Oinarrizko programazioa

Perl

K. Gojenola <jipgogak@si.ehu.es>
K. Sarasola <jipsagak@si.ehu.es>
A. Soroa <ccpsoeta@si.ehu.es>

2 kreditu

Aurkibidea

1 Sarrera

1.1 Motibazioa

Linguistika konputazionalaren arloak informazio linguistikoarekin egiten du lan. Izan ere, konputagailuak linguistikari eskaini dion ekarpen nagusia testu handiekin hainbat datu ateratzea baita, datu horiek analizatu, eta analisi horretatik hainbat ondorio ateratzeko aukera emanez. Hala ere, konputagailua *inozo azkarra* dela esaten da: kalkulu-kopuru handia oso azkar egiteko gai den bitartean, egin behar duen lana erabat espezifikatu beharko diogu, berak ulertzen duen lengoaia bakarrean, bera bakarrik ez baita gai ezer ere egiteko. Hortaz, konputagailuari zerbait egiteko eskatzen diogun bakoitzean, dela testu-editore batekin aritu, dela datu bilduma batetik zenbait estatistika atera, ataza hori beteko duen *programa* bat exekutatuko dugu. Programa bat, konputagailuak ekintza espezifikoren bat egiteko balio duen agindu-sekuentzia bat da, makina-lengoaian idatzia.

Linguistika konputazionalaren arloari itzuliz, gaur egun hainbat eta hainbat lan desberdin egiteko programa sorta izugarria aurki daiteke. Haatik, gerta daiteke guk nahi dugun lan espezifikoa egiteko programarik ez egotea. Horrelako kasuetan, gure arazo espezifikoa ebatziko duen programa guk geuk idatzi beharko dugu edo, aukeran, antzeko gauzak egiten duen programa eskura badugu, programa horretan aldaketa txikiak egin, gure helburua lortzearren.

Programak, *programazio-lengoaia* jakin batean idazten dira. Programazio-lengoaia hauek gizakiok erabiltzen dugun *lengoaia naturala* baino arras sinpleagoak dira eta, gainera, programazio-lengoaietan ez da onartzen inongo anbiguotasunik. Edonola ere, gaur egun ehundaka gora lengoaia daude, lengoaia bakoitzak helburu desberdinetarako balio duelarik.

Perl programazio-lengoaia 1990. hamarkadan jaio zenetik sekulako arrakasta izan du, besteak beste testuen gainean azterketak egiteko. Testuak prozesatzeko duen ahalmen ikaragarriari esker, oso erabilia da linguistika konputazionalean.

1.2 Helburuak

Lehenik eta behin, esan beharra dago ikastaro honen helburua ez dela edozein programa garatzen jakitea, gure helburuak xeheagoak baitira: soilik zenbait arazo espezifiko ebatziko dituzten programa laburrak idazten jakitea, edo beste norbaitek egindako programa txikiak ulertu eta aldatzeko gai izatea. Nolabait, gure asmoa jendeak programazioaren aurrean duen beldurra edo mesfidantza arintzea da.

Zehatzago, ikastaro honen helburuak honako hauek dira:

- Oinarrizko programazioaren elementuak ezagutzea:
  - Programazio-lengoaiak.

    Arazo jakin baten aurrean, azkeneko programa lortu arte bete beharreko urratsak ezagutzea: arazoaren espezifikazioa, algoritmoa, programa eta probak.
  - Programen kontzeptu orokorrak.

    Programatzerakoan erabiltzen diren oinarrizko kontzeptuen azalpena, hala nola objektuak, aldagaiak, aginduak, espresioak, kontrol-egiturak eta azpiprogramak.
- PERL programazio-lengoaia ezagutzea:
  - PERL lengoaiaren objektuak.
  - Espresio erregularrak.
  - Sarrera / Irteera.
  - Arazo sinpleei aurre egiteko, programa laburrak idazteko gai izan.

1.3 Metodologia

Modulu gehienetan bezala klaseak eta lan praktikoa konbinatzea proposatzen da. Dokumentazio honetan oinarrituta zenbait ariketa proposatuko dira eta bukaeran lan praktiko bat burutu beharko da bakarrik edo taldean (gehienez bi ikasle talde bakoitzeko). Ariketak edo lan praktikoa egitean sortzen diren zalantzak e-postaz kontsultatu daitezke irakaslearekin.

Ikastaro honetako ariketa guztien fitxategiak ikastaroan bertan pasako dira, horien gainean ikasleek lan egin dezaten. Lan gehienetan ariketa erdi eginak dauden programak osatzea izango da. Beste batzuetan, aldiz, ikasleak ariketa/programa osoa egin beharko du.

Ikastaroaren amaieran, ariketa horien guztien soluzioak emango dira. Ikasleak hor antzeko problemak ebazteko programategia izango du.

1.4 Ebaluazioa

Modulua gainditzeko asistentzia (lau saioetatik gutxienez hirura etorriz) eta proposatutako lan praktikoetako bat modu egokian aurkeztea (e-postaz bidaltzea gomendatzen da) eskatzen da. Klaseetara etortzerik ez duenak proposatutako ariketak ebatzita bidali beharko ditu modulua gainditu ahal izateko. Klaseetara datozenek ariketak egitean aurkitu dituzten arazoak e-postaz kontsulta ditzakete.

2 Programazioa

2.1 Konputagailua eta programak

Konputagailua luze, zail eta errepikakorrak diren agindu sekuentziak burutzen dituen makina da. Agindu horiek eragiketa logikoak edo kalkulu matematikoak izaten dira orokorrean. Ordenadoreek erabiltzaileak ematen dituen datuak (informazioa) hartu eta lortu nahi diren emaitzak (beste informazio bat) kalkulatzen dituzte, agindu-sekuentzia bat egikarituz. Emaitza horiek lortzeko asmatu behar den agindu-sekuentziari *programa* deitzen zaio.

Nahiz eta konputagailuak berez adimentsuak ez izan, ematen zaien agindu-sekuentzia exekutatzen badakite, eta datuak irakurtzen eta idazten ere bai. Ez dakite, baina, problema baterako programa asmatzen. Datuekin zer nolako aginduak egikaritu behar diren zehatz-mehatz eta zuzen definitzen ez baditugu (hau da, programa egokia ematen ez bazaio), konputagailuak ez du ezer egingo, edo emaitza okerrak eskainiko ditu, programa hori zuzena ez delako.

Programak idazteko, programazio-lengoaiez baliatzen gara. Programatzeko lengoaia bakoitzak bere arauak ditu, batetik datuak memoria barruan adierazteko, eta bestetik exekutatu ahal diren aginduak definitzeko. Beraz, programa bat idazteko, programatzeko lengoaia baten arauak bete behar dira.

2.2 Programazio-lengoaiak. Behe-mailakoak eta goi-mailakoak

Konputagailuarekin komunikatzeko (burutu behar diren lanak deskribatzeko) erabiltzen den lengoaia ez da gizakumeon artean erabiltzen duguna. Gizakumeon arteko lengoaiari *giza-lengoaia* edo *lengoaia naturala* esaten diogu eta konputagailuak ulertzen duenari *makina-lengoai*a.

Nahiz eta programazio-lengoaiak asko izan, zehazki hitz eginez konputagailu batek lengoaia bakar bat ulertzen du: bere ordenadore-ereduari dagokion makina-lengoaia. Beste edozein lengoaiaz adierazitako programa bat ulertu eta egikaritu ahal izateko, ordenadoreak programa horren itzulpena behar du, bere makina-lengoaian idatzita

egongo den itzulpena, hain zuzen ere. Baina makina-lengoaiak ez dira erabiltzeko oso tresna atseginak ondoan aipatzen diren arrazoiengatik:

1. Datu eta eragiketa guztiak kode bitarrez adierazten dira, programak zirriborro ulergaitzak bihurtzen direlarik. Zeroak eta batak besterik ez dira agertzen.
2. Datu bat erabiltzeko bere adierazpen bitarra ezagutu behar da, edota datu horixe daukan memoria-helbidearen zenbakia. Erosoago gertatzen da izen baten bidezko adierazpena, eta ezagunagoa zaigun kode hamartarra erabiltzea.
3. Marka eta eredu desberdinetako konputagailu bakoitzak bere makina-lengoaia desberdina dauka. Hau dela eta, ordenadore baten programek ez dute beste ordenadoreetarako balio.
4. Aginduak oinarrizkoak direnez, horietako asko idatzi behar dira problema bat ebazteko.

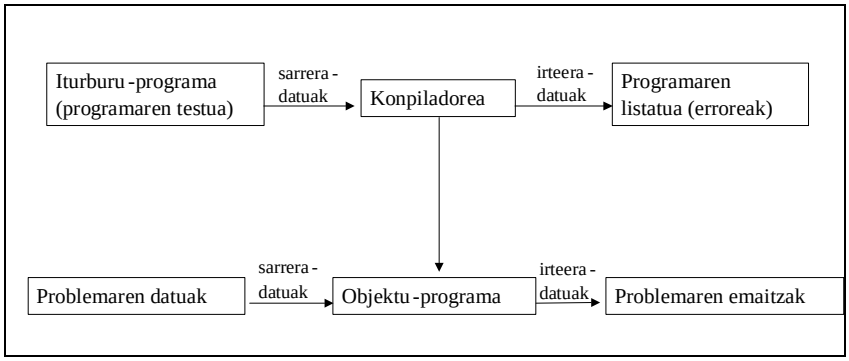
Azaldutako lau oztopo horiek aurkituko ditugu makina-lengoaiaz baliatzen bagara, erabilpena aspergarri eta neketsu bihurtzen delarik. Oztopo gehienak gainditzeko, goi-mailako programazio-lengoaiak asmatu dira. Goi-mailako programazio-lengoaiak giza-lengoaiatik urrun egon arren, makina-lengoaia baino askoz aiseago erabiltzen dira.

Behe-mailako programazio-lengoaiak makina-lengoaiak eta mihiztadura-lengoaiak dira. Mihiztadura-lengoaien erabilpena makina-lengoaiena baino apurtxo bat errazagoa da, kode bitarra ez baita erabiltzen, ez aginduetarako ezta datuetarako ere.

Goi-mailako lengoaia asko dago gaur egunean, bakoitzari bere erabilpen-eremu egokiena dagokiolarik. Zenbait lengoaiak adierazpide algebraiko eta zenbakizkoetarako erraztasuna eskaintzen du eta beraz, kalkulu zientifikoetarako erabiltzen dira; beste zenbaitek hitzak eta zenbaki sinpleak erabiltzen ditu, baina datu-kopuru handietarako balio dute eta ondorioz, enpresa-arazoetarako erabiltzen dira.

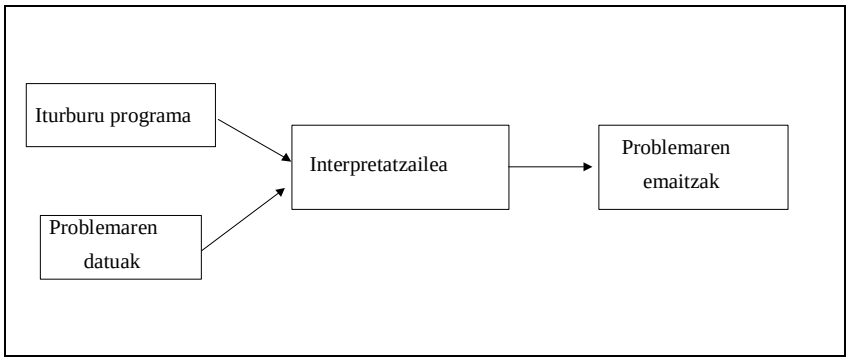
Goi-mailako lengoaiatik makina-lengoaiara itzultzeko programak bi eratakoak dira: konpiladorea eta interpretatzailea. Konpiladoreek programaren agindu guztiak batera itzultzen dituzte, eta programa osoaren itzulpena lortzen dute. Programa

konpiladorea behin bakarrik exekutatzen da, eta bere emaitza lortzen duenean (makina-lengoaiazko programa memorian gorde duenean) konpiladoreak bere lana amaitu du. Programazio-lengoaiaz adierazitako programari *iturburu-programa* izena ematen zaio, eta bere makina-lengoaiazko itzulpenari, berriz, *objektu-programa*. Objektu-programa hau zuzenean exekuta daiteke, makinak ulertzen duen lengoaian baitago.



1. Irudia: Konpiladorearen bidezko exekuzioa

Interpretatzaileak, aldiz, programa exekutatu edo egikaritzen den bakoitzean erabili behar dira. Kasu honetan ez dago objektu-programarik, eta programa interpretatzailea beti egon behar da kargatuta. Iturburu-programaren aginduak banan-banan hartzen ditu, agindu bakoitza lehenengoz itzuliz eta gero exekutatuz.



2. Irudia: Interpretatzailearen bidezko exekuzioa

PERL (*Practical Extraction and Report Language*) 1990. hamarkadan sorturiko programazio-lengoaia dugu. Lengoaia *interpretatua* da eta, jatorrian, testu-fitxategiekin lan egiteko jaio zen. Horrela, beste goi-mailako lengoaietan agindu asko behar dituen zenbait eragiketa, perl-ez modu errazean (edo, behintzat, era laburrean) programatu daitezke; esan ohi da perl lengoaia *lazy-programming* (programazio alferra) egiteko dela. Izan ere, perl lengoaia ez baitago pentsatua

programazio klasikoa egiteko, eta ez da, egia esan, programazio egituratua irakasteko lengoaia hoberena.

Hona hemen programazio-lengoaia honen abantaila nagusiak:

- Testuak tratatzeko oso egokia.
- Hainbat funtziorekin dator, ia *edozer* egiteko balio dutenak.
- Espresio erregularrak erabiltzeko ahalmen handia eta eraginkortasuna.
- Sarrera / Irteera (fitxategietatik irakurri, fitxategietan idatzi) erraz egin daiteke.
- Plataforma independentea. Perl lengoaiaz idatzitako edozein programa, perl interpretatzailea (programa) instalatuta dagoen edozein sistematan (Windows, Unix, Linux, etab.) erabili ahal izango da.
- Sintaxi malgua (beharbada, malguegia).

Baditu, hala ere, zenbait alde ahul:

- Lengoaia interpretatua denez, ez da programazio-lengoaia klasikoak (C, ADA, Pascal, etab.) bezain arina.
- Ez da zorrotza, eta lengoaia honetan modu asko daude gauza bera egiteko. Hori printzipioz ez da txarra baina, programatzen hasteko, hobe da lengoaia zorrotz bat ikastea. Izan ere, programatzen ikasteko, *lagun gaiztoak* ere baitaude, baina perl-ek ez digu hauei buruz inongo oharrik emango.
- Sintaxi traketsa du. Programak, batzuetan nahiko kriptiko eta ulergaitzak izan daitezke.

2.3 Programak eraikitzeko pausoak

Problema bat konputagailuaren bidez ebatzi gura denean ondoko hiru pausoak bete behar dira: zehaztapena, algoritmoa, programa eta proba. Ikus ditzagun banan-banan.

2.3.1 Zehaztapena

Problema bat konputagailu-programa baten bidez ebaztekoa bada, erabat definitua egon behar da. Izan ere, problemaren gorabehera guztiak definituta egon behar dira,

hots, problemaren ebazpenerako zalantza-izpirik ezin da egon, bestela gero nahasteak azalduko direlako. Sarritan problema daukan pertsona ez da programa asmatuko duena, eta horrelakoetan oso garrantzitsua da arazoaren zehaztapen osoa burutzea, zeren pauso horretan zehatz-mehatz ezartzen baita *zer* egin beharko duen programak eta, aurrekoa bezain garrantzitsua dena, *zer* egin behar *ez* duen.

Zehaztapenean, alde batetik, jasoko diren sarrera-datuak, eta beste aldetik, lortu nahi diren irteera-datuak zehaztu behar dira, hau da, zein diren sarrera-datuak, eta irteera-datuek bete beharko dituzten propietateak.

Jo dezagun linguista batek ordenadorez metatutako testu-bilduma handia duela (*corpus* bat, alegia), eta bilduma horren gainean hainbat datu atera nahi dituela. Adibide gisa, corpus horretan, seguruenik, hitzak errepikaturik agertuko dira eta, beharbada, lerroren batean hitzen bat behin baino gehiagotan agertuko da. Gauzak horrela, suposa dezagun linguistak hitz jakin baten agerpen gehien duen lerroaren zenbakia jakin nahi duela. Linguista honek oraindik ez daki programatzen, eta programatzaile batekin hitz egiten du, programaren gorabeherak azaltzeko. Kasu honetan, hau dugu zehaztapena:

Sarrera-datuak: testu-bilduma (fitxategia) eta hitz bat
Irteera-datuak: hitz hori agerpen gehien duen lerro-zenbakia
Zehaztapenean argitu beharreko kontuak:

- Zer gertatzen da hitz jakin hori ez bada bilduman agertzen ?
- Zer gertatzen da bi lerro edo gehiagok hitz horren agerpen-kopuru maximo bera badute? Zein zenbaki atera behar da? Lehenengoa, azkena, denak?
- Etab. ...

Pauso hau funtsezkoa da. Zalantza guztiak argitu behar dira ezinbestean, jatorrizko problema duenak eta programa asmatuko duenak gauza bera uler dezaten; sortuko den programak zerbaiterako balio izatea nahi badugu, behintzat. Izan ere, zehaztapen egokia egiten ez bada, nahiz eta programa ondo eginda egon, eta pertsona eta egun askoren fruitua izan, erabiltzaileak zuen jatorrizko problema ez du ebatziko. Beste problema bat ebazteko balioko du, programa asmatu duenak ulertu zuena ebazteko hain zuzen, baina ez konputagailua erabili nahi duenaren problemari soluzioa emateko

2.3.2 Algoritmoa

Behin problemaren zehaztapena lortuta, eta erabiliko den programazio-lengoaia ondo ezagutuz gero, badirudi programa idazten has daitekeela, besterik gabe. Nahiz eta kasu errazenetan noizbait horrela jokatzen den, hori ez da ohiko prozedura. Zuzenean idatzi gabe, tarteko pauso bat bete behar da: algoritmo bat diseinatzea.

Algoritmoan, programan bezala, agindu-sekuentziak definitzen dira, baina ez programazio-lengoaia jakin baten aginduak erabiliz, programazio-lengoaia guztietan erabiltzen diren aginduen abstrakzioak (lengoaia guztiek konpartitzen dutena edo) baizik. Ondorioz, algoritmoa idazterakoan ez dira programazio-lengoaia batez idazteko gorde behar diren arau sintaktiko estuak jarraitu behar. Zailtasun maila hori geroago egin beharko den kodeketa faserako lagatzen da.

Algoritmo batean bi puntu nagusi definitu behar dira:

- Zelan adieraziko diren datuak, zer nolako aldagaiak erabiliko diren azalduz.
- Programaren mamia, bizkarrezurra. Programaren gorputza eta zeregina ondo definituta egongo da algoritmoan. Bertan, espezifikazioan deskribatutako problemaren soluzioa ebazteko pausoak emango dira.

