 \downarrow \quad \uparrow 4 \\
 (\text{BERREKETA } 2 \ 2) \\
 \downarrow \quad \uparrow 2 \\
 (\text{BERREKETA } 2 \ 1) \\
 \downarrow \quad \uparrow 1 \\
 (\text{BERREKETA } 2 \ 0)
 \end{array}$$

Konputazio-arbola hau ikusi ahal da pantailan ebaluazioaren traza eskatzen badugu. Hori lortzeko nahikoa da lehenago (TRACE BERREKETA) espresioa ebaluatzea. Traza ateratzeko agindua indargabetzeko (UNTRACE) espresioa ebaluatu behar da. TRACE funtzioaren erabilera 8. kapituluaz azalduko da.

2. Adibidea.

FIBONACCI funtzioa definitu. N zenbaki oso positibo bat hartuz, Fibonacci-ren Ngarren zenbakia itzuliko du.

Fibonacciaren zenbakiak honela definitzen dira:

$$\text{fib}(n) = \begin{cases} \text{fib}(n-1) + \text{fib}(n-2) & \text{baldin } n > 1 \\ 1 & \text{baldin } n = 1 \\ 1 & \text{baldin } n = \emptyset \end{cases}$$

Ikus daitekeenez funtzioaren definizio matematikoa bera ere errekurtsiboa da.

1. pausoa. Parametrizazioa.

N izango da parametro bakarra.

2. pausoa. Kasu nabariaren analisia.

Definizio errekurtsiboa denez, erraza da asmatzea bi kasu nabari daudela.

$$\text{FIB}(0) = 1 \quad \text{eta} \quad \text{FIB}(1) = 1$$

3. pausoa. Kasu orokorren analisia.

Definiziotik zuzenean aterata:

$$\text{FIB}(N) = \text{FIB}(N-1) + \text{FIB}(N-2) \quad \text{baldin } n > 1$$

4. pausoa. Funtzioaren eraikuntza.

(DEFUN FIBONACCI (N)

(IF (OR (= 1 N) (= 2 N))

1

(+ (FIBONACCI (- N 1))

(FIBONACCI (- N 2))))))

5. pausoa. Dei-kopurua finitua dela ziurtatzea.

Kasu orokorrean beste bi dei egiten dira hurrengo zenbaki oso positibo txikiagoekin. Kasu nabari bat N zero denean ematen da, baina N bat denean ere beharrezkoa da beste kasu nabari bat definitzea, bestela FIB (1) kalkulatzekoan FIB (-1) beharko genuke eta dei-kopurua infinitua litzateke. Limitean dauden 0 eta 1 kasu nabariak definituz, dei-kateak beti bukatzen da.

Funtzioa inplementatu ondoren komeni da egiaztatzea kasu nabariaren eta kasu orokorren emaitza.

(FIBONACCI 0) → 1

(FIBONACCI 1) → 1

(FIBONACCI 4) → 5

Zein da azken aplikazio horren konputazio arbola ?

3. Adibidea.

Definitu ELEMENTU-P funtzio predikatua. Bere argumentuak X s-espresio bat eta L lista bat izango dira. (ELEMENTU-P X L) formak NIL itzuliko du X L listako elementua ez bada. Bestela L lista itzuliko da baina X baino lehenago zeuden elementuak kenduta.

Adibidez:

(ELEMENTU-P 'A '(B C)) → NIL

(ELEMENTU-P 'A '(B C '(A))) → NIL

(ELEMENTU-P 'A '(B A C)) → (A C)

Hau da, LISpeko MEMBER funtzio aurredefinituak emaitza berdinak itzuliko lituzke.

Funtzioaren kasu orokorra definitzeko nahikoa da listaren lehenengo elementua aztertzea eta listaren hondarraz funtzioak emango duen balioa erabiltzea. Listako lehenengo elementua bilatua baldin bada lista osoa izango da emaitza. Aldiz, lehenengo elementua bilatua ez bada, lista osoaz eta listaren hondarraz lortuko diren emaitzak

berdinak direnez, beste aplikazio sinpleago honek balio du emaitza zehazteko. Bete ditzagun pauso guztiak ideia hauei jarraituz.

1. pausoa. Parametrizazioa.

X s-espresioa eta L lista . Beste parametrorik ez da behar.

2. pausoa. Kasu nabariaren analisia.

Lista hutsa bada ezin izango dugu aurkitu X elementua, beraz emaitza NIL da.

Listaren lehenengo elementua eta bilatua berdinak badira orduan emaitza lista osoa da.

3. pausoa. Kasu orokorren analisia.

Listaren lehenengo elementua eta bilatua berdinak ez badira orduan emaitza listaren hondarraz egindako ELEMENTU-P funtzioaren aplikazioa da.

4. pausoa. Funtzioaren eraikuntza.

```
(DEFUN ELEMENTU-P (X L)
  (COND ((ENDP L) NIL)
        ((EQ (FIRST L) X) L)
        (T (ELEMENTU-P X (REST L))))))
```

5. pausoa. Dei-kopurua finitua dela ziurtatzea.

Emitza zehazteko ELEMENTU-P funtzioaren dei errekursiboak erabiltzen direnean lista argumentuak elementu bat gutxiago du. Listen elementu-kopurua zenbaki oso positibo ez-infinitua denez, kasurik txarrean lista hutsera helduko da beti dei-kopuru finitu bat egin ondoren.

4. Adibidea.

Definitu ATOMO-KOPURU funtzioa lista batean dauden atomoak konta ditzan. Atomoak kontatuko dira nahiz eta listaren elementuak ez izan, adibidez, listaren azpilistetako atomoak ere kontatuko dira.

```
(ATOMO-KOPURU '(A (B C (D E)) (E F) (G (H I)))) → 10
```

1. pausoa. Parametrizazioa.

Lista hartuko duen parametroa behar da.

2. pausoa. Kasu nabariaren analisia.

Lista hutsa bada bere atomo-kopurua zero da.

3. pausoa. Kasu orokorren analisia

Zenbait elementu dauzkan lista bat aztertzen dugunean atomo-kopuru osoa lortu daiteke lehenengo elementuaren atomo-kopuruari listaren hondarreko atomo-kopurua gehituz. Lehenengo elementua atomoa balitz bere atomo-kopurua 1

litzateke bestela ATOMO-KOPURU funtzioa bera erabil daiteke lista den lehenengo elementuaren atomo-kopurua lortzeko.

4. pausoa. Funtzioaren eraikuntza.

```
(DEFUN ATOMO-KOPURU (L)
  (COND ((ENDP L) 0)
        ((ATOM (FIRST L)) (1+ (ATOMO-KOPURU (REST L))))
        (T (+ (ATOMO-KOPURU (FIRST L))
               (ATOMO-KOPURU (REST L))))))
```

5. pausoa. Dei kopurua finitua dela ziurtatzea.

Bi dei errekurtsibo agertzen dira azken kasu orokorrean, biak independenteak dira eta bakoitzak sortzen duen dei-kopurua finitua da.

(ATOMO-KOPURU (REST L)) dei errekurtsiboan elementu bat gutxiago duen lista bat erabiltzen da. Listen elementu-kopuru zenbaki oso positibo finitua denez, lista hutsera helduko da beti dei-kopuru finitu bat eginez.

(ATOMO-KOPURU (FIRST L)) dei errekurtsiboan kabiätze- edo parentesi-maila bat gutxiago duen espresio bat erabiltzen da. Listen kabiätze-maila finitua denez dei-kopuru finitu bat eginez beti aurkituko da lehenengo elementutzat atomo bat edukiko duen lista bat, eta orduan honetako dei errekurtsiboei bukaera emango zaie.

Adibide honekin bukatzeko.

Azter ezazu edozein s-espresiok duen atomo-kopurua lortzen duen honako funtzio hau:

```
(DEFUN ATOMO-KOPURUA (S) ; edozein s-espresio
  (COND ((NULL S) 0)
        ((ATOM S) 1)
        (T (+ (ATOMO-KOPURU (FIRST S))
               (ATOMO-KOPURU (REST S))))))
```

Zeintzu dira bere kasu nabari eta orokorrak? Beti bukatzen dira dei errekurtsiboak?

5. Adibidea.

Definitu TXERTATU funtzioa L lista baten AT1 atomo guztien atzetik AT2 atomoa txerta dezan. AT1 atomorik ez badago ez da AT2 atomorik sartzen. Adibidea:

(TXERTATU '(C (D A) A F) 'A 'B) → (C (D A B) A B F)

1. pausoa. Parametrizazioa.

Hiru parametro azaldu behar dira: L lista, AT1 eta AT2 atomoak.

2. pausoa. Kasu nabarien analisisia.

Lista hutsa denean lista hutsa da emaitza.

3. pausoa. Kasu orokorren analisisia.

Lista hutsa ez bada eta lehenengo elementua atomoa ez denean, lista emaitzaren lehenengo elementua eta hondarra jakinez gero CONS funtzioaren bidez lortzen dugu lista emaitza osoa. Biak lortu daitezke dei errekurtsiboak eginez.

Lista hutsa ez bada eta lehenengo elementua atomoa bada, lehen bezala emaitzaren hondarra errekurtsiboki lor daiteke, baina emaitzaren lehen elementua zehazteko aztertu behar da ea lista argumentuaren lehenengo atomo hori bilatua den. Hala ez bada bera da emaitzaren lehen elementua, bestela atomo horrez gain AT2 bigarren elementu bezala txertatu beharko da.

4. pausoa. Funtzioaren eraikuntza

```
(DEFUN TXERTATU (L AT1 AT2)
  (COND ((ENDP L) NIL)
        ((ATOM (FIRST L))
         (IF (EQL AT1 (FIRST L))
              (CONS AT1
                    (CONS AT2
                          (TXERTATU (REST L) AT1 AT2)))
              (CONS AT1
                    (TXERTATU (REST L) AT1 AT2))))
        (T (CONS (TXERTATU (FIRST L) AT1 AT2)
                  (TXERTATU (REST L) AT1 AT2))))))
```

5. pausoa. Dei-kopurua finitua dela ziurtatzea.

ATOMO-KOPURU aurreko funtzioaren arrazoi berdinak erabiltzen ditugu. Listen elementu-kopurua eta kabiitze-maila finituak dira. (TXERTATU (REST L) AT1 AT2) dei-errekurtsiboan elementu bat gutxiago duen lista bat erabiltzen denez, lista hutseraino helduko da. (TXERTATU (FIRST L) AT1 AT2) dei-errekurtsiboan kabiitze-maila txikiagoa duen espresio bat erabiltzen denez, lehenengo elementua atomoa edukiko duen lista bat aurkitzera helduko gara.

6. Adibidea.

Zenbakizko lista bat hartu eta bere elementurik handiena itzuliko duen MAXIMOA funtzioa definitu. Lista hutsik badago NIL izango da emaitza.

Hiru definizio alternatibo banan banan aurkeztuko ditugu.

Lehenengo definizioaren kasu orokorreko listan bi zenbaki baino gehiago dagoenean beste lista txikiago baten maximo bezala definitu daiteke emaitza. Lista txikiago hori osatzeko lehenengo bi zimbakien arteko handiena hartuko dugu eta gainontzeko zenbaki guztiak.

1. pausoa. Parametrizazioa.

Lista bat baino ez dugu hartzen.

2. pausoa. Kasu nabariaren analisia.

Lista hutsa baldin bada NIL izango da emaitza. Kasu hau hasierako deian soilik gerta daiteke. Beste dei posible guztietan gutxienez elementu bat egongo da listan Listak elementu bakarra baldin badu elementu hori da maximoa.

3. pausoa. Kasu orokorren analisia.

Listak gutxienez bi zenbaki dauzka. Beste lista bat osatu lehenengo eta bigarren zimbakien arteko handienarekin eta gainontzeko zimbakiekin. Hasierako listaren maximoa beste lista txikiago honen maximoa izango da.

4. pausoa. Funtzioaren eraikuntza.

(DEFUN MAXIMOA (L)

(COND ((ENDP L) NIL)

((ENDP (REST L)) (FIRST L))

((> (FIRST L) (SECOND L))

(MAXIMOA (CONS (FIRST L)

(NTHCDR 2 L))))

(T (MAXIMOA (REST L))))))

5. pausoa. Dei-kopurua finitua dela ziurtatzea.

Dei-errekurtsiboetako listek zenbaki bat gutxiago daukate. Listek elementu-kopuru finitua dutenez, dei-kopuru finitu batez bigarren kasu nabariraino helduko gara. Lehenengo kasu nabaria hasierako deian lista hutsa denean soilik gertatzen da.

MAXIMOA definitzeko bigarren aukeran funtzio lagungarri bat erabiliko dugu. MAXIMOLAG funtzioak lista bat eta zenbaki bat hartzen du eta itzultzen du listako zimbakien eta zenbaki argumentuen arteko maximoa, hau da guztien maximoa.

MAXIMOLAG funtzioa definituta badago MAXIMOA oso erraza da eta gainera ez da errekurtsiboa:

(DEFUN MAXIMOA (L)

(COND ((ENDP L) NIL)

(T (MAXIMOLAG L (FIRST L))))))

MAXIMOLAG errekurtsiboa da, baina definitzeko erraza da aurretik baldin badakizkigu listaren lehenengo elementua eta listaren hondarrarekin MAXIMOLAGek itzultzen duen emaitza. Ikus dezagun zelan definitu:

1. pausoa. Parametrizazioa.

L lista bat eta M zenbaki bat.

2. pausoa. Kasu nabariaren analisia.

Lista hutsa baldin bada, orduan M zenbakia da maximoa.

3. pausoa. Kasu orokorren analisia.

Lista hutsa ez denean, lista eta zenbakiaren arteko maximoa beste dei errekurtsibo sinpleago baten bidez definitu daiteke. Beste dei berri horretan lista argumentua lehengo listaren hondarra da eta zenbaki argumentua lehengo zenbakiaren eta lehengo listako lehen zenbakiaren arteko maximoa.

4. pausoa. Funtzioaren eraikuntza.

(DEFUN MAXIMOLAG (L M)

(COND ((ENDP L) M)

((> (FIRST L) M) (MAXIMOLAG (REST L) (FIRST L)))

(T (MAXIMOLAG (REST L) M))))

5. pausoa. Dei-kopurua finitua dela ziurtatzea.

Betiko arrazoia aipatu behar da hemen: dei bakoitzean elementu bat gutxiago daukan lista bat erabiltzen dela.

MAXIMOIA definitzeko hirugarren aukera bigarrenaren moldaketa bat da aukerazko parametro bat erabiliz. Horrela ez dago funtzio lagungarria sartzeko beharrik. Lehenengo deian M parametroak NIL balioa hartuko du. NIL balioa izango da MAXIMOArek emaitza listan zenbaki bat ere ez badago. Bestela hurrengo dei errekurtsiboan listaren hondarra eta listaren lehenengo zenbakia hartuko dira, bigarren definizioan bezalaxe.

(DEFUN MAXIMOIA (L &optional (M NIL))

(COND ((ENDP L) M)

((OR (EQ M NIL) ; Lehenengo deia bada eta lista ..

; hutsa ez bada

(> (FIRST L) M)) ; edo lehen elementua M zenbakia

; baino handiagoa bada

(MAXIMOIA (REST L) (FIRST L)))

(T (MAXIMOIA (REST L) M))))

6.3. ERREKURTSIBITATE-MOTAK.

Errekurtsibitatearen bidez definitutako funtzioak aztertuta, ondoko hiru kontzeptuok azaltzen dira:

- errekurtsibitate bakuna
- errekurtsibitate bikoitza
- buztaneko errekurtsibitatea

Funtzio baten definizio errekurtsiboa bakuna da emaitza itzultzen duten espresioetan gehienez behin bakarrik deitzen bazaio definitzen ari den funtzioari. Esaterako, BERREKETA funtzioaren definizioa.

```
(DEFUN BERREKETA (X Y)
  (IF (= 0 Y)
    X
    (* X (BERREKETA X (1- Y)))))
```

Funtzio baten definizio errekurtsiboa bikoitza da emaitza itzultzen duen espresio batean birritan deitzen bazaio definitzen ari den funtzioari. Esaterako, FIBONACCI eta ATOMO-KOPURU funtzioen definizioa.

```
(DEFUN FIBONACCI (N)
  (IF (OR (= 1 N) (= 2 N))
    1
    (+ (FIBONACCI (- N 1))
      (FIBONACCI (- N 2)))))
```

```
(DEFUN ATOMO-KOPURU (L)
  (COND ((ENDP L) 0)
        ((ATOM (FIRST L)) (1+ (ATOMO-KOPURU (REST L))))
        (T (+ (ATOMO-KOPURU (FIRST L))
                (ATOMO-KOPURU (REST L))))))
```

Buztaneko errekurtsibitatea definitu aurretik problema baten erredukzioa zer den azaldu behar da. Ikusi ELEMENTU-ZENBAKIEN-KOPURU funtzioaren definizio hau:

```

(DEFUN ELEMENTU-ZENBAKIEN-KOPURU (L &OPTIONAL (N 0))
  (COND ((ENDP L) N)
        ((NUMBERP (FIRST L))
         (ELEMENTU-ZENBAKIEN-KOPURU (REST L)
                                       (1+ N))))
  (T (ELEMENTU-ZENBAKIEN-KOPURU (REST L) N)))

```

Antzerako gauza bat egiten duen baina bi argumentu dauzkan funtzio laguntzaile bat erabili da. Funtzioak lista bat eta zenbaki oso bat hartzen ditu, eta itzultzen du zenbakia gehi listaren elementuzenbakiaren kopurua. Aplikazio bat egiten denean, dei guztietarako emaitza beste dei batean lortutako emaitza bera da, beste eragiketarik egin gabe (azkenean ezik, noski). Beraz, nahikoa da funtzio laguntzailearen dei-iladako azkena kalkulatzeko lehenengo deiaren balioa zuzenean jakin ahal izateko, tarteko deiaren emaitzak espreski kalkulatu gabe.

Problema bat beste problema berri baten bidez ebazten dugunean, eta problema berriaren soluzioa jakinda beste konputaziorik egin behar ez bada jatorrizko problemaren soluzioa lortzeko, orduan problema berria *jatorrizko problemaren-erredukzioa* dela esaten da.

Funtzio baten definizioa errekurtsiboa bada eta dei errekurtsibo guztiak erredukzioak badira, orduan funtzioaren definizioa *buztan-errekurtsiboa* dela esaten da. Dei-iladako bukaeran (buztanean) dagoen azken deiaren emaitza jakinda, lehenengo deiaren emaitza jakin ahal dugulako.

Errekurtsibitate bakuna interesgarria da lista baten elementu guztien gainean tratamendu bat egiterakoan soluzio naturala delako.

Errekurtsibitate bikoitza interesgarria da zuhaitzen osagaien gainean tratamendu bat egiterakoan soluzio naturala delako.

Buztaneko errekurtsibitatea interesgarria da eraginkortasunari begira, ondoko sailean azaltzen den arrazoiengatik.

6.4. ERREKURTSIBITATE ETA ERAGINKORTASUNA.

Oro har, LISP oso egokia da datuak egituraz eta tamainuaz dinamikoki aldatzen direnean, edota datuen egitura edukina bezain garrantzitsua denean. Datu-egiturak eta definizio errekurtsiboak errazago eta bizkorrago erabiltzen dira LISPeZ lengoia agintzaileez baino. Hala ere zenbait aplikazio errekurtsiboetan denbora-problemak

agertzen dira. Horrelakoetan soluzio iteratiboetara jo gabe ba dira zenbait puntu kontutan hartzekoak:

- Gehienetan emaitza bera lortu ahal da funtzio errekursibo bakun batez, beste funtzio errekursibo bikoitza batez, edo hirugarren funtzio ez-errekurtsibo batez.
- Egungo LISPeo inplementazio modernoek aparteko trebetasuna dute buztan-errekurtsibo diren definizioekin eraginkortasun maila gorena lortzeko. Horrelako definizio bat ezagutuz gero sortzen diren deiak ez baitira pilaratzen.
- Aurrerago ikusiko diren MAPCAR moduko funtzioek eskema oso erabilgarriak dira zenbait tratamendutarako (listen transformaketa, iragazketa, elementu-kontaketa eta elementu-bilaketa). Ez dira soluzio errekursibo hutsak, baina datu-egitura errekursiboetarako eskema funtzionalak definitzen dituzte, eta gainera tratamendu horiek burutzeko eraginkorragoak dira.

6.5. SASIALDAGAIK.

Funtzio baten definizioa irakurgarriago, ulerterrazago eta eraginkorrago bihurtzarren, espresio bati izen bat eman ahal zaio. Antzeko modura egiten da matematikan.

Adibidez:

$$\text{"Bedi } f(x, y, z) = a + 3a - xy + 2z \text{ non } a = \frac{x+1}{y+z} \text{"}$$

a ez da definitzen aldagai gisa, $a(x, y, z) = \frac{x+1}{y+z}$ funtzio lagungarri gisa baizik. f funtzioaren definizioa irakurgarriago gertatzen da eta gainera amankomuneko espresioa behin baino ez da kalkulatzen.

Definizioetarako lagungarri diren "sasialdagai" hauek LET forma bereziaren bidez erabiltzen dira LISP lengoaiatz. Hona hemen bere erabilpenaren adibidea:

```
(LET ((<izen1> <balio1> ) (<izen2> <balio2> ) ... )
      <espresio1> <espresio2> ... )
```

Forma hau ondokoaren baliokidea da:

```
((LAMBDA (<izen1> <izen2> ... ) <espresio1> <espresio2> ... )
  <balio1> <balio2> ... )
```

LET funtzioak espresioak ebaluatzen ditu, baina lan hori egiten duen bitartean zenbait atomo sinboliko erabili ahal izango ditu berauei lehenago lotu dizkien balioak adierazteko.

Itzulitako emaitza azken espresioaren ebaluazioaren emaitza izango da.

Adibidea: PUNTUAK funtzioak futbol-talde baten izena eta partidu baten emaitza hartuta, taldeari dagozkion puntuak itzuliko ditu (edo NIL partiduaren partaidea ez bada). Partidu baten emaitza lau atomotako lista izango da:

(etxekoa golak kanpokoa golak).

Honako bi definioen artean, zein da irakurgarriago? Zein da ulerterrazago? Zein idatzi daiteke errazago? Zein da kalkulu berdinak errepikatzen ez dituen?

```
(DEFUN PUNTUAK (TALDEA PARTIDUA )
  (COND ((OR (NOT (EQUAL TALDEA (FIRST PARTIDUA ))
              (NOT (EQUAL TALDEA (THIRD PARTIDUA ))
              NIL)
        ((EQUAL TALDEA (FIRST PARTIDUA ))
        (COND ((>(SECOND PARTIDUA)( FOURTH PARTIDUA)) 2)
              ((= (SECOND PARTIDUA)( FOURTH PARTIDUA) 1)
              (T 0 )))
        ((EQUAL TALDEA (THIRD PARTIDUA ))
        (COND ((>(SECOND PARTIDUA)( FOURTH PARTIDUA)) 0)
              ((= (SECOND PARTIDUA)( FOURTH PARTIDUA) 1 )
              (T 2 )))))
```

```

(DEFUN PUNTUAK (TALDEA PARTIDUA )
  (LET ( (T1 (FIRST PARTIDUA ))
        (T2 (THIRD PARTIDUA ))
        (G1 (SECOND PARTIDUA ))
        (G2 (FOURTH PARTIDUA )))
    (COND ((OR (NOT (EQUAL TALDEA T1 )
                (NOT (EQUAL TALDEA T2 ))
                NIL )
          ((EQUAL TALDEA T1 )
           (COND ((> G1 G2 ) 2 )
                  ((= G1 G2 ) 1 )
                  (T 0 ))))
          ((EQUAL TALDEA T2 )
           (COND ((> G1 G2 ) 0 )
                  ((= G1 G2 ) 1 )
                  (T 2 ))))))))

```

LET forma bereziak paraleloki ebaluatzen ditu sinboloei egokituko dizkien balioak, beraz, sasialdagai baten balioa ezin izango da kalkulatu beste sasialdagai bati eman berri zaion balioa erabiliz. Batzutan erabilpen hori lagungarri izan liteke. Honelako kasuetan LET* forma berezira jotzen da. Hori LET bezalakoa da baina sekuentzialki ebaluatu eta egokitzen ditu sasialdagaien balioak.

Adibidea:

```

A → 15
(LET ( (A (+ 3 3 ))
      (B (+ A 2 )))
  (LIST A B )) → (6 17 )
A → 15

```

```

-----
A → 15
(LET* ((A (+ 3 3 ))
      (B (+ A 2 )))
  (LIST A B )) → (6 8 )
A → 15

```

LET* forma berezia ere LAMBDA espresioen bidetz azaldu liteke:

```
(LET* ((<izen1> <balio1> ) (<izen2> <balio2> )....)
      <espresio1> <espresio2> .... )
```

Forma hau ondokoaren baliokidea da:

```
((LAMBDA ( <izen1> )
      ((LAMBDA ( <izen2> )
        <espresio1> <espresio2> ...)
       <balio2> )
      <balio1> )
```

Bukatzeko azaldu baharko dugu *&aux* erako parametroek antzeko helburua betetzeko balio dutela. Ikus dezagun PUNTUAK funtzioa definitzeko beste aukera baliokidea parametro lagungarriak erabiliz:

```
(DEFUN PUNTUAK ( TALDEA
                PARTIDUA
                &AUX (T1 (FIRST PARTIDUA ))
                    (T2 (THIRD PARTIDUA ))
                    (G1 (SECOND PARTIDUA ))
                    (G2 (FOURTH PARTIDUA )))
  (COND ((OR (NOT (EQUAL TALDEA T1 )
              (NOT (EQUAL TALDEA T2 ))
              NIL )
        ((EQUAL TALDEA T1 )
         (COND ((> G1 G2 ) 2 )
               ((= G1 G2 ) 1 )
               (T 0 ))))
        ((EQUAL TALDEA T2 )
         (COND ((> G1 G2 ) 0 )
               ((= G1 G2 ) 1 )
               (T 2 ))))))
```


ARIKETAK

- 6.1. Lista baten lehenengo atomoa itzuliko duen LEHENATOMOA funtzioa definitu. Adibideak:

```
(LEHENATOMOA '((A B) C)) → A
(LEHENATOMOA '(A (B C))) → A
(LEHENATOMOA '()) → NIL
```

- 6.2. Lista baten lehenengo atomoa kentzen duen KENDU-LEHEN-ATOMOA funtzioa definitu.

```
(KENDU-LEHEN-ATOMOA '((A B) C)) → ((B) C)
(KENDU-LEHEN-ATOMOA '(A (B C))) → ((B C))
(KENDU-LEHEN-ATOMOA '() ) → NIL
```

- 6.3. Definitu KENDUATOMOA funtzioa A atomo bat eta L lista bat har ditzan eta L listatik A atomoaren agerpenak kenduz lortzen den lista itzul dezan. Adibidea:

```
(KENDUATOMOA 'C '((A C) D C B)) → ((A) D B)
```

- 6.4. Azaldu ondoko funtzioaren helburua:

```
(DEFUN AUSKALO (S)
  (COND ((NULL S) 1)
        ((ATOM S) 0)
        (T (MAX (1+ (AUSKALO (FIRST S))
                  (AUSKALO (REST S)))))))
```

- 6.5. Azaldu ondoko funtzioaren helburua:

```
(DEFUN HALAKO (L)
  (COND ((NULL L) NIL)
        ((ATOM L) L)
        (T (CONS (HALAKO (FIRST L))
                  (HALAKO (REST L))))))
```

- 6.6. Multzo bat bere elementuek osatzen duten listaren bidez adieraziko dugu. Elementuen ordena ez da esanguratsua. Horrelako multzoak erabiliko dituzten ondoko funtzioak idatz itzazu:

- a) Bi multzoen arteko BILKETA.
- b) Bi multzoen arteko EBAKETA.
- c) Bi multzo ea berdinak diren aztertuko duen BERDINP predikatua.

- 6.7. Lista batetan agertzen diren zenbaki guztien batura kalkulatzeko duen ZBKIBATURA funtzioa idatzi. Emandako listan zenbakiak ez diren balioak eta azpilistak etor daitezke.

$$(ZBKIBATURA \text{ '(1 B (3 B) 7)') } \rightarrow 11$$

- 6.8. LAUTU funtzioa definitu hartzen duen s-espresioko atomo guztiekin lista ez-kabiatu bat itzul dezan. Adibidez:

$$(LAUTU \text{ '(A (A (A (A B))) (A B)))') \rightarrow (A A A A B A B)$$

- 6.9. Bere elementuen artean zero asko dauzkan bektore bat labur-labur adierazi ohi da bikote-lista baten bidez, non bikote bakoitzean ez-nulu baten indizea eta balioa ematen bait dira. Adibidez (1.2, 0, 0, 3, 0, 1, 0) bektorea ((1 1.2) (4 3) (6 1)) listaren bidez adierazten da. Bektorea SPARSE idazkeraz idatzi dela esaten da, edo SPARSE bektorea dela ere bai. Idatz ezazu SPARSE bektore bat eskalar batez biderkatuko duen funtzio bat.

- 6.10. Bi SPARSE bektoreren arteko biderkadura eskalarra kalkulatzeko duen funtzio bat idatzi.

- 6.11. Bi SPARSE bektoreren arteko batura kalkulatzeko duen funtzio bat idatzi.

- 6.12. SPARSE bektore bat eta SPARSE matrize baten arteko biderkadura kalkulatzeko duen funtzioa definitu.

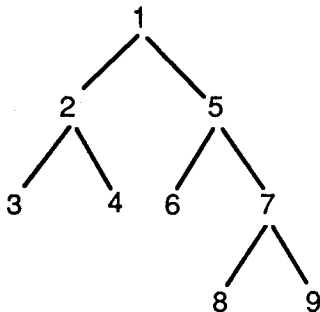
$$\begin{pmatrix} (1 ((1 1))) \\ (2 ((1 1) (3 2))) \\ (3 ((3 1))) \end{pmatrix} \Leftrightarrow \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

6.13. Suposa dezagun A eta B matrizeen arteko biderkadura C matrizea dugula. Modu asko daude eragiketa hori burutzeko, baina guk ondoko hau aukeratu dugu: C matrizearen (i,j) elementua, A matrizearen i-garren lerroa eta B matrizearen j-garren zutabearen arteko biderkadura eskalarra dugu; beraz, Cren j-garren zutabea A matrizea eta B matrizearen j-garren zutabearen arteko biderkadura dugu, hau da, matrize bat eta bektore baten arteko biderkadura.

Gauzak honela, aurreko ariketan eraikitako funtzioa erabiliz definitu bi matrizeen arteko biderkadura burutuko duen funtzio bat.

6.14 Zuhaitz bitar baten pisua 1 da horri huts bat baldin bada. Zuhaitza horri hutsa ez bada, eta bere azpizuhaitz bien pisuak berdinak badira zuhaitzaren pisua azpizuhaitz baten pisua gehi bat izango da, bestela azpizuhaitz bien pisuen arteko handiena da.

Zuhaitz bitarrak LISPez adierazteko listak erabili daitezke. Adibidez:



→ (1 (2 3 4) (5 6 (7 8 9)))

Definitu PISUA funtzioa zuhaitz bitar baten pisua itzul dezan. Adibidez:

(PISUA '(1 (2 3 4) (5 6 (7 8 9)))) → 3

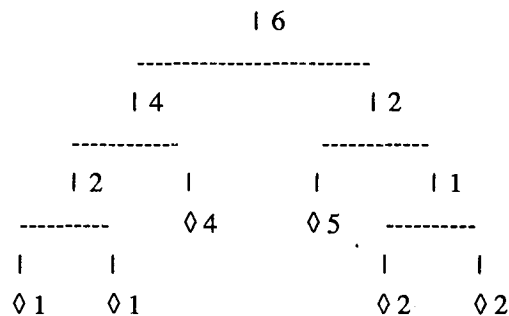
6.15. Zati mugikorrez osatutako eskultura abstraktuei mugikari esaten zaie. Orokorrean zenbait objektu harien bidez puntetatik eskegita dauzkaten ziriez osatzen dira. Mugikari-mota sinple bat errekursiboki defini daiteke ondoko modura: hari batekin eskegita dagoen objektu bat, edo bere puntetan azpimugikari bana eskegita daukan ziri zintzilikatu bat. Ziri bakoitza bere

erditik eskegita dagoela suposatzen badugu, erraz adieraz daiteke zuhaitz bitar baten bidez. Honela bada, objektu sinpleak beren pisuko zenbakiaz errepresentatuko ditugu, eta mugikari konposatuak ordea 3 osagaitako listen bidez. Listaren lehenengo osagaia ziriaren pisua eta beste biak puntetako mugikariak izango dira.

Mugikariak orekatuta egon behar dira, hau da, punta bietan eskegita dauden azpimugikariak pisu berekoak izan behar dira eta gainera bakoitza bere aldetik orekatuta egon behar da.

Definitu MUGIKARIP funtzioa mugikari bat ea orekatuta dagoen azter dezan. Horrela ez badago NIL itzuliko du, baina ondo orekatuta badago bere pisu osoa itzuliko du. Adibidez:

(MUGIKARIP '(6 (4 (2 1 1) 4) (2 5 (1 2 2)))) → 30



- 6.16. Zuhaitz bitarrak espresio aritmetikoak adierazteko erabil daitezke ondoko modura:

(* (+ A B) (- C (/ D E)))

Konpiladoreen betebeharren artean espresio aritmetikoen itzulpena aurkitzen da, beroriek makina-lengoaiaz adierazteko asmoz. Demagun makina-lengoaian 1, 2, 3, ... erregistro-multzoa erabiltzeko aukera dagoela eta bertan tarteko emaitzak gorde ahal direla.

