

TAPE

Testu-analisirako

PERL erremintak

AITZOL ASTIGARRAGA PAGOAGA
KOLDO GOJENOLA GALLETEBEITIA
KEPA SARASOLA GABIOLA
AITOR SOROA ETXABE

```
#!/usr/bin/perl
use warnings;
use strict;
my $fitx = $ARGV[0];
my $patroi = $ARGV[1];
my $zenb = $ARGV[2];
my $kop = 0;
my $lerro;
open(FITX, "$fitx") ||
    die ("Ezin $fitx fitxategia
while ($lerro = <FITX>) {
    while ($lerro =~ /$patroi/g
        $kop++;
        if ($kop == $zenb) {
            print("topatu dut\n");
            print($lerro);
        }
    }
}
```



Udako Euskal Unibertsitatea

TAPE TESTU-ANALISIRAKO PERL ERREMINTAK

Aitzol Astigarraga Pagoaga
Koldo Gojenola Gallettebeitia
Kepa Sarasola Gabiola
Aitor Soroa Etxabe

Udako Euskal **Unibertsitatea**
Bilbo, 2009

© Udako Euskal Unibertsitatea

© Aitzol Astigarraga Pagoaga

ISBN: 978-84-8438-233-1

Lege-gordailua: BI-1972-09

Inprimategia: CUATROAS, Bilbo

Azalaren diseinua: Iñigo Ordozgoiti

Hizkuntza-zuzenketen arduraduna: Ander Altuna Gabiola

Banatzailleak: UEU. Erribera 14, 1. D BILBO telf. 946790546 Faxe. 944793039

Helbide elektronikoa: argitalpenak@ueu.org

www.ueu.org

Elkar Banaketa: Igerabide, 88 DONOSTIA

Galarazita dago liburu honen kopia egitea, osoa nahiz zatikakoa, edozein modutara delarik ere, edizio honen Copyright-jabeen baimenik gabe.

*Eskerrak Ibon Sarasolari,
bere Euskal Hiztegiaren zati
bat erabiltzen uzteagatik.*

Edukien aurkibidea

1. SARRERA	11
1.1. Zertarako programatu?	13
1.2. Zergatik Perl?	13
1.2.1. Perl kultura.	14
1.2.2. Kontuan izan	14
1.3. Nola irakurri liburu hau	15
2. EZERTAN HASI AURRETIK.	19
2.1. Zer behar dut lanean hasteko?	19
2.2. Materiala.	20
2.3. Komando-lerroa	20
2.3.1. Komando-lerroa abiarazi	21
2.3.2. Oinarrizko aginduak.	22
2.4. Perl interpretatzailea.	25
2.4.1. Instalatzeko pausoak	25
2.5. Testu-editorea	29
2.6. Lehen programa	30
2.6.1. Lehen pausoa: edizioa	30
2.6.2. Bigarren pausoa: exekuzioa	32
3. OINARRIZKO PROGRAMAZIOA	35
3.1. Aginduak eta sententziak	35
3.2. Datu motak	39
3.2.1. <i>String</i>	39
3.2.2. Zenbakiak.	41
3.3. Aldagaiak	42
3.3.1. Eskalarrak.	42
3.3.2. <i>Array</i> edo bektoreak.	49
3.4. Kontrol-egiturak	53
3.4.1. Baldintzazko egiturak	54
3.4.2. Begiztak	63
3.5. <i>Array</i> -ekin lan egiteko funtzioak	72

4. SARRERA/IRTEERA	89
4.1. Irakurri eta idatzi	89
4.2. Deskribatzaileak	90
4.3. Sarrera/Irteera estandarra	91
4.4. Komando-lerroa	92
4.5. Fitxategiak	95
4.5.1. Lerroz lerro	99
4.5.2. Fitxategia osorik irakurri	102
4.5.3. Fitxategietan idatzi	104
5. GELDIUNEA ERDIBIDEAN	107
5.1. Ohiko programazio-erroreak	107
5.2. Programazio-estilo egokia	108
5.2.1. Aldagai bereziak	113
5.2.2. Tresna berriak	116
6. ADIERAZPEN ERREGULARARRAK	121
6.1. Patroiak	122
6.2. Karaktere bereziak	125
6.2.1. Bilaketa mugatu	126
6.2.2. Karaktere-klaseak	128
6.2.3. Aukerak: hau, hori edo hura	138
6.2.4. Espresioak bildu	139
6.2.5. Atzera begirako erreferentziak	139
6.2.6. Errepikapenak: zenbat aldiz?	142
6.3. Adierazpen erregularren funtzionamendua	145
6.4. Ordezkapenak	155
6.5. Ordezkapen gehiago	158
6.6. Nola idatzi adierazpen erregularrak	161
6.7. split() eta join(): banatu eta bildu	162
7. HASH EGITURA	173
7.1. delete()	175
7.2. exists()	176
7.3. keys()	177
7.4. values()	178
7.5. each()	180
8. AZPIPROGRAMAK, ERREFERENTZIAK ETA MODULUAK	183
8.1. Azpiprogramak	183
8.1.1. Definizioa	183

8.1.2. Funtzio-deiak	184
8.1.3. Argumentuak	185
8.1.4. Funtzioaren gorputza	186
8.1.5. <i>return</i> agindua	186
8.2. Erreferentziak	189
8.2.1. <i>Array</i> -en erreferentziak	192
8.2.2. <i>Hash</i> -en erreferentziak	193
8.2.3. Eskalarren erreferentziak	194
8.2.4. Datu-egitura habiaratuak	195
8.2.5. Parametroak erreferentziaz	197
8.3. Moduluak	200
8.3.1. Ngramak kontatzen	201
8.3.2. Moduluak instalatzen	202
8.3.3. <i>Text::Ngrams</i> modulua erabiltzen	205
9. KASU PRAKTIKOAK	213
9.1. Azpiprogramak	213
9.2. Bilaketak	216
9.3. Ordezkapenak	222
9.4. Adierazpen erregularrak eta <i>hash</i> egiturak	225
9.5. Erreferentziak	244
9.6. Ariketak hiztegiarekin	247
9.7. Internet corpus gisa	253
10. AMAIERA DA HASIERA	259
10.1. Dokumentazioa	259
10.2. Liburuak	260
10.3. Sarea	260
11. KASU PRAKTIKOEN AURKIBIDEA	263
12. INDIZE ALFABETIKOA	265

1. Sarrera

Testuen azterketa eta prozesamendua, ikerketa-arlo zirrargarria da gaur egun. Testu idatziak gorde eta manipulatzeko konputagailuen erabilerak testuok aztertzeko ate berriak zabaldu ditu. Azken urteotan konputagailuen arloan emandako aurrerapausoek eta, batez ere, Interneten hedatze eta erabilera masiboak, ikaragarriko testu digital masa utzi digute esku-eskura. Adibide pare batekin azalduko dugu hori: batetik, *Berria* egunkariak¹ urtero argiratzen du 10 milioi hitzeko testua, eta bestetik, on-line kontsulta daitezkeen hainbat corpusetan² erraz lor ditzakegu hitz bat duten ehunka esaldi. Interneten ingelesez bilioi bat hitz edo dagoela estimatzen da (milioi bat milioi hitz!). Euskaraz, ingelesez baino mila aldiz edo hamar mila aldiz gutxiago omen dagoenez³, gutxi gorabehera mila milioi hitz izango dira sarean orain. Kopuru horiek zein handi diren erakusteko, nahikoa da jakitea liburu arrunt batean 100.000 hitz inguru sartzen direla. Kalkulu azkar bat eginda, 10 mila liburu betetzeko adina hitz dauzkagu euskaraz sarean. Harrigarria, ez da? Bada, informazio hori guztia eskuz aztertzea, itsasoa koilara batez hustu nahi izatearen pareko lana litzateke, tratamendu automatikorako tresnak behar-beharrezkoak dira.

Liburu honen helburua da konputagailu bidez testuak aztertu eta manipulatu nahi dituztenei euskaraz nolabaiteko abiapuntua eskaintzea. Gaur egun testuekin lan egiteko tresna ugari diren arren (zuzentzaile ortografikoak, hiztegiak on-line, dokumentu-bilatzaileak, etab.), eskura ditugun tresna horiek ez diete beti gure behar zehatzei erantzuten. Zer egin orduan? Gure beharrei erantzungo dieten konputagailu-programak guk geuk idatzi. Horra gure proposamena. Perl programazio-lengoaia aproposa dugu lan horretarako. Programazio-lengoaia ahaltsua da Perl, berariaz testuen tratamendurako sortua izan zena. Garrantzitsua iruditu zaigu, orobat, lengoaia honen zabalkuntza-lana euskaraz egitea, eduki eta aplikazioak euskara eta euskal kulturara egokituz.

1. <http://berria.info>

2. Adibidez: **Corpeus** (<http://corpeus.elhuyar.org>), **Elebila** (<http://www.elebila.eu>), **XX. Mendeko Corpusa** (<http://www.euskaracorpusa.net>, 4,6 Mhitz), **Ereduzko Prosa Gaur** (<http://www.ehu.es/euskara-orria/euskara/ereduzkoa>, 9,6 Mhitz), **ZT** (<http://www.ztcorpusa.net/cgi-bin/kontsulta.py>, 8 Mhitz), **Susa** (<http://www.susa-literatura.com>), eta **Klasikoen Gordailua** (<http://klasikoak.armiarma.com>).

3. Euskarararen presentzia Interneten neurtu nahian: Alegria, I. eta Rodriguez, M. J. (2003): *BAT Soziolinguistika Aldizkaria*, **48**, 89-100.

Ez zaitez beldurtu irakurle, *programazio-lengoaia*, *script* eta *Perl* terminoak, besteak beste, guztiz arrotzak bazaizkizu. Liburu honek ez du programazioari buruzko aurrezagutzarik eskatzen; aitzitik, hutsetik abiatuko gara Perl lengoaiaren ikasketan. Bi aurrebaldintza baino ez ditugu jarriko: bata, testu digitalak aztertu eta manipulatzeko interesa izatea, eta bestea, berriz, konputagailu bat eskura izatea berarekin lan egin ahal izateko. Ez duzu besterik behar.

Liburu honetan, beraz, Perl lengoaiaren oinarriak azalduko ditugu. Perl aukera mordoa eskaintzen duen lengoaia denez, komeni da lehenbizi oinarri bat finkatzea, programazio-lengoaia guztietan funtsezkoak diren kontzeptu eta nozioak azaltzea. Aurrera egin ahala, Perl-en ezagueran sakonduz joango gara, testuen azterketarako eskaintzen dituen funtsezko edukiak geureganatuz. Azalpenak ahalik eta sinpleenak izan daitezen ahalegindu gara, xehetasun erabilgarrienak ahaztu gabe, eta betiere adibide eta kasu praktikoei arreta berezia eskainiz.