Hemen dugu algoritmo baten adibidea:

```
algoritmo Handiena_Kalkulatu
Aurrebaldintza: bi zenbaki osoko teklatutik idatziko dira.
Postbaldintza:  pantailan agertuko da bi zenbakietatik
                handiena
hasiera
    Irakurri_Osokoa (Zenbaki1)
    Irakurri_Osokoa (Zenbaki2)
    baldin  Zenbaki1 > Zenbaki2
    orduan
        Handiena = Zenbaki1
    bestela
        Handiena = Zenbaki2
    ambaldin
    Idatzi_Osokoa (Handiena)
amaia
```

2.3.3 Programa

Pauso honetan eta hautatutako programazio-lengoaiaren arauak ondo ezagututa programatzaileak algoritmoko aginduak programazio-lengoaiara itzultzen ditu. Ondoren programa-itzultzaile bat (konpiladorea edo interpretzailea) beharko da goi-mailako programa hori ordenadorearen makina-lengoaiara itzul dezan. Kasu

gehienetan konpiladoreak programako errore sintaktikoak nabarituko ditu. Horrelako errore horiek guztiak zuzendu arte konpiladoreak ez du sortuko programa itzulia. Orduan prest egongo gara programa lehenengo aldiz egikaritzeko.

Hemen ditugu aurreko algoritmoari dagozkion bi programa, lehenengoa Perl lengoaiaz eta bigarrena Ada lengoaiaz idatzita:

```
#!/usr/bin/perl -w

use strict;

my ($zenbaki1, $zenbaki2, $handiena);

print "Sartu 2 zenbaki, bakoitza lerro batean\n";
$zenbaki1 =<>;
$zenbaki2 =<>;
chomp($zenbaki1);
chomp($zenbaki2);
if ($zenbaki2> $ zenbaki2) {
  $handiena = $zenbaki1;
} else
  $handiena = $zenbaki2;
};
print "$Handiena\n";
```

```
procedure Handiena_Kalkulatu is

   Zenbaki1, Zenbaki2, Handiena: Integer;

begin
    Put("Sartu 2 zenbaki, bakoitza lerro batean");
    New_Line;
    Get(Zenbaki1);
    Get(Zenbaki2);
    if  Zenbaki1 > Zenbaki2
    then
        Handiena := Zenbaki1;
    else
        Handiena := $Zenbaki;
    end if;
    Put(Handiena);
end Handiena_Kalkulatu;
```

2.3.4 Proba

Eraikitako programak erroreak izan ditzake. Frogatu egin behar da programaren zuzentasuna. Honetarako proba-kasuak egikaritzen dira. Programaren zehaztapenaren arabera kasu desberdinek adierazten dituzten sarrera-datuen multzoak definitu eta bakoitzari dagokion emaitza ezarri behar da aldez aurretik. Errorerik aurkitzen bada zuzenketa egin beharko da: programa, edo algoritmoa, edo zehaztapena. Pauso honi errore-arazketa deitzen zaio. Errore-arazketa ez da inoiz ere

amaitzen programa handiekin, ezinezkoa izaten baita zuzentasunerako proba guztiak egikaritzea, oso errepresentagarriak direnak soilik hartzen dira eta. Probetan errorerik agertzen ez bada zuzentzat ematen da programa eta erabiltzen hasteko prest dago, baina erabiltzerakoan kontuan hartu ez diren posibilitateak gertatuz gero, errore berriak ager daitezke, orduan ere zuzenketa berri bat egin beharko delarik.

Programak ahalik eta modu ulergarrienean idatzi behar dira, beren zuzentasunaz edo errore-ezaz seguritatea emateko eta errore-zuzenketa errazagoa izan dadin.

2.4 Estiloa

Programak ahalik eta modu ulergarrienean idatzi behar dira, beren zuzentasunaz edo errore-ezaz ziurtasuna emateko eta errore-zuzenketa errazagoa izan dadin. Programazio-estiloari begira, ondoko puntuak hartu behar ditugu kontuan:

- Programa erabilgarri gehienak, egileez gain, beste programatzaile batzuek ere irakurtzen dituzte.
- Irakurle bakoitzak programak nola lan egiten duen erraz ikusi behar du, erroreak aurkitu ahal izateko eta zuzenketak edo aldaketak egiteko.
- Estilo txarrez eginiko programa bat irakurtzea ez da batere atsegina, ezta egilearentzat berarentzat ere.

3 Eredu algoritmikoa. Algoritmoen osagaiak

Bost dira algoritmoak idaztean menperatu behar diren oinarrizko kontzeptuak: **objektuak** (edo datuak), objektuak manipulatzeko **aginduak**, datu berriak kalkulatzeko **adierazpenak**, aginduak betetzeko ordena alda dezaketen **kontrol-egiturak** eta problema zailak azpiproblema sinpleagotan banatu eta azpiproblema bakoitza bere aldetik ebazteko laguntza eskaintzen duten **moduluak** (funtzioak eta prozedurak).

3.1 Objektuak: aldagaiak, konstanteak eta literalak

Problema baten datuak objektuen bidez adierazten ditugu algoritmoetan. Objektu horiek hiru ezaugarri dituzte beti:

- IZENA: identifikadore bat.
- MOTA: mota batek aspektu hauek definitzen ditu:
  - bere balio posibleen multzoa
  - eta balio horiekin egin daitezkeen eragiketak

    Adibidez: osokoa, erreala, karaktere-katea, bektorea, hash-a
- BALIOA: objektu batek une zehatz batean daukana

Objektuen artean konstanteak eta aldagaiak bereizten dira. Kasu bietan izena eta mota aldaezinak dira algoritmoa egikaritzen denean. Konstanteen kasuan berdin gertatzen da balioarekin, baina aldagaien balioa aldatu egin daiteke.

Konstantea deritzo, behin bere balioa finkatu denetik aurrera, balio hori inoiz aldatzen ez zaion objektuari (adibidez: *Pi* konstantea 3.14159 balio erreala adierazteko defini daiteke, edo *Segundoak_Orduko* konstantea, ordu batek dituen 3600 segundoren kopurua adierazteko).

Aldagaia, berriz, alda daitekeen balioa duen objektua da (adibidez: algoritmoa egikaritzen hasten denean, *Kontagailua* izeneko aldagai batek 0 osoko balioa du eta bukaeran 21 balioa).

Literalak deritzen objektuak ere balio konstanteak adierazteko erabiltzen dira, baina zuzenean balioa idatziz, identifikadorerik erabili gabe. Adibidez:

0 1 60 1_000_000	(osoko literalak)
0.0 3.14158 6.02e23	(literal errealak)
'H' ':' ' '	(karaktere motako literalak)
"Ordua: " "???"	(kate motako literalak)

Objektu bat datu-mota batekoa dela esaten dugunean bi gauza zehazten ditugu: batetik objektuak har ditzakeen **balio posibleen multzoa** (*domeinu* deritzona) zein den, eta bestetik balio horiekin erabil daitezkeen **eragiketak** zein diren.

Objektu batzuk **sinpleak** dira, datu bakar batekin errepresentatzen baitira (*bakunak* edo *eskalarrak* ere esaten zaie); beste batzuk, ordea, konposatuak dira eta datu-mota **egituratuak** erabili behar dira haiek errepresentatu ahal izateko. Azter ditzagun oinarrizko algoritmoetan erabiltzen diren datu-mota arruntenak. Lehenengoz, datu-mota sinpleak, eta gero egituratuak.

Perl-ez, aldagai bat zein motakoa den jakiteko, hasieran karaktere berezi bat ipintzen zaio. Karaktere berezi horiek ondoko hauek dira:

Aldagaiaren hasierako karakterea	**Aldagaiaren mota**	**Adibidea**
$	Eskalarra	$a
@	Bektorea (*array)*	@a
%	*Hash*	%a

Halaber, identifikadore bera erabil daiteke mota desberdineko objektu desberdinak adierazteko, hau da, $a, @a eta %a *hiru objektu desberdinak dira*.

3.1.1 Aldagaien erazagupena

Programazio-lengoaia gehienetan, aldagaiak erabili baino lehen erazagutu egin behar dira. Aldagai bat erazagutzean, aldagai hori existitzen dela adierazten da, hau da, geroago programan zehar erabiliko dela zehazten da. Horretaz gain, erazagupenean aldagaien mota ere definitzen da.

Aldagaiak erazagutzea ohitura ona da programatzeko garaian, hainbat akats identifikatzeko balio baitigu. Gauzak horrela, zenbait programazio-lengoaiak ez du aldagaia erazagutu gabe erabiltzeko aukerarik ematen. Zoritxarrez, Perl-ek ez du hurbilpen hau jarraitzen. Hala ere, badago forma bat Perl-ekin aldagaien erazagupena behartzeko. Horretarako, hau idatzi behar dugu programaren hasieran:

```
use strict;
```

eta, horren ondoren, programan barruan erabiliko diren aldagai guztiak erazagutu beharko ditugu, **my** funtzioaren bitartez:

```
my $a;           # a aldagai eskalarraren erazagupena
my @Bek;         # Bektore motako Bek aldagaia erazagutu
my ($a, $b, @c); # a eta b eskalarrak dira.
                 # c bektore motakoa da
```

use strict agindu berezia erabiliz[1], programak ez du funtzionatuko aurretik erazagutu gabeko aldagairen bat erabiltzen badugu. Arestian aipatu den bezala, hau gauza ona da: konturatu gabe aldagai izenen bat programan idazterakoan akats tipografikoren bat egiten badugu –eta, adibidez, @Bek idatzi ordez @Bec idazten badugu--, programak errore bat emango digu, @Bec bektorea ez baita erazagutu.

Hala ere, $a aldagai eskalarraren balioa edozein balio izan daiteke: zenbaki oso, zenbaki erreal, boolear edo karaktere-katea. Alde batetik erosoa da hori, baina bestetik, babes falta dakar horrek gero aldagaien erabilera okerra edo erroreak detektatzeko.

3.1.2 Datu-mota sinpleak

Perl lengoaiak datu-mota sinpleari eskalarra deritzo. Eskalar hauetan, hainbat datu-motako balioak gorde daitezke. Hauexek dira aldagai eskalar batek eduki ditzakeen motak:

[1] Aginduak zer diren jakiteko, ikus 0 atala.

Datu-mota	**Balioak**	**Adibidea**
Osoko	osoko zenbakiak	8
Erreal	zenbaki errealak	8.34
Karaktere	Karaktereak	'a'
Boolear	balio boolearrak	TRUE(1) edo FALSE(0)
Kate	karaktere-kateak	"eta abar"

Eskalar motako aldagaiak erazagutzeko, aldagaiaren izena zehaztu behar da. Bestalde, aldagaiaren izenari eskalarra dela adierazten duen ikurra ($) jarri behar zaio aurretik. Ikur hau, semantikoki, ingelesezko “the” hitza bezala ikus daiteke: ondoren datorrena balio bakuna duela adierazten da.

Adibidez:

```
my ($adina)    # adina eskalar motako aldagaia da.
my ($helbidea) # helbidea ere bai.
```

3.1.2.1 Osokoak

Zenbaki osokoak adierazteko datu-mota.

DOMEINUA: …, –3, –2, –1, 0, +1, +2, +3, …

ERAGIKETAK:

Eragileak eta emaitzak osokoak dira.

Eragile diadikoak		*Eragile monadikoak*	
+	Batuketa	-	Ukapena
-	Kenketa	abs	Balio absolutoa
*	Biderketa		
%	Modulua		
**	Berreketa		

Adibidez:

```
5           # 5
6 + 5       # 11
2 ** 3      # 8
int(5 / 2)  # 2
5 % 2       # 1
```

Eta eragile erlazionalak:

Eragiketa	
<	Txikiago
>	Handiago
<=	Txikiago berdin
>=	Handiago berdin
==	Berdin
!=	Desberdin

Adibidez:

```
(7 > 8)     # Egiazkoa
(9 >= 2)    # Egiazkoa
(-9 >= -2)  # Faltsua
```

3.1.2.2 Errealak

Zenbaki errealak adierazteko datu-mota.

DOMEINUA:

```
0.0     1.5   386_473.0   3.141_592_65
3.86473e5   3.0e+8      0.1234E-20
```

Adibidez:

```
3.14159265  # Pi
6.02e3      # 6020
```

ERAGIKETAK:

Eragileak eta emaitzak *errealak* dira (berreketako berretzailea izan ezik, *osoko* motakoa izan behar baita):

Eragile diadikoak		*Eragile monadikoak*	
+	Batuketa	-	Ukapena
-	Kenketa	abs	Balio Absolutoa
*	Biderketa		
/	Zatiketa		
**	Berreketa		

Adibidez:

```
1.0 + 4.0   # 5.0
5.0 / 2.0   # 2.5
5.0 ** 2      # 25.0
```

Eta eragiketa erlazionalak:

Eragiketa	
<	Txikiago
>	Handiago
<=	Txikiago edo berdin
>=	Handiago edo berdin
==	Berdin
!=	Desberdin

Adibidez:

```
(7.5 > 8)          # Faltsua
(3.02e2 >= 100.5)  # Egiazkoa
(-9.6 >= -2e2)     # Egiazkoa
```

Zenbaki erreal batetik zenbaki osoa lortzeko **int** funtzioa dago, funtzio honek zenbaki erreal baten zati osoa itzultzen duelarik. Adibidez:

```
5 / 2     # 2.5 Erreala
int(5/2)  # 2   Osoa
```

3.1.2.3 Karaktere-kateak (*string*)

Karaktere-kateak (*string)* adierazteko datu-mota.

DOMEINUA: Karaktere-kateak, adibidez: "Maite", "Kale Nagusia 12, 3.C"

ERAGIKETAK:

Eragiketa	
.	Konkatenazioa
x	Biderketa

Adibidez:

```
"Kaixo"                     # "Kaixo"
"Kepa"." eta "."Aitor"      # "Kepa eta Aitor"
"Kaixo" x 3                   # "KaixoKaixoKaixo"
```

Eragiketa erlazionalak:

Eragiketa	
lt	Txikiago
gt	Handiago
le	Txikiago edo berdin
ge	Handiago edo berdin
eq	Berdin
ne	Desberdin

Kontuan izan Perl-ek eragile desberdinak erabiltzen dituela zenbakiak eta karaktere-kateak konparatzeko. Horrela, “>” eragileak, zenbaki bat bestea baino handiagoa denentz jakiteko balio digu (zenbakizko konparazioa). “gt” eragiketa, berriz, karaktere-kate bat bestea baino “handiagoa” ote den esaten digu, baina kasu honetan konparazio lexikala burutzen du: kate bat bestea baino txikiagoa izango da alfabetikoki bestearen aurrean baldin badago. Bi eragiketa hauen emaitzak, bestalde, ez dira bat etortzen. Adibidez, 1000 zenbakia 200 zenbakia baino handiagoa den bitartean, "1000" katea "200" katea baino txikiagoa da.

Adibidez:

```
"20" gt "100"      # Egiazkoa
20 > 100           # Faltsua
"2"."0" lt "100"   # Egiazkoa ("20" lt "100")
```

Kontuz! Eragile hauek zenbakiekin bakarrik erabil daitezke, karaktere-kateekin ez:

```
<, <=, >, >=, !=, =
```

3.1.2.3.1 Interpolazioa

Perl lengoaian, karaktere-kate batean aldez aurretik definitutako aldagai-izenen bat agertzen bada, agerpen hori aldagaiak duen balioarekin ordezkatzen da normalean. Adibidez, ondorengo lerroek “sagarrak eta udareak“ idazten dute pantailan.

```
my $a = "sagarrak";
my $b = "udareak";
print $a." eta ".$b;
```

"sagarra eta udareak" karaktere-katea osatzeko, kateamendu (**.**) eragilea erabili da, $a-ren balioari ("sagarrak"), " eta " katea eta $b-ren balioa ("udareak") erantsiz. Badago, hala ere, azken lerro hau idazteko era trinkoagorik:

```
print "$a eta $b"; # 'sagarrak eta udareak' idatziko du
```

eta esaten da $a eta $b aldagaiak *interpolatu* egin direla karaktere-katean: $a edo $b aldagaien agerpenak aldagai horiek dituzten balioekin ordezkatu dira. Batzuetan, ordea, ez dugu nahi interpolaziorik gauzatzea, eta horrelakoetan komatxo sinpleak erabil ditzakegu:

```
print '$a eta $b'; # '$a eta $b' idatziko du
```

edo, aldagaiaren mota adierazten duen ikurraren aurretik '\' karakterea jartzen badugu:

```
print "\$a eta \$b"; # '$a eta $b' idatziko du
```

Aldagaien balioez gain, komatxo dobleen artean agertutako hainbat karaktere berezi ere interpolatuko dira. Karaktere berezi horiek lerro-bukaera marka "\n", tabuladorea "\t" eta beste hainbat marka berezi adierazteko balio dute. Hona hemen karaktere berezien zerrenda:

Karakteak	Esanahia
\t	*Tab*
\n	*newline*
\r	*return*
\f	*Form feed*
\b	*backspace*
\a	*alarm (bell)*
\e	*escape*
\033	karaktere zortzitarra
\x1b	karaktere hamaseitarra
\c[	kontrol-karakterea
\l	hurrengo karakterea minuskulaz
\u	hurrengo karakterea maiuskulaz
\L	minuskula \E aurkitu arte
\U	maiuskula \E aurkitu arte
\E	Kasu-modifikatzailea
\Q	metakaraktereak ez interpretatu \E aurkitu arte

Horrela, ondorengo aginduak

```
print "$a\t$b";
```

"sagarrak udareak" karaktere-katea idatziko du pantailan.

Ariketa: ondorengo lerroen emaitza zein da?

```
my $a = "Kaixo";
my $b = "lagun";
my $c = "maitia";

print "$a, $b $c\n";
print '$a\t$b eta $c\n';
print $a.' eta $b'." $c\n2;
print "\l$a, $b \U$c\E\n";
```

Orain arte ariketak `print` komandoa erabiliz egin ditugu, baina berez beste edozein agindu erabil genezakeen, adibidez, esleipena[2]. Adibidez:

```
my $z = "$c\t$a $b";
          # $z-k  'maitia    Kaixo lagun' balio du
```

Interpolazioak, beraz, ez dauka zer ikusirik `print` aginduarekin, hau da, karaktere-kateen *literalen* ezaugarri bat da.

3.1.2.4 Boolearrak

Balio logikoak (egiazkoa ala faltsua) adierazteko datu-mota.

DOMEINUA: {egiazkoa , faltsua }

ERAGIKETAK: eragile erlazionalak.

Eragigai biak mota berekoak dira eta (baina ez dute zertan boolearrak izan behar), emaitza boolearra.

Eragiketa	
<	Txikiago
>	Handiago
<=	Txikiago edo berdin
>=	Handiago edo berdin
==	Berdin
!=	Desberdin

Eragile logikoak.

Eragigai biak eta emaitza ere boolearrak dira.