Beste alde batetik demagun erregistro batetik besteetara balioak pasatzeko aukera dagoela MOVE ekintzaren bidez, eta bi erregistroren balioekin eragiketa aritmetikoak burutzeko ADD, SUB, MUL eta DIV ekintzak daudela, hauen emaitza erabilitako lehenengo erregistroan bertan utziko delarik.

Adibidez, $(* (+ A B) (- C (/ D E)))$ espresioaren itzulpena ondokoa izango da:

```
( (MOVE 1 A)
  (MOVE 2 B)
  (ADD 1 2)
  (MOVE 2 C)
  (MOVE 3 D)
  (MOVE 4 E)
  (DIV 3 4)
  (SUB 2 3)
  (MUL 1 2))
```

Azken emaitza "1" erregistroan uzten da.

Itzulpen hau egingo duen KONPILA-ARIT funtzioa definitu.

- 6.17. Multzo baten elementuen artean baliokidetasun-erlazioak definitzea posible da. Baliokidetasun-erlazio hauek erlazionatzen diren elementu-bikoteen lista batez adieraz daitezke. A eta B baliokideak badira, orduan B eta A ere baliokideak dira. A eta B baliokideak badira eta B eta C baliokideak badira orduan A eta C ere baliokideak dira. Baliokidetasun-klase bat elementu guztien azpimultzoa da, bere elementu guztiak elkarren artean baliokideak direlarik. Lehengo adibidean (A B C) multzoak baliokidetasun-klase bat definitzen du.

Definitu SAILKATU funtzioa baliokidetasun-bikotezko lista bat hartu eta baliokidetasun-klaseen lista itzuliko duena. Adibidea:

```
(SAILKATU '((A B) (B D) (C E)))
→ ((A B D) (C E))
(SAILKATU '((A B) (B D) (C E) (D E)))
→ ((A B D C E))
(SAILKATU '( (A E) (Z F) (M B) (P K) (E I) (F S) (B D) (T P)
              (I O) (S V) (D G) (K P) (O U) (V Z) (G M) (P T)))
→ ((A E I O U) (F S V Z) (B D G M) (K P T))
```

- 6.18. Euklides-en algoritmoa erabiliz bi zenbaki osoren zatitzaile komunetako handiena kalkulatu duen ZKH funtzioa definitu. Algoritmoaren arauera,

A eta B bi zenbaki oso badira eta $A \geq B$ suposatuz, B Aren balio zatitzailea bada, orduan B bera da zenbaki bien zkh-a. Bestela, nahiko da A-B eta Bren zatitzaile komunetako handiena kalkulatzeko. Adibidea:

A=36	B=26	26 ez da 36-ren zatitzailea
A=26	B=36-26=10	10 ez da 26-ren zatitzailea
A=26-10=16	B=10	10 ez da 16-ren zatitzailea
A=10	B=16-10=6	6 ez da 10-ren zatitzailea
A=6	B=10-6=4	4 ez da 6-ren zatitzailea
A=4	B=2	2 zenbakia 26 eta 36-ren zkh-a da

6.19. Zer itzultzen dute ZORO eta ERO funtzioek? Beren aplikazioa azaltzeko eman ezazu zenbait adibide esanguratsu.

```
(DEFUN ZORO (X)
  (IF (NULL X)
    NIL
    (APPEND (ZOROLAG X X)
      (ZORO (REST X))))))
```

```
(DEFUN ZOROLAG (X Y)
  (IF (NULL Y)
    NIL
    (CONS X (ZOROLAG X (REST Y)))))
```

```
(DEFUN ERO (X Y)
  (IF X
    (CONS (FIRST X)
      (ERO Y (REST X)))
    Y))
```

6.20. Polinomio bat adierazteko era honetako lista bat erabili dezakegu:

```
(((<koef-0> <berretz-0>) (<koef-1> <berretz-1>) ... (<koef-n> <berretz-n>)))
```

Adibidez: ((3 0) (-1 1) (5 2)) listak $P(x) = 3 - x + 5x^2$ polinomioa adierazten du.

Horrelako P polinomio bat eta x-entzako n balio konkretu bat hartuta, P(n) itzuliko duen EBALUAPOL funtzioa defini ezazu. Suposatu berredura kalkulatzeko funtzio aurredefiniturik ez dagoela.

- 6.21. Idatz ezazu funtzio errekurtsibo bat x zenbakiaren funtzio esponentziala kalkulatzeko:

$$\begin{aligned} e^x &= 1 + x/1 + x^2/2 + x^3/6 + x^4/24 + \dots + x^n/1*2*\dots*n \\ &= 1 + x/1 + (x/1)*(x/2) + (x/1)*(x/2)*(x/3) + \dots \end{aligned}$$

K zenbaki minimo bat baino handiago diren serieko gaiak bakarrik hartu kontutan. Adibidez:

$$(\text{EXPN } 1 \text{ } 0.01) = 1 + 1 + 0.5 + 0.166666 + 0.041666 = 2.708332$$

Serieko beste gai guztiak 0.01 baino txikiago direnez ez dira kontutan hartzen.

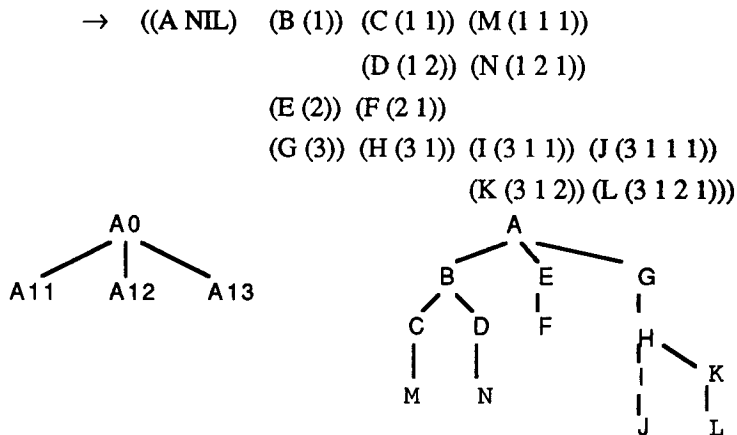
- 6.22. Zuhaitz ez-hutsak adierazteko asmoz lista bat erabiliko dugu. Lehenengo elementua adabegi-erroa da eta beste elementuak bere umeak (hauek ere zuhaitz ez-hutsak dira).

Definitu SAKABANATU funtzioa ondoko zehaztapena bete dezan:

- Zuhaitz ez-huts bat hartzen du.
- Itzuliko duen listak azpilista bat edukiko du zuhaitzeko adabegi bakoitzarentzat.
- Azpilista bakoitzak bi elementu dauzka: adabegia eta bere posizioa.

Adibidez:

```
(SAKABANATU '(A (A11) (A12) (A13))) →
  ((A NIL) (A11 (1)) (A12 (2)) (A13 (3)))
(SAKABANATU '(A (B (C (M)) (D (N)))
  (E (F))
  (G (H (I (J)) (K (L)))))) →
```



6.23. Aurreko ariketako SAKABANATU funtzioa definituta dagoela suposatuz:

- a) Deskriba ezazu zer itzultzen duen *xzuhaitz* funtzioak bere argumentuak edozein zuhaitz ez-huts eta beronen bi adabegi izanez gero.

```

(DEFUN XZUHAITZ (ADABEGI1 ADABEGI2 ZUH)
  (LET* ( (A1 (SAKABANATU ARB))
    (ADABEGI1POS (POSIZIO ADABEGI1 A1))
    (ADABEGI2POS (POSIZIO ADABEGI2 A1)))
    (COND ((HASIERAP ADABEGI1POS ADABEGI2POS)...
      ADABEGI1)
      ((HASIERAP ADABEGI2POS ADABEGI1POS)...
      ADABEGI2)
      (T NIL))))

(DEFUN HASIERAP (L1 L2)
  (COND ((NULL L1) T)
    ((EQUAL (FIRST L1) (FIRST L2))
      (HASIERAP (REST L1) (REST L2)))
    (T NIL)))

(DEFUN POSIZIO (ADABEGI ZUHAITZ_SAKAB)
  (COND ((NULL ZUHAITZ_SAKAB) 0)
    ((EQUAL ADABEGI
      (FIRST (FIRST ZUHAITZ_SAKAB)))
      (SECOND (FIRST ZUHAITZ_SAKAB)))
    (T (POSIZIO ADABEGI (REST ZUHAITZ_SAKAB)))))
  
```


b) Zeintzu dira ondoko espresioen ebaluaketaren emaitzak?

(SETF ZUH1 '(A (B (C) (D)) (E (F)) (G (H (I (J))))))

(XZUHAITZ 'A 'E ZUH1)

(XZUHAITZ 'E 'C ZUH1)

(XZUHAITZ 'F 'J ZUH1)

(XZUHAITZ 'G 'J ZUH1)

7 PROGRAMAZIO AGINTZAILEA LISPeZ.

LISP lengoaiak (programazio funtzionalaren aitzindaria izan zen arren) badauka programa agintzailak eraikitzeke zenbait tresna: agindu-sekuentziaketa, aldagaiak, asignazioa, bigiztak definitzeko kontrol-egiturak, eta etiketak. Beraz guztiz posiblea da programazio klasikoaren arauerako programak eraikitzea LISP erabiliz. Agindu-sekuentziaketa nola implementatzen den 5. kapituluaz azaldu dugu, bertan erakusten bait genuen funtzio baten gorputzean forma bat baino gehiago sartu ahal zela, albo-ondorioak sortarazteko erabiltzen zirelarik. Horrelako jokaera onartzen denean ez da hitz egiten "funtzioetaz", "prozeduretz" baizik. Kapitulua honetan LISPen elementu agintzailak deskribatuko ditugu.

7.1. ALDAGAIAK.

LISPeZ aldagaiak atomo sinbolikoen bidez implementatzen dira. Hiru aldagai-mota nagusi bereizten dira: lexikalak, orokorrak eta bereziak. Mota bakoitzaren erabilera eta ebaluakera desberdina da. Ikus ditzagun banan banan:

a) Aldagai lexikalak.

Funtzioen definizioetan parametro gisa agertzen diren sinboloen funtzioaren "bertako aldagai" edo "aldagai lexikal" deitzen zaie. Funtzioari deitzen zaionean balio bat atxekitzen zaie parametroei, baina funtzioaren emaitza itzuli ondoren desagertu egiten da atxekidura.

Funtzio desberdinetako parametroek sinbolo bera erabil dezakete. Funtzio batek beste bati deitzen badiu eta honen parametro baten sinboloa haren parametro batena bada ez da nahasketarik gertatzen. Balio berria atxekiko zaie sinboloari bigarren funtzioari deitzerakoan, baina funtzio lagungarriaren emaitza lortu ondoren bere lehengo balioa berreskuratuko du parametroak. Adibidez:

```
(DEFUN F1 (X)
  ; f(x)= x4+x
  (+ (KARRATU (KARRATU X))
     X))
```

(DEFUN KARRATU (X)
 (* X X))

(F1 5) forma ebaluatzen denean X parametroak balio desberdinak hartzen ditu funtzio-aplikazio desberdinetan:

```
(F1 5)
|
| (+ (KARRATU (KARRATU X)) X))
|
| (KARRATU (KARRATU 5))
|
|   (KARRATU 5)
|   |
|   | X: 5   (* 5 5)
|   |       → 25
|   |
|   | → 25
|   | (KARRATU 25)
|   | |
|   | | X: 25 (* 25 25)
|   | |       → 625
|   | |
|   | | → 625
|   | |
|   | | → 625
|   | |
|   | | X → 5
|   | |
|   | | → 630
```

Parametroen jokaera hori funtzio-aplikazioak inguratzen dituzten harresien bidez azaltzen da. Hau da, funtzio bat aplikatzerakoan parametroen balioak ezarri ondoren harresi baten bidez isolatuta geratzen dira funtzioaren gorputzeko parametro eta balioak kanpoko aldagaiekin interfeentziarik jaso ez dezaten. Nahiz eta funtzio barruan ("prozedura barruan" hobeto kasu honetan) parametro baten sinbolo batentzat asignazio berri bat egon balio berriak ere ez du inolako eraginik funtzioaren aplikazio horretatik kanpo.

Harresi bakoitzak bere atezaina dauka. Atezainak bere harresiko aldagai lexikalen zerrenda dauka bakoitza bere balio atxekiarekin. Aldagai lexikal bat ebaluatu behar denean gertuen dagoen atezainarengana jotzen da. Atezainak dagokion balioa itzultzen du aldagaia

zerrendan agertzen bada, bestela bere harresia inguratzen duen hurrengo harresira jotzeko eskatuko du.

Funtzio-aplikazio bat ebaluatzerakoan beste funtzio bati deitzen zaionean bigarren deiazen harresiak ez daude sartuta lehenengoaren barruan. Adibidez:

```
(DEFUN MUTUR-BIAK-LAGUNTZEKIN (LISTA-OSOA)
  (LIST (HASIERAKOA LISTA-OSOA)
        (AZKENEKOA LISTA-OSOA))))
```

```
(DEFUN HASIERAKOA (M)
  (FIRST M))
```

```
(DEFUN AZKENEKOA (N)
  (FIRST (LAST N)))
```

Definizioak horrela eginda MUTUR-BIAK-LAGUNTZEKIN funtzioa aplikatzen badugu eta honek beste biei deitzen badie, orduan harresien itxura ondokoa izango da:

MUTUR-BIAK-LAGUNTZEKIN Aldagai-zerrenda: LISTA-OSOA
--

HASIERAKOA Aldagai-zerrenda: M

AZKENEKOA Aldagai-zerrenda: N

Baina ... kontuz! Ez da beste modu honetara izango:

MUTUR-BIAK-LAGUNTZEKIN

Aldagai-zerrenda: LISTA-OSOA

HASIERAKOA

Aldagai-zerrenda: M

AZKENEKO

Aldagai-zerrenda: N

Beraz HASIERAKOA eta AZKENEKO funtzioan LISTA-OSOA erabili izan balitz errorea gertatu zatekeen. Hau da, definizioak honako hauek izan balira:

```
(DEFUN MUTUR-BIAK-LAGUNTZEKIN (LISTA-OSOA)
```

```
  (LIST (HASIERAKOA )
```

```
        (AZKENEKOA )))
```

```
(DEFUN HASIERAKOA ()
```

```
  (FIRST LISTA-OSOA))
```

```
(DEFUN AZKENEKO (N)
```

```
  (FIRST (LAST LISTA-OSOA)))
```

Orduan (MUTUR-BIAK-LAGUNTZEKIN '(GOZARI BAZKARI AFARI)) forma ebaluatzerakoan errorea ageri zatekeen LISTA-OSOA sinboloak balio atxekirik ez duelako.

Noizbait bigarren konfigurazioa behar izanez gero, LABELS forma bereziaz baliatu beharko ginateke:

```
(DEFUN MUTUR-BIAK-LAGUNTZEKIN (LISTA-OSOA)
```

```
  (LABELS ( (HASIERAKOA (M) (FIRST M))
```

```
            (AZKENEKO (N) (FIRST (LAST N)))
```

```
    (LIST (HASIERAKOA LISTA-OSOA)
```

```
          (AZKENEKO LISTA-OSOA)))
```

LET eta LAMBDA forma bereziak funtzio-aplikazioen harresi-ezarpenaren salbuespenak dira. Beroriek ezartzen duten harresia automatikoki sartzen da indarrean zegoen harresiaren barruan LABELS forma berezia erabili gabe. Adibidez, PUNTUAK funtzioaren definizioa eta harresia ondokoak dira:

```
(DEFUN PUNTUAK (TALDEA PARTIDUA )
  (LET ( (T1 (FIRST PARTIDUA ))
        (T2 (THIRD PARTIDUA ))
        (G1 (SECOND PARTIDUA ))
        (G2 (FOURTH PARTIDUA )))
    (COND ((OR (NOT (EQUAL TALDEA T1 )
                (NOT (EQUAL TALDEA T2 ))
                NIL )
            ((EQUAL TALDEA T1 )
             (COND ((> G1 G2 ) 2 )
                   ((= G1 G2 ) 1 )
                   (T 0 ))))
            ((EQUAL TALDEA T2 )
             (COND ((> G1 G2 ) 0 )
                   ((= G1 G2 ) 1 )
                   (T 2 ))))))))
```

PUNTUAK

Aldagai-zerrenda: TALDEA PARTIDUA

LET

Aldagai-zerrenda: T1 T2 G1 G2

(COND ...)

LET forma barruko PARTIDUA sinboloa ebaluatzeko LETen harresia jauzi eta PUNTUAK funtziotik jasotzen da balioa.

Aldagai lexikalei buruz honaino azaldu duguna programazio funtzional zein agintzailentzat biontzat balio du. Baina agintzailean egiten den erabilpena bitxiagoa izaten da: aldagai hauen balioa asgnazioen bidez aldarazi ahal baita funtzioaren gorputzean.

Asignazioa aurreraxeago azalduko diren SETF, SETQ edo SET forma bereziekin burutzen da. Adibidez, ondoko F1 funtzioan X parametroak hasieran balio bat hartzen du eta gorputzeko zenbait forma ebaluatu ondoren Xek 4 balio berria hartzen du.

```
(DEFUN F1 (X)
  ...
  (SETF X 4)
  ...)
```

Hala ere, nahiz eta parametro bati balio berri bat asignatu, funtzioaren aplikazioa bukatzen denean parametroa eta bere balio desagertu egiten dira.

b) Aldagai orokorrak.

Aldagai orokorrak ez daude lotuta funtzio konkretuekin, funtzioetatik aparte definitzen dira sinbolo hauek. Gero edozein funtzioetatik erabil daiteke bere balioa.

Parametro aktibo ez den sinbolo bati asignazio bat egiten zaionean sinboloa aldagai orokortzat hartzen da.

Programazio funtzionalean aldagai orokorrak erabiltzen direnean balio konstante eta korapilotsuak adierazteko erabiltzen dira. Adibidez hizkuntz aplikazio batean kontsultarako dagoen hiztegi handi bat. Aldagai orokorra izanik ez da parametro bezala ageri beharko funtzio askotako definizioetan eta gainera behin baino ez da idatziko.

Programazio agintzailean antzera erabiltzen dira baina balio atxekia aldatu egin daiteke beste asignazio anitzen bidez.

c) Aldagai bereziak.

Noizean behin funtzio baten emaitza adierazteko beste funtzio lagungarri batzu erabiltzen dira eta nagusiaren parametro batzu berriro azaltzen dira funtzio lagungarrien parametroen artean inolako aldaketarik gabe. Parametroen errepikapen horregatik programazioa aspergarri bihurtzen denean edo eraginkortasuna moteltzen denean, orduan egokia izaten da "aldagai berezien" erabilpena.

Aldagai berezien ebaluazioa ez da egiten harresien erizpideari jarraituz. Sinbolo batek aldagai berezi bat adierazten badu, ebaluatu nahi denean deien pilara jo behar da. (Deien pila oraindik kalkulatu gabe geratzen diren funtzio-deiak biltzen dituen da). Pilako tontorretik hasita sinboloari parametro gisa balio bat atxeki dion dei bat aurkitu arte jeisten da. Aurkitutako balio hori izango da ebaluazioaren emaitza.

Aldagai berezien erabilpena hain bitxia denez, beren sinboloak izartxo hasi eta bukatuta idatzi ohi dira nabarmenagoak izan daitezen. Adibidez: *X*

Bi dira sinbolo bat aldagai berezitzat erazagutzeko tresnak:

(DEFVAR <aldagai>)

Aldagai bat bere ondorengo agerpen guztietan berezia izango dela adierazten du. Adib.: (DEFVAR *HIZTEGIA*)

(DECLARE (SPECIAL <aldagai>))

Aldagai berezi bat erazagutzen du baina funtzio baten esparruan soilik. Ondoko adibidean ANALIZATU-LAG funtzioaren inplementazioan *hiztegia* aldagai bereziaren balioak ANALIZATU funtzioaren aplikaziokoa izango dira.

(DEFUN ANALIZATU (ESALDIA GRAMATIKA *HIZTEGIA*)

(DECLARE (SPECIAL *HIZTEGIA*))

(ANALIZATU-LAG ESALDIA GRAMATIKA))

Aldagai orokorrak aldagai berezien kasu partikularizat hartu daitezke, interpretatzailearen mailako parametrotzat hartuz. Beraz, DEFVAR ere erabili daiteke aldagai orokorrak definitzeko funtzioetatik aparte egiten bada eta balioa DEFVAREN bigarren parametroa izanik. Adibidez:

(DEFVAR *HIZTEGIA*

((ETXE (CAT IZENA))

(OGI (CAT IZENA))

(ZURI (CAT ADJEKTIBOA))

...

(EGON (CAT ADITZA))))

7.2. ASIGNAZIOA.

Aldagai-motak eta beren inplementazioak ikusi ondoren ikus dezagun asignazioaren inplementazioa. Hiru aukera dauzkagu asignazioak burutzeko: SETF, SETQ eta SET.

(SETF <aldagai> <balio>)

SETF ez du ebaluatzen bere lehenengo argumentua, aldagaiaren izena dena, baina bai bigarrena (asignatuko duen balioa adierazten duena).

BOKALEAK	→ Unbound variable
	; baliorik gabeko aldagaia
(SETF BOKALEAK '(A E I O U))	→ (A E I O U)
BOKALEAK	→ (A E I O U)
(REST BOKALEAK)	→ (E I O U)
BOKALEAK	→ (A E I O U)
(SETF BOKALEAK '(A E I O U Y))	→ (A E I O U Y)
BOKALEAK	→ (A E I O U Y)

Asignazio bat baino gehiago egin ahal da aldiberean:

```
(SETF L1 '(A) L2 '(A B) X (+ 23 33)) → 56
L1      → (A)
L2      → (A B)
X       → 56
```

(SETQ <aldagai> <balio>)

SETQ primitiboa (SET QUOTEren laburpena) SETF ordeztu daiteke baina asignazioa egiteko baina programatzaile gehienek SETF nahiago dute berau orokorragoa delako, sinboloetarako asignazioaz gain beste datu-moten asignazioetarako ere balio duelako.

(SET <aldagai> <balio>)

SETF bezalakoa da baina ez da forma berezia, beraz aldagaia adierazten duen argumentua ere ebaluatuko du.

```
(SETF L '(A B C)) → (A B C)
(SET (FIRST L) 7) → 7
A → 7
(SET (COND ((EQUAL X Y) 'B)
        (T 'C)))
15) → 15
```

Azken adibidean asignazio honetan 15 balioa B ala C aldagaiari asignatuko zaio X eta Yren balioen arauera.

Aldagaiak eta asignazioa direla eta, honako funtziook ere lagungarri gertatzen dira:

(BOUNDP <sinbolo>)

BOUNDP predikatuak T itzultzen du sinboloari baliorik badatxekio, bestela NIL.

(MAKUNBOUND <sinbolo>)

MAKUNBOUNDek sinboloa itzultzen du eta ondorio gisa aurrerantzean sinboloari ez datxekio lehengo balioa.

A	→ error : Unbound variable
(SETF A 27)	→ 27
A	→ 27
(MAKUNBOUND 'A)	→ A
A	→ error: Unbound variable

7.3. ITERAZIOZKO KONTROL-EGITURAK.

7.3.1. DO, DOTIMES eta DOLIST.

DO forma bereziak era guztietako iterazioak definitzeko aukera eskaintzen du. Bere sintaxia honelakoa da:

```
(DO ((<ald1> <has1> <errep1>) (<ald2> <has2> <errep2>) ..... )
    (<baldintza> <buk1> <buk2> ... <bukn>)
    <esp1>
    <esp2>
    ...
    <espx>)
```

Lehenengo argumentua zero edo gehiago osagai daukan lista da. Honen bidez DO formaren bertako aldagaiak azalduko dira. Lista honen elementu bakoitzean aldagai bakoitzeko bere izena (<aldj>), bere hasierako balioa (<hasj>) eta bigiztako errepikapen bakoitza bukatutakoan hartuko duen balioa (errepj) ematen dira. <errepj> delakoa agertzen ez bada, <aldj> aldagaiaren balioa ez da aldatuko ziklo bakoitza bukatu ondoren. <hasj> hasierako balio bat agertzen ez bada, NIL hartuko da hasierako baliotzat. Aldagaien balioen hasieraketa eta eguneraketa guztiak paraleloan burutzen dira. DO* forma berezia DOren antzekoa da, baina hasieraketak eta eguneraketak sekuentzialki egiten ditu.

DOren bigarren argumentua beste lista bat da. Honen lehenengo osagaiak bigiztatik ateratzeko baldintza adierazten du. Ziklo bakoitzaren exekuzioa hasi aurretik baldintza hori ebaluatzen da, eta emaitza NIL bada blokeko <esp1>, <esp2>, ... forma guztiak ebaluatzen dira. Bestela, baldintzaren emaitza NIL ez bada, <buk1>, <buk2>, ... formak ebaluatuko dira, DOren ebaluazioari bukaera emanez. DOren forma osoaren emaitza ebaluatutako azken <bukn> espresioaren emaitza izango da.

Blokeko formen artean RETURN funtzioari deitzen dion forma bat agertzen bada, berori exekutatzekoan DOren forma osoa bukatutzat emango da. Itzulitako emaitza RETURN formaren argumentuaren balioa izango da.

Adibideak:

n zenbaki positibo edo zero izanik, definitu m^n berreketa kalkulatu duen funtzio bat:

```
(DEFUN BERREKETA (M N)
  (DO ((EMAITZA 1)
      (BERRETZAILE 0))
    ((= BERRETZAILE N) EMAITZA)
    (SETF EMAITZA (* EMAITZA M))
    (SETF BERRETZAILE (1+ BERRETZAILE))))
```

Ikus dezagun bigarren bertsio bat RETURN erabiliz nahiz eta ez oso idazkera dotorea izan.

```
(DEFUN BERREKETA-2 (M N)
  (DO ((EMAITZA 1)
      (BERRETZAILE 0)
      (NIL)
      (IF (= BERRETZAILE N) (RETURN EMAITZA))
      (SETF EMAITZA (* EMAITZA M))
      (SETF BERRETZAILE (1+ BERRETZAILE))))
```

Edo modu trinkoago batez egina:

```
( DEFUN BERREKETA-3 ( M N )
  ( DO   ( ( EMAITZA 1 ( * EMAITZA M ) )
          ( BERRETZAILE 0 ( 1+ BERRETZAILE ) )
          ( (= BERRETZAILE N ) EMAITZA ) ) )
```

Beste adibide bat:

Definitu NIREREVERSE L lista bat alderantziz itzul dezan. Hau da, REVERSE funtzio aurredefinituaren baliokidea izan dadin.

```
( DEFUN NIREREVERSE ( L )
  ( DO (( EMAITZA NIL )
        (( NULL L ) EMAITZA )
        ( SETF EMAITZA ( CONS ( FIRST L ) EMAITZA ) )
        ( SETF L ( REST L ) ) ) ) )
```

Edo modu trinkoago batez egina:

```
( DEFUN NIREREVERSE-2 ( L )
  ( DO   ( ( EMAITZA NIL ( CONS ( FIRST LL ) EMAITZA ) )
          ( LL L ( REST LL ) ) )
        (( NULL LL ) EMAITZA ) ) )
```

DOren kasu partikularrak bezala DOTIMES eta DOLIST funtzioak agertzen zaizkigu. Bi forma hauek ere iterazioak eraikitzeke erabiltzen dira , baina eremu mugatuagoetan.

Horrela bada, iterazio bat aldi-kopuru finko batean egikaritu nahi dugunean DOTIMES forma erabiliko dugu. Bere egitura sintaktikoa hauxe dugu:

```
( DOTIMES ( <kontatzaillea> <goiko-muga> <emaitza> )
           <gorputza> )
```

DOTIMES forma batean sartzean, <goiko-muga> espresioa ebaluatu egiten da, zenbaki bat lortuz, adibidez N. Honela, <kontatzaillea> aldagaiari 0, 1,..., N-1 balioak atxekituko zaizkio bata bestearen atzetik, eta balio bakoitzarentzat <gorputza> egikarituko

da. Bukaeran, <kontatzailea> aldagaiaren atxekidura galdu egingo da, eta <emaitza> forma ebaluatuko da DOTIMES formaren balioa bueltatzeko.

DOTIMES forma batek <emaitza> formarik ez badu, NIL bueltatuko du, eta albo-ondorioak burutzeko eraiki dela suposatuko da.

Adibidea:

Jo dezagun berreketa kalkulatzeko funtzio bat DOTIMES erabiliz definitu nahi dugula:

```
(DEFUN BERREKETA (M N)
  (LET ((EMAITZA 1)) ; emaitza hasieratu
    (DOTIMES (KONT N EMAITZA) ; gorputza N aldiz
      (SETF EMAITZA (* EMAITZA M)))) ; M aldiz bidertu
```

DOLIST funtzioa DOTIMESen antzekoa dugu. Beraien arteko desberdintasuna aldagaiari zenbakiak atxekitu beharrean lista baten elementuak ematen zaizkiola dugu. Bere egitura sintaktikoa:

```
(DOLIST (<elementua> <lista> <emaitza>)
  <gorputza>)
```

DOLIST forman sartzerakoan, <lista> forma ebaluatu egiten da, elementu-lista bat lortuz. Ondoren, lortutako lista honen elementuak bata bestearen atzetik <elementua> aldagaiari esleitzen zaizkio, bakoitzarentzat gorputza ebaluatzen delarik. Bukatzeko, <elementua> aldagaiaren atxekidura galdu egiten da, eta <emaitza> forma ebaluatu egiten da, DOLISTen emaitza lortuz.

DOLISTek <emaitza> formarik ez badu, NIL bueltatuko du, DOTIMESek bezala.

Adibidea:

Suposa dezagun bi mugetik kanpo dauden zenbaki-lista baten elementuak kontatu nahi ditugula:

```
(SETF GOIKO-MUGA 35 BEHEKO-MUGA 10)

(DEFUN KONTATU-KANPOKOAK (LISTA)
  (LET ((EMAITZA 0))
    (DOLIST (ELEMENTUA LISTA EMAITZA)
      (WHEN (OR (> ELEMENTUA GOIKO-MUGA)
        (< ELEMENTUA BEHEKO-MUGA))
        (SETF EMAITZA (+ EMAITZA 1))))))
```

GOIKO-MUGA eta BEHEKO-MUGA sinboloek balio bat atxekituta eduki beharko dute KONTATU-KANPOKOAK funtzioa ebaluatu aurretik. Honela, funtzio honen erabilera bat:

(KONTATU-KANPOKOAK '(15 24 39 12 34 7)) → 2

7.3.3. LOOP eta PROG.

Errepikapenak egiteko beste bi forma orokorrago ditugu: LOOP eta PROG.

LOOP funtzioak gorputza bakarrik izango du, eta iterazio infinituak egiteko erabili ahal daiteke. Bere definizioa honelakoa da:

(LOOP <gorputza>)

Honela, forma honek <gorputza> behin eta berriz egikarituko du, honen barruan RETURN forma bat aurkitu arte. Kasu horretan iterazioa moztuko da eta LOOPek bueltatuko duen emaitza RETURN formak bueltatzen duena izango da. Adibide hau nahiko argia da:

```
(LET ((LISTA '(A B C D)))
      ((KONTA 0))
      (LOOP
        (WHEN (ENDP LISTA) (RETURN KONTA))
        (SETF LISTA (REST LISTA))
        (SETF KONTA (+ 1 KONTA)))) → 4
```

Ikusten denez, lista baten elementuak kontatzeko erabili dugu LOOP forma hori.

DO eta LOOP forma bereziak berritsuak dira. Hauek asmatuak izan ziren arte errepikapenak zehazteko bide bakarra PROG forma berezia zen. Gaur egunean PROGen erabilpena baztertzeko joera dago. Bere garrantzi historikoa dela eta, PROGen azalpena ere erantsiko dugu. Bere sintaxia honako hau da:

```
( PROG (<ald1> <ald2> (<ald3> <has3>) <ald4> (<ald5> <has5>) ..... ))
      < etiketa - edo - espresio 1 >
      < etiketa - edo - espresio 2 >
      .
      .
      .
      < etiketa - edo - espresio n > )
```

Lehengo argumentua lista bat da, bertako aldagaiak deskribatzen dituenak. Aldagaiak beren hasierako balioarekin definitu ahal dira. Hasierako baliorik azaltzen ez bada NIL dela suposatzen da. Hasieraketa hauek paraleloan burutzen dira. PROG* forma berezia PROGen antzekoa da, baina hasieraketak sekuentzialki egiten ditu.

Aldagaiak hasieratu ondoren, blokeko espresioak sekuentzialki ebaluatuz burutzen da PROGen ebaluazioa. Agertzen diren etiketak ez dira ebaluatzen, ondorengo espresiora jauzten delarik. PROGeko azken espresioa ebaluatutakoan bukatutzat ematen da PROGen ebaluazio orokorra eta NIL balioa itzultzen da.

Espresiozko sekuentziaren ebaluazio sekuentziala RETURN eta GO funtzioak erabiliz aldarazi daiteke.

(RETURN <balioa>)

RETURNek PROG osoaren ebaluazioaren bukaera dakar, emaitza gisa balioa itzuliz.

(GO <etiketa >)

GO forma bereziak PROGeko espresioen ebaluazio-ordena mozten du, ebaluatze espresioa PROG barruan etiketaren ondoan agertzen dena izango delarik.

Adibidea:

(DEFUN BERREKETA-4 (M N)

(PROG (EMAITZA BERRETZAILE)

(SETF EMAITZA 1)

(SETF BERRETZAILE 0)

BIGIZTA

(COND ((= BERRETZAILE N) (RETURN EMAITZA)))

(SETF EMAITZA (* M EMAITZA))

(SETF BERRETZAILE (1+ BERRETZAILEA))

(GO BIGIZTA)))

PROGen bidez idatzitako edozein forma DO erabiliz berridatzi daiteke. Egokiagoa da DOren erabilpena bere aurkezpena trinkoago, ordenatuago eta irakurgarriagoa delako.