Hona hemen, adibide gisa, testuak aztertzeko zer-nolako programak idatziko ditugun:

- Hitz-bukaera eman eta testuan errimak aurkitu.
- Bilaketak egin Interneten. Adibidez, zein dira gaur *Berrian* gehien aipatu diren 3 izen bereziak?
- Testu bateko 10 hitz erabilienak aurkitu.
- Testuan erabilitako bokalen maiztasuna kalkulatu eta itzuli. Zein da gehien erabiltzen dena?
- Karaktere kopuruaren arabera testuko hitzak ordenatu. Luzeenak ondo idatzita daude?
- Testu ezberdinen edukia konparatu, hitz eta esaldi komunak bistaratuz. Ea testu bat beste baten kopia ote den aztertzeko balio dezake.
- Karaktere berdinarekin hasi eta amaitzen diren hitzak bilatu testuan.

Eta askoz ere gehiago. Orri hauetan aurkituko duzuna, ahalik eta zehaztasunik handienez azalduko tresna-bilduma izango da, testuen azterketara bideratuak. Programa laburrak dira, di-da batean idaztekoak eta berehalako emaitzak itzultzen dituztenak, programatzen esperientziarik ez duenak ere erraz erabiltzeko modukoak. Batzuk zeure lanerako erabilgarri gerta dakizkizuke dauden daudenean. Edo bestela, aldaketa gutxi batzuk eginda zeurera ekarri ahal izango dituzu. Izan daitezela gutxienez Perl-ekin egin ditzakegun lanen erakusgarri eta argigarri.

Ez beldurtu, beraz, programazioaren uretan inoiz aurretik murgildu ez bazara. Hauxe duzu aukera.

1.1. ZERTARAKO PROGRAMATU?

Liburu honen sarrera irakurtzen ari zarenez gero, badakigu noiz edo noiz testu digitalekin lan egitea egokitu zaizula, eta, hortaz, testuak manipulatu eta aztertzeke tresna egokiak ondo etorriko zaizkizula. Une honetan baliteke duda-mudatan ibiltzea, bi aukera hauen artean: testuak manipulatzeko software-pakete bat eskuratu eta harekin lanean hasi, edo bestela, liburu hau irakurtzen jarraitu eta Perl lengoaiari heldu.

Horrela bada, emaguzu tartetxo bat Perl ikastearen aldeko argudioak azal ditzagun. Hona gure ustez aipagarrienak:

- Programatzeak zehazki egin nahi duzun hura egiteko aukera ematen du, programak nork bere neurri eta beharretara egokituz.
- Testuen azterketara bideratutako software-pakete ugari dagoen arren, ez dugu haien menuetan beti aurkituko egin nahi duguna egiteko funtziorik. Gainera, software-pakete horien erabilera ikasteak denbora eta esfortzua eskatzen du.
- Programatzeak datuen gaineko erabateko kontrola emango dizu. Hurbilago-tik aztertzeak bestela oharkabea pasatuko liratekeen ezaugarriez jabetzea ekar lezake.
- Programatzea aktibitate sortzailea da, aldiro erronka berriak proposatzen dituena. Norberaren beharrei erantzuten dieten programak garatzeak atsegin handia sortzen du.
- Programazio-lengoaia gehienek oinarria berdintsua da. Bat ikasteak beste-tarako bidea asko erraztuko digu.

Hala ere, bide bata ez du bestea kentzen. Erabil dezakezu ohiko programa bat eragiketa orokorrak egiteko, eta gainera Perl apur bat ikasi zeure behar zehatzei erantzuna emango dieten programak idazteko. Gure aholkua badakizu zein den: anima zaitez Perl lengoaia ikasten.

1.2. ZERGATIK PERL?

Programazio-lengoaia mordera dago gaur egun, baina ondoren zehaztuko ditugun arrazoiengatik uste dugu Perl dela gure beharretara hobekien egokitzen dena:

- Testu-fitxategiak aztertu eta manipulatzeko bereziki sorturiko programazio-lengoaia da Perl, egokia beraz, egokirik bada, hizkuntzarekin erlazionatu-tako atazetarako. Fitxategiak lerroz lerro irakurtzeko gai da eta karaktere eta hitz terminoak maneiatzen ditu. Adierazpen erregularren bitartez testue-tan bilaketak eta aldaketak egiteko izugarritzko tresneria eskaintzen du.

- Perl software librea da eta doakoa. Konputagailu eta sistema eragile guztientzat aurki daiteke doako Perl inplementazioa.
- Perl lankidetzan oinarritua dago. CPAN software-artxibategiak Perl komunitateak idatzitako programak biltzen ditu, eta doan jaitsi eta erabil ditzakegu. Haietako asko testuak prozesatu eta manipulatzeko gehigarriak (moduluak) dira.
- Beste lengoia askorekin alderatuz gero, erraza da ikasteko, eta ikasitako apurrari etekin handia atera diezaiokegu.

Ez pentsa irakurle hegan dakien astoa saldu nahi dizugunik. Perl egokia da gure asmoetarako, testuen prozesamendurako, baina ez da horretarako erabil dezakegun lengoia bakarra, badira beste alternatiba batzuk ere: erabilienak Python eta Java, eta horiekin batera baita LISP eta Prolog ere. Aparteko aipamena merezi du NLTK paketeak (www.nltk.org), lengoia naturalaren prozesamendurako Python lengoian garatutako tresna-bilduma.

Bestalde, berriro diogu, ez ditugu hemen Perl-en ezaugarri guztiak azalduko, ezta gutxiagorik ere. Gure helburua xumeagoa da, hutsetik abiatuko gara eta gure beharretarako egokiak diren elementuak aukeratuko ditugu soilik. Zure lanerako abiapuntu baliagarria izatearekin gustura geundeke. Gero, ondo moldatzen bazara Perl-ekin, sakonago aztertu ahal izango duzu lengoia. Liburuaren amaieran, aurrera bidean lagungarri izan ditzakezun liburu eta webguneen erreferentziak topatuko dituzu. Eta arazorik bazenu bide horretan galdetu Interneteko foroetan, edo liburu honen egileoi.

1.2.1. Perl kultura

Programazio-lengoia konputagailuari aginduak emateko arau-multzo formal gisa definitu ohi da. Baina hori baino gehiago ere bada. Bizia ematen dion erabiltzaile taldeak eragin handia dauka lengoiairengan. Perl-ek erabiltzaile-komunitate oso aktiboa dauka, eta bada lengoiairengan inguruan halako ideologia edo filosofia bat, programak idazteko modua eta programatzaileen pentsatzeko modua deskribatzen dituen. Hona filosofia hori hobekien islatzen duten esaldiak:

- Larry Wall-ek sortu zuen Perl, hizkuntzalaria bera. Bere hitzetan: «Perl egin behar duzun hori egiteko lengoia da».
- «Errazak diren gauzek errazak izan behar lukete, eta zailak, berriz, posible».
- «Gauza bera egiteko modu bat baino gehiago daude»

1.2.2. Kontuan izan

Perl bereziki egokia da programa txikiak garatzeko, idazteko errazak direnak eta ataza ugari automatizatzeko edo ebazteko balioko ditutenak. Horren erakusle,

liburu honetako programa gehienak erraz sartzen dira orrialde batean, eta luzeenak ez ditu bi orrialde baino gehiago hartuko. Bestalde, ez da ikaste-prozesu neketsurik gainditu behar programa erabilgarriak idazten hasteko. Hasieratik bertatik ekingo diogu programa praktikoa idazteari. Ahaztu gabe, noski, Perl ezaugarri asko dituen lengoiaia aberatsa dela, ñabardura eta xehetasunez betea, eta programa luze eta konplikatuak idazteko ere erabiltzen dela.

Programatzerakoan, komeni da beti gogoan izatea honako puntu hauek:

- Programak idaztearen eta bestelako edozein dokumentu idaztearen artean diferentzia nabarmenak daude. Baina haxe da seguru asko garrantzitsua: programek sintaxi perfektua behar dute izan, funtzionatuko badute. Sintaxi-aldaketa txikienak programaren semantika erabat alda lezake.
- Perl lengoaiaren sintaxia ikastea erraza da. Edozein lengoiaia naturalena, edozein giza hizkuntzarena, baino askoz errazagoa.
- Perl malgua da eta espresio forma ezberdin asko onartzen ditu, hau da, gauza bera esateko modu bat baino gehiago daude (ia) beti.

1.3. NOLA IRAKURRI LIBURU HAU

Programatzen ikasteak badu bizikletaz ibiltzen ikastearekin antzekotasunik. Teoria modurik argienean azalduta ere, ez dago bizikletaz ikasterik lehenago edo beranduago bizikleta gainean jarri gabe. Modu berean, programazioan trebatu nahi duenak ere ez dauka ordenagailuaren aurrean jarri eta programak idaztea beste biderik. Animatzeko esango dizuegu ez dugula inor ezagutzen behar bezala ahalegindu eta gero bizikletaz ibiltzen ikasi ez duenik.

Ideala litzateke liburua bera ordenagailuaren aurrean irakurtzea, programak eta ariketak ikusi ahala probatuz. Gure aldetik, ahalik eta ariketa gehien jartzen ahalegindu gara, eginez ikasten dela erabat sinetsita. Liburuko programak bi multzotan banatu ditugu: **adibideak** eta **kasu praktikoa**. Lehenbizikoak kode puska txikiak dira, Perl-en zenbait ezaugarri erakusteko baliagarri, edo ariketa txikien soluzio gisa jarriak. Bigarren multzoan, **kasu praktikoa**, erabilgarriak izan daitezkeen programak sartu ditugu, gehienak testu digitalen tratamenduarekin lotutakoak. Azken multzo horretako programak luzeagoak ere badira kode-lerroei begiratuz gero.

Bateko zein besteko, ariketa dezente topatuko dituzue liburuan zehar. Esan gabe doa, ariketak egin ditzazuen jarri ditugu. Baina gogoan izan helburu bera lortzeko modu bat baino gehiago izango direla ia beti, eta guk nahiago genuke ariketa ebazteko zure bidea urratzen ahalegintzea, guk proposatutakoari beti zintzo-zintzo jarraitzea baino.

Guk, badaezpada ere, proposatzen dizkizuegun ariketetan, «gelditu eta hartu tarte bat pentsatzeko» esan nahi duen ikur berezi bat jarri dugu:



.

Badakizue, beraz, zer egin behar duzuen ikur horrekin topo egiten duzuen aldiro: zeuen denbora hartu eta proposatutako ariketa egin.

Hasieran nahikoa izango duzue adibide gisa jarritako programak bere horretan idatzi eta exekutatzearekin. Baina konfiantza hartu ahala, ahalegindu programaren kodean aldaketa txikiak egin eta horien eraginak aztertzen. Gehitu elementu berriak. Aldatu eta ezabatu. Ez izan esperimentatzeari beldurrik.