&& *eta* eragiketa logikoa (and)

| | *edo* eragiketa logikoa (or)

^ *ala* eragiketa logikoa (xor)

! *ez* eragiketa logikoa (not)

[2] Esleipena (edo asignazioa), agindu-mota bat da. Ikus 3.3.1 atala.

and, or, xor eta not eragiketa boolearren egia-taulak:

A	**B**	**A & B**	**A | B**	**A ^ B**	**! B**
Faltsua	Faltsua	Faltsua	Faltsua	Faltsua	Egiazkoa
Faltsua	Egiazkoa	Faltsua	Egiazkoa	Egiazkoa	Faltsua
Egiazkoa	Faltsua	Faltsua	Egiazkoa	Egiazkoa	
Egiazkoa	Egiazkoa	Egiazkoa	Egiazkoa	Faltsua	

Perl-ek modu berezia du `boolearra` datu-mota adierazteko. Berarentzat, 0 zenbakia edo "" karaktere-kate hutsa ez den edozer egiazkoa izango da. Horrela, bada:

```
1    # Egiazkoa
0.5  # Egiazkoa
0    # Faltsua

"motorra" # Egiazkoa
""        # Faltsua
```

3.1.2.5 Espresio erregularrak

Adierazpen erregularrak baliatuz testuetan bilaketak eta ordezteak oso erraz egin daitezke. Ondoko taulan Perl lengoaiak espresio erregularrak adierazteko erabiltzen duen notazioa erakusten da:

Karaktere bat adierazteko espresioak	
a	"a" karakterea
[^a]	"a" ez den beste edozein karaktere
[a-z]	letra xeheen multzoa
[A-Z]	letra larrien multzoa
[a-zA-Z]	letra guztien multzoa
[aeiou]	Bokalak
[0-9]	Zenbakiak
[^aeiou]	bokala ez den edozein karaktere
[^0-9]	edozein karaktere zenbakiak izan ezik
.	edozein karaktere

Kokapena adierazteko espresioak	
^	lerroaren hasiera
$	lerroaren bukaera
\b	hitz-hasiera edo bukaera
\B	hitz-hasiera edo bukaera ez dena
Zenbatetan errepikatu adierazteko espresioak	
*	zero bider edo gehiagotan aurreko espresioa
+	behin edo gehiagotan aurreko espresioa
?	aurreko espresioa behin edo ez egon
{n}	aurreko espresioa n bider
{n, m}	aurreko espresioa n eta m-ren arteko bider
{n, }	aurreko espresioa gutxienez n bider
Bestelakoak	
.*	edozein karaktere-kate
(...)	espresiokoa elkartzea (memoriaratzen da)

Laburpenak	
\\	'\' karakterea
\t	tabuladorea
\n	Newline
\r	Return
\f	form feed
\a	alarma (bell)
\e	escape
\033	zenbaki zortzitarra
\x1B	zenbaki hamaseitarra
\c[	Kontrol karakterea
\l	hurrengo karakterea minuskulaz
\u	hurrengo karakterea maiuskulaz

\L	\E aurkitu arte karaktereak minuskulaz
\U	\E aurkitu arte karaktereak maiuskulaz
\E	bukaera-marka
\Q	\E aurkitu arte ez egin karaktere-itzulpenik
\w	“hitz” karaktere bat (alfanumerikoak gehi "_")
\W	“hitz” karakterea ez dena
\s	zuriunea den karakterea [\t\n\r\f]
\S	zuriunea ez den karakterea
\d	digitu den karaktere bat
\D	digitu ez den karaktere bat

Beraz, x eta y adierazpen erregularrak badira:

```
x*   x zero bider edo gehiagotan
x+   x behin edo gehiagotan
x|y    x edo y
```

karaktere bereziak:

```
\.   puntu karakterea
\+   plus karakterea
```

Adibideak:

b bat baino gehiago gutxienez a baten ondoren:	`a+b*`
Hitz bat:	`[a-zA-Z]+`
edo bestela ere bai:	`\w+`
Hitz bat edo zenbaki bat:	`[a-zA-Z]+|[0-9]+`
Puntuazio ikurrak:	`[\.,;:()]`

Perl-ek, espresio erregularrak eskalar motako aldagaien gainean egikaritzeko aukera ematen du. Espresio erregularrek // karaktere-bikotearen artean egon behar dute. Erabilpen usuena horrelakoa da:

```
$a =~ /ESP_ERREG/;
```

eragiketa honen bidez, $a aldagaiaren balioa *ESPR_ERREG*-ekin parekatzen bada, TRUE itzuliko da, eta bestela FALSE. Albo-ondorio bezala, hainbat aldagairi balioak ematen zaizkio ($&, $’, $`). Adibidez:

```
# Demagun $a eq "Etxe honetan 20 lagun bizi gara";
$a =~ /\d+/; # $a katean, karaktere bat edo gehiago parekatu.
```

Agindu horren ondoren, aldagai bereziek honako balio hauek edukiko dituzte:

Aldagaiaren izena	Balioa	Esanahia
$`	"Etxe honetan "	Hasieratik, parekatu arte dagoen azpi-katea
$&	20	Parekatu den azpi-katea
$’	" lagun bizi gara"	Parekatu ondoren, bukaeraraino dagoen azpi-katea

Honetaz gain, espresio erregularrean agertutako parentesien arteko azpi-parekatzeak ere gordeko dira, $1, $2, ... aldagaietan. Adibidez:

```
# Demagun $a eq "2002/11/18"
$a=~ /(\d+)\/(\d+)\/(\d+)/;
# Orain:
#     $1 == 2002
#     $2 == 11
#     $3 == 18
```

Parekatze-eragiketaren alderantzizkoa ere badago:

```
$a !~ /ESP_ERREG/;
```

Hau da, $a aldagaiaren balioa *ESPR_ERREG*-ekin parekatzen **ez** bada TRUE itzuli, eta, bestela FALSE. Albo-ondorio bezala, hainbat aldagairi balioak ematen zaizkio ($&, $’, $`, $1, ...).

Zehatzago, espresio erregularrak era honetan adieraz daitezke perl-ez

```
[$ALDAGAI_ESK [=|!]~] /ESP_ERREG/PAREKATZE_OPZIOA
[$ALDAGAI_ESK [=|!]~] s/ESP_ERREG1/ESP_ERREG2/PAREKATZE_ERAGILEA
```

non *ESP_ERREG* espresio erregular bat den.

Horrela, bi parekatze-metodo dago: parekatze soila eta ordezkapena. Parekatzearen eragilea hau da:

```
/ESP_ERREG/PAREKATZE_OPZIOA
```

orain arte ikusitako parekatze-eragilea. Espresio erregularra parekatzen saiatzen da. Bigarrena, ordezkapenarena, hau da:

```
s/ESP_ERREG1/ESP_ERREG2/PAREKATZE_OPZIOA
```

`ESP_ERREG1 ESP_ERREG2`-rekin ordezkatzen du. Adibidez:

```
# Demagun $a eq "bost, sei, etab."
$a =~ s/\betab\./eta abar/;
# ESP_ERREG1 = \betab\.
#   hau da, hitz-marka hasiera eta "etab." Katea
# ESP_ERREG2 =eta abar
#   hau da, "eta abar" katea
# aurreko agindu honek, beraz, era laburtuan idatzitako
# "etab." katea "eta abar" katearekin ordezkatuko du.
#
# Ondoren, $a eq "bost, sei, eta abar" izango da.

# Demagun $a eq "Aitor eta Xabier" dela.
$a =~ s/^(\w+) eta (\w+)$/$2 eta $1/;
# $a eq "Xabier eta Aitor"
```

Honetaz gain, `PAREKATZE_OPZIOAK` ere baditugu. Opzio horiek, aginduaren bukaeran jartzen diren karaktereak dira, eta horien bitartez parekatzea nola gauzatuko den kontrola dezakegu. Guri bi parekatze-opzio bi interesatuko zaizkigu:

Opzioa	Esanahia
i	maiuskula/minuskulak kontuan ez hartu parekatzerakoan
g	parekatze globala (orokorra) Ordezkapenetan: ordezkapena behin bakarrik egin ordez, kate osoko agerpen guztietan egin.

Horrela, espresio erregular bati “i” opzioa gehituz, maiuskula eta minuskulen arteko desberdintasunik ez da egongo. Adibidez:

```
# $a eq "Kepa";
$a =~ /kepa/;   # FALTSUA
$a =~ /kepa/i;  # EGIAZKOA
```

“g” opzioa, aldiz, parekatzea kate osoaren zehar egitea ahalbideratzen du. Opzio honen erabilera erraz uler daiteke ordezkapenetan:

```
#Demagun $a eq "UEUn nago. UEU biziki gustukoa dut"
$a =~s/UEU/Udako Euskal Unibertsitatea/; # g opziorik gabe
#$a eq "Udako Euskal Unibertsitatean nago. UEU biziki
#       gustukoa dut"
```

lehenengo adibide honetan, “UEU” azpi-katea “Udako Euskal Unibertsitatea” katearekin ordezkatu du, baina *behin bakarrik*. Izan ere, $a aldagaian bi alditan agertzen baitzaigu “UEU” azpikatea, baina ordezkapena soilik lehenengo agerpenean egin da. Portaera hau aldatzeko, “g” opzioa dugu:

```
# Demagun $a eq ”UEUn nago. UEU biziki gustukoa dut”
$a =~s/UEU/Udako Euskal Unibertsitatea/g; # g opzioarekin
#$a eq “Udako Euskal Unibertsitatean nago. Udako Euskal
#       Unibertsitatea biziki gustukoa dut”
```

hau da, “g” opzioa gehituz, “UEU”ko azpi-kate *guztiak* ordezkatuko dira.

Opzio hau ordezkapenetarako ez ezik, parekatze eragilearekin erabil daiteke, normalean, begiztak[3] egiteko:

```
#demagun $a aldagai eskalarrean hainbat zenbaki daudela,
#espazioekin bereiziak, hau da, $a eq “12 3 543 65 7 ...”
while ($a =~ /\d+/g) {
  my $zbk = $&; # $zbk aldagaiari azkenengo parekatzea
                # esleitu
  ...
}
```

aurreko begizta honek zenbaki guztiak parekatuko ditu, banan-banan. Lehenengo iterazioan $zbk==12 izango da. Hurrengoan $zbk==3 izango da, eta horrela ($a=~/\d+/) espresio erregularrak faltsua itzuli arte, hots, zenbaki gehiagorik geratzen ez den arte.

Espresio erregularrekin bukatzeko, azken ohar bat: espresio erregularretan aldagairik ipintzen ez badugu ($a =~ edo $a !~), parekatzea $_ aldagai lehenetsiarekin burutuko da. $_ aldagaiaren edukia lerro oso bat izaten da eta automatikoki hartzen du balio hori lerro guztiak aztertzen direnean. Honek hainbat programa errazten ditu. Adibidez:

```
while (<>) { # Sarrera estandarretik lerro bat irakurri,
             # eta $_ aldagai lehenetsian sartu
   s/UEU/Udako Euskal Unibertsitatea/g; # ordezkapena $_
                                        # aldagaian burutu
   print;  # $_ aldagai lehenetsia inprimatu
}
```

3.1.3 Datu-mota egituratuak

Orain arte definitutako datu-motak *sinpleak* izan dira: balio bakarra biltzen dute. Datu-mota *egituratuek* balio bat baino gehiago biltzen dute. Datu-mota hauek ikusiko ditugu:

- **Bektorea (*array)***: eskalar motako balio-sorta ordenatu bat. Osagaiak bektorean duten posizioaren bitartez erreferentziatzen dira.
- ***Hash***: (gako, balio) bikoteen sorta bat. Gakoak eta balioak eskalar motakoak izan behar dira. Array bezalako egitura bat da, baina elementu bakoitza lotuta duen gakoaren bitartez erreferentziatzen da.

[3] Adibide hau ulertzeko, begiztak zer diren jakin behar da. Ikusi 3.4.3 atala.

3.1.3.1 Bektoreak (*array)*

Bektore motako aldagai batek bere barruan eskalar motakoak diren hainbat balio dauzka. Osagai horiek indize bat erabiliz erreferentziatzen dira, hots, bektorean duten posizioaren arabera. Kontu berezia eduki behar da, Perl programazio-lengoaian bektoreko lehenengo osagaiaren indizea 0 izaten delako, eta ez 1 beste programazio-lengoaia batzuetan bezala.

Perl-ez bektore motako aldagai (edo objektu) bat erazagutzeko, besterik gabe, bere izena zehaztu behar da. Bestalde, aldagaiaren izenari bektorea dela adierazten duen ikurra (@) jarri behar zaio aurretik. Ikur honek, semantikoki, ingelesezko “these” edo “those” bezala jokatzen du, hau da, ondoren datorrena multzo bat dela adieraziko du. Adibidez:

```
my (@Kalifikazioak)
my (@Batezbestekoak)
```

Bektorearen osagai bat adierazteko, bektorearen izena eta osagaiari dagokion indizea idatzi behar dira. Osagaia eskalarra denez aurretik $ karakterea jarri behar da.

```
$Kalifikazioak[5]  Kalifikazioak deritzon  bektorearen
                     seigarren osagaia da
$Batezbestekoak[1] Batezbestekoak bektorearen bigarren
                    osagaia da
```

Kontuan izan:

- Indizeak 0-tik hasten dira. Beraz, bektore baten *i.* elementuaren indizea (*i-1)* izango da.
- Nahiz eta Kalifikazioak aldagaiaren mota bektorea izan, bere osagaiak eskalarrak dira. Hortaz, osagaia erreferentziatzerakoan erabili behar den ikurra eskalarrei dagokiena da ($). Adibidez:

```
$Kalifikazioak[2] = 7;    # hirugarren osagaia, eskalarra
@Kalifikazioak=(5,4,7,8); # Bektore osoari balio bat eman
```

- Indizea adierazpen bat ere izan daiteke:

```
$Kalifikazioak[1+1]# hirugarren osagaia
$Kalifikazioak[$i] # i zenbaki osoa da, exekuzio garaian
                   # balio zehatzen bat edukiko duena
$Kalifikazioak[$i*2-3] # une horretan i-ren balioa 4
                       # bada, 6. osagaia da aipatzen dena.
                       # Geroago $i-ren balioa 5 bada, 8.
                       # osagaia erreferentziatuko da.
```

- Arrayaren tamaina ez da finkoa. Perl-ek automatikoki handitu edo txikituko ditu osagaiak gehitzen/kentzen ditugun heinean.

- Array baten luzera kalkulatzeko badago *scalar* deritzon funtzio bat:

```
scalar(@Kalifikazioak) # Kalifikazioak bektorearen tamaina
```

ERAGIKETAK:

Bektoreen artean, eragiketa hauek egin daitezke, besteak beste:

- Bektore osoaren asignazioa:

```
my @frutak = (“sagarra”, “udarea”, “platanoa”);
my @zbklista = (1..8);      # (1,2,3,4,5,6,7,8)
@lista2 = @zbklista;
```

- Osagai bakoitzak bere motako aldagai bakun modura jokatzen du:

```
$a = $frutak[0];  # $a-ri “sagarra” asignatu
$frutak[0] = “marrubia”; # @frutak bektorearen
                         # lehenengo elementua aldatu
$zbklista[2] = $zbklista[2]+1; # 3. elementua batean
                               # handitu
```

- Bektore bati, bere azken indizea baino handiagoa den elementu bat asignatzerakoan, bektorea bera dinamikoki handituko da. Adibidez:

```
my @Letrak = (“a”,”b”,”c”); #scalar(@Letrak) == 3
$Letrak[4] = “e”;
# Orain, @Letrak==(“a”,”b”,”c”,,”e”), hau da,
# scalar(@Letrak) == 5 izango da
#    gainera, $Letrak[3] existitzen da, baina ez du
#    baliorik. Bestalde, $Letrak[4] == “e”
```

  Kontuz ibili aukera honekin. Nahi gabe bektore handiegia sortu dezakegulako.

- Bektoreak kateatu:

```
my @arrkonpos = (@frutak, @zbklista);
# @frutak eta @zbklista bektoreak kateatuz lorturiko
# bektorea, kasu honetan:
# @arrkonpos = (“sagarra”, “udarea”, “platanoa”,
#               1,2,3,4,5,6,7,8)
```

- Bektore baten azpi-bektore bat lortu:

```
my @azpBek = @arrkonpos[2..5];
   # @arrkonpos bektorean 3. elementutik
   # 6. elementura osatutako azpibektorea:
   # (“platanoa”, 1,2,3)
```

- Bektore osoen arteko konparazioa (==, !=)

```
@frutak == (“sagarra”, “udarea”, “platanoa”) #Egiazkoa
@arrkonpos[3..10] == @zbklista               #Egiazkoa
(“kaixo”, “lagun”) == (“lagun”, “kaixo”)   #Faltsua
(1..3) != (1,2,3)                          #Faltsua
```

- Elementuak gehitu/kendu:
  - **push** eta **pop** funtzioak: **push**-ek elementu bat gehituko du, bektorearen bukaeran. **pop**-ek, bektorearen azken elementua kenduko du:

```
# Demagun @frutak == (“sagarra”, “udarea”, “platanoa”);
push (@frutak, “marrubia”);
# @frutak == (“sagarra”, “udarea”, “platanoa”,
#             “marrubia”);
pop(@frutak);
# @frutak == (“sagarra”, “udarea”, “platanoa”);
# kendutako balioa itzultzen du, “marrubia”
```

  - **shift** eta **unshift** funtzioak: **unshift**-ek elementu bat gehituko du, bektorearen hasieran. **shift**-ek, bektorearen lehenengo elementua kenduko du:

```
# Demagun @frutak == (“sagarra”, “udarea”, “platanoa”);
shift(@frutak)
# @frutak == (“udarea”, “platanoa”);
# kendutako balioa itzultzen du, “sagarra”
unshift(@frutak, “marrubia”);
# @frutak == (“marrubia”, “udarea”, “platanoa”);
```

  - Funtzio hauek denak **splice** funzio orokorraren bitartez adieraz daitezke:

```
splice BEKTORE1, OFFSET, LUZERA, BEKTORE2
splice BEKTORE1, OFFSET, LUZERA
splice BEKTORE1, OFFSET
```

    Funtzio honek `BEKTORE1`-ean, eta `OFFSET` indizetik aurrera, `LUZERA` elementu kentzen ditu eta `BEKTORE2` elementuekin ordezkatu. `BEKTORE2-rik` pasatzen ez bazaio ez du ordezkapenik egingo. `LUZERA` pasatzen ez bazaio, `OFFSET` posiziotik bukaeraraino egingo du ordezkapena.