DO erabiliz iterazioa kontrolatzeko behar den informazio guztia hasieran aurkezten da eta ez bestelako espresio guztiekin nahasturik. DOren bigarren argumentua bertako aldagaien lista da, PROGen bezalakoa, baina harek hasieratzeko informazioaz gain ziklo bakoitzeko balioa ere zehazten du. DOren bukaera-baldintza bakarra da eta aurkitzeko oso erraza.

Beraz eta gogor hitz eginik PROG forma berezia ez da beharrezkoa.

ARIKETAK.

- 7.1. Zenbaki baten faktoriala kalkulatzeko funtzio iteratibo bat definitu. Hiru bertsio eraiki, DO, DOTIMES eta LOOP erabiliz.

$$\text{Fact}(n) = \begin{cases} 1 & \text{baldin } n = 0 \\ n * \text{Fact}(n - 1) & \text{bestelakoetan} \end{cases}$$

- 7.2. REVERSE funtzioa era iteratiboan definitu DOLIST forma erabiliz
- 7.3. Suposa dezagun MEMBER funtzioa gure inplementazioan ez dagoela, eta horretarako MEMBER-ERREK funtzioa definitua dugula:

```
(DEFUN MEMBER-ERREK (ELEM L)
  (COND ((ENDP L) NIL)
        ((EQL ELEM (FIRST LISTA)) L)
        (T (MEMBER-ERREK ELEM (REST L))))))
```

Idatzi funtzio errekursibo honen bertsio iteratibo bat DO forma erabiliz.

8. PROGRAMAZIO-METODOLOGIA.

SARRERA

Programa batek izan behar dituen ezaugarri nagusiak zuzentasuna, eraginkortasuna eta irakurgarritasuna dira. Programa txikia baldin bada erraz lortu ahal ditugu hiru ezaugarriok batera inolako arreta berezirik gabe, baina programaren tamainua handitzen den heinean teknika edo metodologia bereziak erabili beharko ditugu. Kapitulu honetan, adibide oso sinpleak aurkezten badira ere, LISPez programatzeko erabiltzen diren metodologiak berri ematen da.

8.1. BILKETA PROGRESIBOA (progressive envelopment).

Teknika honen oinarria zera da: emaitza oso egituratu bat itzuli behar denean, lehenengo pausoa bezala ez da bilatzen zein funtzio nagusik emango duen halako emaitza. Alderantzizko prozesua erabili ohi da: lehenengoz emaitza osatuko duten elementu terminalak zehazten dira eta gero LISPeko funtzio primitiboen bidez progresiboki eraikitzen da bilatutako emaitza.

Adibidea:

Lista baten lehenengo eta azken elementuak edukiko dituen beste lista bat itzuliko duen MUTUR-BIAK funtzioa definituko dugu *bilketa progresibo* teknika hau erabiliz. Lehenengo eta behin defini dezagun funtzio honek argumentutzat hartu lezakeen lista bat:

(setf lista-osoa '(gosari bazkari askari afari))

MUTUR-BIAK funtzioari deituko bagenio *lista-osoa* argumentuarekin (GOSARI AFARI) emaitza lortu beharko genuke. Emaita hau eraikitzeko behar ditugun elementuak GOSARI eta AFARI dira.

Nola lortuko ditugu elementu horiek *lista-osoa* delako horretatik? Lehenengo eta azken elementuak direla jakinda, FIRST eta LAST funtzioen bidez erraz lortu ahal dira. Interpretatzailea bera erabil dezakegu asmatutako bideak frogatzeko:

(first lista-osoa) → GOSARIA

(first (last lista-osoa)) → AFARIA

Bitarteko emaitzak egokiak direla ikusita, balio biok LIST funtzioaz bilduz emaitza nagusia lortuko dugu. Berririo LISP erabiltzen dugu hipotesia egiaztatzeko:

```
(list (first lista-osoa)
      (first (last lista-osoa)))
→ (GOSARIA AFARIA)
```

Nahi genuena lortu ondoren, bildu egiten dugu asmatutako espresioa DEFUN erako forma batean *lista-osoa* parametrotzat hartuz:

```
(defun mutur-biak (lista-osoa)
  (list (first lista-osoa)
        (first (last lista-osoa))))
```

Bukatzeko azken egiaztapena:

```
(mutur-biak '(gosari bazkari askari afari)) → (GOSARIA AFARIA)
```

Bilketa progresibo teknikak aukera ematen du datu korapilotsuak eraikitzeko. Pausoz pausoka egiten da. Lehengo pausoan datu terminalak eskuratzen dira. Geroko pauso bakoitzean funtzio bat gehitzen da aurreko pausoetako emaitzak biltzeko. Pauso bakoitza eman ondoren egiaztatu egin ahal da. Orduan errorerik agertzen bada, momentuan bertan zuzendu daiteke, funtzio nagusi osoa definitu arte itxaron gabe.

Dena dela kontuz aukeratu behar dira hasierako adibideak, edozein argumentu posibleren ordezkari egoki izan daitezen. Bestela gerta liteke funtzioa ondo ibiltzea emandako adibiderako baina ez beste argumentu posible batentzat. Esate baterako, lehengo adibidean AFARI elementua ateratzeko (*fourth lista-osoa*) forma erabili bagenu, *mutur-biak-2* funtzioak kasu orokorrean emaitza okerra itzuli zukeen (beste lista bat lista argumentuaren lehenengo eta laugarren elementuekin), nahiz eta (GOSARI BAZKARI ASKARI AFARI) listarako emaitza egokia izan.

```
(defun mutur-biak-2 (lista-osoa)      ; ez da zuzena
  (list (first lista-osoa)
        (fourth lista-osoa)))
(mutur-biak-2 '(gosari bazkari askari afari)) → (GOSARIA AFARIA)
(mutur-biak-2 '(gosari hamaiketako bazkari askari afari))
→ (GOSARIA ASKARI)
Azken emaitza hau okerra da.
```

8.2. PROBLEMA-ERREDUKZIOA (problem reduction).

Funtzioak definitzeko teknika honen oinarria jatorrizko problema azpiproblemetan banatzea da. Azpiproblema bakoitza bere aldetik izango da tratatua, gehienetan funtzio desberdin bat erabiliz.

Problema-erredukzioa nola dabilen azaltzeko berriro hartuko dugu MUTUR-BIAK funtzioa. Aski artifiziala den ikuspuntu batetik ikusita, lau azpiproblema desberdin bereiz dezakegu: zer egin argumentua lista hutsa denean, elementu bakarreko lista denean, bi elementuko lista denean eta bi baino elementu gehiago duenean:

```
(defun mutur-biak (lista-osoa)
```

```
  (case (length lista-osoa)
```

```
    (0 ...) ;lista-hutserako tratamendua
```

```
    (1 ...) ;elementu bakarreko listarako tratamendua
```

```
    (2 ...) ;bi elementuko listarako tratamendua
```

```
    (t ...))) ;kasu orokorrerako tratamendua
```

Kasu-banaketa egin ondoren bakoitza bere aldetik ebatzi ahal da:

```
(defun mutur-biak (lista-osoa)
```

```
  (case (length lista-osoa)
```

```
    (0 NIL) ;lista-hutserako tratamendua
```

```
    (1 (list (first lista-osoa)
```

```
           (first lista-osoa)) ;elementu bakarreko listarako tratamendua
```

```
    (2 lista-osoa) ;bi elementuko listarako tratamendua
```

```
      ;bi elementuko listarako tratamendua
```

```
    (t (list (first lista-osoa)
```

```
           (first (last lista-osoa)))) ;kasu orokorrerako tratamendua
```

8.3. FUNTZIO-ABSTRAKZIOA (function abstraction).

Begira ezazu MUTUR-BIAK funtzioa definitzeko ondoko aukera hau:

```
(defun mutur-biak (lista-osoa) ; Funtzio-abstrakziozko bertsioa
```

```
  (bildu-elementuak ; Bildu lista batean 1.a eta azkena
```

```
    (lortu-lehenengo-elementua lista-osoa)
```

```
    (lortu-azken-elementua lista-osoa))))
```

MUTUR-BIAK era honetara definitzen dugunean, ez zaigu axola nola definituko diren *bildu-elementuak*, *lortu-lehenengo-elementua* eta *lortu-azken-elementua* funtzio lagungarriak. Jakin badakigu posible izango dela definitzea, baina MUTUR-BIAK definitzerakoan abstraditu egiten dugu funtzio lagungarrien betebeharra, aldezturik definituta egongo bailira.

Gero, MUTUR-BIAK funtzioa ontzat eman ondoren, funtzio lagungarrien definizioari ekingo diogu. Esate baterako:

```
(defun bildu-elementuak (elem1 elem2)
  (list elem1 elem2))
```

```
(defun lortu-lehenengo-elementua (lista)
  (first lista))
```

```
(defun lortu-azken-elementua (lista)
  (first (last lista)))
```

Edo beste modu batera egin liteke ere, adibidez:

```
(defun bildu-elementuak (elem1 elem2)
  (cons elem1
        (cons elem2 nil)))
```

```
(defun lortu-lehenengo-elementua (lista)
  (car lista))
```

```
(defun lortu-azken-elementua (lista)
  (first (reverse lista)))
```

Eta bestelako soluzio asko ere asma liteke. Soluzio posible guzti horien artean gehien interesatzen zaiguna hauta dezakegu: ulergarriena, dotoreena edo eraginkorrena. Beste alde batetik funtzio-abstrakzioari esker posible da talde-lanean programatzea, ez baita beharrezkoa beste funtzioak ezagutzea funtzio konkretu baten definizioa idatzi ahal izateko.

Ikuspuntu grafiko batetik esan dezakegu sortutako funtzioak geruza desberdinetan banatzen direla. MUTUR-BIAK funtzioa geruza bateko funtzio bakarra da. Beraren

funtzio lagungarriak azpiko geruzan kokatzen dira. Eta MUTUR-BIAK erabiltzen duen beste edozein funtzio gaineko geruzan dago. MUTUR-BIAK delakoaren definizioa idazteko ez dugu ezertarako jakin behar beste geruzetan egin diren definizioak edo hautaketak.

Funtzio lagungarrien izenak luzeak izaten dira. Programatzaile askok nahiago ohi ditu funtzio-izen luzeak iruzkinak baino. Jokaera horren arrazoia erraza da: nahiz bere definizioan iruzkinik eduki nahiz ez eduki, izen luzeko funtzio batek bere burua esplikatzen du erabiltzen den bakoitzean. Beraz izen luzeen erabilpenak programazioaren *azalpen automatiko* (automatic comenting) moduko zeozer dakar.

Funtzioak geruzetan antolatzen ditugunean *funtzio-abstrakzioaz* programatzen dugula esaten da eta geruzei *abstrakzio-geruzak* edo *abstrakzio-mailak* esaten zaie.

Funtzio-abstrakzio programazio-teknika izugarri ahaltsua da. Programa erraldoiak antolatzeko aukera ematen digu ondoko bi ezaugarri dituelako:

- Funtzio-abstrakzioak posible egiten du programazio beheranzkorra. Goi-maila hutsez pentsatu ahal dugu behe-mailako xehetasunak baztertuz. Goi-mailako programazioa egiten denean behe-mailakoa eginda dagoela suposatzen da. Geroago, goi-mailako programazioa bukatutakoan, orduan idatziko dira behe-mailako funtzioak.
- Funtzio-abstrakzioak laguntza ematen du funtzioen definizioak labur eta ulergarriak izan daitezen.

Errekurtsibitatea funtzio-abstrakzioaren kasu partikularra dela esan ahal da. Kasu honetan egiten den suposaketa aski bitxia da: idazten ari den prozedura bera dela aldeztu aurretik idatzita dagoena.

8.4. DATU-ABSTRAKZIOA.

Data-abstrakzio programazio-teknikaren helburua da programak idatzi ahal izatea bere datuak nola errepresentatu diren jakin beharrik gabe. Adibide baten bidez azalduko dugu.

Pertsonaren batek dauzkan liburuei buruzko informazio bibliografikoa erabiltzeko funtzioak asmatu nahi ditugu. Liburuei buruzko gorde nahi ditugun datuak hiru dira: izenburua, egilea eta sailkapena. Ezaugarri guzti hauek lista batean bilduta adieraziko ditugu:

(setf liburu-adibide-1

```
( ( (Artificial Intelligence)           ; Izenburua
  (Patrick Henry Winston)              ; Egilea
  (Adimen Artifizialeko teknika)))    ; Sailkapena
```

Liburua errepresentatzeko lista bat erabiliko da. Bere lehenengo elementua izenburu-aren hitzak biltzen dituen azpilista bat da. Bigarren elementua egilearen izena eta deiturak dauzkan beste azpilista bat da. Eta hirugarrena sailkapena daukan beste lista bat ere bai. Adierazpide hau aukeratuz gero, erraza da edozein datu hartzea, adibidez egilea:

(second liburu-adibide-1) → (PATRICK HENRY WINSTON)

Hala ere, adierazpide honek bi problema ekarriko ditu laster:

- Hemendik aurrera zehatz-mehatz gogoratu behar da nola gorde den datu bakoitza. Programa handietan ezinezkoa da gogoratzea nola gorde den datu bakoitza.
- Ez da erraza adierazpidea aldatzea. Listako hasieran beste datu berri bat gehitzeko erabakitzen badugu (argitaletxea adibidez), orain arteko idatzitako funtzio guztietan zuzendu beharko ditugu datuak lortzeko erreferentziak. Adibidez, egilea lortzeko lehengo (*second liburua*) erreferentzia guztiak (*third liburua*) formaz ordezkatu beharko dira.

Are gehiago, egilea lortzeko lehengo (*second liburua*) erreferentzia guztiak ez dira erraz topatzen. Sarritan gertatzen baita toki desberdinetan modu desberdinetan idatzi dela: (second liburua), (first (rest liburua)), (car (cdr liburua)) edo (cadr liburua)

Hala eta guztiz ere, demagun lista arrunten adierapidea hautatu dugula. Baina beste egun batean beste adierazpide egokiago bat hartu nahi genezakeen, esate baterako, asoziazio-listak.

Asoziazio-lista bat azpilistaz osatutako lista bat da. Azpilista bakoitzaren lehenengo elementua giltza gisa erabiltzen den sinbolo bat agertzen da. Sinbolo horri dagokion balioa azpilistako bigarren elementua da.

(setf liburu-adibide-2

```
( (izenburu      (Artificial Intelligence))
  (egile         (Patrick Henry Winston))
  (sailkapen     (Adimen Artifizialeko teknika))))
```

Asoziazio-lista baten azpilista bat ASSOC funtzioaren bidez lortzen da nahi dugun sinbolo giltza emanaz:

```
(assoc 'egile liburu-adibide-2) → (EGILE (PATRICK HENRY WINSTON))
(second (assoc 'egile liburu-adibide-2) → (PATRICK HENRY WINSTON))
```

Asoziazio-listen adierazpidera pasatzea posible liteke erabiltzaile bakarra baldin badago, baina programatzaile asko aplikazio berean badabil koordinazio-lan handia egin beharko litzateke.

Hile batzuk pasata, liburu asko mailegatzan dugunez, liburu mailegatu bakoitzari erantsi nahi diogu beste datu bi: *nori-mailegatua* eta *noiz-mailegatua*. Ez bide da modurik egokiena baina adibide hau karikatutara eramatearren, demagun gure adierazpide berrian bi asoziazio-lista batean bilduko ditugula:

```
(setf liburu-adibide-3
  '(((izenburu      (Artificial Intelligence))
    (egile          (Patrick Henry Winston))
    (sailkapen      (Adimen Artifizialeko teknika)))
  ( (nori-mailegatua (Halako Urlia))
    (noiz-mailegatua (89 iraila 1)))))
```

Adierazpide honekin datu bakoitza nola eskuratzen den jakiteko ... A zer nolako buru handi eduki behar den !!

Programatzailea bakarra izanez gero, lana ez da eroso izango hemendik aurrera, baina programatzaile asko aplikazio berean badabil bizitza erdia errore-araketan eman nahi ez badugu hobe dugu data-abstrakzio teknika erabiltzea.

Datu-abstrakzioak datuen adierazpidearen xehetasunak eskutatzea proposatzen du. Programatzaileak ez du erabiltzen azken errepresentazioa, baizik eta hiru funtzio-multzo: lehenengoa datuak eraikitze (eraikitzaileen multzoa), bigarrena datuak eskuratzeko (ikuskatzaileen multzoa) eta azkena datuak eguneratzeko (eguneratzaile edo aldatzaileen multzoa). Orokorki funtzio guzti horiei *atzipen-funtzioak* deitzen zaie.

Adibidez, demagun LIBURU-EGILE dela liburu baten egilea eskuratzeko erabiliko dugun funtzioa. Aplikazioari hasiera eman diogunean, lista arrunten adierazpidea aukeratzen dugunean LIBURU-EGILE honela definitzen dugu:


```
(defun liburu-egile (liburu)          ; datua lista da
  (second liburu))
```

Hona hemen beraren erabilpenaren adibidea:

```
(setf liburu-adibide-1
  '( (Artificial Intelligence)      ; Izenburua
    (Patrick Henry Winston)        ; Egilea
    (Adimen Artifizialeko teknika))) ; Sailkapena
```

```
(liburu-egile liburu-adibide-1) → (PATRICK HENRY WINSTON)
```

Gero, adierazpidea aldatuko den bakoitzean LIBURU-EGILE funtzioa egokitu egin beharko da. Adibidez, asoziazio-listaren adierazpidea hautatzen denean beste modu honetara definitu beharko da:

```
(defun liburu-egile (liburu)          ; datua asoziazio-lista da
  (second (assoc 'egile liburu)))
```

Beronen erabilpenaren adibidea:

```
(setf liburu-adibide-2
  '( (izenburu      (Artificial Intelligence))
    (egile          (Patrick Henry Winston))
    (sailkapen      (Adimen Artifizialeko teknika)))
(liburu-egile liburu-adibide-2) → (PATRICK HENRY WINSTON)
```

Eta geroago bi asoziazio-listetako adierazpidera jotzen badugu, egokipen berri bat egin beharko dugu:

```
(defun liburu-egile (liburu)          ; datua asoziazio-lista da
  (second (assoc 'egile (first liburu))))
```

eta LIBURU-EGILE funtzioaren azken bertsioaren erabilpena:

```
(setf liburu-adibide-3
  '(((izenburu    (Artificial Intelligence))
    (egile       (Patrick Henry Winston))
    (sailkapen    (Adimen Artifizialeko teknika)))
  ( (nori-mailegatua (Halako Urlia))
    (noiz-mailegatua (89 iraila 1)))))
```

(liburu-egile liburu-adibide-2) → (PATRICK HENRY WINSTON)

GARRANTZITSUA:

Adierazpidea aldatu dugun guztietan LIBURU-EGILE funtzioa aldatu behar izan dugu, baina ez dugu aldaketarik egin LIBURU-EGILE-ren erreferentzia egiten den tokietan. Talde-lanean programatzen badugu, inork ez du problemarik jasoko, areago ezta konturatuko ere ez da egingo aldaketaz. Gainera, bilatu nahi badugu programan non erabiltzen den liburu baten egilea, erraz burutuko dugu bilaketa aipatzeko modu bakarra baitago: LIBURU-EGILE. Laburbilduz, lehengo oztopo guztiak gainditu ditugu:

- Datu bakoitza nola errepresentatzen den jakiteko beharrik ez dago.
- Errepresentazioaren antolamendua erraz aldatu ahal da.
- Erraz bilatu ahal dira datuen erreferentziak
- Atzipen-funtzioak gogorazteko errazak dira. Datu bakoitza nola eskuratzen den jakiteko beharrik ez bait dago.

Beraz, datu-atzipenak egiterakoan LISPeke primitibo bat baino gehiago erabili behar denean, hobe legoke atzipen-funtzio multzoa espreski definitzea.

Orain arte funtzio ikuskatzaileez baino ez gara aritu, baina berdin-berdin esan daiteke datuak eraikitze eta eguneratzeko funtzioei buruz.

Adibidez, asoziazio-lista bakarra hartzen zuen adierazpidea aukeratuz gero liburu funtzio eraikitzailea hau da:

```
(defun sortu-liburu (izenburu egile sailkapena)
  (list (list 'izenburu izenburu)
        (list 'egile egile)
        (list 'sailkapena sailkapena)))
```

Hiru argumentu emanda liburu berri bat sortzen du:

```
(setf liburu-adibide-4
      (sortu-liburu '(Common Lisp)
                    '(Guy Steele)
                    '(LISP tekniko)))
→ ( ( IZENBURU (COMMON LISP))
    (EGILE (GUY STEELE))
    (SAILKAPENA (LISP TEKNIKO)))
```

Datu bat eguneratzeko funtzio adibide gisa LIBURU-EGILE-EGUNERATU funtzioa erakutsiko dugu. Honek elementu berri bat eranstean dio buruan asoziazio-listari. ASSOCek sinbolo-giltza duen lehenengo elementua itzultzen duenez, giltza berdina zuen lehengo azpilista ezin izango da ikusi aurrerantzean. Alferrik galtzen da memoria zati bat, baina askoz errazago eta arinagoa da tratamendua.

```
(defun liburu-egile-eguneratu (liburu egile)
  (cons (list 'egile egile) liburu))
```

Ondoan erakusten da nola aldatzen den lehengo LIBURU-ADIBIDE-4 delakoaren egilea Steele-ren bigarren izenaren iniciala sartzearren:

```
(setf liburu-adibide-4
      (liburu-egile-eguneratu liburu-adibide-4
                              '(Guy L Steele)))
→ ( (EGILE (GUY L STEELE))
    (IZENBURU (COMMON LISP))
    (EGILE (GUY STEELE))
    (SAILKAPENA (LISP TEKNIKO)))
```

```
(liburu-egile liburu-adibide-4) → (GUY L STEELE)
```

Nahi izanez gero posible da funtzio errekurtsibo bat definitzea egilearen lehengo balioa ezabatzeko:

```
(defun liburu-egile-eguneratu (liburu egile)
  (if (eql 'egile (first (first liburu)))
      (cons (list 'egile egile) liburu)
      (cons (first liburu)
            (liburu-egile-eguneratu (rest liburu) egile))))
```

Lehengo egile-zuzenketa bera baina bertsio errekurtsiboa erabiliz honela izango da:

```
(setf liburu-adibide-4
      (liburu-egile-eguneratu liburu-adibide-4
                              '(Guy L Steele)))
→ ( (EGILE (GUY L STEELE))
    (IZENBURU (COMMON LISP))
    (SAILKAPENA (LISP TEKNIKO)))
```

```
(liburu-egile liburu-adibide-4) → (GUY L STEELE)
```

Liburuen beste ezaugarrientzako funtzio-eguneratzaileak ere errazak dira, ikusi dugun *liburu-egile-eguneratu* funtzioaren antzerakoak

8.5. PAKETEA.

Paketeak oso tresna interesgarria dira kapitulu honetan azaldu diren lau metodologiaz baliatzeko. Problema handi batean funtzioen definizioak multzoka banatu ahal ditzakegu pakete desberdinetan. Paketeak independenteak dira euren artean. Hau da, pakete baten aldagai edota definizio gehienak ikustezinak dira beste paketeetatik. Pakete bakoitzean espezifikazio batez azaldu behar da zeintzu aldagai eta funtzio ikusi ahal izango dira beste paketeetatik (ingurunetik). Teknika honekin lortzen dugun abantaila nagusia zentzu honetan joango da: azpiproblema euren artean independenteak direnez, pertsona desberdinek ebatzi ahal izango dituzte azpiproblema desberdinak. Honela adibidez, azpiproblema bat ebazterakoan ez dugu kontutan hartu beharko beste azpiproblema ebaztean erabili ditugun identifikadoreak ez errepikatzea.

Paketeak erabiltzeko funtzioak bi ataletan aurkeztuko ditugu:

- Paketeak erazagutzeko funtzioak.
- Beste paketeetako zerbitzuak (funtzio edota aldagaiak) erabiltzeko funtzioak.

Paketeak erazagutzeko funtzioak

(IN-PACKAGE <pakete-izena>)

LISPeko programa bat kargatzean, funtzio eta aldagai guztiak USER pakete lehenetsian sartzen dira. IN-PACKAGE funtzio hau erabiliz, funtzio eta aldagaiak beste pakete batean kargatzeko esaten dugu. Forma hau fitxategi guztien hasieran erabiliko da. Adibidez:

(IN-PACKAGE 'DATU-BASE)

Sententzia hori fitxategi baten hasieran badago, fitxategi hori kargatzean topatzen diren sinbolo guztiak 'datu-base paketearen sartuko dira.

(PROVIDE <pakete-izena>)

Osatzen ari den paketea inguruneari erazagutzen dio. Forma hau ere fitxategi guztien hasieran erabiliko da IN-PACKAGEarekin batera. Adibidez:

(PROVIDE 'DATU-BASE)

(EXPORT <sinbolo-lista>)

Pakete batek eskeintzen dituen zerbitzuak esportatzeko. Forma honen bidez paketetik kanpo erabili ahal izango diren sinboloak erazagutzen dira. Sinbolo horiek eta ez besterik esportatzen dira paketetik. Adibidea:

(EXPORT '(IREKI-DB ITXI-DB IRAKUR-OBJEKTUA
EGUNERATU-OBJEKTUA))

Beste paketeetako zerbitzuak erabiltzeko funtzioak

(REQUIRE <pakete-izena>)

Forma honen bidez pakete baten zerbitzuak eskatzen dira, ondoren erabili ahal izateko.

LISPeko interpretatzaileak kargatu dituen paketeen izenak aldagai batean gordetzen ditu lista gisa. REQUIRE bat ebaluatzen denean aldagai horretan begiratzen da ea paketearen izena dagoen kargatutakoen artean, eta ez badago paketearen izen bera duen fitxategia kargatzen saiatuko da. Adibidez:

(REQUIRE DATU-BASE)

Pakete horretatik hartuko diren sinbolo esportatuak honela erreferentziatuko dira. Adibidez:

DATU-BASE:IREKI-DB
 DATU-BASE:ITXI-DB
 DATU-BASE:IRAKUR-OBJEKTUA
 DATU-BASE:EGUNERATU-OBJEKTUA

(USE-PACKAGE <pakete-izena>)

Goian azaldu dugun sinbolo esportatuak erreferentziazeko modua astun samarra da, paketearen izena beti errepikatu behar baita. USE-PACKAGE forma hau erabiliz gero, sinbolo esportatuak laburrago erreferentziatuko dira, paketearen izena idatzi gabe. Adibidez:

(USE-PACKAGE DATU-BASE)

eta orduan sinboloen erreferentziak honako hauek izango dira:

IREKI-DB
 ITXI-DB
 IRAKUR-OBJEKTUA
 EGUNERATU-OBJEKTUA

Adibideetan azaldu ditugun funtzioen deiak orokorrean honela bilduko lirateke:
 Suposa dezagun “datu-base” fitxategiaren edukina honako hau dela:

; PAKETEAREN EZAUGARRIAK

```
(IN-PACKAGE DATU-BASE) ; fitxategian definitzen den fpaketearen
                        ; izena DATU-BASE DA
(REQUIRE DATU-BASE)   ; paketea inguruneari erazagutu
(EXPORT '( IREKI-DB ITXI-DB IRAKUR-OBJEKTUA
          EGUNERATU-OBJEKTUA ))
                        ; paketeak eskeiniko dituen zerbitzuak
                        ; esportatu
```

; PAKETEAREN ZERBITZUEN DEFINIZIOA

```
(DEFUN IREKI-DB (...)  
    ...)  
(DEFUN ITXI-DB (...)  
    ...)  
(DEFUN IRAKUR-OBJEKTUA (...)  
    ...)  
(DEFUN EGUNERATU-OBJEKTUA (...)  
    ...  
    (F1 (F2 X Y))  
    ...)
```

; PAKETEAREN BARNEKO SINBOLOAK

```
(DEFUN F1 (...)  
    ...)  
(DEFUN F2(...)  
    ...)
```

Beste fitxategi batean ondoko erabilpen hau egin daiteke:

; BEHAR DIREN PAKETEA KARGATU

```
(REQUIRE 'DATU-BASE)           ; Kargatu paketea  
(USE-PACKAGE 'DATU-BASE)        ; Erreferentzien idazkera erraztu
```

; EBALUATZEKO FORMAK

```
(IREKI-DB)  
...  
(IRAKUR-OBJEKTUA X Y Z)  
...  
(IRAKUR-OBJEKTUA X1 Y1 Z1)  
...  
(ITXI-DB)  
...
```

8.6. PROGRAMAZIO-ESKEMAK.

Funtzio askoren helburua antzekoa denean eredu edo eskema orokor bat definitu ohi da beren definizio guztiak biltzearren. Horrela helburu bereko funtzio berri bat definitu

beharko denean nahikoa izango da eskema orokorrari funtzio berriaren ezaugarri bereziak egokitzea.

LISPeko eskema oso orokorrak honako hauek dira:

- Lista bat transformatzea.
- Predikatu bat betetzen duten elementuak lista batetik iragaztea.
- Predikatu bat betetzen duten elementuak lista batean kontatzea.
- Predikatu bat betetzen duten elementuak lista batean bilatzea

```
(DEFUN <transformazio-eskema> (<sarrera-lista>)
  (IF (ENDP <sarrera-lista>)
    NIL
    (CONS (<elementu-transformatzailea> (FIRST <sarrera-lista>))
      (<transformazio-eskema> (REST <sarrera-lista>)))))

(DEFUN <iragazketa-eskema> (<sarrera-lista>)
  (COND ((ENDP <sarrera-lista>) NIL)
        ((<predikatua> (FIRST <sarrera-lista>))
          (CONS (FIRST <sarrera-lista>)
                (<iragazketa-eskema> (REST <sarrera-lista>)))))
        (T (<iragazketa-eskema> (REST <sarrera-lista>)))))

(DEFUN <kontaketa-eskema> (<sarrera-lista>)
  (COND ((ENDP <sarrera-lista>) 0)
        ((<predikatua> (FIRST <sarrera-lista>))
          (+ 1 (<kontaketa-eskema> (REST <sarrera-lista>)))))
        (T (<kontaketa-eskema> (REST <sarrera-lista>)))))

(DEFUN <bilaketa-eskema> (<sarrera-lista>)
  (COND ((ENDP <sarrera-lista>) NIL)
        ((<predikatua> (FIRST <sarrera-lista>))
          (FIRST <sarrera-lista>))
        (T (<bilaketa-eskema> (REST <sarrera-lista>)))))
```

Baina eskema hauen erabilera hain handia denez LISPeK helburu horiexek dituzten funtzio primitiboak eskeintzen dizkio erabiltzaileari programazio-lana erraztearren. Funtzio hauek lehengo bezalako eskema errekurtsiboak baino eraginkorragoak izaten dira. Ondoren banan banan aztertuko ditugu, beraien erabilpena azalduz.

8.6.1. MAPCAR (listen transformazioa)

MAPCAR funtzio primitiboa listen transformaziorako aproposa dugu. Funtzio honek orokorrean bi argumentu hartzen ditu. Lehenengoa funtzio transformatzailea dugu, eta bigarrena transformatuko den lista. Bere sintaxia honelakoa dugu:

(MAPCAR <funtzioa> <lista>)

Forma honek beste lista berri bat ematen du. Lista berri honen osagaiak argumentu bezala eman zaion listaren elementuei funtzioa aplikatuz lortzen dira. Honela, lehengo adibidean:

$L \rightarrow (1\ 4\ 25\ 9\ 84)$

(MAPCAR #'SQRT L) $\rightarrow (1\ 2\ 5\ 3\ 8)$

Funtzio transformatzailearen izenaren aurretik #' ikurra jarri dugu. Horren bidez sinbolo batetik funtzio-objektu bat lortzen dugu (gogoratu sinbolo batek izena, balioa, funtzio bat eta propietate-lista bat eduki ditzakeela).