Hiru aukera zabaltu ditugu liburuan zehar azaltzen diren programak eta fitxategiak lortzeko:

- Liburuarekin batera datorren CDa.
- *Otarrea* Internetzerbitzua. Unibertsitate alorreko lan, txosten, apunte eta bestelako dokumentuak partekatzeko gunea da hori.
<http://www.unibertsitatea.net/otarrea/ingeniaritza-eta-teknologia-1/informatika/tape-testuak-lantzeko-perl-erremintak/>
- Testuak_lantzen bloga. Liburuko kasu praktiko guztiak azaltzen dira hor, blogeko mezu bakoitzean programa bat. Programa horietako bakoitzerako hiru atal aurkituko dituzu: enuntziatua, erabilera-adibide bat, eta kodea.
<http://www.unibertsitatea.net/blogak/testuak-lantzen>

Beraz, liburuko programa bat martxan jarri nahi izango duzunean, dena eskuz kopia beharrik ez duzu. Askiz izango da hiru toki horietako batetik eskuratu, eta gero zuzenean martxan jartzea.

Liburua irakurtzeko ordenari dagokionez, norberaren esku dago nola egin: hasieratik bukaerara jarraian irakurri, edo aurrera eta atzera saltoka. Gure aldetik, gomendio hau ematea besterik ezin dezakegu egin: programatzen esperientziarik ez duenak hobe du orrialde-zenbakiak markatutako ordenari jarraitu. Gure asmoa azalpenak modu progresiboan ematea izan da, eta atal bat ulertu ahal izateko haren aurrekoak irakurtzea komeni da lehenbizi.

Programatzen ikastea eta bizikletaz ibiltzea lotu ditugu lehentxeago. Esan bezala, bizikleta hartu eta bidera ateratzea beste erremediorik ez dugu izango ikasi ahal izateko. Baina ez da komeni errepidera ateratzerik aurrez pedalei nola eragin edo frenoak non dauden jakin ezean. Hori da lehenbiziko kapituluetan egingo duguna hain zuzen, oinarrizko nozio teorikoez jabetzea, bide bazterrean ez aurrera eta ez atzera gelditu ez gaitezen behar-beharrezkoak direnak.

Bizikleta hartzen dugun lehenbiziko aldia bada, nahiko lan izango dugu oreka mantendu eta pedalei eragiten. Aurretik noizbait ibilia denak errazago dominatuko du, eta berehala ibilbide luzeagoak egin nahiko ditu normala den bezala. Horregatik, ibilbide ezberdinak topatuko dituzue liburuan barna: testu orokorra, programatzen inolako esperientziarik ez daukan hura erreferentzi gisa hartuta idatzi dugu. Baina ibilbidea ez da beti laua izango, bere gorabeherak izango ditu, eta tarteka mendateren bat edo beste ere bai. Mendateak, txirrindularitzan bezala, hiru maila-takoak izan daitezke: hirugarren, bigarren edo lehen mailakoak (gogortasun mailaren arabera, hirugarren mailakoak xamurrenak eta lehen mailakoak gogorrenak izanik), eta hasiera eta amaiera zehazteko ikur berezi hau erabili dugu:

2. maila



Liburuaren ibilbide orokorrari jarraitzeko ez dago nahitaez mendate guztiak igotzen ibili beharrik. Norberak ikusiko du noiz igo eta noiz ez. Fresko eta gogotsu sentitzen bazarete, animo eta segi gora. Indarrez juxtu antzean dabilenak hobe izango du mendateaz paso egin eta ibilbide orokorrari jarraitu. Zer topatuko du mendatea igotzen duenak? Bada, mendateetan Perl lengoaiaren ñabardurak eta ezaugarri bereziak landuko dira, horren ohikoak ez direnak; ariketa konplexuagoak planteatu eta beren ebazpena aztertu, edo lehendik ikusitako programa bati beste buelta bat ematen saiatuko gara.

Lehenbiziko itzulian ez bada ere, bigarrenean egin igotzeko ahalegintxoak, geure burua zailtzeko entrenamendu hoberik ez dago eta.

Itzuli on.

2. Ezertan hasi aurretik

Irakurle zaitugun horren lehenbiziko programazio-lengoaia Perl dela suposatuko dugu. Sarreran esan bezala, programazio-lengoiak, konputagailuari zer eta nola egin behar duen deskribatzeko erabiltzen diren lengoiak dira. Gaur egun, programazio-lengoaia ugari topa daitezke, bakoitza bere ezaugarri eta erabilera-eremu konkretuekin. Dena den, konputagailuek, finean, lengoaia bakarra ezagutzen dute: makina-lengoaia deritzona. Izenak berak iradokitzen duen moduan, makina edo konputagailuarentzat espresuki sortutakoa da makina-lengoaia, guretik arras urrun dagoena. Horregatik, gure ideia eta aginduak konputagailuari errazago adierazi ahal izateko, ulerterrazagoak diren goi-mailako programazio-lengoiak asmatu ziren tarteko soluzio bezala. Goi-mailako programazio-lengoiak gure lengoaia naturale-tik urrun daude, baina konputagailuaren makina-lengoaia baino askozaz ere erabilerrazagoak dira. Azken batean, edozein programazio-lengoaia erabiltzean, konputagailuari gure ideiak deskribatzen dizkiogu lengoaia horretako adierazpenak erabiliz; konputagailuak, ondoren, makina-lengoiara itzuliko ditu berak ulertu eta exekutatu ahal izan ditzan.

Iritsiko da noizbait konputagailuekin komunikatzeko lengoaia naturalak erabili ahal izango ditugun eguna, baina ordura arte Perl aukera egokia izan daiteke gure asmoetarako.

Perl, goi-mailako programazio-lengoaia interpretatua da, eta programa bat martxan jartzeko bi pausoko prozesuari jarraitu behar diogu:

- **Edizioa:** aginduak idatzi. Perl lengoiako aginduak testu-fitxategi batean idatziko ditugu.
- **Exekuzioa:** aginduak gauzatu. Guk idatzitako programa makina-lengoiara itzuli eta aginduak gauzatu ditugu Perl interpretatzaileak.

Gure programa guztiek, beraz, editatu-exekutatu 2 pausoko zikloari jarraituko diote.

2.1. ZER BEHAR DUT LANEAN HASTEKO?

Ordenagailua, noski. Berdin dio nolako konputagailua edo sistema eragilea erabiltzen duzun, guztientzat baitago Perl eskuragarri. Dena den, gure azalpenak

Windows edo Linux sistema eragilea daukaten horiengana bideratuko ditugu. Sistema bat ala bestea erabili, ez dago diferentzia nabarmenik ikusiko duzuen bezala.

Badira oinarritzko hiru tresna liburuan zehar etengabe erabiliko ditugunak. **Testu-editorea** da bat. Programa bat idatzi ahal izateko, lehenbiziko pausoa aginduak testu-dokumentu batean idaztea izango da, eta horretarako erabiliko dugu, hain zuzen ere, testu-editorea. Aginduak idatzi ostean, agindu horiek irakurri, makina-lengoiara itzuli eta exekutatu ahal izateko tresna behar dugu: **Perl interpretatzailea**. Bera izango da gure ezinbesteko bigarren tresna. Eta hirugarrena eta azkena: **komando-lerroa**, gure programak abiarazi eta emaitzak pantailaz ikusi ahal izateko erabiliko duguna. Segidan datozen ataletan, hasieran liburu honekin batera erabilgarri dagoen materiala deskribatuko da, eta ondoren tresna bakoitzaren oinarritzko funtzionamendua azalduko dugu.

2.2. MATERIALA

Liburuan zehar tresna, fitxategi eta ariketa franko ikusiko ditugu. Hargatik, egokia iruditu zaigu guztiak CD batean bildu eta eskuragarri jartzea. Liburuarekin batera topatuko duzun CDa honela antolatu dugu:

- **perl:** Perl interpretatzailea, 5.10 bertsioa (Linux erabiltzaileek ez dute hau instalatu beharrik)
- **notepad++:** Notepad++ testu-editorea, 5.4.3 bertsioa (Linux erabiltzaileek ez dute hau instalatu beharrik)
- **programak:** liburuko adibide eta kasu praktiko guztiak
- **fitxategiak:** programetan sarrerako datu gisa erabilitako fitxategiak

Lehen biak, Perl interpretatzailea eta Notepad++ testu-editorea, Internetetik jaitsi ditzakegu zuzenean, eta lana erraztearren soilik gehitu ditugu CDan. Tresna bakoitzaren instalazio eta erabilera-argibideak 2. kapituluan ematen dira zehatz-mehatz. Linux erabiltzaileek, ordea, ez dute ezer instalatu beharrik izango, Linux banaketa gehientsuenek berekin baitaramatzate Perl interpretatzailea eta test-editore hauetako bat: Emacs, gedit, kedit, Vi edo Pico.

Liburuan azaltzen diren programa guztien kodea topatuko du irakurleak **programak** karpetan. Azkenik, **fitxategiak** karpetan, gure programak elikatzeke erabiliko ditugun testu-fitxategi guztiak bildu ditugu.

2.3. KOMANDO-LERROA

Ingurune grafikodun sistema eragilea erabiltzen baduzu, baliteke **komando-lerroa** edo **shell**-a zer den ez jakitea. Leiho eta ikonoen erabilera zabaldu baino lehen, konputagailuan eragiketa guztiak komando-lerroaren bitartez egiten ziren, aginduak

teklatu bidez idatziz (gaur egun ikonoen gainean klik eginez). Linux erabiltzaileen artean zale ugari ditu oraindik komando-lerroak; Windows sisteman, aldiz, haren erabilera ez da oso ohikoa.

Kontuak kontu, Linux zein Windows erabiliz, Perl lengoaiarekin lan egiteko komando-lerroa behar dugu, eta atal honetan topatuko duzu hura erabiltzeko behar duzun guztia.

2.3.1. Komando-lerroa abiarazi

Komando-lerroaren leihoa zabaltzeko egin beharreko pausoak ez dira berdinak Windows-en eta Linux-en.

Windows

Windows erabiltzen ari bagara bi aukera dauzkagu. Bata:

Hasi->Programa Guztiak->Gehigarriak->Komandoaren Gonbita

Inicio->Programas->Accesorios->Simbolo del sistema

Start->All Programs->Accesories->Command Prompt

Edo beste hau, laburragoa dena:

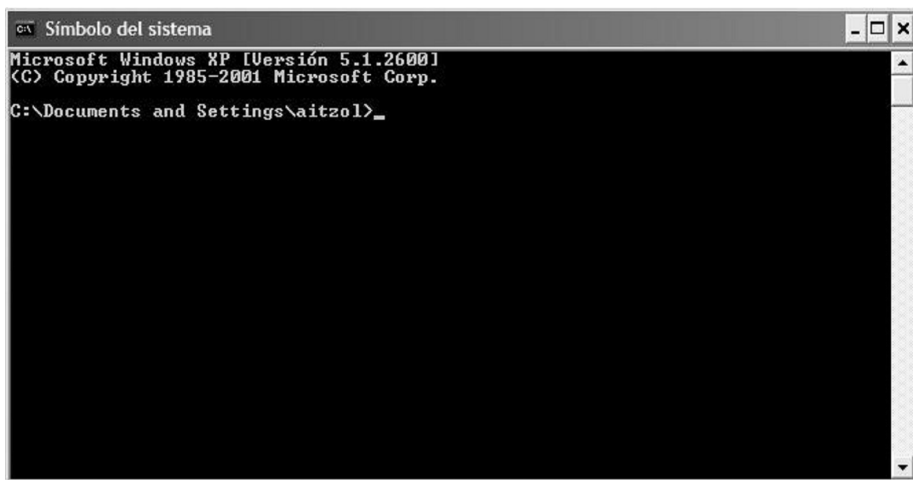
Hasi->Exekutatu

Inicio->Ejecutar

Start->Run

Eta agertuko zaigun leihoan idatzi *cmd*.