Funtzioa	Baliokidea
`push(@a,$x,$y)`	`splice(@a,$#a+1,0,$x,$y)`
`pop(@a)`	`splice(@a,-1)`
`shift(@a)`	`splice(@a,0,1)`
`unshift(@a,$x,$y)`	`splice(@a,0,0,$x,$y)`
`$a[$x] = $y`	`splice(@a,$x,1,$y);`

- Bektore baten alderantzizkoa: **reverse** funtzioa:

```
my @Letrak = (“a”,”b”,”c”);
my @Letrak2=reverse(@Letrak); #@Letrak2==(“c”,”b”,”a”)
```

- Bektore bat ordenatu: **sort** funtzioa:
  - Ordenazio alfabetikoa:

```
my @frutak = (“sagarra”, “udarea”, “platanoa”);
my @frord=sort (@frutak);
         # @frord==(“platano”,”sagarra”,”udarea”)
```

  - Zenbakizko ordena:

```
my @a = (2,10,6,1);
my @b = sort {$a <=> $b} @a; # @b ==(1,2,6,10)
```

- Bektore batetik eskalar bat sortu: **join** funtzioa.

```
join ESPRESIOA, BEKTOREA
```

  funtzio honek bektore batetik eskalar bat itzultzen du. Eskalar horretan bektoreko elementu guztiak kateatuko dira, tartean `ESPRESIOA` txertatuko duelarik. Adibidez:

```
my @frutak = (“sagarra”, “udarea”, “platanoa”);
my $frutak = join(“ eta ”, @frutak);
     # $frutak eq ”sagarra eta udarea eta platanoa”
```

- Eskalar batetik bektore bat lortu: **split** funtzioa.

```
split /ESP_ERREG/, ESPRESIO_ESKALARRA
```

  split funtzioak eskalar bat eta espresio erregularra behar ditu. Eskalarrean espresio erregular hori zenbat aldiz betetzen den kalkulatzen du, eta *parekatu ez diren elementuak* itzultzen ditu, bektore moduan. Adibidez:

```
my @esaldi=split(/\s+/,”hau esaldi bat da”);
   # @esaldi == (“hau”,”esaldi”,”bat”,”da”);
my @h=split(/\d+/, “123hitzak345eta2345zenbakiak”);
   # @h == (“hitzak”,”eta”,”zenbakiak”)
```

3.1.3.2 Hash mota

Hash bat (*elkartze-bektorea* ere deiturikoa) balio eskalarren bilduma da, eta elementu bakoitza indize baten bidez erreferentziatzen da. Bektoreak ez bezala, hash-ak ez du ordenarik. Hortaz, elementua bat erreferentziatzeko erabiltzen den indizeak ez du zertan zenbaki osoko positiboa izan behar[4]. Aitzitik, hash baten elementu orok *gako* bat du esleitua, gako hori edozein eskalar izan daitekeelarik.

Perl-ez, hash motako aldagai (edo objektu) bat erazagutzeko, besterik gabe, bere izena zehaztu behar da. Bestalde, aldagaiaren izenari hash dela adierazten duen ikurra (%) jarri behar zaio aurretik:

```
my (%Adinak)
```

Hash-ean elementuak txertatzeko, gako-balio bat gehitu behar zaio, ondoko eran:

```
$Adinak{“Aitor”} = 31;
$Adinak{“Kepa”} = 18;
```

Aurreko bi agindu hauekin, (“Aitor”=>31) eta (“Kepa”=>18) elementuak txertatu ditugu `%Adinak` hash-ean. Hash bat hainbat elementurekin hasieratzeko:

```
my %Adinak = (“Aitor” -> 31,
              “Eneko”->30,
              “Kepa” -> 18);
```

horrela,

```
my $a = $Adinak{“Aitor”}; # $a == 31
$Adinak{“Kepa”} = $Adinak{“Kepa”} + 1; # Zorionak, Kepa
```

Kontuan izan:

- Hash-en osagaiak, bektoreenak bezala, eskalarrak dira eta, hortaz, osagaia erreferentziatzerakoan erabili behar den ikurra eskalarrei dagokiena da ($). Adibidez:

  ```
  $Adinak{“Koldo”} = 32;
  my %Kat = (“etxe”->”Ize”,
             ”berandu”->”adb”);
  ```

- Indizea adierazpen bat ere izan daiteke:

  ```
  $Adinak{“Kol”.”do”} # $Adinak{“Koldo”}, 32
  ```

- Hash-en tamaina ez da finkoa. Elementu berri bat noiznahi gehitu ahal zaie:

  ```
  my %Kat= (“etxe” -> ”Ize”,
            ”berandu” -> ”Adb”);
  $Kat{“joan”}=”Adi”;
    # Orain, %Kat == (“etxe” -> ”Ize”,
    #                 ”berandu” -> ”Adb”,
    #                  “joan” -> “Adi”)
  ```

[4] Azken finean, bektoreen indizeek elementuen posizioa adierazten dute. Hash-a ordenatua ez dagoenez, ez du zentzurik euren elementuak posizioaren arabera erreferentziatzea.

ERAGIKETAK:

Hash-en artean, eragiketa hauek egin daitezke, besteak beste:

- Hash osoaren asignazioa:

```
my %Adinak = (“Aitor” -> 31,
              “Eneko” -> 30,
              “Kepa” -> 18);
my %h = (); # Hash hutsa
```

- Osagai bakoitzak bere motako aldagai bakun modura jokatzen du:

```
my $a = $Adinak{“Aitor”};  # $a-ri 31 asignatu
$Adinak{“Aitor”}=32; # %Adinak hash-ean,“Aitor” gakoari
                     # dagokion balioa aldatu
$Adinak{“Kepa”} = $Adinak{“Kepa} + 1;
     #“Kepa” gakoari dagokion elementua
    # batean handitu.
```

- Hash osoen arteko konparazioa (==, !=)

```
%Adinak == (“Aitor” -> 31,
            “Eneko”->30,
            “Kepa” -> 18); # TRUE

%Kat == (“etxe”->”Adi”,
         ”berandu”->”Adi”,
         “joan” -> “Ize”)# FALSE
```

- Hash baten elementu bat ezabatu dezakegu, **delete** funtzioaren bitartez:

```
delete($Kat{“etxe”}); #%Kat hashean dagoen
                      #(“etxe”,”Ize”) gako-balio
                     # bikotea ezabatu
```

- Hash batetik bektore bat lor dezakegu:

```
my %Kat = (“etxe”->”Ize”,
           ”goiz”->”Adb”,
           “joan” -> “Adi”);
my @Kat = %Kat;
 #@Kat==(“etxe”,”Ize”,”goiz”,”Adb”,”joan”,”Adi”)
```

- Eta alderantziz (bektore batetik, hash bat lortu):

```
my @bikoteak = (“bero”,”Adj”,”jan”,”Adi”);
my %bikoteak = @bikoteak;
  # %bikoteak == (“bero”=>”Adj”,”jan”=>”Adi”)
```

- Hash batetik, bere gako guztiak atera: **keys** funtzioa:

```
my @hitzak = keys(%Kat);
  # @hitzak == (“etxe”, “goiz”, “joan”)
```

- Hash batetik, bere balio guztiak atera: **values** funtzioa:

```
my @kategoriak = values(%Kat);
  # @kategoriak ==(“Ize”,”Adb”,”Adi”)
```

- Hash baten gako-balioak atera, banan-banan: **each** funtzioa. Hash batean zehar begiztaren bat egin nahi badugu, **each** funtzioa erabil dezakegu.

Hasiera batean, funtzio honek, hash-aren lehenengo gako-balioa emango digu. Berriro hash berean aplikatu ondoren, hurrengo gako-balioak emango dizkigu, banan-banan. Funtzio hau begiztetan[5] erabiltzeko da. Adibidez:

```
while(each($gako,$balio) = %keys(%Kat)) {
   print "$gako hitzaren kategoria $balio da\n";
}
# Honek zera aterako du pantailan:
# etxe hitzaren kategoria Ize da
# berandu hitzaren kategoria Adb da
# joan hitzaren kategoria Adi da
```

3.2 Adierazpenak

Balio berri bat kalkulatzeko formulak dira. Oro har, balio bat erabil daitekeen tokian adierazpen bat ere jar daiteke, adierazpena kalkulatuz lortzen den balioa adierazten duelarik. Adierazpen bat ebaluatzean, bertan dauden aldagaien ordez une horretan aldagaiek dituzten balioak erabiltzen dira kalkuluak egiteko. Adibidez, une batean $I aldagaiaren balioa `5` baldin bada, `2*$I+3` adierazpena ebaluatuz `13` lortuko da. Geroxeago `$I` aldagaiaren balioa `6` baldin bada, orduan `2*$I+3` adierazpen bera ebaluatuz `15` lortuko da.

Adierazpen berean zenbait eragiketa biltzen direnean, ondo zehaztu beharko da eragiketak burutzeko ordena. Ebaluatzeko ordena zein izango den jakiteko, eragiketen arteko lehentasunak ezagutu behar dira. Hala ere, beti izango da posible parentesiak erabiltzea, lehentasun esplizitua ezartzeko.

Adibidez, `$I+$J/$K` adierazpena honako adierazpen hauetako zeinen arabera ebaluatuko da: `$I+($J/$K)` erara? edo `($I+$J)/$K` erara? Maila berean azalduz gero, zatiketak batuketak baino lehentasun handiagoa duenez, `$I+($J/$K)` erara ebaluatuko da.

Lehentasun-maila bereko bi eragile biltzen direnean, ezkerraldetik hasiko da ebaluatzen. Adibidez, `$I/$J*$K` adierazpena `($I/$J)*K` erara ebaluatuko da. Ondoko taulan eragiketa arrunten arteko lehentasunak erakusten dira:

[5] Ikus 3.4.3 atala begiztak zer diren jakiteko.

Eragile diadikoak	Eragile monadikoak	Lehentasuna
**	abs !	handiena
* / %		
+ – &	+ –	
= != < <= >= > lt gt le ge eq ne		
& | ^		txikiena

3.3 Aginduak

Hiru dira datuak erabiltzeko oinarrizko aginduak[6]: balio baten **asignazioa** edo esleipena, datu baten **irakurketa** eta datu baten **idazketa**.

3.3.1 Asignazioa

```
$aldagaia = adierazpena;
```

Asignazioak adierazpena ebaluatuz lortzen den balioa ezartzen dio aldagaiari, balio berri modura. Aldagaiak galdu egiten du aurreko balioa. Adierazpenean aldagairik azaltzen bada, adierazpena ebaluatzean aldagaiak duen balioa erabiliko da, baina aldagaiaren balio hori ez da aldatzen ikusia izateagatik. Adibidea:

Hasierako egoera:

```
$N: 3  $P: 4  $X: 1.0  $Y: 4.5
```

Asignazioak

```
$M = ($N + $P) * 6;
$Z = $Y - $X;
$P = $P + 1;
```

Asignazioak egikaritu ondoko egoera hauxe izango da:

```
$N: 3  $P: 5  $X: 1.0  $Y: 4.5  $M: 42  $Z: 3.5
```

Asignazioa bera, adierazpena ere bada. Horrela, bada, asignazioak kateatu daitezke:

---

[6] Aginduei *ekintza* ere esaten diegu algoritmoez ari garenean.

```
$I = $N = 3;
```

Aurreko agindu hau, eskuinetik ezkerrera norabidean ebaluatuko da, hau da, eta parentesiak erabiliz:

```
$I = ($N = 3);
```

Beraz, lehendabizi parentesien artean dagoen agindua exekutatuko da, `$N` aldagaiak `3`ko balioa jasoko duelarik. Kontua da, baina, agindua adierazpena ere badela, eta agindu honen emaitza `3` balioa izango dela (esleitu berri den balioa, alegia). Hortaz, `$I` aldagaiak ere `3`ko balioa jasoko du.

3.3.2 Datuak irakurri

Programek egin behar duten urrats garrantzitsuenetarikoen artean, sarrera-fitxategi edo teklatu batetik datuak irakurtzea, eta pantailara edo irteera-fitxategietara datuak idaztea da. Perl lengoaia oso egokia dugu datuen sarrera/irteera delakoa burutzeko.

Has gaitezen adibide sinple batekin:

```
my $lerro; # $lerro aldagai eskalarra erazagutu
$lerro = <>; # STDIN, sarrera estandarretik lerro bat
             # irakurri
```

Iruzkinetan azaltzen den bezala, bigarren lerroaren bitartez, sarrera estandarretik lerro bat irakurriko dugu. Sarrera estandarra teklatutik datorrena izan ohi den arren, fitxategi baten edukia izatea lor dezakegu, berrelbideratzearen bitartez.

Edozein fitxategitik ere datuak irakurri/idatzi ahal dira. Horretarako, baina, **open** funtzioa erabili beharko da. Izan ere, fitxategi batekin lan egiteko, fitxategi bera *ireki* egin behar baita:

```
open FITX_DESKR, ESPR
```

fitxategi bat irekiko du, eta fitxategi deskribatzaile batekin erlazionatuko du. Fitxategi horren gainean burutu nahi diren ekintza guztiak, deskribatzailearen bitartez egin beharko dira.

open funtzioaren parametroak hauek dira:

FITX_DESKR Behin fitxategia irekita deskribatzaile bat behar da edozein operazio burutzeko.

ESPR Karaktere-kate bat, fitxategiaren izenarekin. Fitxategiaren izenaz gain, karaktere berezien bitartez adieraziko da fitxategia irakurtzeko, idazteko edo eransteko den.

Adib:

```
open (FI, “fitx.txt”);
```

`fitx.txt` fitxategia irakurtzeko ireki. Ezin izango da fitxategi horretan ezer idatzi.

Perl-etik, bestalde, komando baten irteera irakur dezakegu, fitxategia modu berezi batean irekitzen badugu, hots, “|” *pipe* delako karakterea bukaeran jarriz:

```
open (FI, “ps –x |”)
```

pipe karakterea “|” bukaeran dagoenez zera esan nahi du agindu honek: ‘ps -x’ komandoa exekutatu eta horren irteera FI fitx. deskribatzailearen bitartez irakurri.

Behin fitxategia irekia egon, datuak irakur ditzakegu, “<FITX_DESK>” eragile bereziaren bitartez. Ekintza honek:

```
$lerroa = <FI>;
```

`FI` deskribatzaileak adierazten duen fitxategitik lerro bat irakurriko du.

Perl hasieratzerakoan irakurtzeko deskribatzaile berezi bat definitzen da automatikoki: sarrera estandarra delakoa (`STDIN`). Horrela, bi ekintza hauek erabat baliokideak dira:

```
$lerro = <>;
$lerro = <STDIN>;
```

3.3.3 Datuak idatzi

Datuak idazteko, **print** funtzioa dugu. Adibidez, ekintza honek:

```
print “Kaixo mundua \n”;
```

“`Kaixo mundua`” karaktere-segida idatziko du pantailan, lerro-bukaera ikurrarekin batera (`\n` karaktere berezia).

Funtzio honek definizio hau du:

```
print [FITX_DESK] [ADIERAZPEN_ESKALARRA]
```

non

- `FITX_DESK` fitxategi-deskribatzaile bat den. Deskribatzailerik ipintzen ez bazaio, `STDOUT` irteera estandarrera idatziko du.
- `ADIERAZPEN_ESKALARRA` idatziko den balio bat da. Baliorik jartzen ez bada, `$_` aldagai eskalar berezia idatziko da.

Horrela, bada, fitxategi batean zerbait idatzi nahi badugu, fitxategi bera idazteko moduan ireki egin beharko dugu, lehen bezala, **open** funtzioaren bitartez. Kasu honetan, ordea, fitxategiaren izenari “>” ikur berezia jarri beharko diogu hasieran, fitxategia idazteko moduan irekitzeko dela adierazten duen ikurra. Adibidez:

```
open (FO, “>fitx.txt”);
```

`fitx.txt` fitxategia idazteko moduan ireki (lehendik izen horrekin fitxategiren bat balego ezabatu egingo litzateke)

```
open (FI, “>>fitx.txt”);
```

`fitx.txt` fitxategia eransteko moduan ireki.

Gure irteera beste programa baten sarrera izan daiteke. Horrelakoetan, fitxategia modu berezian ireki beha da, “|” *pipe* delako karakterea hasieran jarriz:

```
open (FI, “| sort |uniq“)
```

`FI` fitxategi deskribatzailera idatzitako guztia ‘`sort | uniq`’ komandoei pasa, *pipe* baten bidez. Kontuan eduki interpolazioa erabiliz aldagaiak erabil ditzakegula `open` funtzioaren `EXPR` espresioan, adibidez: `open (FI, “|sort | uniq > $fizena”);` non `$fizena` fitxategi baten izena den.

Behin fitxategia idazteko edo eransteko moduan irekia egon, bere deskribatzailearen bitartez atzituko da:

```
print FO “Kaixo Mundua\n”;
```

Perl hasieratzerakoan idazteko moduan irekita dauden 2 deskribatzaile berezi definitzen dira automatikoki: irteera estandarra delakoa (`STDOUT`) eta errore-irteera (`STDERR`).

Horrela, bi ekintza hauek erabat baliokideak dira:

```
print “Kaixo mundua\n”;
print STDOUT “Kaixo mundua\n”;
```

3.4 Kontrol-egiturak

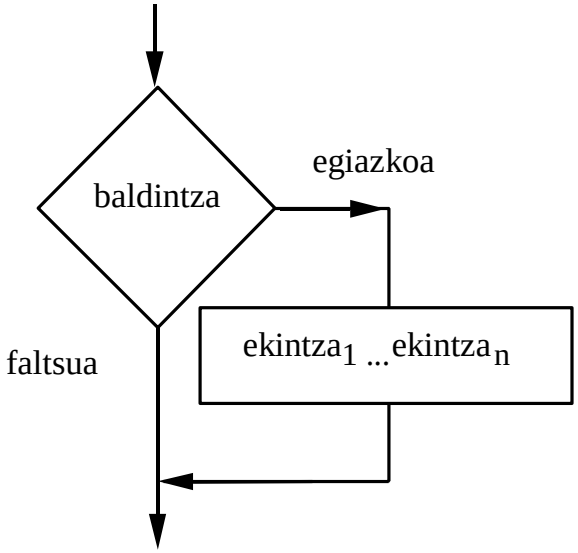
3.4.1 Agindu-sekuentzia

Algoritmoko agindu-sekuentzia pausoz pauso eta ordenan egikaritzen da. Ordena sekuentzial eta lineal hori aldatzeko bi aukera daude: baldintzazko egiturak edo iterazio-egiturak erabiltzea.