Funtzio transformatzaileak argumentu bat baino gehiago behar izango balitu, MAPCARek funtzioaz gain beste horrenbeste lista argumentu beharko lituzke. Honelakoetan lista emaitzaren n-garren osagaia lista argumentuen n-garren osagaiei funtzioa aplikatuz lortzen da. Adibidez:

(MAPCAR #'> '(1 7 3) '(5 2 0)) $\rightarrow (NIL\ T\ T)$

(1 7 3)

(5 2 0)

(NIL T T)

(> 1 5) $\rightarrow NIL$

(> 7 2) $\rightarrow T$

(> 3 0) $\rightarrow T$

LAMBDA funtzioa sarritan erabiltzen da MAPCARekin batera. MAPCAR forma batean erabili nahi dugun funtzio aldatzailea horretan bakarrik erabiltzekoa bada ez dago funtziorako izena asmatu beharrik. Honen adibide bat:

Suposa dezagun zenbakizko lista batetik abiatuz beste lista bat definitu nahi dugula, non osagai bakoitza lehengo listako osagai bat, bere karratua eta bere erro karratua baitira. Horretarako erabili dezakegun forma bat hauxe dugu:

```
(MAPCAR #'(LAMBDA (X) (LIST X (* X X) (SQRT X)))
      '(1 4 25 9))
→ ((1 1 1) (4 16 2) (25 625 5) (9 81 3))
```

Dena den, LAMBDA eta MAPCAR funtzioen beste erabilpen orokorrago bat ba dugu. Kasu askotan lista baten elementu guztiei argumentu bat baino gehiago hartzen duen funtzio bat aplikatu behar izaten zaie, gainontzeko argumentuak listako elementu guztien aplikazioetan berdinak direlarik. Horretarako gure programak honelako egitura hartu dezake :

```
(MAPCAR #'(LAMBDA (X)
              (<funtzio> X Y ... Z))
      LISTA)
```

Erabilpen hau adibide batez erakustearren, suposa dezagun bi zenbaki-multzo ditugula listen bidez adierazirik, eta bata bestearen barruan dagoen ala ez aztertu nahi dugula. Horretarako ondorengo funtzio hau definitu dezakegu:

```
(DEFUN BARNEAN-DAGO (AZPIMULTZO MULTZO)
  (NOT (MEMBER NIL
                (MAPCAR #'(LAMBDA (X)
                                (MEMBER X MULTZO))
                        AZPIMULTZO))))

(BARNEAN-DAGO '(1 3) '(0 1 7 3 2)) → T

(BARNEAN-DAGO '(1 4) '(0 1 7 3 2)) → NIL
```

8.6.2. Listen iragazketa, elementuen kontaketa eta bilaketa.

Listen transformaziorako MAPCAR forma daukagunaren antzera, listen iragazketarako REMOVE-IF funtzio primitiboa agertzen zaigu. Honen egitura sintaktikoa honelakoa dugu:

```
(REMOVE-IF <predikatua> <lista>)
```

Argumentu bezala funtzio predikatu bat eta lista bat hartzen ditu. Forma honen helburua, predikatua betetzen duten listako elementuak kentzea izango da, emaitza horrekin lista berri bat bueltatuko duelarik. Ondoko adibidearekin ikusiko dugu:

$$(\text{REMOVE-IF } \#'\text{NUMBER-P } '(A\ 1\ B\ 2\ 3\ C)) \rightarrow (A\ B\ C)$$

REMOVE-IFek badu bere alderantzizkoa: REMOVE-IF-NOT funtzioa. Azken honek egitura sintaktiko bera dauka, eta bere helburua predikatua betetzen ez duten elementuak listatik kentzea da:

$$(\text{REMOVE-IF-NOT } \langle \text{predikatua} \rangle \langle \text{lista} \rangle)$$
$$(\text{REMOVE-IF-NOT } \#'\text{NUMBERP } '(A\ 1\ B\ 2\ 3\ C)) \rightarrow (1\ 2\ 3)$$

Lista baten osagaien kontaketa eta bilaketa aztertzerakoan COUNT-IF eta FIND-IF funtzio primitiboak agertzen zaizkigu. Funtzio hauek predikatu bat betetzen duten lista baten elementuak kontatzeko eta bilatzeko ditugu. Beraien egitura sintaktikoa REMOVE-IFen berdina dugu.

Honela, atomo-lista batean zenbat zenbaki dauden kontatzeko COUNT-IF forma hau erabil dezakegu:

$$(\text{COUNT-IF } \#'\text{NUMBERP } '(A\ 1\ B\ 2\ 3\ C)) \rightarrow 3$$

FIND-IF funtzioari dagokionez, predikatua betetzen duen listaren lehenengo elementua emango digu:

$$(\text{FIND-IF } \#'\text{NUMBERP } '(A\ 7\ B\ 2\ 3\ C)) \rightarrow 7$$

8.7. PROGRAMAZIO ONERAKO ARAUAK.

Konputagailu-programazioan errore-araketa eta programen mantentzea oso neketsu eta garestiak dira. Beraz, programak idazterakoan oso komenigarria da errorerik ez sartzea. Egia esan, utopikoa da lehenengo idazketa batean programa lortzea, baina *programazio onerako arau* orokor batzuri jarraituz gero errore gutxitako programak sortuko dira, errore-araketa errazago izango da eta programak ulergarriagoak.

Adibide baten bidez azalduko ditugu arauok. Zenbait bloke jaso eta eraman dezakéen errobot batentzako prozedura bat azalduko dugu. Hasieran apropos gaizki idatziko dugu programazio txarrenaren erakuspena izan dadin.

```
(defun jarri-gainean (arg1 arg2)
  (and (ekintza1 arg1)
        (ekintza2 arg1 arg2)
        (ekintza3 arg1)))
```

Definizio honetan ez dago ezer ulertzerik. Egin nahi duena da bloke bat beste baten gainean jartzea. Zeozer ulertzeko lehenengo arau bat hartuko dugu kontutan:

- Eskuzabal izan zaitzez oharrak idazten. Idatz ezazu funtzioaren espezifikazioa eta sartu azalpenak definizio barruan ere bai.

```
(defun jarri-gainean (arg1 arg2)
  ; Bloke bat beste baten gainean jartzen du
  ; ARG1 eta ARG2 blokeak dira
  ; Hiru ekintzak ondo burutzen badira, bukaeran ARG1
  ; ARG2 gainean egongo da.
  (and (ekintza1 arg1) ;Heldu objektuari
        (ekintza2 arg1 arg2) ;Eraman ARG1 ARG2-rengana
        (ekintza3 arg1))) ;Askatu
```

Definizioa hobeto ulertzeko badago beste aukera bat:

- Funtzio eta sinboloei izen esanguratsuak eman.

Horrela, bada, definizioak berak azaltzen du bere burua:

```
(defun jarri-gainean (zein-bloke non-bloke)
  (and (heldu zein-bloke)
        (eraman zein-bloke non-bloke)
        (askatu zein-bloke)))
```

Izen esanguratsuek oharrekin konparatzen baditugu abantaila bat dute: izena funtzioa edo sinboloa erabiliko den toki guztietan ageriko dela eta ez soilik bere definizioan oharrekin gertatzen den bezala.

Hala ere, izen esanguratsuak eta oharrak erabiliz gero, argitasun maximoa lortuko dugu:

(defun jarri-gainean (zein-bloke non-bloke)

; Bloke bat beste baten gainean jartzen du

; zein-bloke eta non-bloke blokeak dira

; Hiru ekintzak ondo burutzen badira,

; bukaeran zein-bloke non-bloke gainean egongo da.

(and (heldu zein-bloke)

;Heldu objektuari

(eraman zein-bloke non-bloke)

;Eraman zein-bloke

; non-bloke gainera

(askatu zein-bloke)))

;Askatu

Hona hemen lehenago kontutan hartu diren beste hiru erregela:

- Idatzi funtzio laburrak.
- Idatzi argumentu gutxitako funtzioak
- Idatzi funtzio bakoitza helburu bat betetzeko

Hiru arau hauek betez gero programak erabat ulergarriagoak dira, momentu bakoitzean kontu gutxi izango bait dugu buruan.

Ondoko arauak babesa eman nahi die funtzioei berentzat pentsatuak izan ziren baldintzetatik kanpo erabiliak izan ez daitezen.

- Funtzio bat idazterakoan ahalik eta baldintza gutxien suposatu funtzioari deitzeko egoerei buruz. Egin ahal diren suposaketak inplizitoki jarri.

Gure adibidean suposatu behar da lekuri egongo dela *non-bloke* gainean. Nola bait baldintza hori inplizito bihurtzeko LEKU-HARTU funtzio lagungarria erabiliko dugu. Leku aurkitzerik ez badago NIL itzuliko du.

(defun jarri-gainean (zein-bloke non-bloke)

; Bloke bat beste baten gainean jartzen du lekuri badago

; zein-bloke eta non-bloke blokeak dira

; Hiru ekintzak ondo burutzen badira,

; bukaeran zein-bloke non-bloke gainean egongo da.

(when (leku-hartu zein-bloke non-bloke)	;Lekurik badago
(and (heldu zein-bloke)	;Heldu objektuari
(eraman zein-bloke non-bloke)	;Eraman zein-bloke
	; non-bloke gainera
(askatu zein-bloke)))	;Askatu

Ondoko araua ere erabiltzeko baldintzei buruzkoa da.

- Funtzio bat bere erabiltzeko baldintzetatik kanpo erabili nahi bada, mezu bat ageri beharko litzateke arrazoia erakusteko.

Gure adibidean leku aurkitzerik ez badago, NIL itzultzeaz gain porrotaren berri emango du:

```
(defun jarri-gainean (zein-bloke non-bloke)
  ;Bloke bat beste baten gainean jartzen du lekurik badago
  ;zein-bloke eta non-bloke blokeak dira
  ;Hiru ekintzak ondo burutzen badira,
  ; bukaeran zein-bloke non-bloke gainean egongo da.
```

(if (leku-hartu zein-bloke non-bloke)	;Lekurik badago
(and (heldu zein-bloke)	;Heldu objektuari
(eraman zein-bloke non-bloke)	;Eraman zein-bloke
	; non-bloke gainera
(askatu zein-bloke)))	;Askatu
(format t "&parka, ez dago lekurik ~a -rentzat ~a gainean."	
zein-bloke	; bestela, azaldu eta
non-bloke)	; itzuli nil

Honaino ikusi ditugun programazio onerako arauak funtzio bakar bat idazteari buruzkoak izan dira, baina aplikazio handiak antolatu behar ditugunean badira beste hiru arau nagusi:

- Erabili funtzio-abstrakzioa.
- Erabili datu-abstrakzioa.

- Antolatu programa osoa modulutan. Modulu bakoitzean sartu elkarren artean lotura handia duten funtzioak, multzo bat osatuz. Moduluen arteko harremanak nuluak ez badira, ahalik eta klaruen zehaztu.

8.8. ERRORE ARAKETA (Debugging).

Nahiz eta arreta handiaz bete programazio onerako erregelak, beti ageriko da errore nahaskanteren bat. Erroreen bilaketa eta araketa errazteko asmoz bi funtzio primitibo nagusi aurkezten ditu LISPeK: TRACE eta STEP.

TRACE primitiboak funtzio baten izena argumentutzat hartzen du. Horrelako forma bat ebaluatu ondoren, argumentu bezala eman den funtzioari deitzen zaion aldi bakoitzean argumentuak eta emaitza erakutsiko ditu LISPeK.

Adibidez, ondoan aurkezten den ATOMO-KOPURU funtzioak espresio batean zenbat atomo dagoen kontatu nahi ditu, baina oker egiten du:

```
(DEFUN ATOMO-KOPURU (ESPRESIO)
  (IF (ATOM ESPRESIO)
    1
    (+ (ATOMO-KOPURU (FIRST ESPRESIO))
      (ATOMO-KOPURU (REST ESPRESIO)))))

(ATOMO-KOPURU '((PROBA BAT) (EGINGO DUGU))) → 7
```

Ebaluazio horrek 7 ematen du, baina guk 4 espero genuen.

Gauzak honela, errorea bilatzen laguntzeko TRACE primitiboa erabiliko dugu. ATOMO-KOPURU funtzioaren dei bakoitzean argumentuak eta emaitzak erakusteko eskatzen diogu:

```
(TRACE ATOMO-KOPURU) ; ohar zaitez: quote-rik ez da erabiltzen!
```

Eta orain lehengo proba berriz ebaluatzen dugu :

```
(ATOMO-KOPURU '((PROBA BAT) (EGINGO DUGU))) →
ENTERING: ATOMO-KOPURU, ARGUMENT-LIST : (((PROBA BAT)
(EGINGO DUGU)))
ENTERING: ATOMO-KOPURU, ARGUMENT-LIST : ((PROBA BAT))
```

```

ENTERING: ATOMO-KOPURU, ARGUMENT-LIST : (PROBA)
ENTERING: ATOMO-KOPURU, VALUE : 1
ENTERING: ATOMO-KOPURU, ARGUMENT-LIST : ((BAT))
    ENTERING: ATOMO-KOPURU, ARGUMENT-LIST : (BAT)
    ENTERING: ATOMO-KOPURU, VALUE : 1
    ENTERING: ATOMO-KOPURU, ARGUMENT-LIST : (NIL)
    ENTERING: ATOMO-KOPURU, VALUE : 1

```

Dena ondo zebilen ATOMO-KOPURU lista hutsarekin deitu arte, orduan emaitza 0 izan beharrean 1 izan da. NIL lista eta atomo bezala hartu daitekeenez, ATOM predikatuak beste sinbolo arrunta bat bezala ikusi du.

ATOMO-KOPURU funtzioa era honetara zuzendu dezakegu:

```

(DEFUN ATOMO-KOPURU (ESPRESIO)
  (COND ((NULL ESPRESIO) 0)
        ((ATOM ESPRESIO) 1)
        (+ (ATOMO-KOPURU (FIRST ESPRESIO))
            (ATOMO-KOPURU (REST ESPRESIO))))))

```

Berriro proba eginda, emaitza egokia dela ikusi ahal izango da.

Aurrerantzean ATOMO-KOPURU funtzioa erabiltzerakoan bere argumentuak eta emaitzak ikusi nahiko ez ditugunez, indarrrik gabe utzi beharko dugu TRACE delako hori.

```
(UNTRACE ATOMO-KOPURU)
```

```
(ATOMO-KOPURU '((PROBA BAT) (EGINGO DUGU))) → 4
```

TRACE eta UNTRACE funtzio bat baino gehiagorekin erabil daitezke. UNTRACE argumenturik gabe erabiltzen bada, funtzio guztien traza ateratzeko beharra geldiarazten da.

STEP primitiboari dagokionez ez da erabiltzen TRACE bezainbatetan, baina zenbait errore latz aurkitzeko oso lagungarria da. Funtzio primitibo honi esker posiblea da interpretatzioari pausoz pausoka jarraitzea. TRACEk baino maila bajuago batetan lan egiten du, eta horregatik LISPeo inplementazio desbesdinetan oso desberdinak izaten dira.

Orokorrean honela lan egiten du: funtzio elkareragilea da, eta pauso bakoitzean erabiltzailearen erantzuna itxaroten du (exekuzioa pausoz pauso egiten duenez, pauso bakoitzaren ondoren erabiltzaileari itxaroten dio). Erabiltzaileak pauso bakoitzean zenbait aukera izango ditu:

- Hurrengo forma pausoka ebaluatzea.
- Hurrengo forma osorik ebaluatzea (pausoka egin gabe).
- Hurrengo forma eta bere maila berekoak diren hurrengoak ere osorik ebaluatzea (pausoka egin gabe)

Adibide gisa saia zaitez ebaluatzen zeure LISPeZ ondoko forma hau:

(STEP (ATOMO-KOPURU NIL))

Laburbilduz zera esan dezakegu: TRACERen bidez funtzio baten sarrerako datuak eta emaitzak ikusi ditzakegu, dei desberdinak aztertuz, eta STEPen bidez funtzioaren gorputz barruan sar gaitezke, pausoz pauso egikarituz.

Errore-araketan gutxiago erabiltzen diren beste bi funtzio BREAK eta TIME ditugu.

BREAK primitiboa funtzio baten definizio barruan erabiltzen da. Ebaluazio batean (BREAK "16. errorea") forma ebaluatzen denean, karaktere-katea idatzi ondoren gelditu egiten da ebaluazioa. Une horretan programatzailcak aukera du edozein forma ebaluatzeko, sinboloen balioak begiratzeko edo sinbolo bati balio bat lotzeko. Honelako eragiketak egin ondoren ebaluazioa aurrera eramateko aukera dago tekla bat zakatzuz, baina tekla desberdina izaten da inplementazio desberdinetan (CTRL-C , CTRL-P , ...?).

TIME funtzioa ebaluazio batek irauten duen denbora kontatzeko dugu. Honela, gure funtzioen eraginkortasuna neurtzeko erabil dezakegu.

Aztertu ditugun funtzio guztiak COMMON LISPeakoak ditugu. Funtzio hauek inplementazioaren arabera aldaketarik jasaten dute, gehienbat erabiltzaileari ateratako mezuetan.

Funtzio hauetaz gain inplementazio desberdinek errore-araketarako tresna bereziak izaten dituzte, "Debugger" izeneko aplikaziotan biltzen direlarik. Hauen artean sarritan agertzen den tresna bat "backtrace" izenekoa dugu. Tresna honekin, errore bat gertatu ondoren ebaluatu gabe gelditu diren LISP forma pilaratuak ikusi ditzakegu, parametro baten bidez azken N formak ikusi ahal direlarik.

Ondorio gisa, errore-araketa aztertzerakoan azaltzen zaizkigun tresnak bi ataletan banaturik daudela esan dezakegu:

- LISPe ematen dizkigun funtzioak, inplementazioaren arabera portaera desbesdinak izaten dituztelarik (TRACE, STEP, ...).
- Inplementazioak eskeintzen dizkigun tresnak ("backtrace", ...).

Ikusten denez inplementazioaren eragina nabarmena dugu tresna guztietan.

ARIKETAK.

- 8.1. Lista edo atomo izan daitekeen s-espresio batetan zenbat atomo agertzen diren kontatuko duen funtzio bat definitu. MAPCAR funtzioa erabili, s-espresioaren osagai bakoitzak zenbat atomo dituen kalkulatzeko.
- 8.2. Lista edo atomoa izan daitekeen s-espresio baten sakonera-maila itzuliko duen funtzio bat definitu. Erabili MAPCAR funtzioa.
- 8.3. Definitu BATULISTAK funtzioa ondoko erara: bere argumentu bakarra zenbakizko listaz osatutako lista bat izango da, eta bere emaitza beste lista bat lehengo azpilista bakoitzaren baturarekin.
- 8.4. Lista bat eta atomo bat emanda, listan atomoa zenbat aldiz agertzen den kontatuko duen KONTATOMOA funtzioa definitu. Erabili MAPCAR funtzioa.
- 8.5. Bi lista argumentu bezala hartzen dituen FMULTIPLE funtzioa definitu. Bi listek elementu kopuru bera izango dute. Lehenengoaren elementuak funtzioen izenak izango dira, eta bigarrenarenak lehenengo listako funtzioentzako argumentuak dituzten azpilistak izango dira. Definitu beharreko funtzioak lista bat bueltatuko du, honen elementuak funtzio bakoitza dagokien argumentuei (bakoitza ebaluatu ondoren) aplikatzerakoan lortuko direlarik.

Adibidez:

```
(FMULTIPLE '(+ FIRST NTH)
              '((2 3) ('(A B)) (3 (APPEND '(2 3) '(5 6)))))
→ (5 A 6)
```

- 8.6. Azter ezazu honako funtzio hau:
- a) Zer itzuliko du XXX funtzioak argumentutzat edozein atomo, edozein lista eta aukerazko zenbaki natural bat hartzen baditu?
- b) Zein da ondoko espresioen ebaluazioen emaitza?

```

(XXX 1 '(1 2 3))
(XXX 'A '(1 2 (A B) (X (A B))))
(XXX 'A '(1 2 (A B) (X (A B))) 2)
(XXX 'A '(1 2 (X Y Z) (M (N O))))

(DEFUN XXX (A L &optional (N 1))
  (COND (ENDP L) 0)
        (T (LET ( (B (MEMBER A L))
                  (Z (APPLY #'APPEND
                          (MAPCAR #'(LAMBDA (X)
                                      (IF (ATOM X)
                                          NIL X))
                                L))))
          (IF B (MAX K (XXX A Z (1+ K)))
              (XXX A Z (1+ K)))))))

```

8.7. Azter ezazu honako funtzio hau:

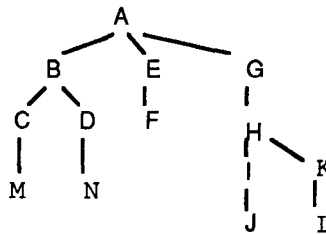
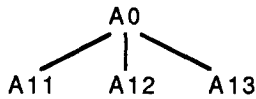
```

(DEFUN XZUHAITZ (EL1 EL2 ZUH)
  (COND ((AND (ADABEGI EL1 ZUH)
              (ADABEGI EL2 ZUH))
        (LET ((XX (APPLY #'APPEND
                          (MAPCAR #'(LAMBDA (ZUH1)
                                      (XZUHAITZ EL1 EL2 ZUH1))
                                (REST ZUH)))))
          (COND ((NULL XX) (LIST (FIRST ZUH)))
                (T (LIST (FIRST XX))))))
        (T NIL)))

```

- a) Zuhaitz bat adierazteko lista bat erabiltzen dugu, non adabegi erroa listaren lehenengo elementua eta umeak gainontzeko elementuak bait dira (umeak ere zuhaitz bezala adierazten dira noski). ADABEGI predikatuak T itzultzen du lehenengo argumentua (atomo bat) bigarrenaren (zuhaitz bat) adabegia bada. Zer itzuliko du XZUHAITZ funtzioak argumentutzat edozein zuhaitz hartzen badu?

b) Zein da ondoko espresioen ebaluazioen emaitza?



```

(SETF Z '(A (B (C (M))
              (D (N)))
          (E (F))
          (G (H (I (J))
                (K (L))))))

(XZUHAITZ 'A11 'A0 '(A0 (A11) (A12) (A13))))
(XZUHAITZ 'A11 'A12 '(A0 (A11) (A12) (A13))))
(XZUHAITZ 'B 'J Z)
(XZUHAITZ 'I 'L Z)
(XZUHAITZ 'B 'N Z)
  
```

c) Definitu ADABEGI funtzio-predikatua.

8.8. Azter ezazu honako funtzio hau:

```

(DEFUN XZUHAITZ (ZUH)
  (COND ((NULL ZUH) NIL)
        ((= (LENGTH ZUH) 1) (LIST (FIRST ZUH)))
        (T (APPLY #'APPEND (MAPCAR #'XZUHAITZ
                                     (REST ZUH))))))
  
```

a) Zuhaitz ez-huts bat adierazteko lista bat erabiltzen dugu. Listaren lehenengo elementua adabegi erroa da eta besteak umeak, hauek ere zuhaitzak direlarik. Deskriba ezazu zer itzultzen duen *xzuhaitz* funtzioak argumentutzat edozein zuhaitz ez-huts hartuz gero.

- b) Zeintzu dira ondoko espresioen ebaluaketaren emaitzak?

(XZUHAITZ '(A0 (A11) (A12) (A13)))

(XZUHAITZ '(A (B (C) (D)) (E (F)) (G (H (I (J))))))

- 8.9. Azter ezazu honako funtzio hau:

```
(DEFUN XZUHAITZ (N A)
```

```
  ;N > 0
```

```
  (COND ((NULL A) NIL)
```

```
        ((= N 0) (LIST (FIRST A)))
```

```
        (T (APPLY #'APPEND
```

```
              (MAPCAR #'(LAMBDA (Y)
```

```
                ..... (XZUHAITZ (1- N) Y))
```

```
              (FIRST A))))))
```

- a) Zuhaitz bat adierazteko lista bat erabiltzen dugu, non adabegi erroa listaren lehenengo elementua eta umeak gainontzeko elementuak bait dira (umeak ere zuhaitz bezala adierazten dira noski). Zer itzuliko du XZUHAITZ funtzioak argumentutzat edozein zuhaitz hartzen badu?

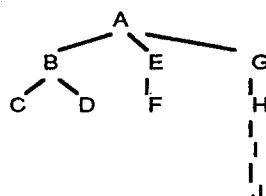
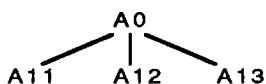
- b) Zein da ondoko espresioen ebaluazioen emaitza?

(XZUHAITZ 1 '(A0 (A11) (A12) (A13)))

(XZUHAITZ 2 '(A0 (A11) (A12) (A13)))

(XZUHAITZ 2 '(A (B (C) (D)) (E (F)) (G (H (I (J))))))

(XZUHAITZ 3 '(A (B (C) (D)) (E (F)) (G (H (I (J))))))



- c) Definitu AURREORDENA funtzioa. Funtzio honek arbola bat emanda bere aurreordenezko korritzea itzultzen du.

9. INFORMAZIO-EGITURAK.

LISPeke eskeintzen dizkigun oinarritzko datu-motak bigarren kapituluaz aztertzen genituen. Horietaz aparte, LISPeke beste datu-mota konplexuago batzu ere onartzen ditu. Kapitulu honetan datu-mota berri hauek azalduko ditugu. Konkreterik elkartzeko-lista, propietate-lista, array eta egitura datu-motak sakonki ikusiko ditugu, baraien funtzio eraikitzaile, ikuskatzaile eta aldatzaileen erabilpena aztertuz.

Dena den, COMMON LISPeke badira beste datu-mota egituratu batzu, adibidez objektuei zuzendutako programazioan klaseak adierazteko erabiltzen diren datu-motak. Testu honetan azken hauek ez ditugu azalduko, bere deskribapen zehatza nahi izanez gero, jo [Winston 89] erreferentziara.

9.1. ELKARTZE-LISTAK.

Elkartze-lista bat azpelistaz osatutako lista bat da. Azpilista bakoitzeko lehenengo elementua giltza-modura erabiltzen da azpilista osoa eskuratu ahal izateko. Adibidez:

ANE → ((pisu 59) (adina 23) (altuera 1.70))

Elkartze-lista hau ANEren ezaugarriak adierazteko erabil daiteke. PISU, ADINA eta ALTUERA sinboloak ezaugarriak adierazteko erabiltzen dira. Ezaugarri hauen balioak eskuratzeko ASSOC funtzio primitiboa daukagu.

(ASSOC <giltza> <elkartze-lista>)

ASSOC funtzioa aplikatuz elkartzeko-lista batean giltza bati dagokion azpilista lortzen da. Gure adibidean ezaugarri-giltza bati dagokion azpilista lortuko dugu:

(ASSOC 'ADINA 'ANE) → (ADINA 23)

Azpilista batek baino gehiagok giltza berdina edukiko balu, lehenengoa itzuliko litzateke, atzeko guztiak ikustezinak izango direlarik.

ANE → ((pisu 59) (adina 23) (altuera 1.70) (adina 22))
 (ASSOC 'ADINA 'ANE) → (ADINA 23)

Giltza elkartze-listan azalduko ez balitz, ASSOC funtzioak NIL bueltatuko du, adibidez:

(ASSOC 'HELBIDEA 'ANE) → NIL

9.2. PROPIETATE-LISTAK.

Sinboloak definitzerakoan, beraien lau ezaugarri posible aipatzen genituen: izena, balioa, funtzio bat eta propietate-lista. Atal honetan propietate-listetaz arduratuko gara.

Sinbolo batek balio arruntak izateaz aparte, propietate-balioak ere eduki ditzake. Propietate-balio hauek era honetako listatan egituratzen dira:

((<propietate-izena1> . <balio1>) (<propietate-izena2> . <balio2>) ...)

Honela balio terminoa bi eratara erabili dezakegu: alde batetik sinboloaren balioaz hitzegiten dugu, eta bestetik sinboloaren propietate konkretu baten balioaz.

Gauzak honela, propietateak maneiatzeko LISPeK eskeintzen dituen funtzio primitiboak aztertuko ditugu, hau da funtzio eraikitzaile, ikuskatzaile eta aldatzaileak.

Bitxia badirudi ere, propietateak eraiki edo beren balioak aldatzeko propietateen balioak irakurtzeko daukagun funtzioaz baliatu behar gara. Horregatik funtzio ikuskatzailea aztertuko dugu lehendabizi.

(GET <sinboloa> <propietate-izena>)

GET funtzio primitiboaren bidez sinbolo baten propietate baten balioa lortuko dugu. Sinbolo horrek propietate hori definitua ez badu, GET funtzioak NIL emango digu.

Adibidez, suposa dezagun MIKEL sinboloak honelako propietate-lista duela:

((GURASOAK . (PATXI AMAIA)) (SEME-ALABAK . (XABIER)))

orduan:

(GET 'MIKEL 'GURASOAK) → (PATXI AMAIA)

(GET 'MIKEL 'SEME-ALABAK) → (XABIER)

Propietateen izenak LISPeke sinboloak direnez, propietateen balioak sinboloen bidez indexatuak daudela esaten da.

Propietateen balioak aldatzeko, honelako forma bat erabiltzen da:

(SETF (GET <sinboloa> <propietate-izena>) <balioa>)

Ikusten denez, SETFren lehenengo argumentua propietateen balioak eskuratzeko erabiltzen den forma dugu. Bigarrena aldiz, propietateari eman nahi diogun balio berria izango da. SETF formak balio berri hori bueltatuko du.

Adibidez:

(SETF (GET 'MIKEL 'SEME-ALABAK) '(XABIER ANE))
→ (XABIER ANE)
(GET 'MIKEL 'SEME-ALABAK) → (XABIER ANE)

SETF funtzioa sinbolo baten propietate batekin lehen aldiz erabiltzean, propietate berria eraiki egiten da automatikoki. Beraz, SETF eta GET formen funtzioen konbinaketa propietateen funtzio eraikitzaile eta aldatzailea dugu.

Garrantzitsua da ikustea (GET <sinboloa> <propietatea>) formak NIL bueltatuko duela sinboloaren propietate hori existitzen ez denean. Honek esan nahi du ezin izango dela bereiztu sinbolo baten propietateak NIL balioa duen ala sinbolo horren propietate hori ez existitzea. Horregatik ez da komenigarria izango propietate baten baliotzat NIL erabiltzea.

Propietateak ezabatzeko nahikoa litzateke honelako forma bat erabiltzea:

(SETF (GET <sinbolo> <propietate-izena>) NIL)

Baina horretarako LISPeke tresna eraginkorrago bat ematen digu, REMPROP funtzioa.

(REMPROP <sinbolo> <propietate-izena>)

Funtzio honek ezabatutako propietatearen balioa itzultzen du, baina aldeztu aurretik propietatea definitu gabe bazegoen NIL ematen du.

Ikus dezagun adibide bat:

(SETF (GET 'ONTZIA 'EDUKINA) 'URA) → URA

(GET 'ONTZIA 'EDUKINA) → URA

(REMPROP 'ONTZIA 'EDUKINA) → URA

(GET 'ONTZIA 'EDUKINA) → NIL

Beste askoren artean, propietateen erabilpen interesgarrienetako bat arbolen eta grafoen adierazpidea dugu.

9.3. ARRAYAK.

Arrayak espresio anitzak adierazteko erabiltzen diren datu-motak edo informazio-egiturak ditugu. Adierazten den espresio-kopurua finkoa izango da eta espresio bakoitza identifikatzeko zenbakizko indizeak erabiliko dira. Ondorioz, array bati lotutako balioak zenbakiz indexatuak daudela esan daiteke.

Hemen daukagu, adibidez, oinarritzko koloreak adierazten dituen dimentsio bateko array bat:

0	1	2
GORRIA	URDINA	HORIA

GORRIA sinboloa arrayaren elementu bat dela esaten da, eta gainera 0 indizea duen lekuan kokatuta dagoela ikusten dugu.

Ondoren bi dimentsioko array bat aztertuko dugu. Suposa dezagun 5x7 dimentsioko matrize bat adierazi nahi dugula. Hortarako ondoko arraya erabili dezakegu, gordetzen diren balioak hauek izanik:

	0	1	2	3	4	5	6
0	1	0	22	14	0	0	0
1	0	0	12	0	0	56	0
2	0	0	12	15	50	0	11
3	25	11	23	33	0	0	0
4	0	0	17	0	0	0	0

Arrayak erabiltzeko, beste datu-mota guztiekin bezala, array bera nola eraiki, datuak nola irakurri eta arrayko datuak nola aldatu jakin behar dugu. Beraz azter ditzagun funtzio eraikitzaile, ikuskatzaile eta aldatzaileak.

Array bat eraikitzeko, zenbat dimentsio dituen eta dimentsio bakoitzak zein luzera duen zehaztu behar dugu. Horretarako MAKE-ARRAY funtzioa daukagu. Hartzen duen argumentu bakarra arrayaren dimentsio-kopurua luzeratzat duen lista bat da, bere elementuek dimentsio bakoitzaren luzera adierazten dutelarik. Adibidez:

(SETF OINARRIZKO-KOLOREAK (MAKE-ARRAY '(3)))

Espresio hori ebaluatuz dimentsio bakarreko array bat lortzen dugu, bere indizeak 0tik 2ra joango direlarik.

(SETF MATRIZEA (MAKE-ARRAY '(5 7)))

Beste hori ebaluatuz bi dimentsioko array bat lortzen dugu, bere lehenengo dimentsioaren indizeak 0tik 4era eta bigarren dimentsioaren indizeak 0tik 6ra joango direlarik.

Arrayak dimentsio bakarra duenean, MAKE-ARRAY funtzioaren argumentua laburtu daiteke, zenbaki-lista baten ordeztu dimentsioaren tamaina adierazten duen zenbaki bat jarritz. Adibidez, OINARRIZKO-KOLOREAK arraya eraikitzeko beste modu hau daukagu:

(SETF OINARRIZKO-KOLOREAK (MAKE-ARRAY 3))

Normalean, array bat eraikitzen dugunean interesgarria izaten da bere posizio guztiak hasierako elementu batez betetzea. MAKE-ARRAY forma barruan :INITIAL-ELEMENT hitz gakoa erabiliz nahi dugun balioaz hasieratuko ditugu arrayaren elementu guztiak. Ondorengo adibidean oinarritzko-koloreak arrayaren elementu bakoitza HUTSA balioarekin hasieratzen da:

(SETF OINARRIZKO-KOLOREAK
(MAKE-ARRAY 3 :INITIAL-ELEMENT 'HUTSA))

Elementu guztien hasierako balioak berdinak ez badira, beste hasieraketa modu bat erabiltzen da. MAKE-ARRAY forma barruan eta :INITIAL-CONTENTS hitz gakoa erabiliz array osorako balioak zehaztuko ditugu. Hasierako elementu hauek espresio

batean bilduko dira. Ondorengo adibidean ikusten denez, hasierako elementuak dituen espresioa azpelistaz osatutako lista bat dugu. Bere lehenengo azpilista, lehenengo indizea 0 duten arrayaren elementuei dagokie, bigarrena lehen indize bezala 1 dutenei, eta horrela:

```
(SETF MATRIZEA
      (MAKE-ARRAY '(5 7)
                    :INITIAL-CONTENTS
                    ( ( 1 0 22 14 0 0 0 )
                      ( 0 0 12 0 0 56 0 )
                      ( 0 0 12 15 50 0 11 )
                      ( 25 11 23 33 0 0 0 )
                      ( 0 0 17 0 0 0 0 ) )))
```

Azter dezagun funtzio ikuskatzailea. Array bat osatua izan ondoren bere datuak irakurtzeko nahikoa da AREF funtzioa erabiltzea. Bere argumentuak arrayaren izena eta nahi dugun elementuaren indizea dira:

```
(AREF OINARRIZKO-KOLOREAK 0) → HUTSA
```

```
(AREF OINARRIZKO-KOLOREAK 2) → HUTSA
```

```
(AREF MATRIZEA 0 3) → 14
```

```
(AREF MATRIZEA 3 0) → 25
```

Bukatzeko, arrayan eguneraketak burutzeko funtzio berezirik gabe nahikoa da AREF eta SETF batera erabiltzea:

```
(SETF (AREF OINARRIZKO-KOLOREAK 0) 'GORRIA)
```

```
(SETF (AREF OINARRIZKO-KOLOREAK 1) 'URDINA)
```

```
(SETF (AREF OINARRIZKO-KOLOREAK 2) 'HORIA)
```

Horrela, SETF funtzioa AREFekin erabiltzen denean arrayen funtzio aldatzailea dugu. Kontutan hartu SETFren lehen argumentua datuak eskuratzeko erabiltzen den forma dela. Propietate-listekin eta beste datu-mota batzuekin gauza bera gertatzen da.