Modu batean ala bestean, ondoko irudiaren pareko leiho beltz bat azalduko zaigu:



Linux

Gure sistema eragilea Linux bada, programen artean hauetakoren bat topatu eta abiatu, bertsio eta hizkuntzaren arabera: *terminala*, *terminal*, *xterm*, *console*, *console*.



2.3.2. Oinarrizko aginduak

Gure programazio-lanetan, eragiketarik gehienak komando-lerroaren bitartez gauzatuko ditugu: fitxategiak editatu, programak abiarazi, gelditu, direktorioz mugitu, etab. Aurretik inoiz erabili ez duenari arrotza suertatuko zaio lan egiteko modu hau hasieran, baina lasai, hiruzpalau komando edo agindu baino ez ditugu beharko, eta pausoak behin eta berriz errepikatuko ditugunez berehala ohituko zarete.

Sistema eragile batetik bestera komando-lerroaren itxura eta aginduak aldatu egiten dira. Lehenbizi, Windows sistema oinarri hartuta emango ditugu azalpenak, eta, ondoren, atalaren amaieran, Linux-erako komandoak ikusiko ditugu.

Windows

Komando-lerroa zabaltzean, *prompt* edo indikatzailak uneko kokapena zein den erakusten digu:

c:>Documents and Settings\aitzol> Hau da, C disko edo partizioan, *Documents and Settings* direktorio edo karpeta *aitzol* karpeta da uneko gure kokapena.

Esan bezala, komando-lerroarekin lan egiteko agindu gutxi batzuk baino ez ditugu beharko:

cd: uneko direktorioa aldatzeko.

Adibidez "ariketak" direktoria sartu nahiko bagenu:

```
c:>Documents and Settings\aitzol>cd ariketak
```

Komando-lerroan idatzi eta "ariketak" karpetan kokatuko ginateke.

```
c:>Documents and Settings\aitzol\ariketak>
```

Uneko direktoriotik atera nahiko bagenu berriz:

```
c:>Documents and Settings\aitzol\ariketak>cd ..
```

Goiko agindua idatzi eta hasierako direktoria bueltatuko ginateke.

```
c:>Documents and Settings\aitzol>
```

Direktorio batetik bestera mugi gaitzkeen bezala, posible da unitate⁴ batetik bestera jauzi egitea ere. Adibidez, demagun *C* diskotik *D* diskora pasatu nahi dugula (lehenbizi begiratu ea badugun *D* bezala etiketatutako diskorik). Nahikoa genuke komando-lerroan unitate izena eta jarraian bi puntu (*d:*) idaztearekin (*cd* agindua ez da beharrezkoa kasu honetan):

```
c:>Documents and Settings\aitzol>d:  
D:\>
```

Unitatez aldatu eta *D*-n kokatu gara. Berrir *C*-ra pasatzeko:

```
D:\>c:  
c:>Documents and Settings\aitzol>
```

mkdir: direktorio edo karpeta berria sortzeko.

Uneko direktorioaren barnean "nerePerl" izeneko karpeta sortu nahi badugu:

```
c:>Documents and Settings\aitzol>mkdir nerePerl
```

dir: uneko direktorioan dauden fitxategi eta karpeten zerrenda erakusten du.

```
c:>Documents and Settings\aitzol>dir
```

"aitzol" karpetako elementuak bistaratuko lituzke.

4. Ordenagailuko disko unitate bakoitzak bereizgarri bezala letra bat dauka. *A* eta *B* letrak disketeentzat daude erreserbatuak. *C* disko gogor printzipalari dagokio. Disko gogor gehiago baldin baditugu, *D*, *E*, *F* etiketak jasoko dituzte hurrenez hurren. Ondoren, *CD*, *DVD* eta gainerako disko unitateei dagozkien letrak etorriko dira, bakoitza dagokion letrarekin.

Linux

Linux sistemaren erabiltzaileentzat komando-lerroaren *prompt* edo indikatzailea ezberdina izango da, baita zenbait komando-izen ere. Gainerakoan, azaldu dugunak berdin balio du sistema baterako zein besterako. Ondoko taulan Windows-eko komando erabilienak azaltzen dira, Linux-erako agindu parekideekin batera:

Windows	Linux	Azalpena
cd	cd	Uneko direktorio edo karpeta aldatu
cls	clear	Pantaila garbitu
dir	ls	Direktorio edo karpeta edukia bistaratu
mkdir	mkdir	Azpidirektorio edo karpeta berria sortu
type	more	Fitxategi baten edukia bistaratu pantailan
del	rm	Fitxategia ezabatu
rmdir	rmdir	Direktorio edo karpeta ezabatu

Ariketa:

Liburu praktikoa da esku artean daukazuna, eta kapituluetan aurrera egin ahala programa mordoxka egitea egokituko zaizu. Ideia ona litzateke Perl programa guztiak aparteko karpeta batean gordetzea. Hemendik aurrera *nerePerl* izeneko karpeta erabiliko dugu Perl lengoaiaz idatzitako programak gordetzeko. Horretarako, komando-lerroa erabiliz lehenbizi *nerePerl* karpeta sortu, eta ondoren sortu berri duzun direktoria mugitu. Amaieran, komando-lerroak honako itxura izango du (zure direktorioaren helbide edo *path*-arekin, jakina):

Windows-en:

```
c:>Documents and Settings\aitzol\nerePerl>
```

Linux-en:

```
aitzol@ordenagailua:~/nerePerl$
```

Laguntza:

Direktorio batetik bestera mugitzeko *cd* eta ondoren izen osoa idaztea neko-soa gerta daiteke. Adibidez, demagun *C:>* direktorioan gaudela (erro direktorioan) eta *Documents and Settings* direktoria sartu nahi dugula. Izen osoa idazten ibili beharrik gabe, nahikoa litzateke *cd Do* idatzi eta tabuladore tekla (→|) sakatzea, eta berak osatuko luke gainerako zatia. *Do* letrez hasten diren direktorio bat baino gehiago izanez gero, tabuladore tekla sakatzean haien artean txandakatuko da, eta guk nahi dugun hura agertzean *Enter* tekla sakatu eta kito. Truku honek Windows-eko komando-lerroan izen luzeekin izan ditzakegun problema asko aurreztuko dizkigu.

2.4. PERL INTERPRETATZAILEA

Gure ordenagailuak Perl lengoaian idatzitako programak ulertuko baditu, ezinbestekoa da aurrez Perl interpretatzailea deituriko programa instalatua izatea. Linux erabiltzaileek ez dute ezer instalatu beharrik izango, banaketa gehientsuenek berekin baitaramate Perl interpretatzailea. Dena den, erraza da gure sisteman Perl daukagun ala ez konprobatzea. Zabaldu komando-lerroa⁵ eta idatzi honakoa bertan (Windows zein Linux):

```
>perl -v
```

Erantzuna `This is perl, v5.10.0` testuarekin hasten bada, zorionak, dagoeneko badaukazu Perl zure sisteman. Horrela ez bada, lasai, oraintxe azalduko dugu-eta pausoz pauso nola instalatu.

Merkatuan Perl interpretatzailearen bertsio bat baino gehiago daude gaur egun. Zein aukeratu? Bi ezaugarri hartu ditugu kontuan aukeraketa egiterakoan: doakoa izatea eta, hutsetik abiatzen garela kontuan izanda, erabilerraza izatea. Programazioan aditu garenean, izango dugu astirik interpretatzaile profesionalagoak probatzeko. Horrela, bada, bi ezaugarri horiek kontuan hartuta *ActiveState* etxearen *ActivePerl* interpretatzailea izan da lan egiteko aukeratu duguna. Programa horrekin, gure Windows makinak Perl lengoaiaren aginduak ulertu eta idazten ditugun *script*-ak⁶ exekutatu ahal izango ditu.

2.4.1. Instalatzeko pausoak

ActiveState etxeak doako Perl interpretatzailea (*ActivePerl*) eskaintzen du bere webgunean. Gunea ingelesez dagoen arren, ez da batere zaila programa topatu eta jaistea. Hona jarraitu beharreko pausoak:

5. Laburdura gisa *prompt*-a adierazteko ">" ikurra erabiliko dugu hemendik aurrera.

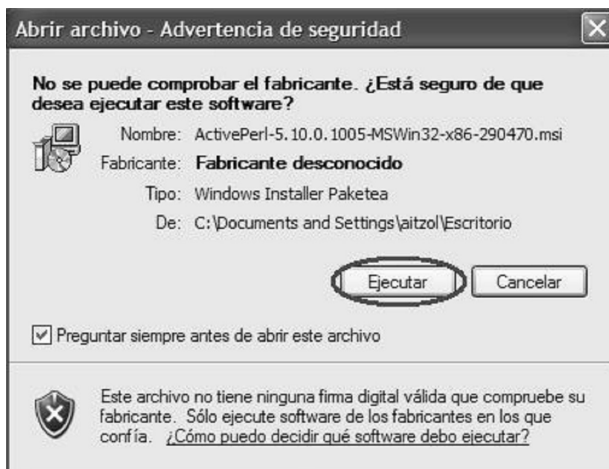
6. *script* eta *programa* termino baliokide gisa erabiliko ditugu testuan zehar.