3.4.2 Baldintzazko egiturak

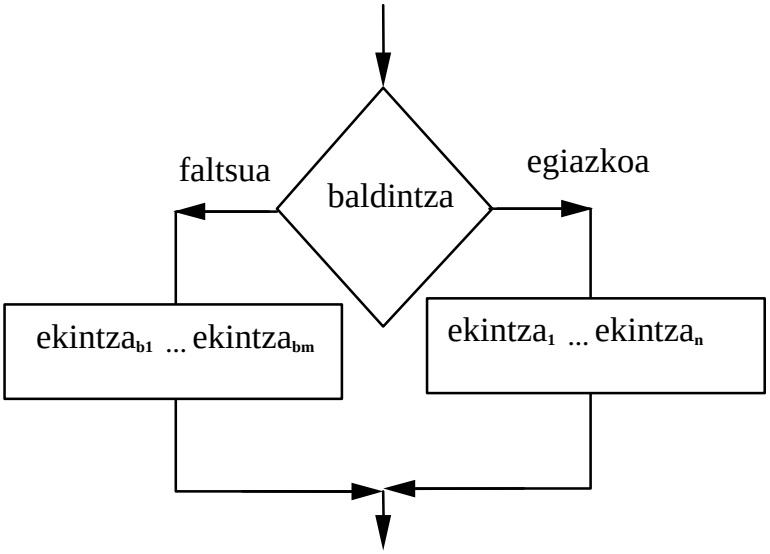
Agindu bat (edo gehiago) baldintza bat betetzen denean bakarrik egin behar bada, *baldin* izeneko kontrol-egitura erabili beharko da. Emaitza boolearra itzuliko duen adierazpena izan behar du baldintzak.

```
if (baldintza) {
     ekintza1
     ...
     ekintzan
}
```



Baldintza betetzen ez denean beste agindu bat (edo gehiago) egin behar bada, *bestela* motako atal bat ere sar daiteke:

```
if (baldintza) {
     ekintza1
     ...
     ekintzan
} else{
  ekintzab1... ekintzabm
}
```



Askotan, baldintza bat faltsua denean beste baldintza bat betetzen denentz egiaztatu behar da. Hau da, eskema hau askotan ematen da:

```
if (baldintza1) {
} else {
   if (baldintza2) {
 ...
   }
}
```

Horrelako kasuetarako, perl-en *elsif* delakoaren bitartez egin daiteke:

```
if (baldintza1) {
  ...
} elsif (baldintza2) {
  ...
} elsif (baldintza3) {
  ...
} else { # Aurreko baldintza guztiak bete ez badira
  ...
}
```

Perl lengoaian badago baldintzazko egiturak idazteko modu gehiago:

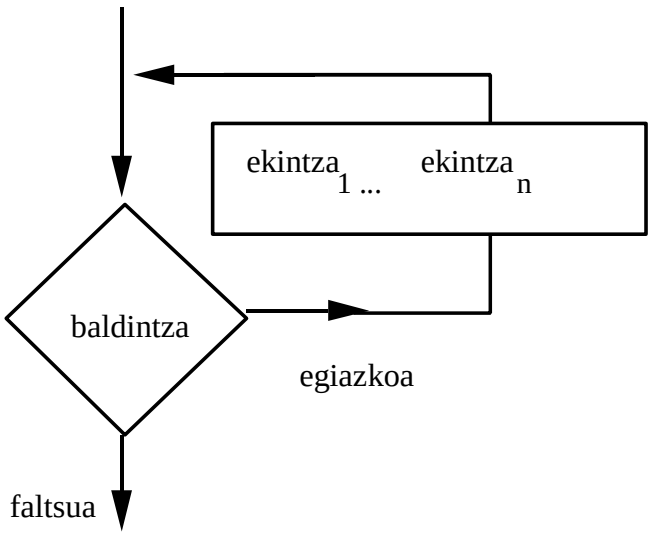
```
ekintza if (baldintza);
```

3.4.3 Iterazio-egiturak

3.4.3.1 Iterazio arrunta

Agindu bat (edo gehiago) errepikatu behar direla adierazteko, *while* izeneko kontrol-egitura erabiltzen dugu:

```
while (baldintza) {
    ekintza1;
    ekintza2;
    ...
    ekintzan;
}
```



3.4.3.1.1 Iterazio arrunta eta Sarrera/Irteera

Askotan, iterazioak fitxategi baten gainean egin behar ditugu, hau da, fitxategiko lerro bakoitzeko agindu-sekuentzia bat egikaritu nahi dugu. Horietan, Perl-ek automatikoki jakingo du fitxategia noiz bukatzen den. Adibidez, hurrengo iterazioak:

```
my $lerro;
while($lerro=<>) {
     ekintza1
     ...
     ekintzan
}
```

sarrera estandarretik (`STDIN`) lerro bat irakurri, `$lerro` aldagaian sartu, eta iterazio bat burutuko du. Hauxe bera edozein fitxategitarako ere balio du:

```
my $lerro;
open (FI, "fitx.txt");
while($lerro=<FI>) {
     ekintza1
     ...
     ekintzan
}
```

agindu hauek "`fitx.txt`" fitxategiaren lerro bakoitzeko iterazio bat burutuko du (eta, iterazio bakoitzean, $lerro aldagaian izango dugu fitxategiko lerroaren balioa). Fitxategiaren amaierara iristerakoan, `($lerro=<FI>)` agindua gezurrezkoa izango da, eta, hortaz, begizta bukatuko da.

Mota honetako iterazioetan, **chomp** eta **split** funtzioak erabili ohi dira. Suposa dezagun hitzei buruzko informazioa gordetzen duen fitxategi bat dugula, "hitzak.txt" izenarekin. Fitxategi horren lerro bakoitzean zera aurkituko dugu:

```
Hitza:Kategoria:adieraZbkia:Definizioa
```

hau da, fitxategi egituratua dugu, lerro bakoitzean ":" ikurraren bidez berezituta dauden hainbat eremu aurkituko ditugularik. Adibidez:

```
abereki:izond.:A1:Aberearen moduan jokatzen dena.
bakartegi:iz.:A2:Leku bakartia.
bakebide:iz.:A1:Baketzeko bidea.
```

Perl-ez, honako iterazio bat burutu beharko da:

```
use strict;
my $lerro;
open (FI, "hitzak.txt");
while ($lerro=<FI>) {
   #lerro bat irakurri dugu $lerro aldagaian
   chomp($lerro); # Lerroari lerro-bukaerako marka kendu
   my @lerro = split(/:/,$lerro);
   ...
}
```

Iterazio honek zera egiten du: lehendabizi, fitxategia irekitzen du. Gero, lerroz-lerroko iterazio batean sartzen da, une bakoitzean `$lerro` aldagaiak fitxategiaren lerro desberdin bat edukiko duelarik. Ondoren, `$lerro` aldagai horri lerro-karaktere berezia kentzen zaio. Azkenik, balio eskalarretik bektore batera bihurtzeko balio duen **split** funtzioaren bitartez, `@lerro` bektorea definitzen da. Bektore horren osagaiak, lerro bakoitzaren eremuekin bat datoz, hau da:

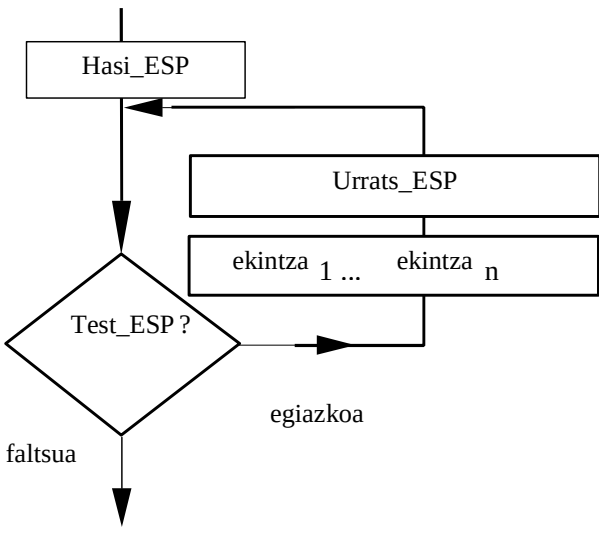
```
$lerro[0] -> hitza
$lerro[1] -> kategoria
$lerro[2] -> Adiera zenbakia
$lerro[3] -> Definizioa
```

puntu honetan, lerro bakoitzaren eremu guztiak bektore batean ditugu gordeta, eta horien gainean edozein eragiketa egin dezakegu. Adibidez, kategoria bakoitza zenbat aldiz agertzen den zenbatu dezakegu. Egin ezazue adibide hori !

3.4.3.2 Aldi-kopuru jakineko iterazioa

Iteratzen hasi baino lehenago aginduak zenbat aldiz errepikatu behar diren jakiterik dagoenean, egokiago izaten da ***guztietarako*** motako kontrol-egitura erabiltzea. Kontrol-egitura honetan, hiru eremu definitu behar dira: kontrol-aldagaiak hartuko duen hasierako balioa (`Hasi_ESP`), kontrol-aldagaia bukaeraraino iritsi denentz jakiteko balio duen espresioa (`Test_ESP`), eta urrats bakoitzean kontrol-aldagaia eguneratzeko behar den espresioa (`Urrats_ESP`).

```
for (Hasi_ESP; Test_ESP; Urrats_ESP) {
    ekintza1
    ...
    ekintzan
}
```



`Hasi_ESP` espresioa behin bakarrik kalkulatzen da, iterazioa hasi aurretik. Gero, `Test_ESP` espresioa konprobatzen da, eta, espresioaren emaitza *egiazkoa* ez bada, iterazioa bukatu egiten da. Bestela, gorputzeko ekintzak exekutatzen dira. Azkenik kontrol-egitura eguneratzen da, `Urrats_ESP` espresioa exekutatuz.

Adibidea:

```
algoritmo Kuboak_1_30
hasiera
    for ($I=1;$I<=30; $I=$I+1) {
        print $I**3;
        print " ";
    }
    print "\n";
amaia
```

Emaitza:

1 8 27 64 125 216 … 27000

Ondoko algoritmoak ere emaitza berdinak ematen ditu baina *bitartean* izeneko kontrol-egitura erabiliz definitu da:

```
algoritmo Kuboak_1_30
hasiera
    $I = 1
    while ($I <=30) {
        print $I**3;
        print " ";
        $I = $I + 1
    }
    print "\n";
amaia
```

Iterazioak bektore baten gainean ere egin daitezke. Kasu honetan, bektorearen balio bakoitzeko iterazio bat egingo da, eta kontrol-aldagaiak elementu horren balioa hartuko du:

```
foreach aldagai_eskalarra (BEKTORE_ESPRESIOA) {
    ekintza1
    ...
    ekintzan
}
```

Adibidez

```
my @frutak = ("sagarra", "udarea", "platanoa");
foreach my $fruta (@frutak) {
        print "$fruta oso goxoa dago\n";
}
```

while izeneko kontrol-egitura erabiliz, kode hau aurrekoaren baliokidea da:

```
my @frutak = (“sagarra”, “udarea”, “platanoa”);
my $i=0;
while($i < scalar(@frutak)) {
     my $fruta=$frutak[$i];
     print “$fruta oso goxoa dago\n”;
     $i=$i+1;
}
```

3.5 Azpiprogramak

Algoritmo zailak azpiproblema sinpleagotan banatu eta azpiproblema bakoitza bere aldetik ebazteko laguntza eskaintzen dute **azpiprogramek**. Azpiprogramak bi eginkizun nagusi betetzeko erabil daitezke:

Funtzioak kalkulatu

- Emaitza bat kalkulatzeko balio du.
- Balio modura erabiltzen da.
- Emaitza hau adierazpen baten barruan erabili behar da derrigorrez. Adibidez, `Pred` azpiprograma zenbaki oso baten aurrekoa itzultzeko definitu bada, `&Pred(8)*2` adierazpenak 14 balioa itzuliko du.

Prozedura

- Aldagai baten balioa aldatu edo sarrera-irteerako agindu bat egikaritzeko balio du.
- Agindu modura erabiltzen da, beste algoritmo edo modulu baten barruan. Adibidez, `Idatzi_Osokoa` prozedura osoko bat idazteko prozedura bada, `&Idatzi_Osokoa($N–4)` agindu posible bat da algoritmo batean.

Perl-ek, azpiprogramen izena izen berezietatik bereizteko, `&` ikurra erabiltzen du. Horrela, bada, *Idatzi_Osokoa* izeneko azpiprograma deitzeko, `&Idatzi_Osokoa` jarri beharko dugu.

Azpiprograma batek parametroak eduki ditzake. Parametroei esker azpiprogramaren definizioa orokorragoa izango da, eta eragin berdina lortu ahal izango da datu edo aldagai desberdinekin. Parametroen bitartez datuak pasatzen zaizkio azpiprogramari eta era berean programa nagusiak emaitzak jasoko ditu.

Azpiprogramak algoritmoak bezala definitzen dira agindu-multzo batekin; eta multzo honi azpiprogramaren *gorputza* deritzo. Gorputz hau azpiprogramari deitzen zaionean egikaritzen da; ondoren algoritmoaren exekuzioak deiaren ondoko puntutik jarraituko du. Azpiprograma barruko aldagaiak eta konstanteak zeintzuk diren hobeto zehaztearren, komeni da agindu-sekuentzia baino lehenago aipatzea, hots, aldagai lokalak erazagutzea. Horrela, objektu horiek azpiprograma horretatik kanpo ezin erabil daitezkeela adierazi nahi dugu.

Programazio-lengoaiek oso erabilgarriak diren azpiprograma edo funtzio estandarrak eskainiko dituzte. Programategi bakoitzak programatzaile-talde berezientzat erabilgarriak diren azpiprogramak izango ditu. Adibidez, programatzaile 'zientifikoek' sinuak, kosinuak, erro karratuak eta horrelakoak kalkulatzeko gai diren funtzio matematikoak beharko dituzte. Horrela, programategiak esfortzu-bikoizketa ekiditen du eta adituek eginiko algoritmo sofistikatuak erabilgarri izango dira esperientziarik ez duten programatzaileentzako ere.

3.5.1 Funtzioak

Imajina ezazu karaktere bat hartu eta letra bat den ala ez aztertzen duen *Alfabetikoa* izeneko funtzio bat. `&Alfabetikoa('K')` eta `&Alfabetikoa($Iniziala)` funtzio--deien adibideak dira (`$Iniziala` delakoa balio gisa karaktere bat duen aldagaia izanik). Parentesi artean dagoen adierazpenari *parametro erreal* deitzen zaio, eta funtzioak parametro erreal horrekin egingo du kalkulua. Funtzio-dei hauetan parametroa karaktere motakoa da, eta emaitza boolear motakoa.

Adibidez, `ZKH` funtzioak bi osoko positiboen zatitzaile komunetatik handiena kalkulatzen badu, `$Y = 1+ &ZKH($N+1, 72)` asignazioan erabili den modura erabil daiteke funtzio-deietan. Funtzio honen bi parametroak eta bere emaitza *Osoko* motakoak izan beharko dira.

Hemen dituzu, aurrez aipaturiko `Alfabetikoa` eta `ZKH` funtzioen erazagupenak:

```
sub Alfabetikoa {
     my ($kar) = @_;
     ...
}

sub ZKH {
     my ($Zenb1, $Zenb2)=@_;
     ...
}
```

Funtzioaren izena *sub* hitz erreserbatuaren jarraian dago. Bestalde, funtzioa deitzeko garaian pasatako parametroak, `@_` bektore berezian ipintzen dira. Hortaz, funtzioen lehenengo agindua parametroak izango diren aldagaiak erazagutzea da, `@_` bektoreko balioak hartuz.

`Alfabetikoa` izeneko funtzioak, `$Kar` izeneko parametro bakarra du. `$Kar` izenari esker, posible izango zaigu funtzio-gorputzaren barrutik parametroa erreferentziatzea, deietan erabiliko den parametro erreala aipatu gabe. Horregatik `$Kar` delakoari funtzioaren *parametro formala* deitzen zaio.

`ZKH` funtzioak bi parametro formal ditu, `Zenb1` eta `Zenb2`. Hala ere, funtzio-erazagupen batek ez du zehazten funtzioaren emaitza nola kalkulatzen den. Horregatik, funtzioaren gorputza ere idatzi behar dugu, honek baititu parametroen balioetatik abiatuz funtzioaren emaitza kalkulatuko duten aginduak.

Ondoko hau `Alfabetikoa` deritzon funtzioaren definizio posible bat duzu:

```
sub Alfabetikoa {
     my ($Kar) = @_;
     if ( (($Kar >= 'A') && ($Kar <= 'Z'))  ||
          (($Kar >= 'a') && ($Kar <= 'z'))) {
         return 1; # Egiazkoa
     } else {
        return 0; # Faltsua
     }
}
```

Edo baliokidea eta laburragoa den beste gorputz batekin:

```
funtzio Alfabetikoa {
     my ($Kar) =@_;

     return ((($Kar >= 'A') && ($Kar <= 'Z')) ||
             (($Kar >= 'a') && ($Kar <= 'z')));
}
```

Funtzio-gorputz batean *return* izeneko aginduak bi xederi erantzuteko balio du:

(a) funtzio-gorputzaren egikaritzapenari bukaera ematen dio; eta

(b) return hitz erreserbatuaren ondoren datorren adierazpena ebaluatzen du, eta hori izango da funtzioaren emaitza.

Jo dezagun, adibidez, `&Alfabetikoa($Iniziala)` delako funtzio-deia ebaluatzen ari garela. Lehenik, `$Kar` parametro formalari dagokion parametro errealaren balioa, `$Iniziala`-ren balioa, ematen zaio. Suposa dezagun bere balioa '?' dela. Orduan giltzen arteko aginduak egikaritzen dira. Hain zuzen, *return* aginduaren adierazpena

0 da (hau da, faltsua), eta beraz, funtzio-deiak 0 emaitza itzuliko du. Parametro errealaren balioa 'K' izango balitz, emaitza 1 izango litzateke (egiazkoa).

Orokorki esanda, funtzio-gorputz batek agindu asko eduki ditzake eta, agian, parametroez gain, beste barne-erazagupen batzuk ere bai.

ZKH funtzioa era askotara inplementa daiteke. Hemen duzu aukeretako bat:

```
sub ZKH {
     my ($Zenb1, $Zenb2) = @_; # Parametro formalak
     my ($M, $N, $R);
                 # Hiru aldagai eskalarren erazagupena
     $M = $Zenb1;
     $N = $Zenb2;
     $R = $M % $N; # % -> modulua
     while ($R != 0) { # jar daiteke ere while ($R) {
         $M = $N;
         $N = $R;
         $R = $M % $N;
     }
     return $N;
}
```

Funtzio-gorputza $M, $N eta $R aldagaiak aurkezten dituen erazagupenarekin hasten da (parametro formalen erazagupena egin ondoren), funtzio-gorputzaren barnean aldagai moduan erabiliko direnak dira horiek. Funtzioaren emaitza, algoritmo euklidearra inplementatzeko $M, $N eta $R erabiltzen dituen begizta baten bidez kalkulatzen da. Halako batean, *return* sententzia egikarituko da, $M, $N-ren multiplo zehatza denean, funtzioaren emaitza gisa $N-ren azken balioa itzuliz.

Funtzio-gorputz batean *return* agindu bat edo gehiago egon daitezke. Horrelakoetan ere, *return* agindu bakoitzak funtzioaren emaitzaren motako adierazpen bat izan behar du. Beharrezkoa da *return* agindu hauetako bat momenturen batean egikaritzea. Hau gertatzen denean, *return* aginduaren ondoren dagoen adierazpena ebaluatzen da funtzioaren emaitza zehazteko, eta funtzio-gorputzaren egikaritzapena bukatu egiten da.