Arrayekin lan egitean askotan erabiltzen den funtzio bat predikatu bat betetzen duten arrayen elementuak kontatzea izaten da. Adibidez, ondoko funtzioak argumentu bat hartzen du eta OINARRIZKO-KOLOREAK arrayan zenbat aldiz agertzen den itzultzen du:

```
(DEFUN KONTATU-KOLOREAK (KOLOREA)
  (LET ((EMAITZA 0))
    (DOTIMES (N 3 EMAITZA)
      (WHEN (EQ KOLOREA
                (AREF OINARRIZKO-KOLOREAK N))
        (SETF EMAITZA (+ EMAITZA 1))))))

(KONTATU-KOLOREAK `HORIA) → 1
```

Ondoren aztertuko dugun funtzioa orokorragoa da, dimentsio bakar bateko edozein arrayekin lan egin bait dezake eta ez OINARRIZKO-KOLOREAKrekin bakarrik.

```
(DEFUN KONTATU-ELEMENTUAK (ELEM ARRAYA)
  (LET ((EMAITZA 0))
    (DOTIMES (N (ARRAY-DIMENSION ARRAYA 0) EMAITZA)
      (IF (EQ ELEM (AREF ARRAYA N))
        (SETF EMAITZA (+ EMAITZA 1))))))
```

Ematen zaion arraya edozein luzeratakoa izan daitekeenez, N kontatzailearen goiko muga lortzeko ARRAY-DIMENSION funtzio primitiboa erabiltzen du. Funtzio honek array bat eta dimentsio bat hartzen ditu argumentu bezala, eta dimentsio horren luzera array horretan bueltatzen du.

9.4. EGITURAK.

Programagile berriek datuak adierazteko lista eta azpilistaz osatutako egitura korapilotsuak erabiltzen dituzte. Esperientzia handiko programagileek aldiz, beste era batera jokutzen dute. Behe-mailako xehetasunetatik aldentzen saiatzen dira, bere burua goi-mailako kontzeptuetan finkatzen dutelarik.

Atal honetan, datu-adierazpidean abstrakzioa nola lortu dezakegun azalduko dugu. Horretarako, LISPen *egitura* datu-motak aztertuko ditugu. Erabiltzaileak honelako egitura mota bat definituz gero datuak erabiltzeko funtzioak sistemak automatikoki sortuko ditu.

9.4.1. Egituren definizioa eta oinarritzko erabilpena.

Datu-abstrakzioaren erabilpena hain handia dela eta, LISPe datu-mota berriak automatikoki eraikitzeke tresnak landu ditu. Datu-mota hauek *egitura* izena hartzen dute orokorrean.

Egitura erako mota berriak eremuen izenak eta balio lehenetsiak espezifikatuz definitzen dira. Egitura mota bat definitu ondoren, mota horretako objektuak erabili ahal izateko instantziak eraiki behar dira. Baina datu abstrakzioaren filosofiarekin bat joateko ezin izango da ezer jakin egitura erako moten instantzien barneadierazpideari buruz.

Egitura erako mota berriak definitzeko daukagun funtzio primitiboa DEFSTRUCT dugu. Ondoren datorren adibidearen bidez bere itxura ikusiko dugu. PERTSONA izeneko datu-mota berria eraikiko dugu, SEXUA eta NORTASUNA eremuak izango dituelarik:

```
(DEFSTRUCT PERTSONA
  (SEXUA NIL)
  (NORTASUNA 'ATSEGINA))
```

SEXUA eremuari NIL eta NORTASUNA eremuari ATSEGINA balio lehenetsiak ematen dizkiegu.

Orokorrean DEFSTRUCTren definizioa honelakoa dugu:

```
(DEFSTRUCT <egitura izena>
  (<eremu-izena 1> <balio lehenetsia 1>)
  (<eremu-izena 2> <balio lehenetsia 2>)
  ...
  (<eremu-izena n> <balio lehenetsia n>))
```

DEFSTRUCTek egiten duena hau da:

- Instantziak sortzeko funtzio eraikitzaile bat definitzen du.
- Instantzien eremuetatik datuak irakurtzeko funtzioak definitzen ditu.
- SETF funtzioa orokorrage bihurtzen du, funtzio ikuskatzaileekin batera erabiltzean egituren idazketarako ere baliagarria izan dadin.
- Objektuak testatzeko predikatu bat eraikitzen du, ea objektu hori DEFSTRUCTen bidez eraikitako datu-mota horren instantzia bat den jakiteko.

Beraz, DEFSTRUCTek egitura mota berriak definitzen ditu, instantziarik eraiki gabe. Honela, DEFSTRUCTen bidez funtzio eraikitzaileak, funtzio ikuskatzaileak, datu-motari buruzko predikatu bat eta datu-mota berriak jasateko SETF funtzio orokorrako bat automatikoki lortzen ditugu.

DEFSTRUCT forma bat ebaluatzean definitzen den funtzio eraikitzailearen izena MAKE- eta egitura motaren izenaren konbinazioa da. Beraz, gure adibidearen funtzio eraikitzailea MAKE-PERTSONA izango da. Bere erabileraren adibide bat:

(SETF PERTSONA-INST-1 (MAKE-PERTSONA))

MAKE-PERTSONA ebaluatzen denean, instantzia berri bat eraikitzen da, eta bere eremuak DEFSTRUCTek pertsona datu-mota eraikitzean ematen zituen espresio lehenetsiekin betetzen dira. Beraz, aurreko forma ebaluatu ondoren PERTSONA-INST-1 sinboloaren balioa ondokoa da:

eremua	balioa	nola lortu da
SEXUA	NIL	DEFSTRUCTren bidez
NORTASUNA	ATSEGINA	DEFSTRUCTren bidez

Instantzia bat eraikitzean eremuai balioak ematea posible da, eremuak balio lehenetsi bat edukita ere. Hortarako eremua identifikatzen duen hitz gakoa eta nahi dugun balioa erabiliko ditugu MAKE-PERTSONA funtzioarekin batera. Adibidez:

(SETF PERTSONA-INST-2 (MAKE-PERTSONA :SEXUA EMEA))

Instantzia berri hau honela geratzen da:

eremua	balioa	nola lortu da
SEXUA	EMEA	MAKE-PERTSONAren bidez
NORTASUNA	ATSEGINA	DEFSTRUCTren bidez

Egitura erako datu-motako objektu baten balioak aztertzeko ez da beharrezkoa izango balio horiek instantziaren eremuetan nola gordetzen diren jakitea, datu horiek lortzeko DEFSTRUCTek automatikoki eraikitzen dituen funtzioak bait ditugu. Funtzio

ikuskatzaile hauen izena, egitura motaren izena eta eremuaren izena gidoi batez bilduz lortzen den sinboloa da:

(PERTSONA-SEXUA PERTSONA-INST-2) → EMEA

(PERTSONA-NORTASUNA PERTSONA-INST-2) → ATSEGINA

Beraz, datuak funtzio espezifikoak erabiliz lortzen ditugunez, funtzioen bidez indexatuak daudela esan dezakegu.

Guzti horretaz gain, existitzen diren eremuen balioak aldatzeko modu bat behar dugu. DEFSTRUCTek hau ere kontutan hartzen du, baina ez du funtzio berezirik eraikiko horretarako. SETF funtzioa orokorrago bihurtzen du egituren instantziekin lan egin dezan. Adibidez, suposa dezagun PERTSONA-INST-1 instantziaren SEXUA eremuan EMEA balioa jarri nahi dugula. Honela egin dezakegu:

(SETF (PERTSONA-SEXUA PERTSONA-INST-1) EMEA)

(PERTSONA-SEXUA PERTSONA-INST-1) → EMEA

PERTSONA-INST-1 instantzia honela gelditzen da:

eremua	balioa	nola lortu da
SEXUA	EMEA	SETFren bidez
NORTASUNA	ATSEGINA	DEFSTRUCTren bidez

Beraz, propietate eta arrayekin bezala, SETFren lehen argumentua eremu balio bat ematen digun atzitze forma bat dugu, eta bigarrena eremuan jarriko den balio berria. Horrela, SETF funtzioa oso orokorra dugu, egituren instantziekin, sinboloekin, propietateekin eta arrayekin lan egiten duelarik. Dena den, bere izena egituren instantziekin duen erabilpenagatik datorkio, *set field*-en akronimoa delarik.

Egitura motako instantzien oinarritzko erabilpena azaltzeko beste funtzio bat aztertzea falta zaigu. Funtzio eraikitzaile eta ikuskatzailatz gain, DEFSTRUCTek datu-motari buruzko predikatu-funtzio bat sortzen du. Adibidez, DEFSTRUCTek pertsona datu-mota eraikitzen duenean PERTSONA-P predikatua sortzen du, ea objektu bat pertsona datu-motako instantzia bat den aztertzeko:

(PERTSONA-P PERTSONA-INST-1) → T

(PERTSONA-P (hau lista bat da, eta ez pertsona bat)) → NIL

Egitura datu-motak erabiltzerakoan kontutan hartu behar da egitura definitzen dugunerako interesatuko zaizkigun eremu guztiak ezagutu beharko ditugula.

Datu-mota eraiki ondoren, ezin izango dugu SETF funtzioa erabili DEFSTRUCT erabili zenean aipatu ez zen eremuren bat eraikitzeko. Beraz, ondoko adibide honek errorea emango luke:

(SETF (PERTSONA-ABIZENA pertsona-instantzia-1) LOPETEGI)

9.4.2. Egituren beste erabilpen batzu.

- Egitura moten deskribapena:

Ikusi dugunez, instantzia baten eremuak irakurtzeko modu bakarra funtzio ikuskatzaileak erabiltzea dugu, eta ezin izango dugu jakin instantzia bat nola adierazten den LISPen barnean. Baina programa baten errore-arazketan ari garenean gerta liteke instantzia baten edukina ikusi nahi izatea. Horretarako, inplementazioaren arabera DESCRIBE funtzioak instantzien informazioa aterako digu. Adibidez:

(DESCRIBE PERTSONA-INST-1)

Eragiketa horren ondoren LISPe instantziaren datu-mota eta bere eremuen balioak erakutsiko dizkigu pantailan.

- Egitura mota batek beste baten eremuak erantsita eduki ditzazke:

Sarritan zenbait gauza beste batzuren espezializazio gisa etortzen zaizkigu burura, hau da, kontzeptuen arteko erlazio hierarkikoaz ohartzen gara. Adibidez, langile bat enpresa batean lan egiten duen pertsona bat dugu, eta ikasle bat ikasketak burutzen ari den pertsona bat. Baina desberdintasun batzuetaz gain, langile eta ikasle guztiak pertsonak dira, hau da, biek dute pertsona guztientzat komunak diren atributuentzako balio bat, adibidez sexua.

Egitura moten bidez espezializazio-erlazioak adieraztea posible da. Horrela egiten da:

```
(DEFSTRUCT PERTSONA
```

```
  (IZENA NIL)
```

```
  (ADINA 'UMEA)
```

```
  (NAZIONALITATEA 'EUSKALDUNA)
```

```
  (SEXUA NIL))
```

```
(DEFSTRUCT (IKASLEA (:INCLUDE PERTSONA))
```

```
  (IKASKETAK 'OINARRIZKOAK))
```

IKASLEAren DEFSTRUCT forman egitura motaren izen hutsaren ordeztu, motaren izenaz gain beronek erantsita edukiko duen egitura datu-mota aipatzen duen lista bat ematen zaio :INCLUDE hitz gakoarekin batera. Horren bidez, IKASLE motak PERTSONA egitura motaren eremuak eta beraien balio lehenetsiak erantsita edukiko ditu. Gauzak honela, IKASLE batek PERTSONA guztiek bezala NAZIONALITATEA eremua izango du, EUSKALDUNA balio lehenetsiarekin:

```
(SETF PERTSONA-1 (MAKE-PERTSON))
```

```
(SETF IKASLE-1 (MAKE-IKASLEA))
```

```
(PERTSONA-NAZIONALITATEA PERTSONA-1)
```

```
→ EUSKALDUNA
```

```
(IKASLEA-NAZIONALITATEA IKASLE-1) → EUSKALDUNA
```

PERTSONA- bezala hasten diren funtzio ikuskatzaileak egitura mota espezializatuaren instantziekin erabili daitezke:

```
(PERTSONA-NAZIONALITATEA IKASLE-1) → EUSKALDUNA
```

PERTSONA egituraren eremuez gain, IKASLEA egiturak IKASKETAK eremua ere dauka, OINARRIZKOAK balio lehenetsiarekin, baina azken hau IKASLEen instantzientzat bakarrik existituko da:

```
(IKASLEA-IKASKETAK IKASLE-1) → OINARRIZKOAK
```

Gerta liteke, beste egitura mota bat erantsita edukitzea eta ondorioz zenbait eremuen izenak errepikatuak agertzea. Kasu honetan espezifikoena den egitura motaren balio lehenetsiak lehenetsia izango du, eta beste balio lehenetsiari itzala egiten diola esaten da.

Ondorengo adibidean :INCLUDE hitz gakoaren bidez LANGILE egitura motak beste edozein pertsona bezala ADINA eremuak dituela adierazten da, baina beste balio lehenetsi bat ematen zaiolarik:

```
(DEFSTRUCT (LANGILEA (:INCLUDE PERTSONA
                        (ADINA 'HELDUA)))
 (LAN-MOTA 'IRAKASLEA))

(SETF LANGILE-1 (MAKE-LANGILEA))

(PERTSONA-ADINA IKASLE-1) → UMEA

(PERTSONA-ADINA LANGILE-1) → HELDUA
```

9.4.3. Egitura moten garrantzia.

Egitura motak oso garrantzitsuak dira sistema handietarako ematen dituzten abantailengatik. Hala ere, lehen hurbilketa batean, badirudi beste datu-mota sinpleagoen bidez ere egitura motek ematen dizkiguten abantaila gehienak lortu ditzakegula. Alegia, datuak adierazteko abstrakzioa lortzeko, nahikoa litzateke elkartzeko-lista edo propietate-listak maneiatzen dituzten makro edo funtzio batzu definitzea, eta azken finean objektuen erabilera egitura motenaren antzekoa litzateke. Dena den, egiturak erabiltzeak emango dizkigun abantailak ondoko zentzu honetan dohaz:

- Funtzio ikuskatzaileak oso eraginkorrak dira:

Ez dago lista luzeak errekorritu beharrik indize bezala lan egiten duen sinbolo baten bila. Horren ordez, LISP inplementatu zutenak asko saiatu ziren DEFSTRUCTek automatikoki definitzen dituen funtzioen eraginkortasuna optimizatzen.

- Egitura moten eremuak aldaciznak dira:

Egitura mota bat definitu ondoren inork ezingo dio eremurik kendu edo gehitu, eta zihurtatu ahal dugu programa handi baten inongo zatitan era horretako aldaketarik egongo ez dela. Hau dela eta, programa bere osotasunean ulertzea errazagoa izango da, eta datuen trinkotasuna handiagoa izango da.

Programagile taldeek, normalean, programa batek erabiltzen dituen egitura moten definizioak ondo komentatutako fitxategi batean sartzen dituzte argitasun hobea lortuz.

ARIKETAK.

- 9.1. Pertsona baten aitaren aitaren izena itzuliko duen AITITA funtzioa idatzi. Zenbait sinbolorentzat AITA propietatea definitu dela suposatuko da. Aitita jakiteko modurik ez badago NIL itzuliko da.
- 9.2. EVA funtzioa definitu amen bidetik arbaso urrutiena itzul dezan pertsona batentzat. AMA propietatea erabili. Pertsona baten emakumezko arbasorik ezagutuko ez balitz, pertsonaren izena bera itzuliko litzateke.
- 9.3. ARBASOAK funtzioa definitu pertsona baten izena hartu eta lista batean pertsonaren izena eta bere arbaso guztiena itzuliko dituen. Erabili AITA eta AMA propietateak.
- 9.4. Definitu funtzio bat lista bateko osagai bakoitzarentzat AURREKOAK propietatea eraikitzeko. Osagai baten AURREKOAK propietatearen balioa listan bere aurrean dauden osagaiek osatzen duten lista izango da. Adibidez, funtzioa aplikatzen badiogu (A C H Y F) listari atomo bakoitzari lotzen zaion AURREKOAK propietatearen balioa honako hau izango da:

A	-----	()
C	-----	(A)
H	-----	(A C)
Y	-----	(A C H)
F	-----	(A C H Y)

- 9.5. Suposa dezagun herrialde bateko hiriak sinboloen bidez adieraziak ditugula. Gainera hiri bakoitzak BIZILAGUNAK propietate bat izango du, bertan hiri horrek errepide batez zuzenean loturik dituen hiriak daudelarik.

Definitu LOTU funtzio bat bi hiri hartuta bata bestearen BIZILAGUNAK propietate-listan sartuko dituen. Kontutan hartu konexio bat jadanik egina egon daitekeela, eta kasu horretan funtzioak ez du ezer egingo eta NIL bueltatuko du; bestela T bueltatuko du.

- 9.6. 5x7 dimentsioko zenbakizko array baten elementuen batzbestekoa kalkulatzeko funtzio bat definitu.
- 9.7. Suposa dezagun elkartze-listen bidez hiztegi bateko hitzen ezaugarriak adieraziak ditugula. Hitz bakoitzak lista batean honako informazioa eduki dezake loturik:

```
(( KATEGORIA-SINTAKTIKOA <kategoria-balioa> )  
 ( ESANGURA <esangura-balioa> )  
 ( SINONIMOAK <sinonimo-lista> )  
 ( HIZKUNTZA <hizkuntza-balioa> ) )
```

Adibidez:

PERTSONA →

```
(( KATEGORIA-SINTAKTIKOA IZENA )  
 ( ESANGURA "Giza espezieko izaki bakoitza." )  
 ( SINONIMOAK ( GIZAKI ) )  
 ( HIZKUNTZA EUSKARA ) )
```

IDATZI →

```
(( KATEGORIA-SINTAKTIKOA ADITZA )  
 ( ESANGURA "Ikur grafikoen bidez ideiak adieraztea." )  
 ( HIZKUNTZA EUSKARA ) )
```

- a) Defini itzazu ezaugarri bakoitzaren balioak atzitzeko funtzioak.
- b) Hitzak egituren bidez adierazteko zer egin beharko genuke? Nola aldatuko litzateke problema? Aukera berri hau aztertu.
- 9.8. Grafo bat definitzerakoan korapiluneak atomo sinbolikoen bidez adierazten dira. Sinbolo bakoitzaren HAUZOKOAK propietatearen balioaren bidez arku batez loturik dauden beste korapiluneez osatzen duten lista izango dugu.
- Definitu MUNDU funtzioa ondoko erara: argumentu bakartzat korapilune bat hartzen du eta korapilune horretatik abiatuz eta arku bat edo gehiago zeharkatuz ikutu daitezkeen korapilune guztiak itzultzen ditu lista bat osatuz.

Adibideak:

A ----- B	H
C ----- F	G
D ----- E	

(GET 'A 'HAUZOKOAK) → (B C)
 (GET 'B 'HAUZOKOAK) → (A)
 (GET 'C 'HAUZOKOAK) → (A F D)
 (GET 'D 'HAUZOKOAK) → (C E)
 (GET 'E 'HAUZOKOAK) → (F D)
 (GET 'F 'HAUZOKOAK) → (C E)
 (GET 'G 'HAUZOKOAK) → (H)
 (GET 'H 'HAUZOKOAK) → (G)

(MUNDU 'F) → (A B C D E F)
 (MUNDU 'C) → (A B C D E F)
 (MUNDU 'G) → (G H)

- 9.9. Grafo bat definitzeko asmoz korapiluneak atomo sinbolikoen bidez adierazten dira eta sinbolo bakoitzaren HAUZOKOAK propietatearen balioa berarekin arku batez loturik dauden beste korapilunek osatzen duten lista izango da.

A ----- B----- D	(GET 'A 'HAUZOKOAK)→(B C)
	(GET 'B 'HAUZOKOAK)→(A C D)
	(GET 'C 'HAUZOKOAK)→(A B)
	(GET 'D 'HAUZOKOAK)→(B)
C	

Definitu ITZULBIDERIK funtzioa. Argumentu bakartzat korapilune bat hartu eta NIL itzuliko du grafoko arkuak jarraituz korapilunera itzultzeko biderik ez badago bestela bide bat itzuliko du. Ezin daiteke pasa birritan korapilune beretik.

(ITZULBIDERIK 'D) → NIL
 (ITZULBIDERIK 'A) → (A B C A)
 edo (A C B A)

- 9.10. Familia bat adierazteko asmoz sinbolo atomiko bana definitu da senide bakoitzarentzat. Pertsona bakoitzaren izenari ondoko propietateak definitu zaizkio:

SEME-ALABAK: Seme-alaben lista, nagusitik txikienera ordenatuta.

SEXUA: GIZONEZKO edo EMAKUMEZKO.

BIZIRIK: BAI ala EZ.

- a) Definitu ONDOKOEN-ORDENA funtzioa modu honetara: izen bat (edo anai-arreben lista ordenatu bat) hartu eta bere (edo beren) ondoko guztiak itzultzen ditu lista batetan bilduta, ondorengotzarako lehentasunaren arabera ordenatuta. Pertsona baten lehentasuna bere anai-arreba zaharragoena baino txikiagoa da, eta anai-arreba zaharragoen ondokoena baino txikiagoa ere bai. Seme-alabek beren gurasoek baino lehentasun txikiago dute. Norbera bere buruaren ondokoa dela eman.

Adibidez: Azpimarratuak senide hilak direla emanda, irudiko familiarako:

(ONDOKOEN-ORDENA R1) → (R11 R111 R113 R12 R132 R133)

<u>R1</u> _____	R11 _____	R111
	_____	<u>R112</u>
	_____	R113
_____	R12	
_____	<u>R13</u> _____	<u>R131</u>
	_____	R132
	_____	R133

- b) Aldaketa txiki bat eginez, definitu ORDMATXISTA funtzioa lehengoaren antzera baina funtzio berri honen arabera emakumeen lehentasuna bere edozein nebarena eta neben ondokoena baino txikiagoa izango da. Lagungarri gisa definitu GIZONAK-AURRERA funtzioa. Beronek adinaz ordenatuta dagoen pertsona-lista bat hartu eta berrodenatuta itzultzen du: hasieran gizonezkoak adinaz ordenatuta eta gero emakumezkoak adinaz ordenatuta.

10. SARRERA/IRTEERAKO FUNTZIOAK.

Kapitulu honen helburua LISPez sarrera/irteerak egiteko dauzkagun aukerak azaltzea da. Hau da, programa eta erabiltzailearen arteko komunikaziorako tresnak ikusiko ditugu.

Horretarako, agertzen zaigun funtzio bakoitza aztertzean adibideetan oinarrituko gara bere portaera hobeto ikusteko. Adibideetan erabiliko dugun idazkera aldatu egingo dugu, argitasun handiagoa lortu nahiean. Beste ataletako adibideetan, forma baten ebaluazioaren emaitza adierazteko "→" ikurra erabili dugu. Orain ez dugu horrelakorik erakutsiko. Kapitulu honetako adibideek LISPeke edozein interpretatzailetan froga batek hartzen duen itxura hartuko dute. Honela adibide bakoitza terminalaren aurrean egongo bagina bezala ikusiko dugu, interpretatzailearen prompt-a (">" karakterea hartuko dugu prompt-tzat), guk teklatutako LISP forma eta ondoko lerroetan LISPeke bueltatutako emaitza ikusiko direlarik. Emaita itzuli aurretik formak zerbait idazten badu, emaitza baino lehenago azalduko da. Kasu askotan, LISPeke emaitzak bueltatzen dituen lerroetan ';' eta ohar batzu agertzen dira. Ohar hauek guk jarritakoak dira LISPeke idazten duena argitzeko asmoz.

10.1. SARRERA/IRTEERAKO OINARRIZKO FUNTZIOAK.

READ eta PRINT.

READ eta PRINT funtzioek programa eta erabiltzailearen arteko komunikazioa posible egiten dute.

PRINT funtzioak bere argumentu bakarra ebaluatu eta emaitza idazten du irteeraren lerro berri batean. Ondorengo adibidean, PRINT erabiltzen dugu sinbolo baten balioa idazteko:

```
> (SETF temperatura 100)
100
> (PRINT temperatura)
100 ; PRINTek idatzia (albo-ondorioa).
100 ; PRINT formak bueltatzen duena.
```

Ikusten dugunez, PRINTek bueltatzen duen balioa pasatzen zaion argumentua ebaluatzearen emaitza dugu. Ondorengo adibidean PRINT formak bueltatutako balioa beste forma baten argumentu bezala erabiltzen da:

```
> (IF (< (PRINT temperatura) 25)
      NORMALA
      BEROA)
100                                ; PRINTek idatzitako balioa.
BEROA                             ; IFek bueltatutako balioa.
```

PRINT forma baten argumentua edozein espresioa izan daiteke, eta ez sinbolo bat bakarrik:

```
> (SETF PROGRAMAGILEA 'Mikel LENGOAIAK '(LISP C))
(LISP C)
> (PRINT (LIST PROGRAMAGILEA
              'programagilea
              (LENGTH LENGOAIAK)
              'lengoaia
              LENGOAIAK))
(MIKEL PROGRAMAGILEA 2 LENGOAIA (LISP C)) ; albo-ondorioa
(MIKEL PROGRAMAGILEA 2 LENGOAIA (LISP C)) ; balioa
```

Interpretatzaileak READ funtzioa duen forma bat aurkitzen duen aldi bakoitzean erabiltzaileak espresio bat tekleatu arte itxaroten du. Beraz, funtzio hau erabiltzen duguncan edozein espresio tekleatu arte programa geldi egongo da. Hurrengo adibidean, funtzioaren ondoren "Kaixo" tekleatu dugu eta atzetik zurigune bat:

```
> (READ)Kaixo
KAIXO                             ; READ formak bueltatutakoa
```

READ funtzioak ez du ezer idazten itxaroten dagoela ikustarazteko eta hau arazo bat izan daiteke programa luzeak egikaritzeraoan,

Horregatik, argitasuna lortu nahiean interesgarria izaten da PRINT funtzioarekin batera erabiltzea. Honela, erabiltzailea geldialdiaz konturatzen da, eta gainera programak espero duen erantzunaren itxura azaldu diezaioke. Ondorengo adibide honetan definitzen den funtzio hau nahiko adierazgarria dugu:

```

(DEFUN KARRATUA ()
  (PRINT '(ZENBAKI BAT TEKLEATU:)) ; Erabiltzailea ohartu.
  (SETF ZENBAKI (READ)) ; Zenbaki bat jaso.
  (LET ((EMAITZA (* ZENBAKI ZENBAKI)))
    (PRINT (APPEND (LIST ZENBAKI) ; mezua osatu eta
                  '(ren karratua) ; inprimatu.
                  (LIST EMAITZA)
                  '(da))
      EMAITZA)) ; azken forma.

```

Ondoren funtzioa erabiltzen badugu:

```

> (KARRATUA)
(ZENBAKI BAT TEKLEATU:) 21 ; oharra eta erabiltzailearen erantzuna.
(21 REN KARRATUA 441 DA) ; erragiketaren oharra.
441 ; bueltatutako balioa.

```

10.2. SARRERA/IRTEERA KO ERABILPEN BITXIAGOAK.

10.2.1. FORMAT. Idazkera dotorea.

LISPeko programagile trebeek programen irteerak dotoreago aurkezten dituzte, majuskula, minuskula eta puntuazio ikurrak dituzten mezuak ateraz. Aukera hauek posible egiteko FORMAT funtzioa agertzen zaigu, PRINT funtzioaren osagarri malguago bezala.

FORMAT funtzioa karaktere-kateekin lan egiteko prestatuta dago. Bigarren kapituluaren ikusi genuenez, karaktere-kateak komatxoaren artean mugatutako karaktere sekuentziak ditugu. FORMATen formarik sinpleenak terminalean karaktere-katea bat idazten du, komatxorik gabe eta lerro berri batera pasa gabe:

```

> (FORMAT T "Kaixo !") Kaixo! ; FORMATEk Kaixo! idazten du.
NIL ; FORMATEk bueltatzen duena.

```

T argumentuak terminalean idazteko esaten dio FORMATi. Bere ordeztu, hurrengo atalean ikusiko denez, irteera-fitxategi bati lotzen dion sinbolo bat etor daiteke.

Karaktere-katea bat lerro berri batean idazteko, nahikoa da lerro berria hasi nahi dugun lekuan tilde karakterea eta bere atzetik portzentaiaren ikurra jartzea:

```
> (FORMAT T "~%Kaixo!")
Kaixo!                ; FORMATEk Kaixo! idazten du.
NIL                   ; FORMATEk bueltatzen duena.

> (FORMAT T "~%Kaixo! ~%Zer moduz zaude?")
Kaixo!                ; FORMATEk idatzia.
Zer moduz zaude?     ; FORMATEk idatzia.
NIL                   ; FORMATEk bueltatzen duena.
```

‘~’ karakterea kode agintzaileak sartzeko erabiltzen da. % kode agintzaileak FORMATi lerro berri batean hasteko esaten dio. & kode agintzaileak FORMATi lerro berri batean hasteko esaten dio ere, baina ez du lerro berri batera jauziko jadanik lerro berri batean badago. Ondorengo adibidean, bi % kode agintzaile bata bestearen atzean jarrita bi lerro berri jauzten dira, eta % eta & kode agintzaileak jarraian jarrita aldiz, lerro berri bakarra ematen dute:

```
> (DEFUN LERRO-JAUZIAK ( )
  (FORMAT T "~%Lerro honek % darama aurrean eta atzean~%")
  (FORMAT T "~%Lerro honek % darama aurrean eta atzean~%")
  (FORMAT T "~&Lerro honek & darama aurrean")) )
LERRO-JAUZIAK
> (LERRO-JAUZIAK)
Lerro honek % darama aurrean eta atzean

Lerro honek % darama aurrean eta atzean
Lerro honek & darama aurrean
NIL
```

Beraz, lerro berri batean hasi nahi dugunean eta zihur ez gaudenean beste edozein FORMAT forma bat lerro berri batekin bukatu duen & kode agintzailea erabiliko dugu.

Beste kode agintzaile asko daude, % eta &ez aparte. Horrenbeste aukera ematen zaizkigu, kode agintzaile guztiak ikastea beste lengoaia bat ikastea bezala izango litzatekeela. Dena den, idazketa arruntak egiteko nahikoa izan dezakegu %, & eta A kode agintzailearekin. Azken honen bidez karaktere-katearen barruan beste argumentu batzuren balioa sartzea posible izango da:

```
> (SETF IZENA 'PELLO)
...
> (FORMAT T "~%IZENA aldagaiaren balioa ~a da." IZENA)
IZENA aldagaiaren balioa PELLO da. ; IZENAREN balioa ~a-ren ordeez.
NIL ; Bueltatzen duen balioa.
```

FORMAT forma batean A kode agintzailea behin baino gehiagotan erabiltzen bada, beste horrenbeste argumentu eman beharko da karaktere-katearen atzetik:

```
> (FORMAT T
    "~%~a programagileak ~a lengoiaia ezagutzen ditu: ~a"
    PROGRAMAGILEA
    (LENGTH LENGOAIAK)
    LENGOAIAK)
MIKEL programagileak 2 lengoiaia ezagutzen ditu: (LISP C) ;albo-ondorioa.
NIL ;balioa.
```

FORMATek tabulatutako zutabeetan idazteko aukera ere ematen du. Hau egiteko, A kode agintzailearekin batera zutabe-zabalera adieraziko duen zenbaki bat sartuko dugu. Honela, 10A kode agintzaileak ondoren etorriko den balio bat idazteaz gain, 10 karaktereko zabalera betetzeko behar diren zuriguneak idazten ditu:

```
> (FORMAT T
    "~%Programagilea: ~10a Lengoaiak: ~a"
    PROGRAMAGILEA
    LENGOAIAK)
Programagilea: MIKEL Lengoaiak: (LISP C) ; albo-ondorioa.
NIL ; balioa.
```

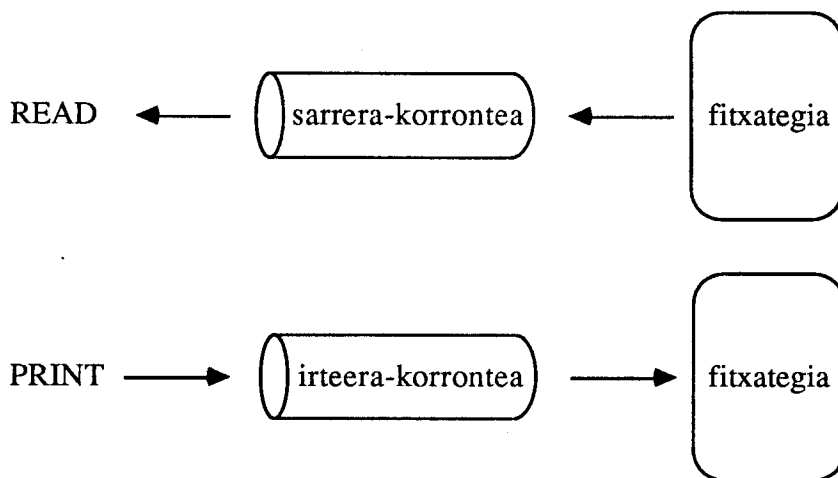
10.2.2. WITH-OPEN-FILE.Sarrera/irteerako fitxategiak.