1. Nabigatzailea zabaldu, eta idatzi helbide barran honakoa:

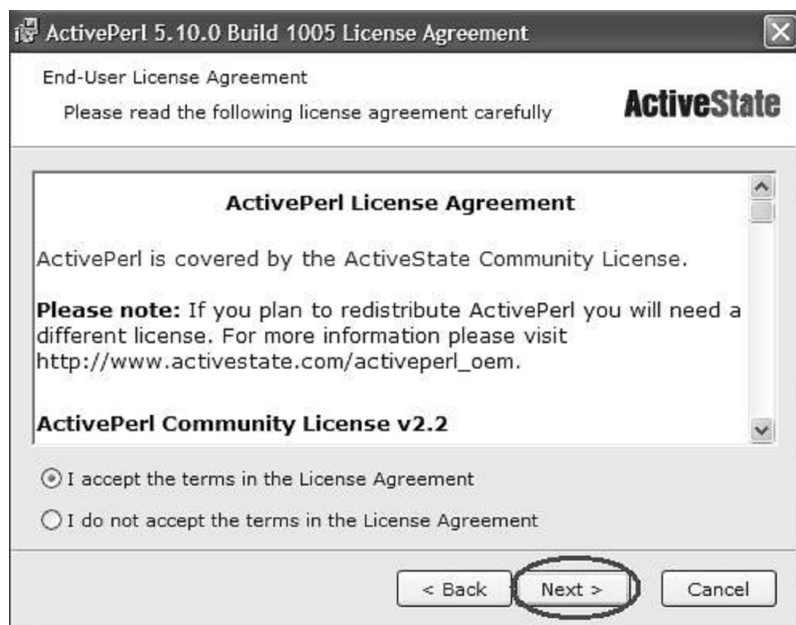
<http://www.activestate.com/activeperl/>

ActiveState webguneko *ActivePerl* atalera eramango gaitu zuzenean. *ActivePerl* produktua identifikatu eta *Download* jartzen duen lekuan klik egin.

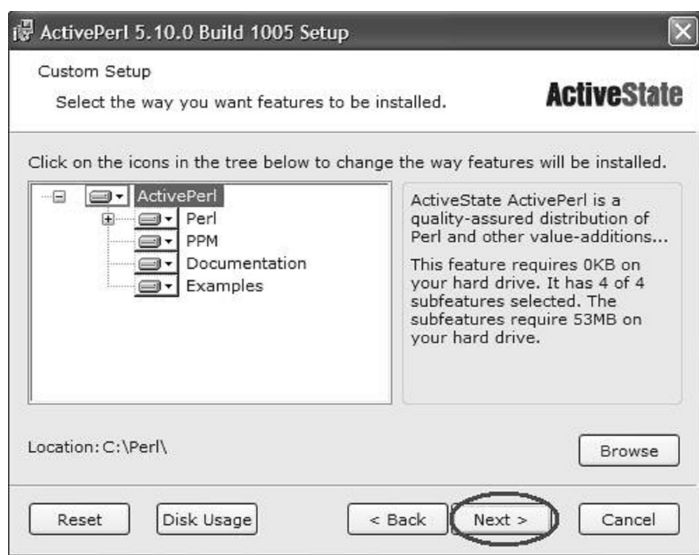
2. Programa jaisteko leihoa agertuko zaigu. Gorde fitxategia disko gogorrean (*Mahaigainean* edo beste nonbaiten). Jaitsi ondoren, bere gainean klik bikoitza egin eta Perl instalatzailea abiaraziko da.



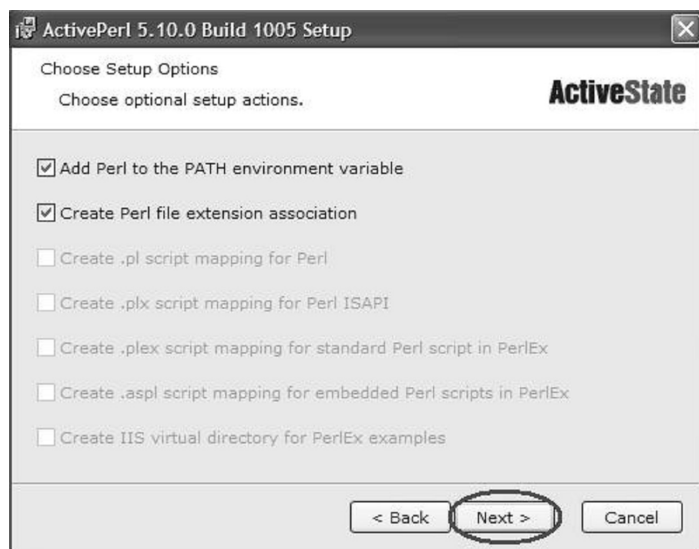
3. Itxaron pixka bat instalatzailea abiatu arte. Instalazio-prozesua oso erraza da: *Next* botoia sakatzea besterik ez da egin behar agertzen den leiho bakoitzean.



4. Pauso honetan *ActivePerl*-en instalazio-aukerak zehazteko aukera daukagu, baita Perl non instalatu nahi dugun adierazteko ere. Programak instalatzen esperientzia baldin badaukazu, eta lehenetsi gisa datorren `c:\Perl` instalazio-direktoria aldatzea nahiko bazenu, haxe da momentua. Printzipioz, dagoenean utzi eta aurrera jarraituko dugu Next botoia sakatuta.



5. Azken pantaila honetako aukerak ere bere horretan utziko ditugu. Hala ere, badaezpada, ziurtatu lehen biak markatuta dauzkazula, azpiko irudian agertzen den bezala. Ondoren sakatu *Next*.

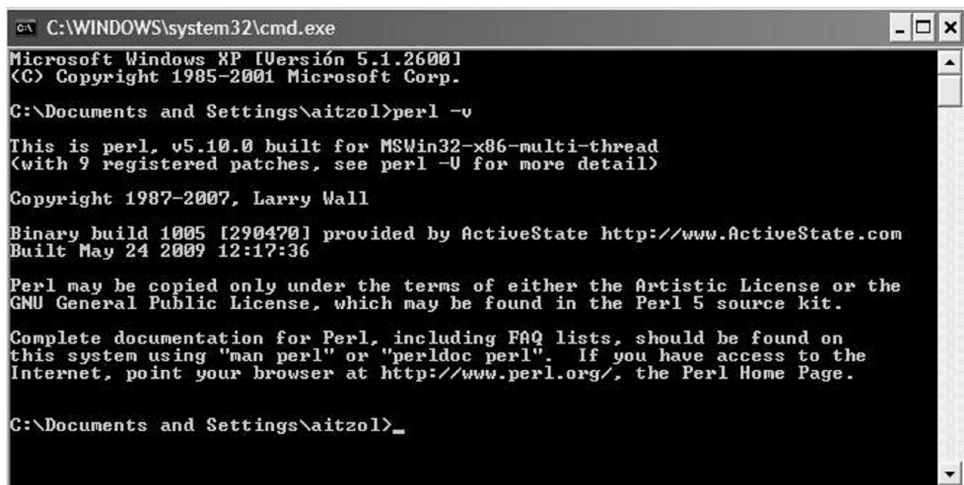


6. Puntu honetara iritsita, *ActivePerl* instalatzaileak badauka programa instalatzeko behar duen informazio guztia. *Install* botoia sakatzean instalazio-prozesua abiatuko du. Denbora-tarte baten ondoren prozesua amaitu duela adieraziz leiho bat agertuko zaigu pantailan. *Finish* botoian klik egin eta kito, Perl interpretatzaile berri-berria daukagu gure konputagailuan.

Instalazioa behar bezala burutu dela egiaztatzeko, komando-lerroa zabaldu eta idatzi honakoa bertan:

```
>perl -v
```

Aurretik esan bezala, `This is perl` eta ondoren bertsio-zenbakia erakutsiz erantzuten badigu Perl-ek, dena ondo egin dugun seinale.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\aitzol>perl -v

This is perl, v5.10.0 built for MSWin32-x86-multi-thread
(with 9 registered patches, see perl -U for more detail)

Copyright 1987-2007, Larry Wall

Binary build 1005 [290470] provided by ActiveState http://www.ActiveState.com
Built May 24 2009 12:17:36

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl". If you have access to the
Internet, point your browser at http://www.perl.org/, the Perl Home Page.

C:\Documents and Settings\aitzol>
```

2.5. TESTU-EDITOREA

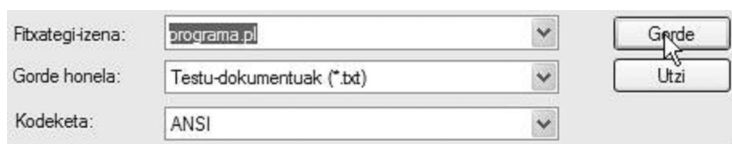
Programa edo *script*-ak idazteko erabiliko dugun tresna da testu-editorea. Programa, Perl lengoaian idatzitako agindu sorta da, eta lehenbizi agindu horiek testu-fitxategi batean gorde behar dira ondoren exekutatu ahal izateko. Eskura daukagun edozein testu-editore erabil dezakegu lan honetarako, betiere testu hutsa, formaturik gabe, gordetzeko aukera ematen baldin badu. Hala ere, biziki gomendatzen dugu Microsoft Word edo OpenOffice Writer testu-prozesadoreak, esaterako, ez erabiltzea. Izan ere, prozesadore horiek testuarekin batera informazio gehigarria gordetzen dute eta arazo-iturri bihurtu ohi dira programazio-lanetan erabiltzean. Merkatuan programaziorako editore bereziak dauden arren, mementoz ez dugu tresna berririk instalatuko, eta gure hautua konputagailuak berak dakarren testu-editore sinple eta erabilerrazaren aldekoa izango da. Ahalik eta indarrik gutxien gastatu nahi genuke tresna berriak ikusi eta ikasten, eta ahalik eta gehien gorde Perl lengoia ikasteko. Aurrerago izango dugu astirik editore aurreratuagoak probatu eta erabiltzeko.

Sistemaren arabera, testu-editore hauetakoren bat erabil genezake:

Windows: Notepad (Ohar-bloka), Wordpad

Linux: Emacs, Vi, Pico, gedit, kedit

Gure aukera edozein delarik ere, programak idazteko ez da gauza handirik jakin behar: dokumentu bat zabaldu, testua idatzi, aldaketak gorde eta itxi. Idatzitakoa beti testu-dokumentu bezala gorde behar da, ".pl" luzapenarekin. Luzapen hori ez da nahitaezkoa, baina bai gomendagarria. Izan ere, Windows sistema eragileak fitxategiaren luzapenari begiratzan dio bere edukiaren berri jakiteko (*doc* Word fitxategien kasuan, *mp3* soinu-fitxategiak, etab.), eta *pl* luzapena jarritz gero, Perl programa gisa identifikatu eta besteetatik bereiziko du ikono propioarekin.



2.6. LEHEN PROGRAMA

Programatzeko behar ditugun tresna guztiak inguratu ditugu jada. Prest gaude, beraz, amorratzen ere bai, gure lehen programa idazteko. Ez gaitezen, bada, alferrikako luzamenduetan galdu. Aurretik esan bezala, bi pausoko prozesuari jarraituko diogu:

1. **Edizioa:** Perl lengoaiako aginduak erabiliz programa idatzi testu-fitxategian.
2. **Exekuzioa:** komando-lerrotik Perl interpretatzaileari dei egin programa exekuta dezan.

2.6.1. Lehen pausoa: edizioa

Lehen programa idazteko, zabaldu komando-lerroaren leihoa eta *cd* komandoa erabiliz mugitu *nerePerl* direktoria. Hemendik aurrera programa guztiak direktorio horretan gordeko ditugu, gainerako fitxategietatik aparte. Laneko direktorioan gaudela, programa-fitxategia editatuko dugu honakoa idatziz komando-lerroan:

Windows