Zenbait moduluk, bestalde, ez dute inongo baliorik itzuliko, eta soilik betebehar jakin bat burutuko dute. Horietan, ez da *return* agindurik egongo edo, baldin badago, soilik funtzioa amaitzeko balioko dute, hau da, ez dute baliorik itzuliko. Funtzio hauei, formalki, *prozedurak* esaten zaie, baina Perl-ek ez du prozedura eta funtzioen artean desberdintzeko inongo sintaxi berezirik ezartzen. Moduluaren semantikak esango digu funtzioa edo prozedura den.

Hemen duzu zuriune-kopuru jakin bat idatzi behar duen prozedura baten definizioa:

```
sub Zuriuneak_Idatzi {
     my ($Zuriune_Kop) = @_;
     my ($Kont); # Kontrol-aldagaia
     for ($Kont=0; $Kont <$Zuriune_Kop; $Kont++) {
         print ' ';
     }
}
```

Funtzio-gorputz bateko *return* aginduak funtzioaren emaitza zehaztu eta funtzio--gorputzaren egikaritzapena bukatzeko balio du. Prozedura-gorputz batean lehenengo xedea ez da beharrezkoa; beraz, ez da egongo jarraian adierazpenik duen *return* agindurik. Gainera, bukaera inplizitu bat dago prozedura-gorputza bakoitzaren azkenean.

Azpiprograma bat exekutatu ondoren, azpiprograma barruan erazagututako parametro eta objektu formal guztiak desagertzen dira (horrela, hauek betetzen zuten memoria libre uzten da beste xede batzuetarako).

3.6 Moduluak

Azpiprogramak erabiliz programazio-lanak erraztu egiten dira. Izan ere, zenbait ataza espezifikoren kodea funtzio edo prozedura[7] desberdinetan “estali” daiteke, arestian aipatu dugun legez. Funtzio eta prozedurak erabiltzeak bi abantaila nagusi ditu programazio-lanetan diharduguneanean, eta, bereziki, programa handi eta konplexuak egin nahi dituguneanean:

- Errepikatzen den zenbait kode-lerro azpiprograma baten azpian koka ditzakegu. Horrela, bada, kode-lerro horiek ez ditugu errepikatu behar izango programa nagusian zehar. Honek hasiera batean dirudiena baino garrantzi handiagoa du, batez ere programak mantentzerakoan: kode-lerro horiek aldatu nahi izanez gero, toki bakar batean soilik egin behar izango dugu aldaketa.
- Azpiprogramak erabiltzeak ideiak egituratzen lagunduko digu, hau da, handi eta konplexu izan daitezkeen programak zati txikiagoetan banatzen lagunduko digute, horrelako zati bakoitzaren kodeak azpiprograma batean joan beharko lukeelarik.

[7] Ikusi dugu, baita ere, perl lengoaiak ez duela funtzio eta prozeduren arteko bereizketarik egiten.

Dakigunez, azpiprograma bat idazterakoan bere sarrera- zein irteera-parametroak zeintzuk diren zehaztu beharko dira. Esate baterako, bi zenbaki osoren zatitzaile komunetatik handiena ematen digun `&ZKH` funtzioak bi sarrera-parametro behar ditu eta, emaitza gisa, beste zenbaki osoko bat itzuliko digu. Demagun `&ZKH` funtzio hori programa konplexu baten zatia dela, eta programa konplexu hori lan-talde baten emaitza dela. Demagun, bada, `&ZKH` funtzioa lagun batek idatzi duela, eta beste batek erabiliko duela, programa nagusian. Testuinguru honetan, enkapsulazio deituriko kontzeptu arras garrantzitsua azalduko zaigu: funtzio horri deitzen dionak, funtzioak behar dituen eta itzultzen duen parametroei buruzko zehaztapena behar du soilik, *eta besterik ez*. Bereziki, ez du zertan jakin behar funtzio horrek emaitza hori itzultzeko zein aldagai erabili dituen, zein algoritmo egikaritu duen, edota beste azpiprogramaren bat erabili duenentz.

Free On-line Dictionary of Computing hiztegian, “encapsulate” hitzerako definizio hau irakur daiteke:

> The ability to provide users with a well-defined interface to a set of functions in a way which hides their internal workings. In object-oriented programming, the technique of keeping together data structures and the methods (procedures) which act on them.

Azpiprogramen bitartez enkapsulazioa burutu daiteke eta aukera hori ezinbesteko laguntza da programazio-lana ganoraz antolatu nahi bada. Hala ere, programa aurreratuak egin ahal izateko azpiprograma ugari erabili beharko ditugunez, azpiprograma horiek guztiak modu antolatuan bildu eta erabili ahal izateko beste tresna-mota bat behar izaten dugu: moduluak. Moduluak zeregin berezi bat burutzen duten prozeduren eta datu-egituren multzoak dira, edozein programatzailek erabili ahal izateko diseinatuak. Izan ere, perl-eko edozein instalazio hainbat modulurekin batera etorriko da, sistema kudeatzeko edo beste zeregin askotarako baliagarriak direnak.

Modulu bat erabiltzeko, lehendabizi gure programan modulu hori erabili nahi dugula adierazi behar dugu. Ondoren, modulu horren objektu bat (edo gehiago) sortu behar da eta, azkenik, objektu horri zenbait eginkizun betetzeko eskatu behar zaio, *metodoen* bidez. Adibide bat ikustearren, badago mezu elektronikoak zuzenean bidaltzeko perl-ekin datorren modulu estandar bat, `Net::SMTP` delakoa. Ikus dezagun, modulu hori erabiliz, mezu bat bidaltzen duen programatxo txiki bat:

```
#!/usr/bin/perl

use strict;
use Net::SMTP;  # Net::SMTP modulua erabili ahal izateko

my $smtp;

$smtp = Net::SMTP->new('localhost'); # Objektu berri bat
                                     # sortu

$smtp->mail(‘ccpsoeta@si.ehu.es’); # mail berri bat.
                                   # Igorlea:ccpsoeta
$smtp->to('siahitek@localhost');   # Jasotzeailea:siahitek

$smtp->data();  # Orain mezuaren gorputza dator.
$smtp->datasend("To: siahitek\n");
$smtp->datasend("\n");
$smtp->datasend("Mezu hau perl-en bidez idatzi dizut!\n");
$smtp->dataend(); # Bukatu dugu gorputzarekin

$smtp->quit; # Azkenik, konexioa itxi
```

Programa honek baditu orain arte ikusi ez ditugun gauza franko. Lehenengoa honako lerro hau da:

```
use Net::SMTP;  # Net::SMTP modulua erabili ahal izateko
```

Lerro horren bitartez, perl programari modulu berezi bat erabiliko dugula adierazten diogu, kasu honetan, moduluaren izena `Net::SMTP` delarik. Modulua erabiltzeko, baina, objektu bat erazagutu beharko dugu, ondoko eran:

```
$smtp = Net::SMTP->new('localhost'); # Objektu berri bat sortu
```

Horren bidez, `Net::SMTP`-ko moduluko objektu berri bat definitu dugu, **new** *metodoaren* bitartez. Bere sintaxi orokorra hauxe da:

```
$aldagaiEskalarra = ModuluarenIzena->new([opzioak]);
```

Kasu honetan, `Net::SMTP` objektu bat sortuko dugu, hots, mezu elektronikoen protokoloa (Simple Mail Transfer Protocol deiturikoa) erabiltzeko aukera ematen duen objektu bat. Parametro bakar batekin sortu dugu: mezu elektronikoa bidaliko duen makinaren izena, “localhost”[8] alegia. Momentu horretan, `$smtp` objektua “localhost” izeneko mail zerbitzariarekin konektatzen da. Gero, mezu berri bat idazteko eskatuko diogu:

```
$smtp->mail('ccpsoeta@si.ehu.es');
```

mail metodoaren bitartez, mezu berri bat idatzi nahi dugula adierazten dugu. Parametro bakarra du, mezuaren igorlea adierazten duelarik, maila iristerakoan "From:" eremuan agertuko dena, alegia. Ondoren, mezua norentzat izango den adieraziko diogu, **to** metodoaren bitartez:

```
$smtp->to('siahitek@localhost');
```

Mezua, beraz, *siahitek@localhost* kontura bidaliko dugu. Behin parametro hauek guztiak ezarrita egonda, mezuaren edukia zein izango den adieraziko dugu, instrukzio hauen bitartez:

```
$smtp->data();
$smtp->datasend("To: siahitek\n");
$smtp->datasend("\n");
$smtp->datasend("Mezu hau perl-en bidez idatzi dizut!\n");
$smtp->dataend();
```

Ikusten den bezala, **data** metodoaren bitartez mezuaren gorputza izango dena ezartzen hasiko garela adierazten dugu. Gero, **datasend** metodoari hainbatetan deituko diogu, behin mezuaren lerro bakoitzeko. Azkenik, mezuaren gorputzarekin bukatu dugula adierazten dugu, **dataend** metodoaren bidez.

Bukatzeko, konexioa ixteko eskatuko dugu:

```
$smtp->quit; # Azkenik, konexioa itxi
```

Adibidetxo horrekin moduluen izaera ikus daiteke. Modulu batek objektuak sortzeko ahalmena du, eta objektu horiei eskatu ahal izango zaie zenbait eginkizun burutzeko, metodoen bidez. `Net::SMTP` moduluan mezu elektronikoak bidaltzeko hainbat azpiprograma bildu dira, denak multzo integratu eta oso gisa aurkezten direlarik. Jakina, modulu bat erabili ahal izateko, **objektuek onartzen dituzten metodoen zerrenda ezagutu behar dugu**, nahitaez. Modulu batek onartzen dituen metodo desberdinen erazagupenari, **moduluaren interfazea** deituko diogu. Bestalde, modulu horrek bere eginkizunak betetzeko erabiltzen dituen aldagaiak, prozedurak eta algoritmo multzoei **moduluaren inplementazioa** deituko diogu; inplementazioa ezkutuan egoten da moduluaren erabiltzailearentzat. Ohar zaitezte programatzeko era

---

[8] Unix sistema eragilearen arloan, *localhost* hitza lan egiten ari garen makinaren ezizena da.

honen abantaila apartak: edozein arrazoi dela medio, modulu batean bere atazak burutzeko era (inplementazioa) aldatzen bada ere, moduluaren erabiltzailea ez da ezertaz arduratu behar izango, inplementazioaren aldaketak erabat gardenak baitira berarentzat.

3.6.1 Modulu estandarrak.

Esan bezala, perl-eko distribuzio estandarrekin hainbat eta hainbat modulu datoz. Zoritxarrez, ez dago modulu horiek zeintzuk diren jakiteko metodo orokorrik, batez ere, plataforma desberdinetan (Windows, Unix/Linux, Mac, ...) modulu-sorta desberdinak instalatzen direlako. Unix/Linux sistema batean instalatutako moduluen zerrenda nahi izanez gero, hau exekutatu beharko da komando-lerroan:

```
find `perl -e 'print "@INC"'` -name '*.pm' –print
```

Modulu jakin baten informazioa nahi izanez gero, berriz, perldoc komandoa exekutatu beharko da. Adibidez, lehen aipatutako `Net::SMTP` moduluko metodoen informazioa lortzeko, hau exekutatu behar da:

```
perldoc Net::SMTP
```

Moduluaren dokumentazioa irakurtzea ezinbestekoa da modulu hori erabili nahi bada, dokumentazio horretan agertuko baita moduluak onartzen dituen metodoen zerrenda, hots, moduluaren interfazea, eta metodo bakoitza nola erabili behar den.

3.6.2 SILemati modulua

Ikastaro honetan zehar, lematizatutako testuekin lan egingo dugu. Lematizatzaile batek, testuan agertutako hitz bakoitzeko, bere lema eta kategoria sintaktikoa emango dizkigu, besteak beste. Lematizatzaile honen irteerari begiratu azkar bat botako diogu. Adibidez, sarrerako esaldia “Gaur lau (4) ordu egin ditut lo.” Bada, hau izango da lematizazioaren emaitza:

```
/<Gaur>/<HAS_MAI>/
     ("gaur"  ADB ADOARR  @ADLG)
/<lau>/
     ("lau"  ADJ IZO DEK ABS MG  @OBJ @SUBJ @PRED)
/<(>/<BEREIZ>/
/<4>/<ZEN>/
     ("4"  DET DZH NUMP DEK ABS MG  @OBJ @SUBJ @PRED)
/<)>/<BEREIZ>/
/<ordu>/
     ("ordu"  IZE ARR DEK ABS MG  @OBJ @SUBJ @PRED)
/<egin>/
```

```
     ("egin"  ADI SIN AMM PART ASP BURU  @-JADNAG)
/<ditut>/
     ("*edun"  ADL A1 NR_HK NK_NI  @+JADLAG)
/<lo>/
     ("lo"  ADB ADOARR  @ADLG)
     ("lo"  IZE ARR)
/<.>/<PUNT_PUNT>/
```

Ikus daitekeenez, irteerak orain arte ikusi duguna baino egitura aski konplexuagoa du, batez ere hitz batzuek interpretazio bat baino gehiago eduki ditzaketela kontuan hartzen badugu. Lematizazioaren gainean edozein lan egin nahi izanez gero, hitzak eta bere interpretazioak nolabait irakurri beharko ditugu, banan-banan. Guk, horretarako, bereziki idatzitako modulu sinple bat erabiliko dugu, **SILemati** deiturikoa. Atal honetan, modulu hau erabili ahal izateko bete beharreko urratsak azalduko ditugu.

Lehenik eta behin, eta perl-ek modulua aurkitu ahal izateko, gu gauden azpidirektorio berean “SILemati.pm” fitxategiak ere egon behar du. Modulua erabiltzeko, hau jarri beharko dugu gure programaren hasieran:

```
require SILemati;
```

Gero, objektu berri bat sortu beharko dugu:

```
my $si = new SILemati(lematizazioaren_irteeraren_izena);
```

Aurreko agindu honen bidez, gure programak lematizatzailearen irteerarekin lan egin ahal du. Hitzak eta interpretazioak irakurtzen hasteko, berriz, gure programak begizta nagusi bat beharko du, iterazio bakoitzean fitxategiko hitz desberdin bat irakurriko duelarik. Begiztaren bizkarrezurra honako hau da:

```
while ($si->hurrengo_sarrera()) {
   my @HitzInterp = $si->sarrera_ekartzan();
   ...
}
```

Begizta honen barruan, `@HitzInterp` bektoreak hitz bakoitza gordeko du, bere interpretazioekin batera. Moduluari aurrean ikusitako fitxategia irakurtzeko eskatzen badiogu, `@HitzInterp` bektoreak bi osagai hauek edukiko ditu begiztaren lehenengo iterazioan:

```
$HitzInterp[0] == “/<Gaur>/<HAS_MAI>/”
$HitzInterp[1] == “  ("gaur"  ADB ADOARR  @ADLG)”
```

Hona hemen, SILemati moduluak onartutako metodoen zerrenda:

`SILemati->new(fitxategiaren_izena)`

Fitxategia ireki eta fitxategiaren goiburukoa[9] irakurri. Metodo honek balio eskalar bat itzuliko du. Hortik aurrera, operazio guztiak itzulitako objektu honen bitartez egin beharko dira.

`$obj->goiburukoa_ekartzan()`

Fitxategiaren goiburukoa itzuli, bektore batean. Beraz, bektore bat itzuliko du.

`$obj->hurrengo_sarrera()`

Fitxategiko hurrengo sarrera irakurri. Egiazko balioa itzuliko du sarrera berri bat irakurri baldin badu. Fitxategi-bukaerara iritsi bagara, berriz, Faltsua itzuliko du. Beraz, eskalar bat itzuliko du.

`$obj->sarrera_ekartzan()`

Irakurri berri den sarrera bektore batean itzuli. Beraz, bektore bat itzuliko du.

`$obj->hitza_ekartzan()`

Irakurritako azkenengo hitza emango digu. Beraz, eskalar bat itzuliko du. Aurreko adibidearekin jarraituz:

```
$si->hitza_ekartzan() == “Gaur”
$obj->id_ekartzan()
```

Irakurritako azkenengo hitzaren identifikatzailea emango digu (puntuazio-ikurra izan bada, maiuskulaz hasi bada, etab.). Beraz, eskalar bat itzuliko du. Aurreko adibidean:

```
$si->id_ekartzan() == “HAS_MAI”
$obj->lerro_zenbakia_ekartzan()
```

Irakurritako azkenengo hitzak fitxategian duen lerro-zenbakia emango digu. Beraz, eskalar bat itzuliko du.

[9] Fitxategiaren goiburukoa, lehenengo hitza agertu baino lehen dagoen guztia da.

4 Programazio-estiloa

Programak irakurterrazak izan daitezen, zenbait estilo-erregela jarraitzea komeni da :

- Erabil itzazu iruzkinak lasai, programaren funtzioa adierazi eta zati bakoitzaren funtzionamendua azaltzeko. Iruzkinak "#" karakterearen ondoan idatziko ditugu. Lerro bukaeraraino dauden gainontzeko karaktereak, ohar gisa hartuko dira.
- Aukera itzazu ahalik eta identifikadore deskriptiboenak.
- Erabili maiuskulak edo minuskulak identifikadoreetan, baina beti era berean! Liburu honetan minuskulaz idatzi ditugu baina lehenengo letra beti maiuskulaz. Objektuen identifikadoreak izenak izaten dira; prozedurenak, aldiz, aditzak. Kontuan izan Perl-ek maiuskula eta minuskulen artean bereizten duela. Horrela, bada, $ald, $Ald, $aLd edo $ALD identifikadoreak denak desberdinak dira.
- Ez idatzi algoritmo luzeegiak. Saia zaitez problema nagusia azpiproblema errazagoetan banatzen eta, geroago, hauek aparte definitzen.

Programatzaile askok sententziak bikoiztea ekiditeko tresna erabilgarri huts gisa ikusten dituzte moduluak. Ikuspegi honek, ordea, moduluak erabiliz lortzen diren onurak gutxiesten ditu. Hemen duzu horien laburpen bat:

- Modulu bat, *nola* funtzionatzen duen jakin gabe izan daiteke deitua; moduluaren erabiltzaileak moduluak *zer* egiten duen bakarrik jakin behar du. Modulu bat zertan erabiliko den jakin gabe irakur eta uler daiteke. Gai hauek bereizteari *abstrakzio* deritzo eta tresna intelektual ahaltsua da problema konplexuak ebazteko.
- Azpiprogramek zati askeez osatutako programa bat eraikitzeko balio dute, non modulu bakoitzak xede bakun bat izango duen. Horrela programa luze baten eraikuntza asko errazten da modulu bakoitza bereizita idatz eta azter baitaiteke.
- Azpiprogramek izenak ematen dizkiete programen zatiei. Azpiprogramentzako (eta objektu eta datu-motak moduko hainbat

entitateentzako) izen zentzudunak aukeratuz gero, programa "testu" gisa irakur daiteke. Moduluarentzat izen egokia aukeratzea errazagoa da moduluak xede bakun eta sinpleren bat baldin badu.