Orain arte terminala erabili dugu irteerak eta sarrerak egiteko. Kasu askotan interesgarria izaten da eragiketa hauek fitxategiekin egitea, gehienbat datu-kopuru handiak maneiatu nahi direnean. Adibidez, suposa dezagun fitxategi batean honelako datu-kopuru handi bat dugula:

((Xabier Jauregi) (LISP PASCAL))
 ((Pello Lopetegi) (PASCAL ADA FORTRAN))
 ((Arantxa Etxeberria) (LISP PROLOG))
 ((Mikel Agirre) (PASCAL ADA C))
 ((Itziar Zubillaga) (C PASCAL))
 ((Karmen Ugalde) (LISP PROLOG C))
 ...

Datu horiek LISPez erabili ahal izateko, irakurtzeko modu bat behar dugu. Hau da, READ formak terminalari ordeztu fitxategiei lotzeko modu bat interesatzen zaigu.

Hori lortzeko, *korronte*-ak agertzen zaizkigu. Kontzeptualki, korronteak datu-fitxategiekiko lotura ematen diguten LISPeko objektuak ditugu. Datuak irakurtzeko erabiltzen diren fitxategiei lotutako korronteak sarrera-korronteak ditugu, eta datuak idazteko erabiltzen diren fitxategiei lotutakoak aldiz irteera-korronteak. Ondorengo irudi hauek oso adierazgarriak ditugu:



Korronteak sortu eta fitxategi bati lotzeko WITH-OPEN-FILE funtzio primitiboa daukagu. Funtzio honi hiru argumentu eman behar zaizkio: sinbolo baten izena (korrontea adierazteko), korronteari lotuko zaion fitxategiaren izena eta korrontearen norabidea adierazten duen balio bat, sarrera- edo irteera-korronte bat eraikiko den jakiteko (:INPUT ala :OUTPUT).

Funtzioaren egitura orokorra:

```
(WITH-OPEN-FILE (<korronte izena> <fitxategia>
                  :DIRECTION <:input edo :output>)
  gorputza)
```

Gorputzaren barruan, korrontearen norabidearen arabera READ edo PRINT funtzioak erabiliko ditugu. Funtzio horiei, maneiatuko duten korrontearen izena pasako zaie argumentu bezala:

```
(READ <korronte-izena>)
(PRINT <inprimatzeko espresioa> <korronte-izena>)
```

Ikus ditzagun bere erabilpenaren adibideak:

A. Datuak fitxategi batetik irakurri.

Ondorengo formaren bidez sarrera-korronte bat eraikiko dugu, PROG-KORR aldagaiaren bidez erabiliko dena, lehen aipatu dugun fitxategiari lotua egongo delarik. Fitxategiaren izena karaktere-katea baten bidez adierazten da. Karaktere-katea honetan fitxategiaren izena katalogo eta guzti etorri daiteke, lan egiten ari garen sistema eragilearen arabera adieraziko delarik (gure kasuan UNIX). Korrontearekin bukatzeko, sarrera-korronte bat dela adierazteko :DIRECTION eta :INPUT hitz gakoak erabiliko ditugu.

Gorputzaren barruan PROG-KORR READ formatan ager daiteke. Gure kasuan hiru espresio irakurtzen dira eta ondoren inprimatu:

```
> (WITH-OPEN-FILE (PROG-KORR
                    "/lisp/datuak/programagileak.lsp"
                    :DIRECTION :INPUT)
  (DOTIMES (N 3) (PRINT (READ PROG-KORR))))
((XABIER JAUREGI) (C PASCAL))           ; lehen espresioa.
((PELLO LOPETEGI) (PASCAL ADA FORTRAN)) ; bigarrena.
((ARANTXA ETXEBERRIA) (LISP PROLOG))    ; hirugarrena.
NIL                                       ; balioa.
```

Ikusten denez, WITH-OPEN-FILE formak bere gorputzean dagoen azken formaren balioa bueltatzen du. Adibidean DOTIMES formak emandako NIL balioa bueltatzen du.

B. Datuak fitxategi batetik irakurri eta emaitzak beste batean idatzi.

Adibide honetan sarrera- eta irteera-korronteak erabiliko ditugu. Gure helburua LISP programazio-lengoaia ezagutzen duten programagileak bilatzea eta fitxategi batean idaztea izango da. Datu-fitxategiko bost lehenengo programagileak aztertuko ditugu.

Horretarako BADAHI-LISP-P predikatua beharko dugu:

```
(DEFUN BADAHI-LISP-P (ESPRESIOA)
  (MEMBER 'LISP (SECOND ESPRESIOA)))
```

Erabiliko dugun LISP forma:

```
> (WITH-OPEN-FILE.(PROG-KORR
    "/lisp/datuak/programagileak.lsp"
    :DIRECTION :INPUT)
  (WITH-OPEN-FILE (LISP-KORR
    "/lisp/datuak/lisp_dakite.lsp"
    :DIRECTION :OUTPUT)
    (DOTIMES (N 5)
      (LET ((MORROIA (READ PROG-KORR)))
        (IF (BADAHI-LISP-P MORROIA)
          (PRINT MORROIA LISP-KORR))))))
```

NIL

Terminalean ez da beste daturik aterako, baina fitxategi berriak edukin hau izango du:

```
((XABIER JAUREGI) (LISP PASCAL))
((ARANTXA ETXEBERRIA) (LISP PROLOG))
```

Bukatzeko, kontutan hartzeko gauza batzu:

- WITH-OPEN-FILEekin sarrera-fitxategi bat irekitzen den bakoitzean, LISPe hasieratik hasten du irakurketa.

- WITH-OPEN-FILE erabiliz irteera-fitxategi bat irekitzen duguncan, fitxategi berri bat eraikitzen da.
- WITH-OPEN-FILE funtzioaz gain, badaude LISPeZ korranteak fitxategiei lotzeko beste modu batzu. Dena den, funtzio hau erabiltzea modu hoberena iruditzen zaigu, fitxategiak egoera zuzen batean uztea zihurtatzen delako beti, gorputzaren barruan errore bat gertatuta ere.

10.2.3. Fitxategi-bukaerako tratamendua.

Aurreko ataletan azaldutako adibideetan irakurketa mugatuak egin ditugu, fitxategietako lehen datuak bakarrik irakurri ditugularik. Baina normalagoa izaten da fitxategi bat hasieratik bukaeraraino irakurri nahi izatea. Hori lortu nahirik arazo bat agertzen zaigu: programak ez du jakingo fitxategia noiz bukatuko den eta fitxategi-bukaeran irakurtzen saiatuko balitz errore bat gertatuko litzateke.

Arazoa konpontzeko, fitxategian espresio berezi baten bidez fitxategi-bukaera markatu dezakegu. Horrela, irakurketa bat egiten dugun bakoitzean ea espresio berezi hori topatu den testeatuko dugu, aurkitu ondoren irakurketa gehiago ez egiteko. Dena den, hurbilketa hobe iruditzen zaigu fitxategi-bukaerara heltzen denean errorerik ez gertatzea. Hori lortzeko bigarren argumentu bat pasa beharko diogu READ funtzioari, NIL balioarekin:

```
(READ <korrante-izena> NIL)
```

Erabilera honen bidez, fitxategi-bukaera topatzen duenean errore bat eman ordez NIL bueltatzen du.

READen erabilpen hau argitzeko aurreko atalean aztertu dugun azken adibidea berriro hartuko dugu. Kasu honetan bost lehenengo programagileak aztertu beharrean fitxategi osoa irakurriko dugu. Horretarako, READ funtzioaren forma berriaz gain, WITH-OPEN-FILEren gorputzan DO forma bat erabiliko dugu DOTIMES formaren ordez:

```
> (WITH-OPEN-FILE (PROG-KORR
                    "/lisp/datuak/programagileak.lsp"
                    :DIRECTION :INPUT)
  (WITH-OPEN-FILE (LISP-KORR
                    "/lisp/datuak/lisp_dakite.lsp"
                    :DIRECTION :OUTPUT)
```

```
(DO ((MORROIA (READ PROG-KORR NIL)
              (READ PROG-KORR NIL)))
    ((NOT MORROIA))
    (IF (BADA KI-LISP-P MORROIA)
        (PRINT MORROIA LISP-KORR))))
```

NIL

Irteerako fitxategiaren edukina honela geratzen da:

```
((XABIER JAUREGI) (LISP PASCAL))
((ARANTXA ETXEBERRIA) (LISP PROLOG))
((KARMEN UGALDE) (LISP PROLOG C))
...
```

10.2.4. Karaktere-kateen irakurketarako funtzioak.

Karaktere-kateak irakurtzeko tresna berezi bat eskeintzen digu LISPEk: READ-LINE funtzioa.

Funtzio honek <RETURN> karakterea edo fitxategi-bukaera aurkitu arte hartzen ditu karaktereak, bi balio bueltatzen dituelarik. Lehenengoa hartu dituen karakteretaz osatutako karaktere-katea dugu. Bigarrena normalean NIL izaten da, baina READ-LINE fitxategi-bukaerara heldu bada T ematen du. Hurrengo adibidean ikusiko denez, READ-LINE forma ebaluatu ondoren karaktere-katea bat eta <RETURN> karakterea tekleatu arte ez da ezer gertatuko:

```
> (READ-LINE)Hau READ-LINE funtzioarena da.
"Hau READ-LINE funtzioarena da."
NIL
```

Beraz, karaktere-kateen lerrokako irakurketarako READ-LINE funtzioa daukagu. Hala ere, kasu askotan interesgarria izaten da irakurketak karakterez-karakteregitea. Horretarako READ-CHAR funtzioa daukagu:

```
> (READ-CHAR)A
#A

> (READ-CHAR)a
#a
```

Bi funtzioek aukerazko argumentuak onartzen dituzte READek bezala. Lehen argumentu bat ematen zaienean, sarrera-korrante baten izena izango da, datuak handik irakurtzeko. Bigarren bat ematen bazaie, READ-LINE eta READ-CHAR funtzioek fitxategi-bukaera aurkitzen dutenean errore bat eman ordez, bigarren argumentu horren balioa bueltatuko dute.

Bere erabilpena hobeto ikusteko ondorengo adibidea aztertuko dugu:

Suposa dezagun karaktere-katea bat fitxategi baten lerroen artean bilatzen duen funtzio bat eraiki nahi dugula. Karaktere-katea aurkitzen bada, funtzioak fitxategiaren lerro osoa inprimatuko du terminalean, eta T itzuliko du.

WITH-OPEN-FILE funtzioaren bidez korrante egokia definituko dugu, ondoren READ-LINERen bidez lerroak irakurtzeko. Lerro bakoitza SEARCH funtzioarekin aztertuko dugu emandako karaktere-katea aurkitu arte.

```
(DEFUN BILATU (KATEA FITXATEGIA)
  (WITH-OPEN-FILE (LERRO-KORRONTEA FITXATEGIA
    :DIRECTION :INPUT)
    (DO ((LERROA (READ-LINE LERRO-KORRONTEA NIL)
      (READ-LINE LERRO-KORRONTEA NIL)))
      ((NOT LERROA) (FORMAT T "~%Sarrera hori ez dator!"))
      (WHEN (SEARCH KATEA LERROA :TEST #'CHAR-EQUAL)
        (FORMAT T "~%~A" LERROA)
        (RETURN T))))))
```

Frogak egiterakoan, datu bezala "programagileak.dat" fitxategia erabili dezakegu, programagileen telefonoa eta herria gordetzen dituen:

Jauregi, Xabier	651037	TOLOSA
Lopetegi, Pello	512044	ORERETA
Etxeberria, Arantxa	881620	ORDIZIA
Agirre, Mikel	273390	DONOSTIA
Zubillaga, Itziar	653429	TOLOSA
Ugalde, Karmen	590867	ANDOAIN

Erabilpenaren adibideak:

> (BILATU "MIKEL" "programagileak.dat")

Agirre, Mikel 273390 DONOSTIA

T

> (BILATU "Iker" "programagileak.dat")

Sarrera hori ez dator!

NIL

ARIKETAK

- 10.1. Karaktere-katea bat fitxategi baten lerroetan bilatzen duen funtzio bat definitu nahi da. Funtzio honek karaktere-katea barruan duten lerro guztiak beste fitxategi batean idatziko ditu. Karaktere-katea bilatzen ez badu NIL bueltatuko du, eta bestela T.

11. MAKROAK

11.1. MAKROEN HELBURUAK.

Demagun FIRST funtzioaren izena ez dugula egokia ikusten, eta LEHENENGO izenaz erabili nahi genuke. Funtzio berri bat definituz helburu hori lortuko genuke (aurreko ariketa batetan proposatu zen bezala):

(DEFUN LEHENENGO (X) (FIRST X))

Modu honetara LEHENENGO erabili ahal da FIRSTen ordeez, baina ebaluazio berri bat egikaritu behar denez (LEHENENGO eta FIRSTena) luzcago gertatzen da ebaluazio osoa.

Gure helburua betetzeko modurik onena, egikaritu baino lehenago (LEHENENGO X) moduko formak (FIRST X) espresioetara itzuliko lituzkeen itzultzaile bat edukitzea litzateke. Konpiladoreaz lan egiten bada, itzulpen hori konpilatze-denboran burutuko da, eta beraz egikaritzapenean ez da inolako luzapenik gertatuko. Areago, zenbait interpretatzailek ere ba dute ahalmena era honetako itzulpenen errepikaketak baztertzeko.

Itzulpen hauek MAKROen bidez burutzen dira, eta itzulpenaren prozesuari MAKROAREN HEDAKETA esaten zaio.

Makroen funtsezko beste ezaugarria zera da deietako argumentuak ez direla ebaluatzen itzulpena egin aurretik. Espresio hedatuaren argumentuak bai, horiek ebaluatuko dira espresio hedatua ebaluatzeko. Makroen ezaugarri hau dela eta, aukera dugu forma berezi berriak definitzeko. DEFUNen bidez ezinezkoa da forma berezien definizioa argumentu guztien ebaluazioa bait dakar.

Adibidea:

BALDIN makroa erabiltzen duen ondoko s-espresiotik

(BALDIN (NUMBERP X) (* X X) X)

ondoko s-espresioa lor liteke bere hedadura gisa:


```
(COND ((NUMBERP X) (* X X))
      (T X))
```

Argumentuen ebaluazioa itzulpena egin aurretik burutuko balitz, (* X X) espresioaren ebaluazioa beti egingo litzateke eta Xen balioa zenbakia ez izatekotan errorea gertatuko ere.

11.2. MAKROEN DEFINIZIOA ETA EBALUAZIOA.

Makroak definitzeko DEFMACRO forma berezia erabiltzen da, ondoko sintaxiari jarraituz:

```
(DEFMACRO izena arg-lista gorputza)
```

izena: makroaren izena

arg-lista: makroaren argumentu-lista

gorputza: makroaren hedadura lortzeko ebaluatu behar den LISP-eko s-espresioa. Gehienetan espresio bakar batek osatzen du "gorputza" baina bat baino gehiago ere azaltzen dira batzutan albo-ondorioak lortzearren. Bat baino gehiago azaltzekotan azkenaren emaitza izango da makroaren hedadura.

Adibidea:

Lehen aipatu dugun LEHENENGO makroa honela definitzen da:

```
(DEFMACRO LEHENENGO (X) (LIST FIRST X))
```

Makro bati egindako dei bat bi pausotan ebaluatzen da:

1.- Makroaren hedaketa. Argumentuak ebaluatu gabe, makroaren gorputzeko espresioa ebaluatzen da makroaren hedadura lortzeko.

2.- Makroaren hedaduraren ebaluaketa, edozein s-espresio ebaluatzen den modura ebaluatzen da lortutako hedadura.

Adibidea:

```
(LEHENENGO (CONS 'A '(B C)))
```

Lehenengo pausoan (LIST FIRST X) espresioa ebaluatzen da non X-en balioa (CONS A '(B C)) argumentua da ebaluatu gabe. Beraz emaitza (makroaren hedadura) ondoko hau izango da.

```
(FIRST (CONS 'A '(B C)))
```

Bigarren pausoan azken espresio hori ebaluatzen da. FIRST funtzioa forma berezia ez denez bere argumentua ebaluatuko da (A B C) lista lortuz.

```
(FIRST (CONS 'A '(B C))) → A
```

Makroaren ebaluazioa laburbilduz

```
(LEHENENGO (CONS 'A '(B C))) → A
```

Adibidea: BALDIN makroa definitu, zeinek " baldintza ", " esp-arrakasta ", " esp-porrot " hiru espresioak hartuko ditu eta COND forma batetara itzuliko du. Hau da: eraiki BALDIN makroa, hiru argumentutako IF forma bereziaren modura erabili ahal izateko, IF formarik ez daukan LISP batetarako.

```
(DEFMACRO BALDIN (BALDINTZA ESP-ARRAKASTA ESP-PORROT)
  (LIST 'COND
        (LIST BALDINTZA ESP-ARRAKASTA)
        (LIST 'T ESP-PORROT)))
```

X sinboloaren balioa A dela emanda, kalkulatu ondoko espresioaren emaitza:

```
(BALDIN (NUMBERP X) (* X X) X)
```

Lehenengo pausoaren emaitza ondokoa da:

```
(COND ((NUMBERP X) (* X X))
      (T X))
```

Bigarren pausoaren emaitza: A

Zer gertatuko zatekeen ondoko funtzio hau erabili balitz:

```
(DEFUN BALDIN (BALDINTZA ESPARRAKASTA ESPPORROT)
  (COND (BALDINTZA ESPARRAKASTA)
        (T ESPPORROT)))
```

Bigarren argumentua ebaluatzerakoan errorea gertatu zatekeen. * eragiketak argumentu zenbakiak eskatzen baiditu.

11.3. MAKROAK DEFINITZEKO LAGUNTZA SINTAKTIKOAK. KOMATXO KARAKTEREA.

Makroak definitzen direnean CONS eta LIST funtzioetarako dei ugari azaltzen da orokorrean. Definizioak laburrago eta ulergarriago bihurtzearren idazkera berezi bat proposatzen da. (QUOTE X) espresioa `X bezala labur zitekeen, apostrofo karakterea QUOTE funtziorako deiaren adierazlea bait zen. Makroen idazkera erraztatzeko komatxo karakterea ( `) erabiliko da antzeko modura.

Komatxoa (`) BACQUOTE funtzioari deitzeko adierazlea da. BACQUOTE funtzioa QUOTE funtzioa bezala ebaluatzen da, baina bere argumentu barruan komarik (,) agertzen bada, orduan ebaluatu egin beharko da koma ondoan dagoen s-espresioa.

Adibidea:

```
(SETQ ALD `ADIBIDEA) → ADIBIDEA
`(HAU ,ALD DA) → (HAU ADIBIDEA DA)
```

, @ Karaktere-bikotekarekin ere erabili ahal da BACQUOTE forma berezia. , @ karaktere-bikoteak bere ondoan dagoen espresioaren ebaluazioa dakar, baina kasu honetan emaitza lista bat izango da derrigorrez. Sortzen ari den BACKQUOTEren emaitzan ez da sartzen lista bezala, listako elementu guztiak banan banan sartzen dira baina lista bat osatu gabe.

Adibidea:

```
( SETQ ALD `( ADIBIDE ZAILAGOA ))  
      → ( ADIBIDE ZAILAGOA )  
`( HAU ALD DA )  
      → ( HAU ALD DA )  
`( HAU , ALD DA )  
      → ( HAU ( ADIBIDE ZAILAGOA ) DA )  
`( HAU ,@ ALD DA )  
      → ( HAU ADIBIDE ZAILAGOA DA )
```

ARIKETAK

- 11.1. Definitu WHEN eta UNLESS makroak forma berezi aurredefinitu bezala ez dauzkaten sistemetarako.

(WHEN test esp1 esp2 ...) = (COND (test esp1 esp2 ...))
(UNLESS test esp1 esp2 ...) = (COND ((NOT test) esp1 esp2 ...))

- 11.2. Definitu HONDARRA eta ERAIKI makroak CDR eta CONS baino izen esankorragoak erabili ahal izateko.

- 11.3. Definitu BALDIN makroa IF forma berezia aurredefinitu bezala ez dauzkaten sistemetarako.

- 11.4. Definitu IZANIK makroa LET forma berezia aurredefinitu bezala ez dauzkaten sistemetarako.

- 11.5. DOLIST eta DOTIMES makroak definitu iterazio-eskema hauek erabiltzeko aukera eransteko horrelakoak definituta ez dauzkaten sistemetan.

- 11.6. Golden Common Lispez PUTPROP funtzio aurredefinitua ez da existitzen. <sym> sinboloaren <prop> propietaren balio berria <bal> balioa izan dadin ondoko forma ebaluatu behar da:

(SETF (GET <sym> <prop>) <bal>)

Definitu PUTPROP makroa lehendik beste dialekto batetarako idatzita geneuzkan PUTPROP guztiak Golden Common Lispek ulertu ahal ditzan.

- 11.7. APPLY funtzioa ez daukan LISPeke sistema batetan definitu APLIKA makroa APPLYren baliokide gisa erabili ahal izateko.

- 11.8. FUNCALL funtzioa ez daukan LISPeke sistema batetan definitu FUNDEITU makroa FUNCALLen baliokide gisa erabili ahal izateko.

11.9. Aztertu honako definizio hau:

```
(DEFMACRO CASE (GILTZA &REST BALIO-EXP-LISTA)
  `(COND ,@(MAPCAR #'CONS
                    (MAPCAR #'(LAMBDA (X)
                                (LIST  EQ
                                      GILTZA
                                      `(QUOTE ,(FIRST X))))
                    BALIO-EXP-LISTA)
          (MAPCAR #'REST BALIO-EXP-LISTA))))
```

Zeintzu dira makroaren hedakuntza eta emaitza ondoko formak ebaluatzen badira X, Y eta Z sinboloen balioak hurrenes hurren 3, 3 eta 8 direnean?

```
(CASE X
  (1 (+ Y Z))
  (2 (- Y Z))
  (3 (* Y Z))
  (4 (/ Y Z)))

(CASE X
  (Y (+ Y Z))
  (Z (- Y Z)))
```

11.10. LISPen dialekto batzuk DEFINE forma berezia erabiltzen dute funtzio berriak definitzeko, DEFUN erabili ordez. Bien artean dagoen diferentzia bakarra syntaxian dago, DEFINE erabiltzeko modua ondokoa da eta:

```
(DEFINE ( funtzioaren_izena parametro-lista )
        funtzioaren_gorputza )
```

Sortu ezazu DEFINE makroa, forma hori Common LISPez erabili ahal dezazun.

11.11. Aztertu honako definizio hau:

```
(DEFMACRO IFX (N E1 E2)
  (COND ((ODDP N) `(1+ ,E1))
        (T  `(1- ,E2))))
```

Zeintzu dira makroaren hedakuntza eta emaitza ondoko formak ebaluatzen badira X eta Y sinboloen balioak hurrenez hurren 4 eta A direnean?

```
(IFX 5 (* X 4) Y)
(IFX 6 (* X 4) Y)
(IFX X (* X 4) Y)
(IFX (1+ X) (* X 4) Y)
```

11.12. Aztertu aurreko ariketakoarekin antza handia duen honako definizio hau:

```
(DEFMACRO IFX2 (N E1 E2)
  (COND ((ODDP (EVAL N)) `(1+ ,E1))
        (T `(1- ,E2)))))
```

Zeintzu dira makroaren hedakuntza eta emaitza ondoko formak ebaluatzen badira X eta Y sinboloen balioak hurrenez hurren 4 eta A direnean?

```
(IFX2 5 (* X 4) Y)
(IFX2 6 (* X 4) Y)
(IFX2 X (* X 4) Y)
(IFX2 (1+ X) (* X 4) Y)
```

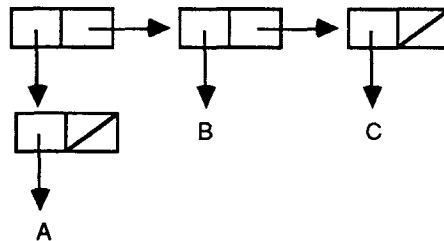
ARIKETEN EBAZPENAK

Bigarren kapitulua

2.1. Ariketa

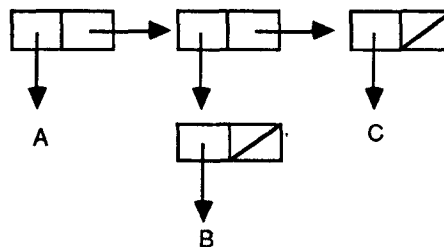
a.-

$$\begin{aligned} ((A) B C) &= ((A . NIL) B C) = ((A . NIL) . (B C)) = \\ &= ((A . NIL) . (B . (C))) = ((A . NIL) . (B . (C . NIL))) \end{aligned}$$



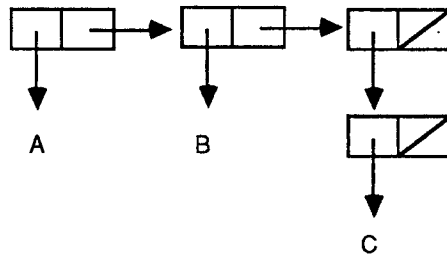
b.-

$$\begin{aligned} (A (B) C) &= (A . ((B) C)) = (A . ((B) . (C))) = \\ &= (A . ((B . NIL) . (C . NIL))) \end{aligned}$$



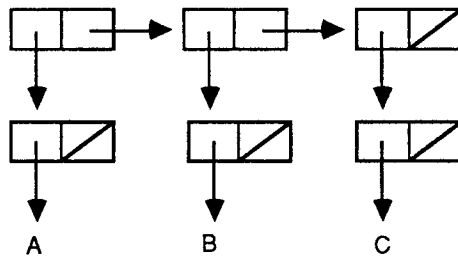
c.-

$$\begin{aligned} (A B (C)) &= (A . (B (C))) = (A . (B . ((C)))) = (A . (B . ((C) . NIL))) = \\ &= (A . (B . ((C . NIL) . NIL))) \end{aligned}$$



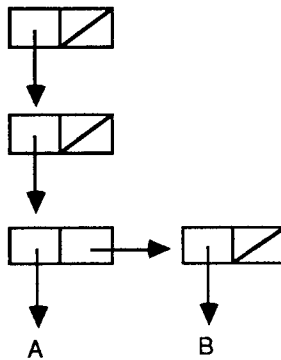
d.-

$$\begin{aligned}
 ((A)(B)(C)) &= ((A) \cdot ((B)(C))) = ((A \cdot \text{NIL}) \cdot ((B) \cdot ((C)))) = \\
 &= ((A \cdot \text{NIL}) \cdot ((B \cdot \text{NIL}) \cdot ((C) \cdot \text{NIL}))) = \\
 &= ((A \cdot \text{NIL}) \cdot ((B \cdot \text{NIL}) \cdot ((C \cdot \text{NIL}) \cdot \text{NIL})))
 \end{aligned}$$



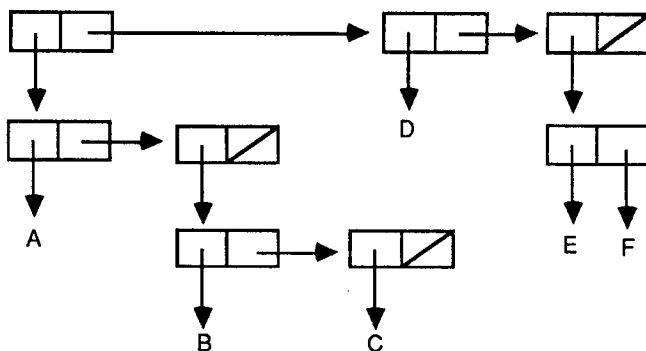
e.-

$$\begin{aligned}
 (((A \ B))) &= (((A \ B)) \cdot \text{NIL}) = (((A \ B) \cdot \text{NIL}) \cdot \text{NIL}) = \\
 &= (((A \cdot (B)) \cdot \text{NIL}) \cdot \text{NIL}) = (((A \cdot (B \cdot \text{NIL})) \cdot \text{NIL}) \cdot \text{NIL})
 \end{aligned}$$



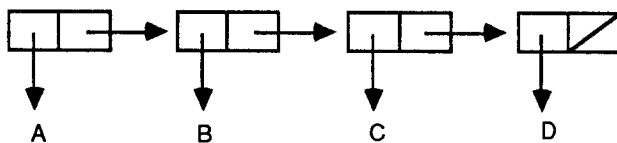
f.-

$$\begin{aligned}
 ((A (B C)) D (E . F)) &= ((A (B C)).(D (E . F))) = \\
 &= ((A . ((B C))) . (D . ((E . F)))) = \\
 &= ((A . ((B C) . NIL)) . (D . ((E . F) . NIL))) = \\
 &= ((A . ((B . (C)) . NIL)) . (D . ((E . F) . NIL))) = \\
 &= ((A . ((B . (C . NIL)) . NIL)) . (D . ((E . F) . NIL)))
 \end{aligned}$$



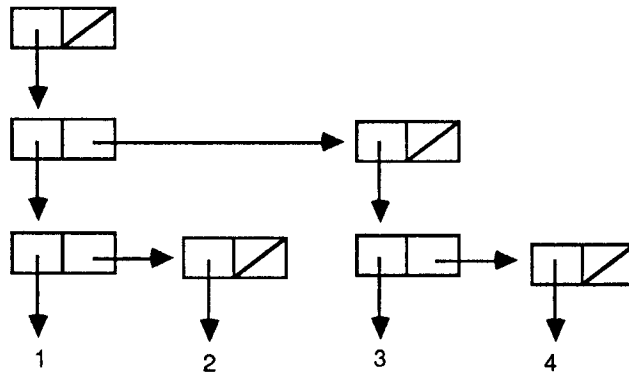
g.-

$$(A . (B C D)) = (A . (B . (C . (D . NIL))))$$



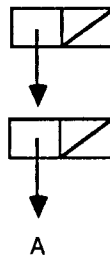
h.-

$$\begin{aligned}
 (((1 2) (3 4)) . NIL) &= (((1 2) . ((3 4)) . NIL) . NIL) \\
 &= (((1 . (2)) . ((3 4) . NIL)) . NIL) \\
 &= (((1 . (2 . NIL)) . ((3 . (4)) . NIL)) . NIL) \\
 &= (((1 . (2 . NIL)) . ((3 . (4 . NIL)) . NIL)) . NIL)
 \end{aligned}$$



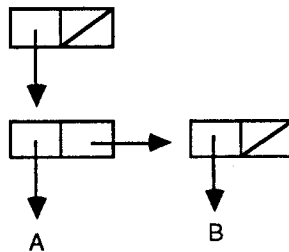
i.-

$$((A)) = ((A) . \text{NIL}) = ((A . \text{NIL}) . \text{NIL})$$



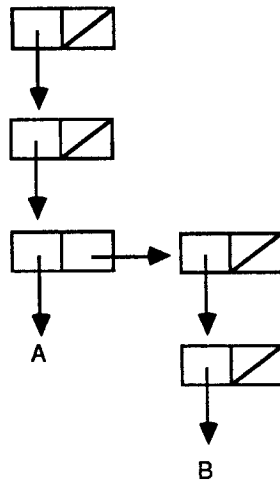
j.-

$$((A\ B)) = ((A\ B) . \text{NIL}) = ((A . (B)) . \text{NIL}) = ((A . (B . \text{NIL})) . \text{NIL})$$

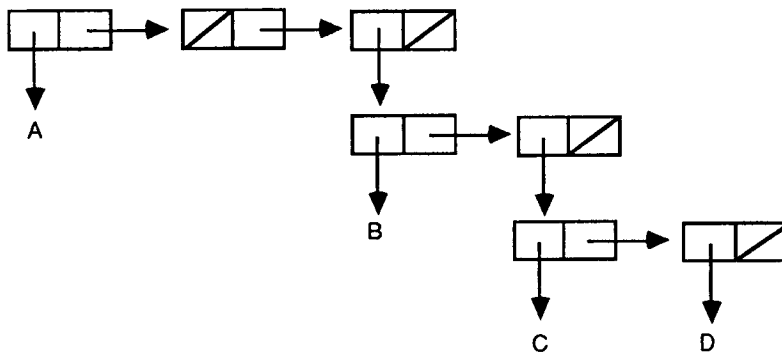


k.-

$$\begin{aligned} (((A\ (B)))) &= (((A\ (B))) . \text{NIL}) = (((A\ (B)) . \text{NIL}) . \text{NIL}) = \\ &= (((A . ((B)))) . \text{NIL}) . \text{NIL}) = \\ &= (((A . ((B) . \text{NIL})) . \text{NIL}) . \text{NIL}) = \\ &= (((A . ((B . \text{NIL}) . \text{NIL})) . \text{NIL}) . \text{NIL}) \end{aligned}$$



1.-

$$\begin{aligned}
 (A \text{ NIL } (B (C D))) &= \\
 &= (A . (NIL (B (C D)))) = (A . (NIL . ((B (C D))))) = \\
 &= (A . (NIL . ((B (C D)) . NIL))) = \\
 &= (A . (NIL . ((B . ((C D)) . NIL))) = \\
 &= (A . (NIL . ((B . ((C D) . NIL)) . NIL))) = \\
 &= (A . (NIL . ((B ((C . (D)) . NIL)) . NIL))) = \\
 &= (A . (NIL . ((B . ((C . (D . NIL)) . NIL)) . NIL)))
 \end{aligned}$$