```
>notepad kaixoMundua.pl
```

Linux

```
>emacs kaixoMundua.pl
```

Aurreko aginduak *kaixoMundua.pl* testu-fitxategia zabalduko du testu-editorearekin (NotePad programa Windows-en, eta Emacs, berriz, Linux-en), bertan gure kodea idatzi ahal izateko. Windows erabiltzaileei, *kaixoMundua.pl* izeneko fitxategirik existitzen ez bada, NotePad-ek ea izen horrekin fitxategi berri bat sortu nahi duten galdetuko die. Baietz erantzun.

Fitxategi zabaldu berrian, idatzi zehatz-mehatz ondoko lerroak:

```
#!/usr/bin/perl
use strict;
use warnings;
# gure lehen programa
print "Kaixo, Mundua!\n";
```

Idatzitako programa gordetzeko jarraitu pauso hauei:

NotePad:

Fitxategia—>*Gorde*

Segidan, NotePad programa itxi eta komando-lerrora itzuli ahal izateko:

Fitxategia—>*Atera*.

Emacs:

File—>*Save (current buffer)*

Edo teklatura erabiliz, *Ctrl-c Ctrl-s* (*Ctrl* sakatuta mantendu, eta lehenbizi *c* eta ondoren *s* teklak sakatu).

Emacs programa itxi eta komando-lerrora bueltatzeko:

File—>*Exit Emacs*

Teklatua erabiliz, *Ctrl-x Ctrl-c* (*Ctrl* sakatuta, lehenbizi *x* tekla eta ondoren *c*).

Behin komando-lerrora bueltatuta, exekutatzeko prest dago *kaixoMundua.pl* gure lehen programa. Ez pentsa edozein programa denik, ez gero! *KaixoMundua* programa berezia da, programatzaile ororen bataio informatikoa. Ez beraz utzi lehen adibide hau egin gabe!

Idatzitako lerroei antzik hartzen ez badiezu, ez arduratu, berehala aztertuko dugu-eta lerroz lerro *kaixoMundua.pl* programa. Baina lehenbizi exekuta dezagun zer egiten duen ikusteko.

2.6.2. Bigarren pausoa: exekuzioa

Exekuzio-fasean, Perl lengoaiari idatzitako programa interpretatzaileari pasatu eta honek makina-lengoiara itzuliko du konputagailuak uler dezan. Konputagailuak, aginduak jaso eta exekutatu egingo ditu. Hau guztia, komando-lerroko agindu bakar batean biltzen da:

```
>perl kaixoMundua.pl
```

Honakoa jarrita ere nahikoa litzateke Windows sistemetan:

```
>kaixoMundua.pl
```

Dena ondo joan bada, konputagailuak *Kaixo, Mundua!* mezua idatzi behar luke pantailan. Tamalez, nahikoa izan ohi da errore txiki bat programaren exekuzioa eten eta pantailan ingelesez mezu ulertezin bat ager dadin. Hala gertatu bada, lasai, bi errore hauetako batek izan behar du eta: lehenbizi, egiaztatu *kaixoMundua.pl* izeneko fitxategia sortu duzula, baliteke idaztean nahastu eta beste izen bat eman izana, *kaxoMundua.pl* edo *kaixomundua.pl* adibidez (Linux-ek letra larrien eta xeheen arteko bereizketa egiten du). Fitxategiari jarri diogun izena eta komando-lerrotik deitzean erabili duguna berdin-berdinak izan behar dira. Erabili `dir` agindua uneko direktorioan izen horretako fitxategirik baden ala ez egiaztatzeko. Fitxategiaren izena zuzena izanik programak oraindik funtzionatzen ez badu, editatu berriro fitxategia NotePad edo Emacs erabiliz eta egiaztatu lerroz erro agindu guztiak behar bezala idatzi dituzula. Baliteke idaztean akatsen bat egin izana.

3. maila

Plataforma ezberdinak

Programa bat exekutatu nahi dugun aldiro komando-lerroan **Perl** eta ondoren programaren izena idazten aritzea ez da beharrezkoa. Nola egin idatzi beharrik izan ez dezagun? Tira, erantzuna sistema eragilearen arabera da.

Windows

Windows-ek fitxategiaren luzapenari begiratzen dio fitxategi hori zein programarekin zabaldu edo exekutatu behar duen jakiteko. Gure programa exekutatzeke nahikoa litzateke komando-lerroan honakoa idaztea:

```
>kaixoMundua.pl
```

Fitxategiaren luzapena begiratzean **".pl"** topatu eta Windows-ek zuzenean fitxategia **Perl** interpretatzaileari pasatuko dio, exekuta dezan. Kontuz, soilik **".pl"** luzapena daukaten fitxategiekin funtzionatzen du honek!

3. Oinarrizko programazioa

3.1. AGINDUAK ETA SENTENTZIAK

Programak edo *script*-ak, lengoaia jakin batean elkarren segidan idatzitako aginduak baino ez dira. Programako kode-lerro bakoitzari sententzia deitzen zaio, eta konputagailuak gauzatu beharreko agindua da funtsean («egin hau» edo «idatzi mezu hori» gisakoak). Sententzia bakoitza aparteko lerro batean idatzi ohi da, nahiz eta sententzia batek lerro bat baino gehiago izan ditzakeen. Sententzia, ia beti, puntu eta koma ikurraz (;) ixten edo amaitzen da.

programa.pl

```
#!/usr/bin/perl

sententzia1;
sententzia2;
...
sententzian;
```

Programa bat exekutatzean, Perl interpretatzaileak komando-lerroan zehaztutako fitxategia zabaldu eta bertako aginduak banan-banan exekutatuko ditu, lehen lerrotik hasi eta azkenekora iritsi arte. Hori bezain sinplea da prozedura.

Jar dezagun adibide bat, eta bide batez, gure bigarren programa idatzi. *kaixo.pl* deituko diogu eta bi sententzia izango ditu. Gogoratu programa bat idazteko eman beharreko bi pausoak. Lehenbizi komando-lerroan *kaixo.pl* izeneko fitxategia zabalduko dugu editorearekin (Notepad edo Emacs):

```
>notepad kaixo.pl
```

Fitxategian honako lerro hauek idatziko ditugu segidan (goiburukoa, ***kaixo.pl***, fitxategiaren izena da eta ez da testuan idatzi behar):

kaixo.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# gure bigarren programa
print("Kaixo!"); # mezua pantailaz bistaratu
print("Hau da gure\n bigarren programa"); # mezua bi lerrotan
```

Idatzi berri duguna exekutatzeko, komando-lerrora joan, eta idatzi perl kaixo.pl. Honako mezua hau jaso beharko zenuke pantailaz:

```
>perl kaixo.pl
Kaixo!Hau da gure
  bigarren programa
>
```

Azter dezagun programa lerroz lerro:

```
#!/usr/bin/perl
```

Lehenbiziko lerroa, *shebang*-a, gure programak Windows zein Linux sistemetan arazorik gabe funtziona dezan jarri beharrekoa da. Gure programa guztien lehen lerroa *shebang*-a izango da beti.

```
# gure bigarren programa
```

Bigarren lerro hau iruzkina da eta Perl interpretatzaileak ez dio kasurik egingo. Programari buruzko argibideak emateko erabiliko ditugu iruzkinak, bere funtzionamenduan inolako eraginik izan gabe. Lerroaren edozein puntutan idatz genitzake, kontuan izanda betiere # karakterearen atzetik datorren guztia lerro-amaiera arte Perl-ek ez duela kontuan hartuko.

```
print("Kaixo!"); # mezua pantailaz bistaratu
```

Programazio-lengoaia guztiek daukate beren lexiko edo hitz berezien multzoa. Perl lengoaian, `print` funtzio motako termino berezia da. Funtzioek ekintzak adierazten dituzte programazio-lengoaian. `print` irakurtzean, Perl-ek segidan datorren argumentua pantailaratuko du. Argumentu edo mezua komatxoaren artean jarri duguna da, *Kaixo!* goiko adibidean. Bukaerako puntu eta komak (;) agindua amaitu dela adierazten dio Perl-i. Lerroaren gainerako zatia sententziari buruzko iruzkina da, eta Perl-ek ez du interpretatuko.

```
print("Hau da gure\n bigarren programa"); #mezua bi lerrotan
```

Konturatuko zinen bezala, bigarren `print` agindu honen mezua bi lerrotan bistaratu du Perl-ek. Hala egin du tartean *return* edo lerro-jauzia adierazten duen karaktere berezia, `\n`, idatzi dugulako. Baina bestalde, lehenengo eta bigarren `print` funtzioen mezuak elkarren segidan idatzi ditu, nahiz eta guk bi `print` agindu desberdinetan idatzi ditugun mezuak. Bigarren mezua lerro berri batean agertzea nahiko bagenu, lerro-jauzi edo *return* karakterea (`\n`) erabili beharko genuke berriro.

Ondoko programak egiten du hori:

kaixo2.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# print aginduak lerro jauziekin
print "Kaixo!\n"; # lehenbiziko print agindua
print "Programa honek lerro jauziak\n erabiltzen ditu\n";
```

Programa honetan, `print` funtzioaren mezua ez dugu parentesi artean idatzi, baina portaera berbera izango da. Ez da beharrezkoa parentesiak jartzea, eta zure esku gelditzen da horiek erabili edo ez.

Komando-lerrora joan eta programa exekutatzean, haxe da lortuko dugun erantzuna:

```
>perl kaixo2.pl
Kaixo!
Programa honek lerro jauziak
  erabiltzen ditu
>
```

Badago lerro jauziarekin `"\n"` batera `print` aginduetan oso erabilia den beste karaktere berezi bat ere: tabuladorea `"\t"`. Berridatzi aurreko programa, `print` aginduaren argumentuan tabuladorea jarritz lehenbiziko karaktere:

```
print "\tKaixo!\n";
```

Gorde aldaketak eta exekutatu programa. Zein da diferentzia?

Segurtasun-kontua

Lehen bi programetan, nahita ez ditugu aipatu ere egin bigarren eta hirugarren sententziak: `use warnings` eta `use strict`. Laster konturatuko zarete oso ohikoa dela programak idaztean akatsen bat egitea. Nahikoa da karaktere bakar bat ahaztu edo behar bezala ez idaztea programa hankaz gora joan dadin.

Begiratu bestela programa hori:

kaixo3.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# programa akats eta guzti
print \tKaixo!\n Zer moduz?";
```

Exekutatzen ahalegintzean, honela erantzungo digu Perl-ek:

```
>perl kaixo3.pl
syntax error at kaixo3.pl line 5, near "\tKaixo!"
Can't find string terminator ' ' anywhere before EOF at
kaixo.pl line 3.
>
```

Ez beldurtu mezuarekin. Perl-ek sintaxi-errore bat aurkitu eta programaren exekuzioa bertan behera utzi du. Pantailaratu duen mezua arretaz irakurtzen badugu, errorea 5. lerroan (*line 5*) dagoela jakingo dugu (ez beti fidatu itsu-itsuan Perl-ek adierazitako lerro-zenbakiarekin, baliteke errorea aurreko edo ondorengo lerroan egotea). Berriro editatuko dugu *kaixo3.pl* fitxategia eta arretaz begiratuko diogu 5. lerroari. Berehala konturatuko gara `print` funtzioaren mezuari hasierako komatxoak falta zaizkiola. Horiek jarri, aldaketak gorde eta exekutatu berriro. Zer moduz?