- Modulu bati toki desberdin askotatik dei dakioke, nahi izanez gero parametro erreal desberdinekin. Honek programa-testuaren bikoizte aspergarria (eta errore-sortzailea) ekiditen du.
- Erraza da modulu-gorputzaren bertsio bat beste batekin ordezkatzea. Moduluaren *erazagupena* aldatzen ez den bitartean, programaren gainerako zatiek ere ez dute aldatu beharrik.

5 Perl komando-lerroan

Askotan, egin nahi ditugun zereginak nahiko sinpleak dira. Perl-ek badu “Fitxategi honetako lerro guztiei hasierako espazioak kendu”, “fitxategi honetan hitz jakin bat agertzen diren lerroak soilik inprimatu” etab. bezalako atazak burutzeko era trinko bat, *switch-ak*[10] erabiliz. Metodo honen bidez, hainbat programa lerro bakar batean idatz ditzakegu.

Perl-ek *switch* sorta handia onartzen badu ere[11], guk bakarrik opzio pare batean jarriko dugu arreta. Konparazio batera, honako programa honek:

```
% perl –e ‘print “Kaixo mundua\n”;’
```

“Kaixo mundua” idatziko du pantailan. *Switch*-ak perl interpretatzailea idatzi ondoren jarri behar dira, - ikurraren atzetik. Adibide honetan, (-e) *switch*-a erabili dugu. Opzio honen bitartez, perl interpretatzaileak, fitxategi batean idatzita dagoen programa bat egikaritu ordez, opzioaren ondoren datorren karaktere-segida exekutatuko du. Beste honek, berriz:

```
% perl –e ‘print (64*5)/2;’
```

160 zenbakia idatziko du pantailan. (-e) opzioaren bitartez idatzitako programetan aldagaiak ere erabil daitezke, aldagai hauek aurretik erazagutu behar ez direlarik. Horrela, bada,

```
% perl –e ‘$ema = (64*5)/2; print “Emaitza $ema da\n”;’
```

programak “Emaitza 160 da” idatziko luke pantailan.

Edonola ere, programa hauek nekez izango dira erabilgarriak fitxategiekin lan egin ezin badute. Izan ere, gure interesa askotan fitxategiekin lan egitea baita, hau da, fitxategi batetik lerroak irakurri, lerro horietan zenbait operazio burutu, eta azkenik pantailan zerbait idatzi. Hau horrela izanik, perl-en (-n) *switch*-ak fitxategien gainean lan egiteko aukera emango digu. Adibidez, aurreko “Fitxategi bateko lerro guztietan hasierako espazio guztiak kendu” programa horrela idatz dezakegu:

```
% perl –n –e ‘chomp();s/^ +//;print “$_\n”;’
```

---

[10] *Switch* hitza “komando-lerroko opzioa” bezala itzul genezake.

[11] “perldoc perlrun” exekutatu perl-eko switch guztien informazioa lortzeko.

Adibide honetan, (-n) *switch*-ak erabili dugu, aurrean ikusitako (-e) opzioarekin batera. Idatzitako aginduek zera egingo dute:

- `chomp(); $_` aldagai lehenetsian lerro-bukaerako karakterea kendu.
- `s/^ +//; $_` aldagaian gorderiko katearen hasiera-espazioak kendu.
- `print "$_\n"; $_` aldagaia pantailan idatzi, lerro-bukaerako karakterearekin batera

Baina zer egiten du (-n) *switch*-ak ? Bada, opzio horrek ezkutuko begizta bat egingo du inplizituki, sarrera-fitxategiko lerro bakoitzeko aurreko agindu hauek egikarituko dituelarik. Hau da, aurreko programa beste honen baliokidea da:

```
while (<>) {
   chomp;
   s/^ +//;
   print "$_\n";
}
```

(-n) *switch*-aren bitartez fitxategi bateko baldintza bat betetzen duten lerroak soilik idazteko programa bat erraz egin dezakegu:

```
% perl –ne 'if (/\bgorri/i) {print;}' testua.txt
```

Programatxo honek "testua.txt" fitxategian begiratuko du, eta "gorri"z hasten diren hitzak dituzten lerroak soilik inprimatuko ditu.

Azkenik, (-a) eta (-F) *switch*-ak ditugu. (-n) eta (–e) opzioekin, hainbat eremutan banatutako fitxategiekin lan egiteko aukera emango da. Suposa dezagun hitzei buruzko informazioa gordetzen duen fitxategi bat dugula, "hitzak.txt" izenarekin. Fitxategi horren lerro bakoitzean zera aurkituko dugu:

```
Hitza:Kategoria:adieraZbkia:Definizioa
```

hau da, fitxategi egituratua dugu, lerro bakoitzean ":" ikurraren bidez berezituta dauden hainbat eremu aurkituko ditugularik. Adibidez:

```
abereki:izond.:A1:Aberearen moduan jokatzen dena.
bakartegi.:iz.:A2:Leku bakartia.
bakebide:iz.:A1:Baketzeko bidea.
```

Fitxategi honekin zuzenean komando-lerrotik lan egiten duen programa bat hau litzateke:

```
% perl –F"/:/" –a –n –e 'print "$F[0]\n" if ($F[1] =~ /\biz\b/);'
```

edo, trinkoago:

```
% perl –F"/:/" –ane 'print "$F[0]\n" if ($F[1] =~ /\biz\b/);'
```

(-a) *switch*-ak, (-n) *switch*-arekin konbinatuz, lerro bakoitza hainbat eremutan banatuko du inplizituki, emaitza @F bektorean utziko duelarik. Eremu-banatzaile gisa, (-F) switch-ean ezarritakoa erabiliko du (aurreko adibidean "/:/" espresio erregularra, hots, ":" karakterea), besterik ezean, espazioak erabiliko dituelarik. Aurreko programaren baliokidea hau izango litzateke:

```
$eremuBanatzaile = "/:/";#-F switch-aren bitartez adierazia
while (<>) {
   @F = split($eremuBanatzaile, $_);
   print "$F[0]\n" if ($F[1] =~ /\biz\b/);
}
```

Zer egingo du idatzi berri dugun programa honek ?

Azkenik, zenbait *switch* gure programa sintaktikoki zuzena denentz esango digu:

```
% perl –cw programa.pl
programa.pl sintax OK
```

Honek guk idatzitako "programa.pl" egiaztatuko du.

6 Ariketa-bilduma

6.1 Espresio aritmetikoen kalkulua

6.1.1 Segundoak ordutan

Gauerdiaz gero pasatu diren segundoak irakurri eta 24 orduko adierazpidera pasa. Adibidez: 4005 irakurrita, *1 ordu, 6 minutu eta 45 segundo* idatzi behar da, zeren gauerdiaz gero 4005 segundo pasa direnean, ordu bat 6 minutu eta 45 segundo pasa baitira. Adibidez:

```
%segundo2ordu.pl
Eman segundo-kopurua:
4005
4005 segundoak, 1 ordu, 6 minutu eta 45 segundo dira
%
```

6.1.2 Zenbat segundo

Alderantzizkoa: hainbat ordu, minutu eta segundo emanda, kalkulatu denera zenbat segundu diren. Adibidez: *1, 6 eta 45* zenbakiak irakurrita, 4005 idatzi behar da, zeren *ordu bat, 6 minutu eta 45 segundo*, guztiak batera, 4005 segundo dira. Adibidez:

```
%ordu2segundo.pl
Eman 3 zenbaki, bakoitza lerro batean: orduak, minutuak eta segundoak:
1
6
45
Segundo-kopurua: 4005
%
```

6.1.3 Celsiusetik Farenheitera

Farenheit eskalan hainbat gradu adierazten dituen zenbaki bat irakurri eta Celsius eskalan dagokion tenperatura idatzi.Gogora ezazu formula hau: C/100=(F-32)/180 Adibidez:

```
% celsius2farenheit.pl
Eman Farenheit sisteman adierazitako tenperatura:
32
32 -> Farenheit
0 -> Celsius

% celsius2farenheit.pl
Eman Farenheit sisteman adierazitako tenperatura:
212
212 -> Farenheit
100 -> Celsius
%
```

6.1.4 Pezeta eta euroen bihurtzailea

Euro eta zenbakien artean bihurketak egiten dituen programatxo bat egin. Adibidez:

```
% euroKonbert.pl
Sartu kopuru bat:6
6 kopurua sartu duzu. Egin zure aukera.
 1) Euro->pta
 2) pta->Euro
1
6 euro 998.316 pezeta dira.

% euroKonbert.pl
Sartu kopuru bat:5000
5000 kopurua sartu duzu. Egin zure aukera.
 1) Euro->pta
 2) pta->Euro
2
5000 pezeta 30.0506052191891 euro dira.
```

Hobekuntza: Emaitza zenbaki erreal izan ordez, zenbaki erreal bat osoko bihurtzen duen `int` funtzioa erabiliz, idatzi zenbat euro eta zenbat zentimo diren.

6.1.5 Triangeluaren azaleraren kalkulua

Emanda triangelu baten oinarria (A) eta altuera (H) triangeluaren azalera idatzi.

```
%triangelu.pl
Eman 2 zenbaki, bakoitza lerro batean: oinarria  eta altuera:
1
6
Triangeluaren azalera: 3.00
%
```

6.2 Espresio erregularrak eta Perl

6.2.1 Bilaketa. Komando laburtuaren bidez

Bilatu eta idatzi *egunkaria.txt* fitxategian zein lerrotan azaltzen diren hiru silaba jarraian *p* , *t* edo *k* letrarekin hasten direnak. Adibidez, aurkituko dira: *kokatua, kontratu*, eta *kostape.*

6.2.2 Bilaketa. Programa baten bidez

Bilatu egunkaria.txt fitxategian zein lerrotan azaltzen den hiru silaba jarraian p , t edo k letrarekin hasten direnak. Hori lortzeko egokitu ezazu zenbakia duten lerroak idazten duen ondoko programa:

```
#!/usr/bin/perl -w
use strict;
my $lerro;

$lerro=<>;
chomp($lerro); # Lerro-bukaera marka kendu
if ($lerro !~ /\d/) {
   print "$lerro\n";
}
```

6.2.3 Ordezkapena. Komando laburtuaren bidez

Lerro-hasieretan dauden zuriuneak kendu.

6.2.4 Ordezkapena. Programa baten bidez

Lerro-hasieretan dauden zuriuneak kendu. Hori lortzeko egokitu ezazu UEUren ordez 'Udako Euskal Unibertsitatea' ipintzen duen ondoko programa:

```
#!/usr/bin/perl -w
use strict;
my $lerro;

$lerro = <>; # lerroa irakurri
chomp($lerro) # lerro-bukaerako marka kendu
$lerro =~ s/UEU/Udako Euskal Unibertsitatea/;   # soilik behin
# edo $lerro =~ s/UEU/Udako Euskal Unibertsitatea/g;   # agerpen guztiak
print "$lerro\n";
```

6.3 Kontrol-egiturak: baldintzapeko eskema

6.3.1 Ordenatu bi zenbaki

Zenbaki bi irakurri eta ordenatuta idatzi (hasieran handiena eta ondoren txikiena).

```
% txikiHandi.pl
Sartu 2 zenbaki, bakoitza lerro batean
2
7
7, 2
% txikiHandi.pl
Sartu 2 zenbaki, bakoitza lerro batean
7
2
7, 2
%
```

6.3.2 Asmatu zenbaki bat

Zenbaki bat irakurri eta beste erabiltzaile bati zenbaki hori asmatzen lagunduko dion programa bat egin ezazu. Adibidez:

```
% zbkAsmatu.pl
Inork ikusi gabe, sar ezazu zenbaki bat:17

Zenbaki bat gordeta dut. Zein izango da ?
Egin zure aukera:33
... mmmm, nik daukadan zenbakia 33 baino txikiagoa da
Egin zure aukera:25
... mmmm, nik daukadan zenbakia 25 baino txikiagoa da
Egin zure aukera:3
... mmmm, nik daukadan zenbakia 3 baino handiagoa da
Egin zure aukera:10
... mmmm, nik daukadan zenbakia 10 baino handiagoa da
Egin zure aukera:20
... mmmm, nik daukadan zenbakia 20 baino txikiagoa da
Egin zure aukera:17
Arraioa. Asmatu duzu eta !
%
```

6.3.3 Balio absolutua

Zenbaki bat irakurri eta bere balio absolutua idatzi.

```
%absolutu.pl
Eman zenbaki bat:
-1
1
%absolutu.pl
Eman zenbaki bat:
1
1
%
```

6.3.4 Lehenengo zenbakia bigarrenaren multiploa

A eta B bi zenbaki oso emanda, ea A Bren multiploa den ala ez esango digun algoritmoa idatzi.

```
%multiplo.pl
Eman 2 zenbaki, bakoitza lerro batean:
12
3
Multiploa da

%multiplo.pl
Eman 2 zenbaki, bakoitza lerro batean:
13
3
Ez da multiploa
```

6.4 Kontrol-egiturak: iterazio-eskema

6.4.1 Zenbat bokal lerroan

Konta ezazu zenbat bokal dauden lerro batean. Adibidez:

```
% zenbatBokal.pl
Idatzi lerro oso bat:hau da lerro bat idatzi behar dudana
Lerroan 14 bokal daude.
% zenbatBokal.pl
Idatzi lerro oso bat:HAU da hau
Lerroan 5 bokal daude.
%
```

6.4.2 Zenbat letra lerroan

Konta ezazu zenbat letra dauden lerro batean. Adibidez:

```
% zenbatLetra.pl
Idatzi lerro oso bat:hau da lerro bat idatzi behar dudana
Lerroan 30 letra daude.
%
```

6.4.3 Karakterea zenbatetan lerroan

Irakurri karaktere bat eta lerro bat, eta konta ezazu zenbatetan dagoen karaktere hori lerroan. Adibidez:

```
% zenbatKar.pl
Idatzi lerro oso bat:hau da lerro bat idatzi behar dudana
Zein karaktere zenbatu nahi duzu:z
z karakterea 1 aldiz agertzen da lerroan.

% zenbatKar.pl
Idatzi lerro oso bat:hau da lerro bat idatzi behar dudana
Zein karaktere zenbatu nahi duzu:a
a karakterea 7 aldiz agertzen da lerroan.

% zenbatKar.pl
Idatzi lerro oso bat:hau da lerro bat idatzi behar dudana
Zein karaktere zenbatu nahi duzu:d
d karakterea 4 aldiz agertzen da lerroan.
```

6.4.4 Zenbat karaktere fitxategian

Irakurri karaktere bat eta fitxategi baten izena eta konta ezazu zenbatetan dagoen karaktere hori fitxategian. Adibidez:

```
% zenbatKarFitx.pl
Zein karaktere zenbatu nahi duzu ?:a
Zein fitxategian begiratu nahi duzu ?:txikiHandi.pl
a karakterea 6 aldiz agertzen da txikiHandi.pl fitxategian.
%
```

6.5 Ariketa orokorrak

6.5.1 Letra bakoitza zenbatetan fitxategian

Irakurri fitxategi baten izena eta konta ezazu letra bakoitza zenbatetan azaltzen den fitxategian. Idatzi maiztasun horiek bi modutan letraren arabera ordenatuta eta maiztasunaren arabera ordenatuta. Adibidez:

```
% letraMaiztasuna.pl
Zein fitxategian begiratu nahi duzu ?: txikiHandi.pl
a karakterea 6 aldiz agertu da
b karakterea 4 aldiz agertu da
c karakterea 3 aldiz agertu da
e karakterea 7 aldiz agertu da
f karakterea 1 aldiz agertu da
h karakterea 2 aldiz agertu da
i karakterea 8 aldiz agertu da
k karakterea 2 aldiz agertu da
l karakterea 3 aldiz agertu da
m karakterea 3 aldiz agertu da
n karakterea 9 aldiz agertu da
o karakterea 4 aldiz agertu da
p karakterea 6 aldiz agertu da
1 karakterea 7 aldiz agertu da
2 karakterea 8 aldiz agertu da
r karakterea 9 aldiz agertu da
S karakterea 1 aldiz agertu da
s karakterea 4 aldiz agertu da
t karakterea 8 aldiz agertu da
u karakterea 3 aldiz agertu da
w karakterea 1 aldiz agertu da
y karakterea 1 aldiz agertu da
z karakterea 16 aldiz agertu da
------ Ordenatua -------
z karakterea 16 aldiz agertu da
n karakterea 9 aldiz agertu da
r karakterea 9 aldiz agertu da
```

```
2 karakterea 8 aldiz agertu da
t karakterea 8 aldiz agertu da
i karakterea 8 aldiz agertu da
1 karakterea 7 aldiz agertu da
e karakterea 7 aldiz agertu da
a karakterea 6 aldiz agertu da
p karakterea 6 aldiz agertu da
b karakterea 4 aldiz agertu da
o karakterea 4 aldiz agertu da
s karakterea 4 aldiz agertu da
m karakterea 3 aldiz agertu da
l karakterea 3 aldiz agertu da
c karakterea 3 aldiz agertu da
u karakterea 3 aldiz agertu da
h karakterea 2 aldiz agertu da
k karakterea 2 aldiz agertu da
S karakterea 1 aldiz agertu da
w karakterea 1 aldiz agertu da
y karakterea 1 aldiz agertu da
f karakterea 1 aldiz agertu da
```

6.5.2 Bigramak zenbatetan fitxategian

Irakurri fitxategi baten izena eta konta ezazu letra fitxategiko karaktere-bikote bakoitza zenbatetan azaltzen den fitxategian. Idatzi maiztasun horiek bi modutan bigramak fitxategian agertu direnaren arabera ordenatuta, eta maiztasunaren arabera ordenatuta. Adibidez:

```
% bigramaMaiztasuna.pl
Zein fitxategian begiratu nahi duzu ?: txikiHandi.pl
y-z bigrama 1 aldiz agertu da
S-a bigrama 1 aldiz agertu da
b-a bigrama 3 aldiz agertu da
...
2-z bigrama 3 aldiz agertu da
t-z bigrama 3 aldiz agertu da
------ Ordenatua -------
z-1 bigrama 7 aldiz agertu da
z-2 bigrama 7 aldiz agertu da
r-i bigrama 4 aldiz agertu da
i-n bigrama 4 aldiz agertu da
n-t bigrama 3 aldiz agertu da
...
a-n bigrama 1 aldiz agertu da
n-b bigrama 1 aldiz agertu da
```

6.5.3 Hitzen maiztasuna lerroan

Irakurri lerro bat eta konta ezazu zenbat bider azaltzen den lerroko hitz bakoitza (*lerro1HitzMaiztasun.pl*).

6.5.4 Hitzen maiztasuna fitxategian

Irakurri fitxategi baten izena eta konta ezazu hitz bakoitza zenbat bider azaltzen den fitxategian (*fitxHitzMaiztasun.pl*).