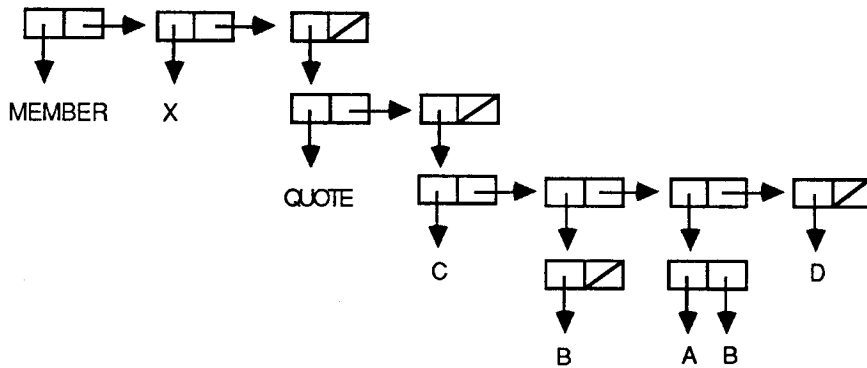
m.-

$$\begin{aligned}
 (\text{MEMBER } X (\text{QUOTE } (C (B) (A . B) D))) &= \\
 &= (\text{MEMBER } .X (\text{QUOTE } (C (B) (A . B) D))) = \\
 &= (\text{MEMBER } .X .((\text{QUOTE } (C (B) (A . B) D)))) = \\
 &= (\text{MEMBER } .X .((\text{QUOTE } (C (B) (A . B) D)) . \text{NIL})))
 \end{aligned}$$

```

= (MEMBER .(X .(( QUOTE .(( C ( B ) ( A . B ) D ))) . NIL)))
= (MEMBER .(X .(( QUOTE .(( C ( B ) ( A . B ) D ) . NIL)) . NIL)))
= (MEMBER .(X .(( QUOTE .(( C .(( B ) ( A . B ) D ))) . NIL)) . NIL)))
= (MEMBER .(X .(( QUOTE .(( C .(( B ) .(( A . B ) D ))) . NIL)) .
NIL)))
= (MEMBER .(X .(( QUOTE .(( C .(( B . NIL ) .(( A . B ) .(D)))) .
NIL)) . NIL)))
= (MEMBER .(X .(( QUOTE .(( C .(( B . NIL ) .(( A . B ) .(D . NIL)))) .
NIL)) . NIL)))

```

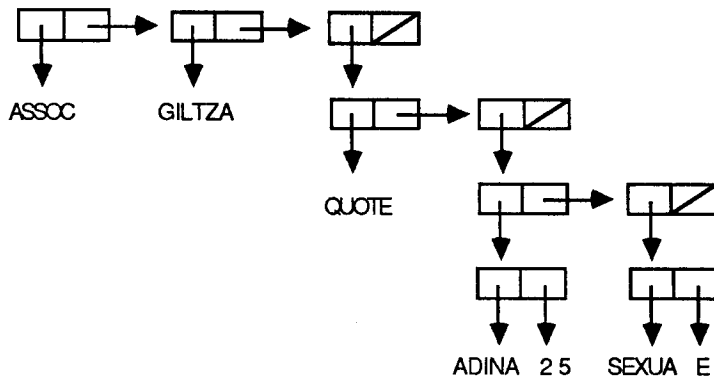


n.-

```

( ASSOC GILTZA ( QUOTE `(( ADINA . 25 ) ( SEXUA . M )))
= (ASSOC .(GILTZA (QUOTE ((ADINA . 25 ) ( SEXUA . M )))))
= (ASSOC .(GILTZA .((QUOTE ((ADINA . 25 ) ( SEXUA . M )))))
= (ASSOC .(GILTZA .((QUOTE ((ADINA . 25 ) ( SEXUA . M ))) .
NIL)))
= (ASSOC .(GILTZA .((QUOTE .( ((ADINA . 25 ) ( SEXUA . M )))) .
NIL)))
= (ASSOC .(GILTZA .((QUOTE .( ((ADINA . 25 ) ( SEXUA . M ) .
NIL)) . NIL)))
= (ASSOC .(GILTZA .((QUOTE .( ((ADINA . 25 ) .(( SEXUA . M )) .
NIL)) . NIL)))
= (ASSOC .(GILTZA .((QUOTE .( ((ADINA . 25 ) .((SEXUA . M ) .
NIL)) . NIL)) . NIL)))

```



2.2. Ariketa.

a.-

$$((A. NIL). (B. (C. NIL))) = ((A). (B. (C))) = ((A). (B C)) = ((A) B C)$$

b.-

$$(A. ((B. (C. NIL)). (C. NIL))) = (A. ((B. (C)). (C))) = (A. ((B C). (C))) = (A. ((B C) C)) = (A (B C) C)$$

c.-

$$\begin{aligned} & ((HEMEN. (BI. NIL)). ((LISTA. (DITUGU. NIL)). NIL)) = \\ & = ((HEMEN. (BI)). ((LISTA. (DITUGU)). NIL)) = \\ & = ((HEMEN BI). ((LISTA DITUGU). NIL)) = \\ & = ((HEMEN BI). ((LISTA DITUGU))) = \\ & = ((HEMEN BI) (LISTA DITUGU)) \end{aligned}$$

d.-

$$((A. B). (C. (D. E)))$$

e.-

$$\begin{aligned} & ((A. (B. (C. NIL))). (((D. NIL) E. NIL)). (F. (H. NIL)))) = \\ & = ((A. (B. (C))). (((D). (E)). (F. (H)))) = \\ & = ((A. (B C)). (((D) E). (F H))) = \\ & = ((A B C). (((D) E) F H)) = \\ & = ((A B C) ((D) E) F H) \end{aligned}$$

f.-

(KONTA . ((FIRST . (L. NIL)). ((MEMBER. ((QUOTE . (X . NIL)) . ((A
 . LI) . NIL))) . NIL))) =
 = (KONTA . ((FIRST . (L)) . ((MEMBER. ((QUOTE . (X)) . ((A . LI)
)) . NIL))) =
 = (KONTA . ((FIRST L) . ((MEMBER . ((QUOTE X) . ((A . LI)))))) =
 = (KONTA . ((FIRST L) . ((MEMBER . ((QUOTE X) (A . LI)))))) =
 = (KONTA . ((FIRST L) . ((MEMBER (QUOTE X) (A . LI))))) =
 = (KONTA . ((FIRST L) (MEMBER (QUOTE X) (A . LI)))) =
 = (KONTA (FIRST L) (MEMBER (QUOTE X) (A . LI)))

Laugarren kapitulua

4.1. Ariketa

a.-

(FIRST '(P H W)) 0 P
 (REST '(B K P H)) 0 (K P H)
 (FIRST '((A B) (C D))) 0 (A B)
 (REST '((A B) (C D))) 0 ((C D))
 (FIRST (REST '(A B) (C D))) 0 (C D)
 (REST (FIRST '(A B) (C D))) 0 (B)
 (REST (FIRST (REST '(A B) (C D)))) 0 (D)
 (FIRST (REST (FIRST '(A B) (C D)))) 0 B
 (FIRST (REST (FIRST (REST '(A B) (C D) (E F))))) 0 D
 (FIRST (FIRST (REST (REST '(A B) (C D) (E F))))) 0 E
 (FIRST (FIRST (REST '(REST ((A B) (C D) (E F))))) 0 (A B)
 (FIRST (FIRST '(REST (REST ((A B) (C D) (E F))))) 0 ERRORA
 (FIRST '(FIRST (REST (REST ((A B) (C D) (E F))))) 0 FIRST
 '(FIRST (FIRST (REST (REST ((A B) (C D) (E F))))) 0
 '(FIRST (FIRST (REST (REST ((A B) (C D) (E F)))))

b.-

(FIRST (REST (REST
 '(SAGARRA LARANJA MADARIA MAHATSA)
))) 0 MADARIA
 (FIRST (FIRST (REST
 '(SAGARRA LARANJA) (MADARIA MAHATSA))
))) 0 MADARIA

```

(FIRST (FIRST (REST (REST (FIRST
  '(((SAGARRA) (LARANJA) (MADARIA) (MAHATSA)))
  )))) 0 MADARIA
(FIRST (FIRST (REST (FIRST (REST
  '(SAGARRA (LARANJA (MADARIA (MAHATSA))))
  )))) 0 MADARIA
(FIRST (FIRST (REST (REST
  '(((SAGARRA))) ((LARANJA)) (MADARIA) MAHATSA)
  )))) 0 MADARIA
(FIRST (REST (FIRST
  '(((SAGARRA) LARANJA) MADARIA) MAHATSA)
  ))) 0 MADARIA

```

4.2. Ariketa

```

( APPEND L L) 0 (A B A B)
( APPEND L L L) 0 (A B A B A B)
( APPEND '(A) '() '(B) '() ) 0 (A B)
( APPEND 'L L) 0 ERROREA
( APPEND '(A) '(B) '((C) (D)) ) 0 (A B (C) (D))
( APPEND '((AUTOA FORD) (EDARIA SAGARDOA)))
  0 ((AUTOA FORD) (EDARIA SAGARDOA))
( LIST L L) 0 ((A B) (A B))
( LIST L L L) 0 ((A B) (A B) (A B))
( LIST 'L L) 0 (L (A B))
( LIST '(A) '(B) '((A) (B)) ) 0 ((A) (B) ((A) (B)))
( LIST '(A B) '(C D)) 0 ((A B) (C D))
( APPEND '(A B) '(C D)) 0 (A B C D)
( CONS L L) 0 ((A B) A B)
( CONS 'L L) 0 (L A B)
( LENGTH L) 0 2
( LENGTH 'L) 0 ERROREA
( LENGTH '(A B)) 0 2
( LENGTH '((A B) (C D))) 0 2
( LENGTH (APPEND L L)) 0 4
( LENGTH '(PLATON SOCRATES ARISTOTELES)) 0 3
( LENGTH '((PLATON) (SOCRATES) (ARISTOTELES))) 0 3

```



```

(LENGTH '((PLATON SOCRATES ARISTOTELES))) 0 1
(REVERSE '(A B)) 0 (B A)
(REVERSE '( (A B) '(C D))) 0 ( '(C D) (A B) )
(REVERSE L) 0 (B A)
(REVERSE (APPEND L L)) 0 (B A B A)
(REVERSE '(PLATON SOCRATES ARISTOTELES))
      0 (ARISTOTELES SOCRATES PLATON)
(REVERSE '( (PLATON) (SOCRATES) (ARISTOTELES) ))
      0 ( (ARISTOTELES) (SOCRATES) (PLATON) )
(REVERSE '((PLATON SOCRATES ARISTOTELES)))
      0 ( (PLATON SOCRATES ARISTOTELES) )
(REST '(A B) ) 0 (B)
(REST (LIST L L) ) 0 ((A B))
(NTH 4 (APPEND L L)) 0 ERROR
(NTHCDR 3 (APPEND L L)) 0 (B)
(LAST L) 0 (B)
(LAST '((A B) (C D))) 0 ((C D))
(LAST 'A) 0 ERROR
(LAST '(A B C D E)) 0 (E)
(APPEND '(A B C) '() ) 0 (A B C)
(LIST '(A B C) '() ) 0 ((A B C) NIL)
(CONS '(A B C) '() ) 0 ((A B C))

```

4.3. Ariketa

a.-

```

X 0 (MARITXU NORA ZOAZ)
Y 0 (1 2 3)
(NCONC X Y) 0 (MARITXU NORA ZOAZ 1 2 3)
X 0 (MARITXU NORA ZOAZ 1 2 3)
Y 0 (1 2 3)
(NCONC Y X) 0 (1 2 3 MARITXU NORA ZOAZ 1 2 3)
X 0 (MARITXU NORA ZOAZ 1 2 3)
Y 0 (1 2 3 MARITXU NORA ZOAZ)

```

b.-

X 0 (MARITXU NORA ZOAZ)
Y 0 (1 2 3)
(APPEND X Y) 0 (MARITXU NORA ZOAZ 1 2 3)
X 0 (MARITXU NORA ZOAZ)
Y 0 (1 2 3)
(APPEND Y X) 0 (1 2 3 MARITXU NORA ZOAZ)
X 0 (MARITXU NORA ZOAZ)
Y 0 (1 2 3)

c.-

X 0 (MARITXU NORA ZOAZ)
Y 0 (1 2 3)
(RPLACA X Y) 0 ((1 2 3) NORA ZOAZ)
X 0 ((1 2 3) NORA ZOAZ)
Y 0 (1 2 3)

d.-

X 0 (MARITXU NORA ZOAZ)
Y 0 (1 2 3)
(RPLACD X Y) 0 (MARITXU 1 2 3)
X 0 (MARITXU 1 2 3)
Y 0 (1 2 3)

4.4. Ariketa

(ATOM 'DIGITUAK) 0 T
(ATOM 'HIRU) 0 T
(ATOM DIGITUAK) 0 NIL
(ATOM '(ZERO BAT BI HIRU LAU)) 0 NIL
(EQUAL DIGITUAK DIGITUAK) 0 T
(EQUAL 'DIGITUAK 'DIGITUAK) 0 T
(EQUAL DIGITUAK '(ZERO BAT BI HIRU LAU)) 0 T
(EQUAL DIGITUAK DIGITUAK) 0 NIL
(NULL '()) 0 T
(NULL T) 0 T

(NULL ()) 0 T
(NULL DIGITUAK) 0 NIL
(NULL 'DIGITUAK) 0 NIL
(NULL NIL) 0 T
(MEMBER 'HIRU DIGITUAK) 0 (HIRU LAU)
(MEMBER 'BOST DIGITUAK) 0 NIL
(MEMBER 'BI 'DIGITUAK) 0 ERROREA
(MEMBER 'BI '((BAT BI HIRU) (LAU BOST))) 0 NIL
(SYMBOLP DIGITUAK) 0 NIL
(SYMBOLP 'UNO) 0 T
(SYMBOLP 2) 0 NIL
(NUMBERP (FIRST DIGITUAK)) 0 NIL
(NUMBERP 2) 0 T
(NOT NIL) 0 T
(NOT T) 0 NIL
(NOT 'TXAKURRA) 0 NIL
(NOT (MEMBER 'BAT DIGITUAK)) 0 NIL
(NOT (MEMBER 'SEI DIGITUAK)) 0 T
(AND 'TXAKURRA 'KATUA) 0 KATUA
(OR 'TXAKURRA 'KATUA NIL) 0 TXAKURRA

4.5. Ariketa

(FUNCALL + A1 A2 A3) 0 ERROREA
(FUNCALL #' + A1 A2 A3) 0 6
(FUNCALL #' + 'A1 'A2 'A3) 0 ERROREA
(FUNCALL + 1 2 3) 0 ERROREA
(FUNCALL #' + 1 2 3) 0 6
(FUNCALL #' + '1 '2 '3) 0 6
(FUNCALL (FIRST ERL) L) 0 ERROREA
(APPLY + A1 A2 A3) 0 ERROREA
(APPLY (FIRST ERL) A1 A2 A3) 0 ERROREA
(APPLY (FIRST ERL) L) 0 6
(+ L) 0 ERROREA
(+ A1 A2 A3) 0 6
(EVAL (CONS (FIRST ERL) '(A1 A2 A3))) 0 6
(EVAL (APPEND '(+ 1) '(A2 A3))) 0 6

Bostgarren kapitulua**5.1. Ariketa**

(DEFUN LEHEN (L)
 (FIRST L))

(DEFUN HONDAR (L)
 (REST L))

(DEFUN KATEATU (L1 L2)
 (APPEND L1 L2))

(DEFUN LISTA (X Y)
 (LIST X Y))

5.2. Ariketa

(DEFUN TRUKABI (L)
 (CONS (FIRST (REST L))
 (CONS (FIRST L)
 (REST (REST L)))))

5.3. Ariketa

(DEFUN PORTZENTAIA (X Y)
 (/ (* X 100)
 Y))

5.4. Ariketa

(DEFUN COTAN (X)
 (/ 1 (TAN X)))

(DEFUN SEC (X)
 (/ 1 (COS X)))

(DEFUN COSEC (X)
 (/ 1 (SIN X)))

5.5. Ariketa

```
(DEFUN EMAITZA (L)
  (AND (EQUAL (LENGTH L) 4)
    (OSOPOSITIBO (SECOND L))
    (OSOPOSITIBO (FOURTH L))))
```

```
(DEFUN OSOPOSITIBO (X)
  (AND (PLUSP X)
    (= X (TRUNCATE X))))
```

5.6. Ariketa

```
(DEFUN EZKER-BIRA (L)
  (APPEND (REST L)
    (LIST (FIRST L))))
```

5.7. Ariketa

```
(DEFUN ESKUIN-BIRA (L)
  (APPEND (LAST L)
    (REVERSE (REST (REVERSE L)))))
```

5.8. Ariketa

```
(DEFUN IZPILU (L)
  (APPEND L (REVERSE L)))
```

5.9. Ariketa

```
(DEFUN TRIANGELU (HIP KAT1 KAT2)
  (> 2
    (PORTZENTAIA (ABS (- (* HIP HIP)
      (+ (* KAT1 KAT1)
        (* KAT2 KAT2))))
      (* HIP HIP))))
```

5.10. Ariketa

```
(DEFUN ERROAK (A B C)
  (LIST (/ (+ (- 0 B)
    (SQRT (DISKRIM A B C))
```

```

      (* 2 A))
(/ (- (- 0 B)
      (SQRT (DISKRIM A B C))
      (* 2 A))))

```

```

(DEFUN DISKRIM (A B C)      ; B2 - 4AC diskriminatzailea
  (- (* B B)                ; kalkulatzeko du.
    (* 4 A C)))

```

5.11. Ariketa

```

(DEFUN KONPLEXUAK (A B C)
  (> 0 (- (* B B)
    (* 4 A C))))

```

edo lehengo DISKRIM funtzioa erabiliz:

```

(DEFUN KONPLEXUAK (A B C)
  (> 0 (DISKRIM A B C)))

```

5.12. Ariketa

(FUN1 7) 0	X:7	L:NIL	M:10	N:NIL	R:NIL
(FUN1 7 4 3) 0	X:7	L:4	M:3	N:NIL	R:NIL
(FUN1 7 4 3 5) 0	X:7	L:4	M:3	N:5	R:NIL
(FUN1 7 4 3 5 '(A) '(B)) 0	X:7	L:4	M:3	N:5	R:((A) (B))
(FUN2 2 3) 0	X:2	Y:3	L:NIL	M:10	N:NIL
(FUN2 2 3 :L 3) 0	X:2	Y:3	L:3	M:10	N:NIL
(FUN2 2 3 :N 2 :M 3 :L 4) 0	X:2	Y:3	L:4	M:3	N:2

5.13. Ariketa

```

(DEFUN ORDUZ (MIN &optional (SEG 0))
  (/ (+ (* MIN 60.0) SEG)
    3600))

```

5.14. Ariketa

```

(DEFUN MAXIMO (X &rest L)
  (COND ((ENDP L) X)

```

```
( (>= X (FIRST L) )  
  (EVAL (CONS MAXIMO  
          (CONS X (REST L))))))  
(T (EVAL (CONS MAXIMO L))))
```

5.15. Ariketa

- a)
(DEFUN ORDEZKATU (AT1 AT2 L &key (ALDIAK 1) ATZERA)
- b)
(ORDEZKATU V B L :ALDIAK 3)
- c)
(ORDEZKATU V B L :ATZERA BAI)

Seigarren kapitulua

6.1. Ariketa

```
(DEFUN LEHENATOMOA (L)  
  (COND ( (NULL L) NIL)  
        ((ATOM L) L)  
        (T (LEHENATOMOA (FIRST L))))))
```

6.2. Ariketa

```
(DEFUN KENDU-LEHEN-ATOMOA (L)  
  (COND ((NULL L) NIL)  
        ((ATOM (FIRST L)) (REST L))  
        (T (CONS (KENDU-LEHEN-ATOMOA (FIRST L))  
                  (REST L)))))
```

6.3. Ariketa

```
(DEFUN KENDUATOMOA (L A)  
  (COND ((NULL L) NIL)  
        ((ATOM (FIRST L))  
         (COND ((EQUAL A (FIRST L))  
                (KENDUATOMOA (REST L) A))
```

```

(T (CONS (FIRST L)
          (KENDUATOMOA (REST L) A))))
(T (CONS (KENDUATOMOA (FIRST L) A)
          (KENDUATOMOA (REST L) A))))

```

6.4. Ariketa

Espresio baten kabiaketa maila maximoa ematen digu.

6.5. Ariketa

Funtzio identitatea du. Espresio bera itzultzen du.

6.6. Ariketa

```

(DEFUN BILKETA (L1 L2)
  (COND ((NULL L1) L2)
        ((NOT (MEMBER (FIRST L1) L2))
         (CONS (FIRST L1)
                (BILKETA (REST L1) L2)))
        (T (BILKETA (REST L1) L2))))

```

```

(DEFUN EBAKETA (L1 L2)
  (COND ((OR (NULL L1) (NULL L2)) NIL)
        ((MEMBER (FIRST L1) L2)
         (CONS (FIRST L1)
                (EBAKETA (REST L1) L2)))
        (T (EBAKETA (REST L1) L2))))

```

```

(DEFUN BERDINP (L1 L2)
  (AND (BARNEAN-DAGO L1 L2) (BARNEAN-DAGO L2 L1)))

```

```

(DEFUN BARNEAN-DAGO (L1 L2)
  (COND ((NULL L1) T)
        ((MEMBER (FIRST L1) L2)
         (BARNEAN-DAGO (REST L1) L2))
        (T NIL))) ; azken forma hau kendu daiteke

```


6.7. Ariketa

```
(DEFUN ZENBAKIBATURA (L)
  (COND ((NULL L) 0)
        ((ATOM (FIRST L))
          (COND ((NUMBERP (FIRST L))
                  (+ (FIRST L)
                     (ZENBAKIBATURA (REST L))))
              (T (ZENBAKIBATURA (REST L)))))
        (T (+ (ZENBAKIBATURA (FIRST L))
               (ZENBAKIBATURA (REST L))))))
```

6.8. Ariketa

```
(DEFUN LAUTU (L)
  (COND ((NULL L) NIL)
        ((ATOM L) (LIST L))
        (T (APPEND (LAUTU (FIRST L))
                    (LAUTU (REST L))))))
```

6.9. Ariketa

```
(DEFUN BEKT-ESK-BIDERKADURA (E B)
  ;; SPARSE bektore bat eta eskalar baten arteko biderkadura
  (COND ((OR (NULL B) (ZEROP E)) NIL)
        (T (CONS (LIST (FIRST (FIRST B))
                        (* E (SECOND (FIRST B))))
                   (BEKT-ESK-BIDERKADURA E (REST B)))))
```

6.10. Ariketa

```
(DEFUN BIDER-ESKALARRA (B1 B2)
  ;; bi sparse bektoreen arteko biderkadura
  (COND ((OR (NULL B1) (NULL B2)) 0)
        ((= (FIRST (FIRST B1)) (FIRST (FIRST B2)))
          (+ (* (SECOND (FIRST B1))
                (SECOND (FIRST B2)))
             (BIDER-ESKALARRA (REST B1) (REST B2))))
        (> (FIRST (FIRST B1)) (FIRST (FIRST B2))))
```

```

(BIDER-ESKALARRA B1 (REST B2)))
((< (FIRST (FIRST B1)) (FIRST (FIRST B2))))
(BIDER-ESKALARRA (REST B1) B2)))

```

6.11. Ariketa

```

(DEFUN BEKTORE-BATURA (B1 B2)
;; bi sparse bektoreen arteko batura
(COND ((ENDP B1) B2)
      ((ENDP B2) B1)
      ((< (FIRST (FIRST B1)) (FIRST (FIRST B2))))
      (CONS (FIRST B1)
             (BEKTORE-BATURA (REST B1) B2)))
      ((< (FIRST (FIRST B2)) (FIRST (FIRST B1))))
      (CONS (FIRST B2)
             (BEKTORE-BATURA B1 (REST B2))))
      (T (CONS (LIST (FIRST (FIRST B1))
                     (+ (SECOND (FIRST B1))
                       (SECOND (FIRST B2))))
                (BEKTORE-BATURA (REST B1)
                                  (REST B2)))))))

```

6.12. Ariketa

```

(DEFUN BEKT-MAT-BIDERKADURA (M B)
;; sparse bektore bat eta matrize baten arteko biderkadura
(COND ((OR (ENDP B) (ENDP M)) NIL)
      ((< (FIRST (FIRST B)) (FIRST (FIRST M)))
       (BEKT-MAT-BIDERKADURA M (REST B)))
      ((< (FIRST (FIRST M)) (FIRST (FIRST B)))
       (BEKT-MAT-BIDERKADURA (REST M) B))
      (T (BEKTORE-BATURA
          (BEKT-ESK-BIDERKADURA (SECOND (FIRST B))
                                  (SECOND (FIRST M)))
          (BEKT-MAT-BIDERKADURA (REST M)
                                  (REST B))))))

```

6.13. Ariketa

```
(DEFUN MATRIZE-BIDERSKADURA (M1 M2)
;; bi matrizeen arteko biderskadura
(COND ((NULL M1) NIL)
      (T (CONS (LIST (FIRST (FIRST M1))
                     (BEKT-MAT-BIDERSKADURA
                      M2
                      (SECOND (FIRST M1))))
                (MATRIZE-BIDERSKADURA (REST M1) M2)))))
```

6.14. Ariketa

```
(DEFUN PISUA (ZUHAITZ)
(COND ((ATOM ZUHAITZ) 1)
      ((= (PISUA (SECOND ZUHAITZ))
          (PISUA (THIRD ZUHAITZ)))
       (1+ (PISUA (SECOND ZUHAITZ))))
      (T (MAX (PISUA (SECOND ZUHAITZ))
               (PISUA (THIRD ZUHAITZ)))))
```

6.15. Ariketa

```
(DEFUN MUGIKARIP (M)
(COND ((ATOM M) M)
      (T (MUG-EGITURATUA (FIRST M)
                          (MUGIKARIP (SECOND M))
                          (MUGIKARIP (THIRD M)))))
```

```
(DEFUN MUG-EGITURATUA (Z EZK ESK)
;Z, EZK eta ESK mugikari konposatu bat definitzen dute
;orekatuta ez badago NIL itzuliko du, bestela bere pisu osoa
(AND EZK
     ESK
     (= EZK ESK)
     (+ Z EZK ESK)))
```

6.16. Ariketa

```
(DEFUN KONPILA-ARIT (S)
  (KONPILA-ARIT-LAG 1 S))
```

```
(DEFUN KONPILA-ARIT-LAG (R S)
  (COND ((ATOM S) (LIST (LIST 'MOVE R S)))
        (T (APPEND (KONPILA-ARIT-LAG R (SECOND S))
                     (KONPILA-ARIT-LAG (1+ R) (THIRD S))
                     (LIST (LIST (ERAGIKETA (FIRST S))
                                   R
                                   (1+ R)))))))
```

```
(DEFUN ERAGIKETA (OP)
  (COND ((EQUAL OP '+) 'ADD)
        ((EQUAL OP '-') 'SUB)
        ((EQUAL OP '*') 'MUL)
        ((EQUAL OP '/') 'DIV)))
```

6.17. Ariketa

```
(DEFUN SAILKATU (L)
  ;L: elementuen arteko erlazioak (bi elementutako azpilistazko lista).
  ;Emitza: baliokidetasun klaseak (elementuzko azpilistazko lista).
  (SAILKATU-LAG L NIL))
```

```
(DEFUN SAILKATU-LAG (LP LK)
  ;LK: aldeaz aurretik sortutako baliokidetasun klaseak.
  ;LP: erlazio berrizko lista bat.
  ;Emitza: baliokidetasun klaseak erlazio berriak kontutan hartuta.
  (COND ((NULL LP) LK)
        (T (SAILKATU-LAG (REST LP)
                           (BILDU (FIRST LP) LK)))))
```

```
(DEFUN BILDU (P LK)
  ; P erlazio berri bat.
  ; LK baliokidetasun klaseak P erlazioa kontutan hartu gabe.
  ; Emitza: baliokidetasun klaseak P erlazioa kontutan hartuta.
```

```
(CONS (BILKURA (KLASE (FIRST P) LK)
              (KLASE (SECOND P) LK))
      (BESTEKLASEAK P LK)))
```

```
(DEFUN KLASE (E LK)
; E elementu bat. LK baliokidetasun klaseak (lista bat).
; Eraitza: E elementuaren baliokidetasun klasea LK listan.
; (E) lista E elementua inongo klasetan agertzen ez bada.
(COND ((NULL LK) (LIST E))
      ((MEMBER E (FIRST LK)) (FIRST LK))
      (T (KLASE E (REST LK)))))
```

```
(DEFUN BILKURA (C1 C2)
; C1 eta C2 multzoen bilkura (listak elementuak errepikatu gabekoak).
(COND ((NULL C1) C2)
      ((MEMBER (FIRST C1) C2) (BILKURA (REST C1) C2))
      (T (CONS (FIRST C1)
                (BILKURA (REST C1) C2)))))
```

```
(DEFUN BESTEKLASEAK (P LK)
; P erlazioa (bi elementutako lista bat).
; LK baliokidetasun klaseak (lista bat).
; Eraitza: LK bera baina Pren elementuen klaseak kenduta.
(COND ((NULL LK) NIL)
      ((OR (MEMBER (FIRST P) (FIRST LK))
            (MEMBER (SECOND P) (FIRST LK)))
       (BESTEKLASEAK P (REST LK)))
      (T (CONS (FIRST LK)
                (BESTEKLASEAK P (REST LK)))))
```

6.18. Ariketa

```
(DEFUN ZKH (X Y)
  (LET ( (TXIKIENA (COND ((>= X Y) Y)
                        (T X)))
        (HANDIENA (COND ((>= X Y) X)
                        (T Y))))
    (COND ((ZEROP (REM HANDIENA TXIKIENA)) TXIKIENA)
          (T (ZKH TXIKIENA (- HANDIENA TXIKIENA)))))
```

6.19. Ariketa

ERO funtzioak argumentu bezala bi lista hartzen ditu, eta emaitza bezala lista berri bat itzultzen du. Lista honek, aurreko bi listen elementuak biltzen ditu beraien ordena mantenduz. Orden hori adibide hauen bidez ikusten da:

```
(ERO '(A) '(B))    0    (A B)
(ERO '(A B C) '(1 2 3)) 0    (A 1 B 2 C 3)
(ERO '(A B C D) '(1 2)) 0    (A 1 B 2 C D)
(ERO '(A B C) '(1 2 3 4 5)) 0    (A 1 B 2 C 3 4 5)
(ERO NIL NIL)    0    (A B)
(ERO NIL '(A B C))    0    (A B C)
(ERO '(A B C) NIL)    0    (A B C)
```

ZORO funtzioak lista bat hartzen du eta emaitza bezala beste bat itzultzen du. Lista berri honek azpilistak izango ditu elementu bezala, azpilista hauek emandako listaren hondarrak izango direlarik. Azpilista bakoitza N alditan agertuko da, N azpilistaren elementu kopurua izango delarik. Adibidez:

```
(ZORO '(A))    0    ((A))
(ZORO '(A B))    0    ((A B) (A B) (B))
(ZORO '(A B C))    0    ((A B C) (A B C) (A B C)
                        (B C) (B C)
                        (C) )
(ZORO NIL)    0    NIL
(ZORO '(1 2 3 4))    0    ((1 2 3 4) (1 2 3 4) (1 2 3 4) (1 2 3 4)
                        (2 3 4) (2 3 4) (2 3 4)
                        (3 4) (3 4)
                        (4) )
```

6.20. Ariketa

```
(DEFUN EBALUAPOL (N POL)
  (COND ((NULL POL) 0)
        (T (LET ( (KOE (FIRST (FIRST POL)))
                  (BERRETZ (SECOND (FIRST POL))))
            (+ (* (BERREDURA N BERRETZ)
```

KOEF)
(EBALUAPOL N (REST POL))))))

(DEFUN BERREDURA (ZBKI BERRETZ)
(COND ((ZEROP BERRETZ) 1)
(T (* (BERREDURA ZBKI (1- BERRETZ))
ZBKI))))

6.21. Ariketa

(DEFUN EXPON (X K)
(EXPON-LAG X K 0 0))

(DEFUN EXPON-LAG (X K INDIZEA BALIOA)
; INDIZEA: batutako azken osagaiaren indizea.
; BALIOA: metatutako balioa.
(COND (((< (SERIEKO-GAIA X INDIZEA) K) BALIOA)
(T (EXPON-LAG X
K
(1+ INDIZEA)
(+ (SERIEKO-GAIA X INDIZEA)
BALIOA))))))

(DEFUN SERIEKO-GAIA (X N)
(COND ((ZEROP N) 1)
(T (* (/ X N) (SERIEKO-GAIA X (1- N))))))

Ondorengo bertsio hau eraginkorragoa dugu:

(DEFUN EXPN (X K)
(EXPN-LAG X K 1 0 1))

(DEFUN EXPN-LAG (X K BALIOA INDIZEA OSAGAIA)
; BALIOA: batura partziala.
; INDIZEA: batutako azken osagaiaren indizea.
; OSAGAIA: batutako azken osagaia.
(LET ((OSAG-BERRIA (/ (* OSAGAIA X) (1+ INDIZEA))))

```

(COND ((< OSAG-BERRIA K) BALIOA)
      (T (EXPN-LAG X
                  K
                  (+ BALIOA OSAG-BERRIA)
                  (1+ INDIZEA)
                  OSAG-BERRIA))))))