Oraindik ez dugu azaldu, dena den, zertarako diren `use warnings` eta `use strict`. Programazioan ohikoak diren erroreak harrapatzeko behar ditugu. Gure programek izan ditzaketen akatsak identifikatzen lagunduko digute. Eta ez da laguntza makala gero! Oraingoz ez dugu bi sententzia horiei buruzko azalpen gehiagorik emango. Aztertuko ditugu zehatzago bere garaian.

Atalarekin amaitzeko, esango dugu programazio-estilo onaren erakusgarri izateaz gain, luzera begira buruko minak aurrezteko modua ere badela programa guztiak hiru lerro hauekin hastera:

```
#!/usr/bin/perl
use warnings;
use strict;
```

Ariketak:

1. Moldatu *kaixo3.pl* programa, zure izenaz agurtu zaitzan. Honelako mezua lortu beharko zenuke pantailan:

```
Kaixo, zure_izena
      Zer moduz?
```

Gorde programa berria *kaixo4.pl* izenarekin.

2. Honako programa honek erroreak eman dizkigu exekutatzekoan. Saiatu konpontzen.

```
#!/usr/bin/perl
use warnings;
use strict;
print ("Programa honek\n")    # lehenbiziko print agindua
print ("akatsak besterik \n ez ditu!");
iruzkina: bi print agindu eta 3 akats
```

3.2. DATU MOTAK

Programa baten lehengaia datuak dira. Funtsean programek datu horien gainean eragiketak egin eta emaitza berriak kalkulatu besterik ez dute egiten. Perl-ek bi multzotan banatzen ditu datuak: **string** edo karaktere-kateak batetik, eta **zenbakiak** bestetik. Atal honetan bi datu mota horietaz arituko gara, baita beraiekin gauza daitezkeen eragiketez ere.

3.2.1. String

String terminoa arrotza gerta dakigukeen arren, dagoeneko erabili dugu datu mota hau `print` aginduarekin batera. Karaktere motako datua adierazten du *string*-ak. Adibidez, *string* motako datuak dira: "Kaixo, Mundua!", "2001 Odisea espazioan", "Kale Nagusia 12, 3.C"

Karaktere motako datuak komatxo artean idatzi behar dira beti, komatxoak sinpleak edo bikoitzak izan daitezkeelarik. Adibidez: "hau da hau!" edo 'hau da hau!'. Kontuz baina, aurreko bi *string*-ak ez baitira berdinak Perl-entzat: komatxo bikoitzak erabiltzean Perl-ek tarteko karaktere bereziak (`\n` eta `\t` adibidez) interpolatu edo interpretatu egingo ditu, eta komatxo sinpleekin, aldez, ez. Programa honek lagunduko digu ulertzen ezberdintasuna:

komatxoak.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# komatxo bikoitz eta sinpleen arteko ezberdintasuna
print ("\tKomatxo bikoitzak\n");
print ('\tKomatxo sinpleak\n');
```

Komando-lerrora joan eta exekutatzean, hona emaitza:

```
>perl komatxoak.pl
      Komatxo bikoitzak
\tKomatxo sinpleak\n
>
```

Komatxo bikoitzen artean idatzitako mezuan karaktere bereziak interpretatu egin ditu Perl-ek. Komatxo sinpleekin, mezua literalki bistaratu du.

Bi eragiketa egin ahal ditugu *string* edo karaktere motako datuekin: kateaketa eta biderketa. Kateaketak bi *string* edo gehiago elkarrekin lotzen ditu eragile bezala puntu karakterea (.) erabiliz.

kateaketa.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# string-en arteko kateaketa
print ("string-ak" . " " . "eta" . " " . "zuriuneak" .
      " " . "kateatuta\n");
```

Komando-lerrotik programa abiatuz gero, pantailan string-ak eta zuriuneak kateatuta testua azalduko zaigu. Proba egin!

Biderketa-eragilea *string* jakin bat nahi beste aldiz errepikatzeko erabiltzen da. Hona hemen nola:

stringBider.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# string-en arteko eragiketak
print ("HIRU BIDER " x 3);          #biderketa
print ("\n" . "pa" . "ta" x 2);
      #kateaketa eta biderketa nahastuta
```


Exekuta dezagun zer egiten duen ikusteko:

```
>perl stringBider.pl
HIRU BIDER HIRU BIDER HIRU BIDER
patata
>
```

Lehenbiziko eragiketa *string* biderketa arrunta izan da, baina bigarrenenean kateaketa eta biderketa nahastu ditugu: lerro-jauzia kateatu "pa" *string*-arekin, eta honi kateatu bi aldiz "ta".

Bi eragileak, "." eta "x" ez dira programan komatxo artean jarri behar, karaktere arrunt bezala interpretatuko bailituzke bestela Perl-ek.

3.2.2. Zenbakiak

Oinarrizko bigarren datu mota zenbakiak dira: 60, 24, 7, 3,5, etab. Zenbakiak inprimatzeko `print` funtzioa erabiliko dugu ere, baina komatxorik gabe:

```
print(7); # 7 zenbakia inprimatu pantailan
```

Zenbakien arteko eragiketetarako ohiko eragileak ditugu, erabilienak hauexek: +, -, *, /, ** eta %. Jarraian datorren programak bakoitzaren erabilera erakusten du:

eragileak.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# zenbakien arteko eragiketak
print(2 + 4);          # Batuketa: 6
print("\n");
print(4 - 5);          # Kenketa: -1
print("\n");
print(7 * 3);          # Biderketa: 21
print("\n");
print(12 / 3);         # Zatiketa: 4
print("\n");
print(2 ** 3);         # Berreketa: 8
print("\n");
print(20 % 3);         # Hondarra: 2
print("\n");
```

Azken bi eragileak ez dira agian oso ezagunak. Bi zenbakiren arteko berreketa kalkulatzeko ** eragilea erabiltzen da. Adibidez:

```
3 ** 2          # 9 , hau da, 32
(2 ** 3) ** 2   # 64, hau da, (23)2
```

Modulua % eragileak bi zenbakiren arteko zatiketaren hondarra itzultzen du. Adibidez:

```
21 % 3      # 0, zatiketaren emaitza 7 eta hondarra 0
7 % 3       # 1, zatiketaren emaitza 2 eta hondarra 1
```

3.3. ALDAGAIAK

Aldagaiaren kontzeptua oinarri-oinarrizkoa da programazio-lengoaia guztietan. Atal honetan, zer diren eta zertarako erabil ditzakegun azaltzen saiatuko gara. Ikusi berri dugunez, gure programetan erabil ditzakegun datu motak bi dira: *string*-ak eta zenbakiak. Bada, aldagaiak, balio horiek gordetzeko edukiontziak dira. Adibide baten laguntzaz hobeto ulertuko delakoan, demagun urte oso bateko segundo kopurua kalkulatu eta bistaratzen duela gure programak agindu honekin:

```
print(365*24*60*60);
```

Zer egin, programa berean aurrerago balio hori berriro behar badugu? Berririko kalkulatzea beste erremediorik ez genuke izango. Aldagaiekin, nahi ditugun balioak ordenagailuaren memoriako txoko batean gorde eta aurrerago balio horiek berreskuratzeko modua izango dugu. Berehala ikusiko dugu nola.

Aldagaia, aurrerago ere beharko dugun balio edo espresioa gordetzeko edukiontzia izango da. Baina programan erabili ahal izateko, aurretik definitu egin behar dira aldagaiak. Definitzeak, aldagaiari izen bat ematea esan nahi du, programan zehar erreferentziatu ahal izateko. Izena jartzean, letra, zenbaki eta azpimarren konbinazioak onartzen ditu Perl-ek: *izena*, *lehen_abizena*, *NAN*. Aldagai guzti-guztiek izenaren aurretik ikur berezia daramate, aldagai mota adierazten duena. Hiru dira Perl-ek erabiltzen dituen aldagai motak: **eskalar**, **array** eta **hash** motakoak. Lehen biak ikusiko ditugu atal honetan eta hash motakoak aurreragorako utzi.

3.3.1. Eskalarrak

Aldagai eskalarretan *string* edo zenbakiak gorde ditzakegu; edozein datu mota beraz. Bereizgarri bezala, izenaren aurretik dolar karakterea "\$" daramate. Aldagai eskalarrek lirateke, adibidez: *\$izena*, *\$lehen_abizena*, *\$NAN*. Aldagai eskalarrek ez daukate muga edo toperik, edozein tamainatako datuak gorde ditzakegu bertan. Muga bakarra, gure konputagailuaren memoriak ezartzen diguna da.

Honako programa honek erakutsiko digu nola erabili aldagai eskalarrak:

eskalar.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# Aldagaiak erabiliz izena eta adina pantailaratu
my $izena = "Aitor";    # Aitor balioa gorde $izena aldagaian
my $adina = 36;         # 36 balioa gorde $adina aldagaian
print("Izena: ");
print($izena);
print("\n");
print("Adina: ");
print($adina);
print("\n");
```

Honela erantzungo du `eskalar.pl` programak exekutatzean:

```
>perl eskalar.pl
Izena: Aitor
Adina: 36
>
```

Aldagaietan gordetako datuak eskura izango ditugu programan noiznahi erabili ahal izateko. Aldagai batean balio bat gordetzeko, "=" esleipen-eragileak erabili behar da. Ezkerraldean aldagaia eta eskuinean bertan gorde nahi dugun balioa jarriko dugu beti. Esleipen-eragileak, "=", lehenbizi espresioaren eskuinaldea ebaluatuko du eta ondoren emaitza ezkerraldean jarritako aldagaian gorde.