6.5.5 Hitzen maiztasuna eta maiztasun erlatiboa fitxategian

Irakurri fitxategi baten izena, konta ezazu hitz bakoitza zenbat bider azaltzen den fitxategian, eta idatzi ere bere maiztasun erlatiboa, hau da, hitz guztien artean zein portzentaiatan azaldu den hitz bakoitza (*fitxHitzMaiztasunerlat.pl*).

6.5.6 “garri”-z bukatzen diren hiztegiko sarrerak

EH_hiztek.txt fitxategian Ibon Sarasolaren Euskal Hiztegiko zatitxo bat daukagu. Lerro bakoitzean hiztegiko sarrera baten definizio bat sartu da, bakoitzean tabuladore batez banaturik honako lau eremu sartu direla: definizioko sarrera, bere kategoria, adiera-zenbakia eta definizioa (*EHgarriIzond.pl*). Adibidez:

```
a       interj  H2.     Oi.
aba     iz              Elizgizonen titulua, izen-deituren ondoan ezartzen dena.
ababor  iz              Ontziaren ezkerraldea, brankara begiratuz.
abade   iz      A2.     Gizonezkoentzako monasterio bateko burua, apaizteko esku duena.
abadegai        iz              Apaizgaia.
```

Lor itzazu *EH_hiztek.txt* hiztegi azpimultzo horretan “*garri*”-z bukatzen diren sarrerak.

6.5.7 “garri”-z bukatzen diren hiztegiko sarrerak beren definizioekin

Aurrekoan bezala baina sarrera bakoitzarekin bere definizioa erakutsi (*EHgarriIzond_gehi_def.pl*).

6.5.8 “garri”-z bukatzen diren sarreren definizioetako hitzen maiztasuna

Aurreko ariketan lortu diren definizio horietan azaltzen diren hitzen maiztasunak lortu. Lortu aparte hiztegi osoko hitzen maiztasuna, eta konparatu zerrenda biak *garri*-z bukatzen diren hitzekin lotuago dauden hitzak aztertzeko. Zein hitz azaltzen dira askoz maizago *garri*-z bukatzen diren hitzekin, orokorrean baino (*EHgarriMaiz.pl*).

6.5.9 Akatsen bila: definizioko sarrera definizioan (zirkulartasuna)

EH_hiztek.txt hiztegitxoan txarto eratutako definizio batzuk gehitu ditugu. definizioko sarrera definizioan erabili dugu. Txarto eratutako horrelako definizioak, bakoitza bere sarrerarekin, idatziko dituen programa bat idatzi (*EHzirk.pl*).

6.5.10 Hitz bat duten lerroak idatzi aurreko lerroarekin batera

Irakurri hitz bat eta idatzi corpus.txt fitxategian hitz hori duten lerroak, baina beti aurreko lerroarekin batera, testuingurua hobeto ikustearren (*lerro1.pl*). Unix komandoak erabiliz lor daiteke emaitza hori?

6.5.11 Hitz bat duten lerroak idatzi aurreko eta hurrengo lerroarekin batera

Irakurri espresio bat eta idatzi corpus.txt fitxategian espresio hori duten lerroak, baina beti aurreko eta hurrengo lerroarekin batera, testuingurua hobeto ikustearren (*lerro2.pl*). Unix komandoak erabiliz lor daiteke emaitza hori?

6.5.12 Hitz bat duten lerroak idatzi aurreko eta hurrengo N lerroekin

Irakurri hitz bat eta N zenbaki bat, idatzi corpus.txt fitxategian hitz hori duten lerroak, baina beti aurreko eta hurrengo N lerroekin batera, testuingurua hobeto ikustearren.

6.5.13 Ingelesezko testutik aditzak atera

Egizu programa bat Hobbes_Leviathan testutik aditz guztiak ateratzeko (eman dezagun aditzak aurretik "to" hitza daukatenak direla) (*aditzak_bilatu.pl*).

```
% aditzak_bilatu.pl
Aditz bat: go
Aditz bat: take
...
```

6.5.14 Ingelesezko testutik aditzak ordenatuta atera

Egizu programa bat Hobbes_Leviathan testutik aditz guztiak ateratzeko (eman dezagun aditzak aurretik "to" hitza daukatenak direla). Orain aditzak testuko agerpen-ordenan eta maiztasunaren arabera ordenatuta ere agertuko dira (*aditzak_bilatu_eta_ordenatu.pl*).

```
% aditzak_bilatu_eta_ordenatu.pl
Agerpen-kopurua:     12 go
Agerpen-kopurua:     7 take
Agerpen-kopurua:     24 come
...
Orain ordenatuta:
Agerpen-kopurua eta hitza:     0112 make
Agerpen-kopurua eta hitza:     0024 come
Agerpen-kopurua eta hitza:     0012 go
Agerpen-kopurua eta hitza:     0007 take
...
```

6.5.15 Ingelesezko testutik adjektiboa eta bere ondoko hitza lortu

Egizu programa bat Hobbes_Leviathan testutik "artificial" adjektiboaren ondoko izenak ateratzeko (*artifizial_bilatu.pl*).

```
% artifizial_bilatu.pl
Adibide bat: artificial life
...
```

6.5.16 Hiztegia kontsultatzeko sistematxoa

Prestatu hiztegitxoa kontsultatzeko sistematxo bat. Hainbat kontsulta jarraian erantzungo ditu sistemak, kontsultak menu baten arabera eskatuko direla. Adibidez:

a. sarrera baten definizioa
b. azpikatea bat duten sarrerak
c. azpikatea bat duten definizioen sarrerak
d. azpikatea bat duten definizioen sarrerak eta definizioak
e. kategoria eta sarreran azpikatea duten sarreren lista

0- bukatu

6.5.17 Tokenizatzaile sinplea. Esaldiak hitzetan banatu.

Ezer eskatu gabe, irakurri esaldi bat sarrera estandarretik, eta hitzetan banatu. Idatzi lerro bat hitz zein ikur bakoitzeko.

```
% tokenizatzailea.pl
Gaur lau (4) ordu egin ditut lo.
Gaur
lau
(
4
)
ordu
egin
ditut
lo
.
```

6.5.18 Esaldiak berrantolatzen.

Aurreko adibidearen alderantzizkoa. Tokenizatzailearen irteera hartu, eta esaldia berrantolatu, hau da, lerro bakoitzeko hitzak eransten joan esaldi-bukaerako ikur bat azaldu arte.

```
% esaldika.pl tokenizatzailearen_irteera.txt
 Gaur lau ( 4 )  ordu egin ditut lo .
```

6.6 Azpiprogramak

6.6.1 Bi zenbaki osoen zatitzaile komunetatik handiena lortu.

Oinarrizko programazioko ikastaroaren apunteetan agertzen den ZKH izeneko funtzioa inplementatu. Bi zenbaki eskatu eta bien arteko zatitzaile komunetatik handiena dena pantailaratu.

```
% zkh.pl
Sakatu bi zenbaki oso, espazioekin bereiziak.
Sartu lerro hutsa programatik ateratzeko
100 16
100 eta 16 zenbakien arteko zatitzaile komunetatik handiena 4 da.
Sakatu bi zenbaki oso, espazioekin bereiziak.
Sartu lerro hutsa programatik ateratzeko
```

6.6.2 Interpretazio-fitxategien lema/kategoriak atera.

Demagun interpretazioak dituen fitxategi bat dugula, honako egitura batekin:

```
("norbera"  IOR IZGMGB DEK GEN NUMS MUGM  @IZLG> @<IZLG)
("sen"  IZE ARR  @KM>)
("on"  ADJ IZO DEK NUMS MUGM DEK ABL  @ADLG)
("hasi"  ADI SIN AMM PART  @-JADNAG)
...
```

Gure programak, lerro bakoitzeko lema (komatxoen artekoa) zein kategoria idatziko ditu interpretazio-lerro bakoitzeko.

```
% perl lemaKategoia.pl lemaFroga.txt
norbera  IOR
sen   IZE
on    ADJ
```

```
hasi  ADI
...
```

6.6.3 Fitxategi batean, eta bi karaktere emanda, karaktere horietatik zein agertzen den usuen erabakitzea.

Eskatu fitxategi-izen bat, eta bi karaktere. Gero, lerroz lerro, karaktere horien agerpenak zenbatu. Azkenik, bi karaktere horietatik usuena zein izan den esan. Emandako karaktereen agerpen-kopurua berdina bada, hala esan.

```
% zeinKarGehiago.pl
Eman fitxategi-izen bat:corpus.txt.lem
Sakatu lehenengo karakterea:b
Sakatu bigarren karakterea:b
b karakterearen agerpen-kopurua eta b karakterearen agerpen- rua berdinak
dira corpus.txt.lem fitxategian

% zeinKarGehiago.pl
Eman fitxategi-izen bat:corpus.txt.lem
Sakatu lehenengo karakterea:a
Sakatu bigarren karakterea:b
a (2648) karakterea b (747) baino gahiagotan azaltzen da corpus.txt.lem
fitxategian
```

6.6.4 Zenbaki bat emanda, izartxoez osatutako piramide-erdia pantailatik inprimatu.

Eskatu zenbaki bat, eta, horren arabera, hainbat lerro pantailan idatzi, non lerro bakoitzak aurrekoak baino izartxo bat gehiago duen, lehenengo lerroak izartxo bakarra duelarik.

```
./izartxo.pl
Sartu zenbaki bat (zero zenbakia bukatzeko)
2
*
**
Sartu zenbaki bat (zero zenbakia bukatzeko)
10
*
**
***
****
*****
******
*******
********
*********
**********
Sartu zenbaki bat (zero zenbakia bukatzeko)
0
```

6.7 Moduluak

6.7.1 Fitxategi jakin batean dauden helbide elektroniko guztietara mezu bat idatzi.

```
% mezuak.pl helbideak.txt

Mezua bidali dut ccpsoeta@si.ehu.es helbidera.
Mezua bidali dut jipsagak@si.ehu.es helbidera.
Mezua bidali dut jipgogak@si.ehu.es helbidera.
```

6.7.2 Helbideen fitxategi bat eta testu-fitxategi bat emanda, testu hori helbide fitxategiko helbide guztiei igorri.

```
% mezuak2.pl helbideak.txt testua.txt

Mezua bidali dut ccpsoeta@si.ehu.es helbidera.
Mezua bidali dut jipsagak@si.ehu.es helbidera.
Mezua bidali dut jipgogak@si.ehu.es helbidera.
```

6.7.3 Testu lematizatu baten esaldiak berrosatzen.

Parametro gisa testu lematizatu baten izena jasoz, testu horren esaldiak berrosatu (kategoria eta lemaren informazioari jaramonik egin gabe). Demagun honako testu lematizatu hau dugula:

```
/<Norberaren>/<HAS_MAI>/
      ("norbera"  IOR IZGMGB DEK GEN NUMS MUGM  @IZLG> @<IZLG)
/<sen>/
```

```
     ("sen"  IZE ARR  @KM>)
/<onetik>/
     ("on"  ADJ IZO DEK NUMS MUGM DEK ABL  @ADLG)
/<hasi>/
     ("hasi"  ADI SIN AMM PART  @-JADNAG)
/<eta>/
     ("eta"  LOT JNT EMEN  @PJ)
/<goi-mailakoteknologiaraino>/
     ("  /goi-mailakoteknologia/  IZE ARR DEK NUMS MUGM DEK ABU)
     ("  /goi-mailakoteknologia/  IZE ARR MAR IZE ARR DEK NUMS MUGM)
/<XXI>/<ERROM>/
     ("XXI"  DET DZH NUMP DEK ABS MG  @OBJ @SUBJ @PRED)
/<.>/<PUNT_PUNT>/
/<MENDERANTZ>/<DEN_MAI>/
     ("mende"  IZE ARR DEK NUMS MUGM DEK ABZ  @ADLG)
...
```

Programak zera egingo luke:

```
% perl lematizatu2esaldi.pl testuLematizatua.etiketatua3
Norberaren sen onetik hasi eta goi-mailakoteknologiaraino XXI .
 MENDERANTZ ...
```

6.7.4 Hitz anbiguoenak ezagutzen.

Lematizatutako fitxategi bat parametro gisa jasoz, fitxategi horretan anbiguotasun-neurri handiena duen hitza (eta bere interpretazioak) eman. Hitz hori fitxategiko zein lerrotan dagoen ere erakutsi.

```
% perl anbiguoHandiena.pl corpus.txt.etiketatua3
/<zentzudunentzat>/
     ("zentzu+dun"  IZE ARR ATZ ADJ IZO DEK DES MG  @ADLG)
     ("zentzu+dun"  IZE ARR ATZ ADJ IZO DEK DES NUMP @ADLG)
     ("zentzudun"  ADJ IZO DEK DES MG  @ADLG)
     ("zentzudun"  ADJ IZO DEK DES NUMP MUGM  @ADLG)
1302 lerroan.
```

6.7.5 Lema desberdinak jaso duten hitzak erakutsi.

Lematizatutako fitxategi bat parametro gisa jasoz, lema bat baino gehiago duten hitzak (eta bere interpretazioak) erakutsi, hitz hori fitxategiko zein lerrotan dagoen ere adieraziz.

```
% perl lemaAnbiguoa.pl corpus.txt.etiketatua3
/<ERAMANGARRIA>/<DEN_MAI>/
     ("eraman+garri"  ADI SIN ATZ ADJ IZO DEK ABS NUMS MUGM)
     ("eramangarri"  ADJ IZO DEK ABS NUMS)
42 lerroan.
/<diogu>/
     ("*io"  ADT A1 NR_HU NK_GU  @-JADNAG)
     ("ukan"  ADT A1 NR_HU NI_HU NK_GU  @-JADNAG)
242 lerroan.
/<estaltzen>/
     ("estaldu"  ADI SIN AMM ADOIN ASP EZBU  @-JADNAG)
     ("estali"  ADI SIN AMM ADOIN ASP EZBU  @-JADNAG)
644 lerroan.
/<eramangarriaren>/
     ("eraman+garri"  ADI SIN ATZ ADJ IZO DEK GEN NUMS MUGM)
     ("eramangarri"  ADJ IZO DEK GEN NUMS MUGM)
723 lerroan.
....
```

7 Lan praktikoak

1) Egindako perl programa bat aplikatu beste datu batzuekin
2) Egindako perl programa bat egokitu beste gauza bat egin dezan.

```
Asmatu perl programa bat
```

8 Bibliografia

Agirre E., Soroa A. *Perl programazio-lengoaia* ikastaroaren apunteak. UEU. 1999.

Schwartz R *Learning PERL.(the "Llama Book"). Second Edition.* Ed. O'Reilly & Associates 1997.

Wall L. R *Programming Perl (the "Camel Book"). Second Edition*. Ed. O'Reilly & Associates 1996.

9 Eranskinak

9.1 Perl nola eskuratu

- PERL GNUko lizentzian kokatzen da eta berez dohainik eskura daiteke
- Internet-en
- Programak lortzeko (PERL bera, moduluak, etab.)
    - CPAN (Comprehensive Perl Archive Network): PERLi buruzko ia edozer
    - ftp.rediris.es/mirror/CPAN
    - ftp.funet.fi/pub/languages/perl/CPAN
- Informazioa lortzeko:
    - http://www.perl.com
- Eztabaida-guneak (News)
    - comp.lang.perl.announce
    - comp.lang.perl.misc
    - comp.lang.perl.modules

9.2 SILemati modulua

```
package SILemati;
$VERSION = "0.1";

use Carp;
use FileHandle;
use strict;
use locale;

my %kontrol = (fh => undef,
               fname => undef,    # fitxategiaren izena
               lz => undef,       # lerro zenbakia
               hlz => undef,      # hurrengo lerro zenbakia
               hlerr => undef,    # hurrengo lerroa
               posiz => undef,    # chunkaren hasiera-posizioa
               hposiz => undef,   # hurrengo posizioa
               hegoera => undef,  # Hurrengo egoera
               egoera => undef,   # 1 sarrera
```

```
                                    # 0 EOF
               ema => [],
               goiburu => []
             );

sub new {

  my $that = shift;
  my $class = ref($that) || $that;
  my $self = { %kontrol };
  bless $self, $class;
  $self->_hasieratu(@_);
  return $self;
}

sub _hasieratu ($$) {

  my ($this) = shift;

  my $fh = new FileHandle;
  my ($eof);

  ($fh->open($_[0])) ||
    croak "Ezin dut $_[0] fitxategia ireki:$!\n";
  $this->{fh} = $fh;
  $this->{posiz} = $this->{lz} = 0;
  $this->{hposiz} = $this->{hlz} = 0;
  $this->{goiburu} = [];
  $eof = $this->_hlerr($this->{goiburu});
  $this->{fname} = $_[0];
  $this->{egoera} = $eof ? 0 : 1;
}

sub _hlerr ($$$) {

  my ($this, $aref) = @_;

  $this->{hlerr} = "";
  while (!$this->{fh}->eof) {
    $this->{hposiz} = $this->{fh}->tell();
    $this->{hlz}++;
    chomp($this->{hlerr} = $this->{fh}->getline);
    next if ($this->{hlerr} =~ /^\*+$/o);
    last if ($this->{hlerr} =~ /^\/\<[^\@]/);
    push(@{$aref}, $this->{hlerr});
    $this->{hlerr}="";
  }
  return 0 if ($this->{hlerr});
  return 1;
}

sub hitza_ekartzan ($) {

  my $this = shift;

  if ($this->{ema}->[0] =~ /^\/\<([^\>]+)\>/) {
    return $1;
  }
  return "";
}
```

```
sub id_ekartzan ($) {

  my $this = shift;

  if ($this->{ema}->[0] =~ /\>\/\<([^\>]+)\>/) {
    return $1;
  }
  return "";
}

sub hgoiburu_ekartzan ($) {

  my $this = shift;

  return $this->{goiburu};

}

sub goiburu_ekartzan ($) {

  my $this = shift;

  return @{$this->{goiburu}};

}

sub hsarrera_ekartzan ($) {

  my $this = shift;

  return $this->{ema};
}

sub sarrera_ekartzan ($) {

  my $this = shift;

  return @{$this->{ema}};
}

sub lerro_zenbakia_ekartzan ($) {

  my $this = shift;

  return $this->{lz};
}

sub pos_eman ($) {

  my $this = shift;

  return $this->{posiz};

}

sub goto_pos ($$) {

  my $this = shift;
  my $pos = shift;
```

```
  my ($eof, @aux);
  $this->{fh}->seek($pos, 0); #SEEK_SET
  $eof = $this->_hlerr (\@aux);
  $this->{egoera} = $eof ? 0 : 1;
}

sub hurrengo_sarrera ($) {

  my ($this) = shift;

  my $eof;

  return 0 unless ($this->{egoera});

  $this->{ema} = [$this->{hlerr}];
  $this->{posiz} = $this->{hposiz};
  $this->{lz} = $this->{hlz};
  $eof = $this->_hlerr(\@{$this->{ema}});
  $this->{egoera} = $eof ? 0 : 1;
  return 1;
}

1;
```