```

6.22. Ariketa

```

(DEFUN SAKABANATU (ZUH)
  (SAKABANATU_LAG NIL ZUH))
; Sakabanatu zuhaitza erroaren posizioa NIL dela suposatuz.

(DEFUN SAKABANATU_LAG (ERROAREN_POSIZIOA ZUH)
  ; zuhaitz bat sakabanatzen du, baina erroa ERROAREN_POSIZIOAn
  ; dagoela suposatzen da.
  ; Beraz, adabegien posizio guztiek erroaren posizio horretaz hasiko dira.
  (CONS (LIST (FIRST ZUH) ERROAREN_POSIZIOA )
        (SAKABANATU_UMEAK ERROAREN_POSIZIOA
                          1
                          (REST ZUH)))))

(DEFUN SAKABANATU_UMEAK ( ERRO_POS
                          LEHEN_UME_ZBKIA
                          UME_LISTA)
  (COND ((NULL UME_LISTA) NIL)
        (T (APPEND (SAKABANATU_LAG
                      (APPEND ERRO_POS
                              (LIST LEHEN_UME_ZBKIA))
                      (FIRST UME_LISTA))
                    (SAKABANATU_UMEAK
                      ERRO_POS
                      (1+ LEHEN_UME_ZBKIA )
                      (REST UME_LISTA)))))))

```

6.23. Ariketa

- a) *xzuhaitz* funtzioak aztertzen du ea *zuh* zuhaitzean *adabegi1* eta *adabegi2* adar berean dauden, eta hala bada adabegi biotako bat itzultzen du, errorik gertuena dagoena hain zuzen ere; bestela NIL itzultzen du.

b)

```
(SETF ZUH1 '(A (B (C) (D)) (E (F)) (G (H (I (J))))))
(XZUHAITZ 'A 'E ZUH1) 0 A
(XZUHAITZ 'E 'C ZUH1) 0 NIL
(XZUHAITZ 'F 'J ZUH1) 0 NIL
(XZUHAITZ 'G 'J ZUH1) 0 G
```

Zazpigarren kapitulua

7.1. Ariketa

```
(DEFUN DOTIMES-FAKTORIALA (N)
  (LET ((EMAITZA 1))
    (DOTIMES (KONTA N EMAITZA)
      (SETF EMAITZA (* (+ 1 KONTA) EMAITZA)))))
```

```
(DEFUN DO-FAKTORIALA (N)
  (DO ((EMAITZA 1 (* N EMAITZA))
      (N N (- N 1)))
    ((ZEROP N) EMAITZA)))
```

```
(DEFUN LOOP-FAKTORIALA (N)
  (LET ((EMAITZA 1))
    (LOOP
      (IF (ZEROP N) (RETURN EMAITZA))
      (SETF EMAITZA (* EMAITZA N))
      (SETF N (- N 1)))))
```

7.2. Ariketa

```
(DEFUN DOLIST-REVERSE (LISTA)
  (LET ((EMAITZA NIL))
    (DOLIST (L LISTA EMAITZA)
      (SETF EMAITZA (CONS L EMAITZA)))))
```

7.3. Ariketa

```
(DEFUN DO-MEMBER (ELEMENTUA LISTA)
  (DO ((L LISTA (REST L)))
    ((OR (ENDP L) (EQL ELEMENTUA (FIRST L))) L)))
```

Zortzigarren kapitulua

8.1. Ariketa

```
(DEFUN ATKONTA (S)
  (COND ((NULL S) 0)
        ((ATOM S) 1)
        (T (APPLY #' + (MAPCAR #'ATKONTA S)))))
```

8.2. Ariketa

```
(DEFUN SAKONERA (S)
  (COND ((NULL S) 1)
        ((ATOM S) 0)
        (T (1+ (APPLY #'MAX (MAPCAR #'SAKONERA S)))))
```

8.3. Ariketa

```
(DEFUN BATULISTAK (L)
  (MAPCAR #'(LAMBDA (X) (APPLY ' + X)) L))
```

8.4. Ariketa

```
(DEFUN KONTA (E S)
  (COND ((NULL S) 0)
        ((ATOM S) (COND ((EQUAL S E) 1)
                          (T 0)))
        (T (APPLY #' + (MAPCAR #'(LAMBDA (X) (KONTA E X))
                          S)))))
```

8.5. Ariketa

Lehenengo bertsio bat:

```
(DEFUN FMULTIPLE (LF LARG)
  (MAPCAR #'APPLY LF (MAPCAR #'(LAMBDA (X)
    (MAPCAR #'EVAL X))
    LARG)))
```

Bigarren bertsio bat:

```
(DEFUN FMULTIPLE1 (LF LARG)
  (COND ((OR (NULL LF) (NULL LARG)) NIL)
    (T (CONS (APPLY (FIRST LF)
                     (MAPCAR #'EVAL (FIRST LARG)))
              (FMULTIPLE1 (REST LF) (REST LARG))))))
```

8.6. Ariketa

a)

L lista barruan A agertzen bada, hau aurkitzen den sakonera maila maximoa gehi N itzultzen du. A listan agertzen ez bada, 0 itzultzen du.

b)

```
(XXX 1 '(1 2 3))      0      1
(XXX 'A '(1 2 (A B) (X (A B)))) 0      3
(XXX 'A '(1 2 (A B) (X (A B))) 2) 0      4
(XXX 'A '(1 2 (X Y Z) (M (N O)))) 0      0
```

8.7. Ariketa

a)

XZUHAITZ funtzioaren emaitza honela defini daiteke:

- EL1 eta EL2 sinbolo biak ZUH zuhaitzaren adabegiak ez badira, NIL.
- Bestela, zuhaitzaren adabegi bat, hain zuzen, EL1 eta EL2 adabegi bien arbaso direnen artean errolik urrutien dagoena

b)

```
(XZUHAITZ 'A11 'A0 '(A0 (A11) (A12) (A13))) 0 (A0)
(XZUHAITZ 'A11 'A12 '(A0 (A11) (A12))) 0 (A0)
(XZUHAITZ 'B 'J Z) 0 (A)
(XZUHAITZ 'I 'L Z) 0 (H)
(XZUHAITZ 'B 'N Z) 0 (B)
```

c)

```
(DEFUN ADABEGI (ELEM ZUH)
  (COND ((ZUHAITZ_HUTS ZUH) NIL)
    ((EQL ELEM (ERROA ZUH)) T)
    (T (IF (MEMBER T (MAPCAR #'(LAMBDA (X)
```

(ADABEGI ELEM X))
(UMEAK ZUH))

T
NIL))))

(DEFUN ZUHAITZ_HUTS (ZUH)
(NULL ZUH))

(DEFUN ERROA (ZUH)
; zuh ez da hutsa
(FIRST ZUH))

(DEFUN UMEAK (ZUH)
; zuh ez da hutsa
(REST ZUH))

8.8. Ariketa

a)
XZUHAITZ funtzioak zuhaitzaren hostoak itzultzen ditu.

b)
(XZUHAITZ '(A0 (A11) (A12) (A13))) 0 (A11 A12 A13)
(XZUHAITZ '(A (B (C) (D)) (E (F)) (G (H (I (J)))))) 0 (C D F J)

8.9. Ariketa

a)
Zuhaitzaren N sakonera mailan dauden adabegiak itzultzen ditu.

b)
(XZUHAITZ 1 '(A0 (A11) (A12) (A13))) 0 (A11 A12 A13)
(XZUHAITZ 2 '(A0 (A11) (A12) (A13))) 0 NIL
(XZUHAITZ 2 '(A (B (C) (D)) (E (F)) (G (H (I (J)))))) 0 (C D F H)
(XZUHAITZ 3 '(A (B (C) (D)) (E (F)) (G (H (I (J)))))) 0 (I)

c)
(DEFUN AURREORDENA (ZUH)
(COND ((NULL ZUH) NIL)
(T (CONS (FIRST ZUH)
(APPLY #'APPEND
(MAPCAR #'AURREORDENA
(REST ZUH))))))

Bederatzigarren kapitulua

9.1. Ariketa

```
(DEFUN AITITA ( X )  
  (COND ((GET X 'AITA) (GET (GET X 'AITA) 'AITA))))
```

9.2. Ariketa

```
(DEFUN ADAN ( X )  
  (COND ((GET X 'AITA) (ADAN (GET X 'AITA )))  
    (T X )))
```

9.3. Ariketa

```
(DEFUN ARBASOAK ( X )  
  (COND ((NOT X ) NIL )  
    (T ( CONS X (APPEND (ARBASOAK (GET X 'AITA ))  
                        (ARBASOAK (GET X 'AMA ))))))))
```

9.4. Ariketa

Lehenengo ebazpena:

```
(DEFUN SORTU_AURREKOAK ( L )  
  (SORTU_AURREKOAK_LAG NIL L ))  
  
(DEFUN SORTU_AURREKOAK_LAG( L1 L2 )  
  (COND ((NULL L2 ) NIL )  
    (T (SETF (GET (FIRST L2) 'AURREKOAK) L1 )  
      (SORTU_AURREKOAK_LAG  
        (APPEND L1 (LIST (FIRST L2 )))  
        (REST L2 ))))))
```

Bigarren ebazpena:

```
(DEFUN SORTU_AURREKOAK ( L )  
  (MAPCAR #'(LAMBDA (X) (SETF (GET X 'AURREKOAK)  
                              (AURREKOEN_LISTA X L)))  
    L))  
  
(DEFUN AURREKOEN_LISTA ( X L )  
  (COND ((EQUAL X ( FIRST L )) NIL )  
    (T ( REST ( MEMBER X ( REVERSE L ))))))
```

Azkeneko funtzio honen beste bertsio bat:

```
(DEFUN AURREKOEN_LISTA ( X L )
  (COND ((EQUAL X (FIRST L)) NIL)
        (T (CONS (FIRST L) (AURREKOEN_LISTA X (REST L))))))
```

9.5. Ariketa

```
(DEFUN LOTU (A B)
  (LET ((A-BIZILAGUNAK (GET A 'BIZILAGUNAK))
        (B-BIZILAGUNAK (GET B 'BIZILAGUNAK))
        (EMAITZA NIL))
    (UNLESS (MEMBER B A-BIZILAGUNAK)
      (SETF (GET A 'BIZILAGUNAK)
            (CONS B A-BIZILAGUNAK))
      (SETF EMAITZA T))
    (UNLESS (MEMBER A B-BIZILAGUNAK)
      (SETF (GET B 'BIZILAGUNAK)
            (CONS A B-BIZILAGUNAK))
      (SETF EMAITZA T))
    EMAITZA))
```

9.6. Ariketa

```
(DEFUN BATAZBESTEKO (MATRIZEA)
  (LET ((EMAITZA 0))
    (DOTIMES (M 5 (/ EMAITZA 35))
      (DOTIMES (N 7)
        (SETF EMAITZA (+ (AREF MATRIZEA M N)
                          EMAITZA))))))
```

9.7. Ariketa

Hitzak honelako listen bidez adieraziko ditugu:

```
( (KATEGORIA-SINTAKTIKOA <kat-balioa>)
  (ESANGURA <esangura-balioa>)
  (SINONIMOAK <sinonimo-lista>)
  (HIZKUNTZA <hizkuntza-balioa>))
```

Era horretako objektuak eraikitzeke listen funtzio eraikitzaileak erabiliko ditugu.

Atzitze funtzioak:

```
(DEFUN EMAN-KATEGORIA (HITZA)
  (LET ((PAREA (ASSOC 'KATEGORIA-SINTAKTIKOA HITZA)))
    (IF (NULL PAREA) NIL (SECOND PAREA))))
```

```
(DEFUN EMAN-ESANGURA (HITZA)
  (LET ((PAREA (ASSOC 'ESANGURA HITZA)))
    (IF (NULL PAREA) NIL (SECOND PAREA))))
```

```
(DEFUN EMAN-SINONIMOAK (HITZA)
  (LET ((PAREA (ASSOC 'SINONIMOAK HITZA)))
    (IF (NULL PAREA) NIL (SECOND PAREA))))
```

```
(DEFUN EMAN-HIZKUNTZA (HITZA)
  (LET ((PAREA (ASSOC 'HIZKUNTZA HITZA)))
    (IF (NULL PAREA) NIL (SECOND PAREA))))
```

Hitzak egituren bidez adierazteko, nahikoa da datu-mota definitzea (DEFSTRUCT erabiliz), eta horrekin eremuen balioak atzitzeko funtzioak automatikoki edukiko ditugu.

```
(DEFSTRUCT HITZA-DATU-MOTA
  (KATEGORIA-SINTAKTIKOA NIL)
  (SINONIMOAK NIL)
  (ESANGURA NIL)
  (HIZKUNTZA 'EUSKARA))
```

9.8. Ariketa

```
(DEFUN MUNDU (X)
  (AZTERTU NIL (LIST X)))
(DEFUN AZTERTU (LBISITATUAK LBISITATZEKOAK)
  (COND ((NULL LBISITATZEKOAK) LBISITATUAK)
        ((MEMBER (FIRST LBISITATZEKOAK) LBISITATUAK)
         (AZTERTU LBISITATUAK (REST LBISITATZEKOAK)))
        (T (AZTERTU (AZTERTU (CONS (FIRST LBISITATZEKOAK)
```

```

        LBISITATUAK)
      (GET (FIRST LBISITATZEKOAK)
        'HAUZOKOAK))
    (REST LBISITATZEKOAK))))))

```

9.9. Ariketa

```

(DEFUN ITZULBIDERIK (K)
; K korapilune bat hartuta, NIL itzuliko du grafoko arkuak jarraituz
; K korapilunera itzultzeko biderik ez badago, eta bestela, bidea itzuliko du
; (korapilune edo arku batetik bi aldiz pasa gabe).
  (ZIRKUITOL (ZABALDU (LIST K))))

```

```

(DEFUN ZABALDU (L)
; Bide bat hartuta (pasatutako korapiluneen lista bezala, non abiatzeko
; korapilunea listako azkena den), bisitatutako azken korapilunearen hauzoko
; bakoitzarekin bide berri bat eratuko du, bide berri guztiekin lista bat itzuliz.
  (MAPCAR #'(LAMBDA (X) (CONS X L))
    (GET (FIRST L) 'HAUZOKOAK)))

```

```

(DEFUN ZIRKUITOL (BIDE-LISTA)
; Bide-lista bat hartuta, bideetako bat edo bere luzapenetako bat zirkuito bat
; bada, emaitza gisa zirkuitoa itzuliko du; bestela bide guztien artean
; zirkuitorik ez badago NIL itzuliko du.
  (COND ((NULL BIDE-LISTA) NIL)
    ((ZIRKUITO (FIRST BIDE-LISTA))
      (ZIRKUITO (FIRST BIDE-LISTA)))
    (T (ZIRKUITOL (REST BIDE-LISTA)))))

```

```

(DEFUN ZIRKUITO (BIDE)
; Bide bat hartuta, berau edo bere luzapenetako bat zirkuitoa bada zirkuitoa
; itzuliko du; bestela, bide horretatik zirkuitorik ezin bada aurkitu NIL itzuliko
; du.
  (COND ((EQL (FIRST BIDE) (FIRST (LAST BIDE))))
    (COND ((> 4 (LENGTH BIDE)) NIL)
      (T BIDE)))
  ((MEMBER (FIRST BIDE) (REST BIDE)) NIL)
  (T (ZIRKUITOL (ZABALDU BIDE)))))

```


9.10. Ariketa

a)

```
(DEFUN ONDOKOEN_ORDENA (X)
  (COND ((NULL X) NIL)
        ((ATOM X)
         (COND ((EQL BAI (GET X 'BIZIRIK))
                (CONS X
                      (ONDOKOEN_ORDENA (GET X 'SEME-ALABAK))))
              (CONS X
                    (ONDOKOEN_ORDENA (GET X 'SEME-ALABAK))))
        (T (ONDOKOEN_ORDENA (GET X 'SEME-ALABAK))))
```

```
(ONDOKOEN_ORDENA (GET X 'SEME-ALABAK))))
(T (ONDOKOEN_ORDENA (GET X 'SEME-ALABAK))))
(T (APPEND (ONDOKOEN_ORDENA (FIRST X))
            (ONDOKOEN_ORDENA (REST X))))))
```

b)

```
(DEFUN ORDMATXISTA (X)
  (COND ((NULL X) NIL)
        ((ATOM X)
         (LEFT ((S-A (GET X 'SEME-ALABAK))
                (COND ((EQL BAI (GET X 'BIZIRIK))
                      (CONS X
                            (ORDMATXISTA (GIZONAK_AURRERA S-A))))
                    (CONS X
                          (ORDMATXISTA (GIZONAK_AURRERA S-A))))
              (ORDMATXISTA (GIZONAK_AURRERA S-A))))
        (T (APPEND (ORDMATXISTA (FIRST X))
                    (ORDMATXISTA (REST X))))))
```

```
(DEFUN GIZONAK_AURRERA (L)
  (COND ((NULL L) NIL)
        (T (TXERTATU (FIRST L) (GIZONAK_AURRERA (REST L))))))

(DEFUN TXERTATU (X L)
  (COND ((NULL L) (LIST X))
        ((EQL GIZONEZKO (GET X 'SEXUA)) (CONS X L))
        ((EQL GIZONEZKO (GET (FIRST L) 'SEXUA))
         (T (COND ((EQL GIZONEZKO (GET (FIRST L) 'SEXUA))
                  (CONS (FIRST L) (TXERTATU X (REST L))))
              (T (CONS X L))))))
```

ORDMATXISTA funtzia beste modu batera:

```

(DEFUN ORDMATXISTA (X)
  (COND ((NULL X) NIL)
        ((ATOM X)
         (COND ((EQL 'BAI (GET X 'BIZIRIK))
                  (CONS X
                        (ORDMATXISTA (GET X 'SEME-ALABAK))))
              (T (ORDMATXISTA (GET X 'SEME-ALABAK))))))
  (T (LET ((XORD (GIZONAK_AURRERA X)))
      (APPEND (ORDMATXISTA (FIRST XORD))
              (ORDMATXISTA (REST XORD))))))

```

Hamargarren kapitulua

10.1. Ariketa

```

(DEFUN BILATU-DENAK (KATEA DATU-FITX EMAITZA-FITX)
  (WITH-OPEN-FILE (SARRERA DATU-FITX :DIRECTION :INPUT)
    (WITH-OPEN-FILE (IRTEERA EMAITZA-FITX
                      :DIRECTION :OUTPUT)
      (DO ( (LERROA (READ-LINE SARRERA NIL)
                    (READ-LINE SARRERA NIL))
          (EMAITZA NIL) )
        ( (NOT LERROA) EMAITZA)
        (WHEN (SEARCH KATEA LERROA
                      :TEST #'CHAR-EQUAL)
          (FORMAT IRTEERA "~%~A" LERROA)
          (SETF EMAITZA T))))))

```

Hamaikagarren kapitulua

11.1. Ariketa

```

(DEFMACRO NIRE-WHEN (BALDINTZA &REST GORPUTZA)
  `(COND (,BALDINTZA ,@GORPUTZA)))

(DEFMACRO NIRE-UNLESS (BALDINTZA &REST GORPUTZA)
  `(COND ((NOT ,BALDINTZA) ,@GORPUTZA)))

```

11.2. Ariketa

```
(DEFMACRO HONDARRA (LISTA)
  `(REST ,LISTA))
```

```
(DEFMACRO ERAIKI (ELEMENTUA LISTA)
  `(CONS ,ELEMENTUA ,LISTA))
```

11.3. Ariketa

```
(DEFMACRO BALDIN (BALDINTZA ORDUAN
                  &OPTIONAL BESTELA)
  `(COND (,BALDINTZA ,ORDUAN)
        (T ,BESTELA)))
```

11.4. Ariketa

LAMBDA eta LET funtzioen arteko baliokidetasuna erabiliz:

```
(DEFMACRO IZANIK (ALDAGAIK &REST FORMAK)
  `((LAMBDA ,(MAPCAR #'FIRST ALDAGAIK)
    ,@FORMAK)
    ,(MAPCAR #'SECOND ALDAGAIK)))
```

11.5. Ariketa

```
(DEFMACRO NIRE-DOLIST (KONTROL-LISTA &REST GORPUTZA)
  `(DO* ( (LL ,(SECOND KONTROL-LISTA) (REST LL))
    ,(FIRST KONTROL-LISTA) (FIRST LL) (FIRST LL)))
    ( (NULL LL) ,(THIRD KONTROL-LISTA))
    ,@GORPUTZA))
```

```
(DEFMACRO NIRE-DOTIMES (KON-LISTA &REST GORPUTZA)
  `(DO ( ,(FIRST KON-LISTA) 0 (1+ ,(FIRST KON-LISTA)))
    ( (EQUAL ,(FIRST KON-LISTA) ,(SECOND KON-LISTA))
      ,(THIRD KON-LISTA))
    ,@GORPUTZA))
```

11.6. Ariketa

```
(DEFMACRO PUTPROP (SINBOLOA PROPIETATEA BALIOA)
  `(SETF (GET ,SINBOLOA ,PROPIETATEA) ,BALIOA))
```

11.7. Ariketa

```
(DEFMACRO APLIKA (FUNTZIOA ARGUMENTUAK)
  `(,FUNTZIOA ,@ARGUMENTUAK))
```

11.8. Ariketa

```
(DEFMACRO FUNDEITU (FUNTZIOA &REST ARGUMENTUAK)
  `(,FUNTZIOA ,@ARGUMENTUAK))
```

11.9. Ariketa

a)

```
(CASE X
  (1 (+ Y Z))
  (2 (- Y Z))
  (3 (* Y Z))
  (4 (/ Y Z)))
```

X: 3 Y: 3 Z: 8

Hedaketa:

```
(COND ((EQL X (QUOTE 1)) (+ Y Z))
      ((EQL X (QUOTE 2)) (- Y Z))
      ((EQL X (QUOTE 3)) (* Y Z))
      ((EQL X (QUOTE 4)) (/ Y Z)))
```

Forma hori ebaluatzen badugu:

```
(COND ((EQL X (QUOTE 1)) (+ Y Z))
      ((EQL X (QUOTE 2)) (- Y Z))
      ((EQL X (QUOTE 3)) (* Y Z))
      ((EQL X (QUOTE 4)) (/ Y Z)))    0    24
```

b)

```
(CASE X
  (Y (+ Y Z))
  (Z (- Y Z)))
```

X: 3 Y: 3 Z: 8

Hedaketa:

```
(COND ((EQL X (QUOTE Y)) (+ Y Z))
      ((EQL X (QUOTE Z)) (- Y Z)))
```

Forma hori ebaluatzen badugu:

```
(COND ((EQL X (QUOTE Y)) (+ Y Z))
      ((EQL X (QUOTE Z)) (- Y Z)))
0    NIL
```

Zerbait egiteko Xen balioa 'Y edo 'Z izan behar du.

11.10. Ariketa

```
(DEFMACRO DEFINE (BURUKOA &REST GORPUTZA)
  `(DEFUN ,(FIRST BURUKOA) ,(SECOND BURUKOA)
    ,@GORPUTZA))
```

11.11. Ariketa

a) N: 5 E1: (* X 4) E2: Y

Hedaketa:

(1+ (* X 4))

Ebaluazioa:

(1+ (* X 4)) 0 17

b) N: 6 E1: (* X 4) E2: Y

Hedaketa:

(1- Y)

Ebaluazioa:

(1- Y) 0 ERRORA: A EZ DA ARGUMENTU
EGOKIA 1- FUNTZIOARENTZAKO.

c) N: X E1: (* X 4) E2: Y

Hedaketa:

0 ERRORA: X EZ DA ARGUMENTU EGOKIA ODDP
FUNTZIOARENTZAKO.

d) N: (1+ X) E1: (* X 4) E2: Y

Hedaketa:

0 ERRORA: (1+ X) EZ DA ARGUMENTU EGOKIA ODDP
FUNTZIOARENTZAKO.

11.12. Ariketa

a) N: 5 E1: (* X 4) E2: Y

Hedaketa:

$$(1+ (* X 4))$$

Ebaluazioa:

$$(1+ (* X 4)) \quad 0 \quad 17$$

b) N: 6 E1: (* X 4) E2: Y

Hedaketa:

$$(1- Y)$$

Ebaluazioa:

$$(1- Y) \quad 0 \quad \text{ERROREA: A EZ DA ARGUMENTU EGOKIA}$$

1- FUNTZIOARENTZAKO.

c) N: X E1: (* X 4) E2: Y

Hedaketa:

$$(1- Y)$$

Ebaluazioa:

$$(1- Y) \quad 0 \quad \text{ERROREA: A EZ DA ARGUMENTU EGOKIA}$$

1- FUNTZIOARENTZAKO.

d) N: (1+ X) E1: (* X 4) E2: Y

Hedaketa:

$$(1+ (* X 4))$$

Ebaluazioa:

$$(1+ (* X 4)) \quad 0 \quad 17$$

BIBLIOGRAFIA

Bat ere dudarik gabe esan ahal da LISP ikasteko libururik onena [Winston, 89] dela, bere hirugarren argitaralpenean aurrekoari funtsezko hobekuntzak gehitzen zaikiolarik. Azalpen eta ariketak tartekatzen ditu eta kontzeptuen azalpenak erizpide didaktikoen arauera ordenatuta agertzen dira. Ondorioz, testu hau bereziki egokia da nork bere kabuz lengoaia ikasteko.

[Steele, 84] ere oso erabilia da, baina honek ez du helburu pedagogikorik. LISP lengoaia ondo ezagutzen duen programatzailearentzat ezinbesteko kontsulta-liburua edo erreferentzi liburua da lengoaiari buruzko edozein zalantza argitzeko.

Helburu mistoa dute [Tratar, 86], [Wilensky, 86] eta [Hennessey, 89] erreferentziek, hirurek COMMON LISP erabiltzen dutelarik.

LISPeke beste deskribapen orokorrak baina COMMON LISP kontutan hartzen ez dutenak: [Touretzky, 86]-k sarrera oso erraz bat eskaintzen die informatikari ez direnei (ingelesaz, frantsesaz eta espaineraz argitaratua); [Farreny, 86]-k eta [Wertz, 86]-k Frantzia oso zabalduta dauden VLISP eta LELISP dialektoak erabiltzen dituzte (frantsesaz eta espaineraz argitaratuak); [Cortés, 87]-k Bartzelonan inplementatu duten UPCLISP erabiltzen du (espaineraz).

LISPeke sarrera eta garapenari buruz hitz egiterakoan beharrezko aipamenak dira [McCarthy, 62] eta [McCarthy, 78].

Azkenik, beste erreferentzi talde batean, LISPeke erabilpen aurreratuentzakoa da: [Winston, 84a], [Winston, 84b] eta [Keene, 89].

Cortés, U. y Sierra, C.

LISP

Marcombo, Boixareu editores, 1987.

Farreny H.

Introducción a LISP El lenguaje básico para la inteligencia artificial

Masson 1986

Hennessey, W. H.

COMMON LISP

McGraw Hill 1989

Keene, S.E.

Object-Oriented Programming in COMMON LISP

Addison-Wesley 1989

McCarthy, J. Abrahams, P. Edwards, D. & Levin, M.

LISP 1.5 Programmer's Manual

M.I.T. Press 1962

McCarthy, J.

History of LISP

ACM SIGPLAN Notices, Vol. 13 No. 8 1978

Steele, Guy L. Jr.

COMMON LISP Reference Manual

Digital Press 1984

Touretzky, D. S.

LISP. Introducción al cálculo simbólico.

Ediciones Díaz de Santos, 1986.

Wertz, H.

LISP. Introducción a la programación.

Masson, 1986.

Wilensky, R.

COMMON LISPcraft

W.W. Norton New York 1986

Winston, P.H..

Artificial Intelligence. 2nd edition

Addison-Wesley, 1984

Winston, P.H.. & Prendergast, K. A. (editoreak)

The AI Business: Commercial Applications of Artificial Intelligence

M.I.T. Press 1984

Winston P. , Horn B.

LISP. 3rd edition

Addison-Wesley, 1989

KONTZEPTUEN AURKIBIDEA

- 41
&aux 72
&key 72
&optional 70
&rest 71
* 42
+ 41
, @ Karaktere-bikotea 184
/ 44
1+ 42
1- 42
= 51

A

ABS 41
adimen artifiziala 9
agindu-sekuentziaketa 105
ahotsaren analisia 10
aldagai 77; 103
aldagai 111
aldagai orokor 108
aldagai berezi 108
aldagai lexikal 103
AND 55
aplikatzaile 77
APPEND 35
APPLY 60
AREF 154
ARRAY-DIMENSION 155
arrayak 152
asignazio 103; 109
ASSOC 149
ATOM 48
atomo 13
atxekitako balio 14
aukerazko parametroak 70

B

BACQUOTE 184
bertako aldagai 103
bigizta 103
bikote-idazkera 17
bilketa progresibo 119
binding 14
BOUNDP 50
BOUNDP 111

BREAK 142
BUTLAST 37
buztaneko errekurtsibitate 88

C

CADDR 34
CADR 34
CAR 33
CASE 57
CDR 33
CHAR-EQUAL 47
CHAR= 47
COMMONLISP 8
COND 56
CONS 32
CONSP 49
COS 45
COUNT-IF 136

D

datu-abstrakzioa 123
debugging 140
DECLARE 109
DEFMACRO 182
DEFSTRUCT 156
DEFUN 67
DEFVAR 109
dei errekurtsibo 79
DESCRIBE 159
DIRECTION 173
DO 111
DOLIST 114
DOTIMES 113

E

ebaluazio 25; 29
ebaluazio nagia 4
egiturak 155
elkartze-listak 149
ELT 46
ENDP 49
EQ 51
EQL 51
EQUAL 50
eraginkortasun 89
errealak 13

erredukzio 89
 erreferentzi gardentasun 78
 erreferentzi gardentasuna 77
 errekurtsibitate 88
 errekurtsibitate bakuna 88
 errekurtsibitate bikoitza 88
 errore-araketa 140
 espezializazio-erlazioak 159
 espresio sinboliko 16
 etiketa 103; 115
 EVAL 6
 EVAL 59
 EVENP 54
 EXP 45
 EXPORT 130
 EXPT 45

F

FIND-IF 136
 FIRST 33
 fitxategi-bukaera 175
 FLOAT 43
 forma 25
 forma berezi 28
 FORMAT 169
 FUNCALL 60
 funtzio 73
 funtzio aurredefinitu 31
 Funtzio suntsitzaile 38
 funtzio-abstrakzio 121
 funtzio-aplikazio 1
 funtzio-definizio 1; 14
 funtzio-konposaketa 1

G

garbage collection 21
 GCD 42
 GET 150
 GO 116
 gorputz 67

H

harresi 105

I

IN-PACKAGE 130
 INCLUDE 160
 informazio sinboliko 8
 INITIAL-CONTENTS 153
 INITIAL-ELEMENT 153
 inprimatzeko izena 14
 INTERLISP 2; 8
 irazkin 69
 iterazio 111

K

karaktere bereziak 13
 karaktere-kate 15
 karektere-kate 13
 kasu nabari 80
 kasu orokor 80
 kode agintzaileak 170
 koma 184
 komatxo karakterea 184
 konputagailu bidezko Ikusmena 10
 konputagailuz Lagunduriko
 Irakaskuntza Inteligentea 10
 kontrol-egitura 78; 103
 korronteak 172

L

LABELS 106
 LAMBDA 69; 107; 135
 LAST 37
 LELISP 8
 lengoaia agintzaile 3
 lengoaia erazagutzaile 4
 lengoaia funtzional 3; 77
 Lengoaia Naturalaren
 Prozesamendua 10
 LENGTH 37; 46
 LET 90; 107
 LET* 93
 LISP 1.5 1; 2; 8
 LISP makina 2
 LIST 35
 lista 16
 lista-idazkera 17
 LISTP 50
 LOOP 115

M

MACLISP 2; 8
 MACSYMA 11
 MAKE- 157
 MAKE-ARRAY 153
 makro 181
 makroen hedaketa 181
 MAKUNBOUND 111
 MAPCAR 90; 134
 MAX 41
 McCarthy 1
 MEMBER 52
 memoriako adierazpidea 18
 MIN 42
 MINUSP 53

N

NCONC 39
NIL 8; 15; 16
NOT 54
NTH 36
NTHCDR 36
NULL 49
NUMBERP 50

O

objektu atomiko 13
ODDP 53
OR 55
osoak 13
OTHERWISE 58

P

paketeak 129
parametro erregularrak 70
parametro giltzadunak 72
parametro laguntzaileak 72
parametro-lista 67
parametro-mota 73
PLUSP 53
pname 14
PRINT 167
print name 14
problema-erreduzioa 121
PROG 115
programazio agintzaile 77
programazio funtzional 77
programazio Modular 67
programazio-eskemak 132
programazio-inguruneak 2
propietate-lista 152
PROVIDE 130
prozedura 73
puntu-idazkera 17

Q

QUOTE 28

R

READ 167
READ-CHAR 176
READ-LINE 176
REM 44
REMOVE-IF 135
REMOVE-IF-NOT 136
REMPROP 151
REQUIRE 130
REST 33
RETURN 112

REVERSE 38
REVERSE 46
ROUND 43
RPLACA 38
RPLACD 38

S

s-espresio 16
sarrera/irteerak 167
sasiadagai 90
SEARCH 47
SECOND 36
SET 110
SETF 40; 110; 154; 158
SETF 151
SETQ 110
SIN 45
sinbolo 13; 14
sistema adituak 10
SPECIAL 109
SQRT 45
STEP 140
STRING-EQUAL 46
STRING= 46
STRINGP 50
SYMBOLP 49

T

T 15
TAN 46
TENTH 36
THIRD 36
TIME 142
TRACE 140
TRUNCATE 43

U

UNLESS 57
UNTRACE 141
UPCLISP 8
USE-PACKAGE 131

W

WHEN 57
WITH-OPEN-FILE 171

X

XLISP 8

Z

zabor-bilketa 21
zenbakiak 13
ZEROP 53