```
$x = 5;          # $x aldagaian 5 balioa gorde
5 = $x;          # errorea!
```

Konturatuko zinen `eskalar.pl` programan aldagaiaren aurretik `my` gako-hitza jarri dugula. Izan ere, `use strict` sententziak, lehenbiziko aldiz aldagaia erabili aurretik erazagutzeroa behartzen gaitu `my` erabiliz. Aldagai bat erazagutzean, Perl-i adierazten diogu aldagai hori geroago programan erabiliko dugula. Perl-ek ez gaitu aldagaiak erazagutzeroa behartzen, baina gomendagarria da hala egitea, programa idaztean egin ditzakegun akats tipografikoak identifikatzen lagunduko baitigu besteak beste. Adibidez, `print($izena)` idatzi ordez, nahastu eta `print($izewna)` jarri izan bagenu, `use strict` jarrita programak errorea itzuliko liguke exekutatzerakoan, `$izewna` aldagaia ez baita aurretik erazagutua izan. `use strict` sententzia gabe, Perl-ek bi aldagai ezberdini buruz ari garela usteko luke eta ez luke erroreak emango exekutatzerakoan. Baina programak ez luke guk nahi dugun balioa (`$izena` aldagaiarena) pantailaratuko eta buruhauste askoren iturri bihurtuko litzateke. Egiatzapen mota hau lagungarria da programazio-erroreak aurkitzeko orduan. Hori dela-eta, liburuko adibideetan beti erabiliko ditugu `my` eta `use strict`.

Terminoak berak iradokitzen duen moduan, aldagaien edukia aldatu egin daiteke programan zehar. Begira hona bestela:

eskalar2.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# Aldagaiak edukia aldatuz
my $izena = "Aitor"; # Aitor balioa gorde $izena aldagaian
print("Izena: ");
print($izena);
print("\n");
$izena = "Nagore"; # Nagore balioa gorde $izena aldagaian
print("Izena: ");
print($izena);
```

Exekutatzean, \$izena aldagaiak hartzen dituen balio ezberdinak pantaila-ratuko ditu Perl-ek:

```
>perl eskalar2.pl
Izena: Aitor
Izena: Nagore
>
```

Eragiketa matematikoetan, zenbakiak eta aldagaiak nahasian erabil daitezke:

```
$zenb1 = 2 * 3 + 4;          # $zenb1 aldagaian 10 balioa gorde
$zenb2 = $zenb1;            # $zenb2 = 10
$zenb2 = (2 * zenb1) - 7;    # $zenb2 = 13
$zenb2 = $zenb2 + $zenb1;    # $zenb2 = 23;
```

Aldagaiak erabiltzen hasi berritan, ohiko errorea izaten da aldagaiaren izena eta bere balioa nahastea. Kontuz gero, aldagaiaren izena eta bere baitan gordetzen duen balioa bi gauza zeharo ezberdin dira eta. Begira bestela ondoren datorren adibideari:

```
$txakurra = "katua";
$bi = 3;
print($txakurra); # katua karaktere-katea pantailaratuko du
print($bi);       # 3 zenbakia pantailaratuko du
```

Komatxoaren erabilera

Ikusi dugu dagoeneko `print` funtzioa erabiltzean, komatxo bikoitzen artean idatzitako *string*-etan karaktere bereziak interpretatu egiten dituela Perl-ek. Aldagai eskalarrekin ere, beste horrenbeste gertatzen da. Aldagaia komatxo bikoitzen artean jarritz gero, aldagaiaren balioa jarriko du bere lekuan Perl-ek. Komatxo sinpleen artean jarritz gero, Perl-ek ez du interpretatuko eta dagoen dagoenean erakutsiko du. Begiratu ondoko adibideari:

komatxoAldagai.pl

```
#!/usr/bin/perl
use warnings;
use strict;
my $zenbakia = 4;
# komatxoak eta aldagaiak
print("Hau da gure zenbakia: $zenbakia\n");      # bikoitzak
print('Hau da gure zenbakia: $zenbakia\n');      # sinpleak
```

Programaren irteera:

```
>perl komatxoak.pl
Hau da gure zenbakia: 4
Hau da gure zenbakia: $zenbakia\n
>
```

Komatxo bikoitzak erabiltzean, "\$" ikurraz hasten diren karaktere-kateak aldagai bezala interpretatuko ditu Perl-ek, eta bere balioarekin ordezkatu. Gerta liteke inoiz edo behin "\$" ikurra bere horretan erabili nahi izatea ere. Zer egin? Bi aukera, edo komatxo sinpleak erabili *string*-ean, edo "\" karakterea jarri "\$" karakterearen aurretik. Izan ere, "\" erabilita, ondoren datorren karakterea Perl-ek ez du karaktere berezi bezala interpretatuko, nahiz eta komatxo bikoitzak erabili. Adibide honek hobeto erakusten du azaldutakoa:

karakBereziak.pl

```
#!/usr/bin/perl
use warnings;
use strict;
my $prezioa = 200;
print("Brent upelak $prezioa \ $eko salneurria iritsi du \n");
```

Komando-lerrora joan eta exekutatzean:

```
>perl karakBereziak.pl
Brent upelak 200 $eko salneurria iritsi du
>
```

Zenbaki eta karaktere motako datuekin baliatu ditugun eragileak, erabilgarriak dira aldagaiekin ere. Honako programa honek eragiketa sinpleak egiten ditu aldagaiekin:

aldEragiketak.pl

```
#!/usr/bin/perl
use warnings;
use strict;
# eragiketak aldagaiekin
my $adina = 12;
my $izena = "Xabat";
my $abizena = "Urreisti";
my $izen_abizenak = $izena . " " . $abizena;
#string-en kateaketa
$adina = $adina * 3;      #zenbakien biderketa
print ("Izen-abizenak: $izen_abizenak \tAdina: $adina\n");
print($izena);
```

Programa exekutatzean:

```
>perl aldEragiketak.pl
Izen-abizenak: Xabat Urreisti      Adina: 36
>
```

1. kasu praktikoa: moneta-bihurtzailea

Euro kopuru jakin bat emanda, kopuru hori zenbait txanponetara bihurtuko duen programa idatziko dugu lehen kasu praktikoan. Aldagaiak eta hauen arteko eragiketak lantzeko proposa.

Lehen hurbilpen posible bat honakoa litzateke:

monetak.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# euro kopuru bat moneta ezberdinetara bihurtzen duen programa
my $kop = 30;                # euro kopurua
my $dolar = $kop * 1.577;    # dolarretan
my $libra = $kop * 0.797;    # liberatan
my $yen = $kop * 166.825;    # yen-etan
print("$kop euro:\n");
print("\t$dolar dolar dira\n");
print("\t$libra libra dira\n");
print("\t$yen yen dira\n");
```

Editorea erabiliz idatzi programa zehatz-mehatz goian agertzen den bezala, eta ondoren gorde *monetak.pl* izenarekin. Komando-lerrotik exekutatzean honako irteera lortu beharko zenuke gauzak ondo bidean:

```
>perl monetak.pl
30 euro:
    47.31 dolar dira
    23.91 libra dira
    5004.75 yen dira
>
```

Programak bere horretan ez du ezer berririk erakusten. Programa guztietan jarri beharreko lehen hiru lerroen ondoren, `$kop` aldagaian 30 balioa gordetzen da. Kopuru hori eurotan oinarritzat hartuta, moneta ezberdinetara bihurtzeko kalkuluak egiten dira segidan eragiketa aritmetiko sinpleak erabiliz. Azkenik, programak bata bestearen ondoren kalkulaturako balioak pantailaratzen ditu.

Erabilgarritasunaren aldetik, oso mugatua da idatzi berri dugun programa. Beti euro kopuru bera bihurtzen duen programak ia ez dauka erabilerarik. Interesgarriagoa litzateke, dudarik gabe, programak hiru pauso hauei jarraituko balie:

1. Erabiltzaileari galdetu zein den bihurtu beharreko € kopurua.
2. Kopuru hori beste txanponetara bihurtu.
3. Bihurketak pantailan bistaratu.

Lehen pausoa emateko, programak modua izan behar luke kanpotik datuak jasotzeko. Hau da, programa abiatzean, erabiltzaileak (programa abiarazi duena) euro kopurua idatzi eta programak hura jaso behar luke. Gai hau zehatz-mehatz hurrengo kapituluan landuko badugu ere, aurrerapen txiki bat eskainiko dizuegu orain, goiko moneta-bihurtzaileari (eta ondoren datozen programei) erabilgarritasuna gehitze aldera.

Oinarrizko sarrera/irteera

Gure programetan kanpoko datuak jasotzeko bi modu nagusi dauzkagu: bata, erabiltzaileak teklatu bidez zerbait idatzi eta hura jasotzea, eta, bestea, datuak fitxategi batetik irakurtzea. Lehenbiziko aukera da orain labur-labur jorratuko duguna, bestea aurreragorako utziz.

Erabiltzaileak teklatu bidez idatzitakoa gure programak jaso dezan, `<STDIN>` eragilea behar dugu. Adibidez, gure programan sententzia hau idatziko bagenu:

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

Sententzia irakurtzean, Perl zai-zai geldituko litzateke erabiltzaileak teklatu bidez zerbait idatzi eta lerro-jauzia (*return*) tekla sakatu arte. Idatzitako guztia \$lerro aldagaian gordeko luke Perl-ek. Laburdura gisa, <STDIN> jarri ordez <> ere erabil daiteke.

Teklatu bidez datuak jasotzeko oinarrizko teknika ezagutzen dugunez, erabil dezagun *moneta-bihurtzailea* programa eguneratzeko. Bigarren bertsio honetan, programak erabiltzaileari bihurtu beharreko euro kopuruaz galdegingo dio:

moneta2.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# euro kopurua teklatutik jaso eta
# moneta ezberdinetara bihurtzen duen programa
print("Idatzi diru kantitatea eurotan: ");
my $kop = <STDIN>;
my $dolar = $kop * 1.577;
my $libra = $kop * 0.797;
my $yen = $kop * 166.825;
print("$kop euro dira:\n");
print("\t$dolar dolar\n");
print("\t$libra libra\n");
print("\t$yen yen\n");
```

Exekutatzerakoan, bihurtu beharreko kopuruaz galdegingo digu programak. Nahi dugun kantitatea idatzi, *return* sakatu, eta bihurketak egin ondoren emaitzak pantailan bistaratuko ditu programak.

Hona exekuzio adibidea:

```
>perl moneta2.pl
Idatzi diru kantitatea eurotan: 3
3
euro dira:
    3.154 dolar
    1.594 libra
    333.65 yen
>
```

Konturatuko zinen agian bigarren bertsio honek badaukala akats bat: X euro dira mezua ez du lerro bakarrean pantailaratzen lehen bezala, bi lerrotan banatuta baizik. Zergatik ote? <STDIN> erabilia teklatu bidez idatzitakoa jasotzen du

programak, guzti-guztia, baita bukaerako lerro-jauzia (*return*) tekla ere. Ondoren, datu bera pantailan bistaratzean, lerro-jauzi eta guzti bistaratzen dugu, horregatik ageri da bi lerrotan banatuta. Lerro-jauzia ezabatzeko `chomp()` agindua erabiltzen da. Aurreko programan, teklatutik datua jaso ondoren `chomp($kop)`; sententzia idatziko bagenu, ondoren `print` funtzioak lerro bakarrean bistaratuko lituzke datuak. Egin proba.

3.3.2 Array edo bektoreak

Aldagai eskalarretan zenbaki edo *string*-ak gorde ditzakegu, bai, baina eskalar motako edukiontziaaren ahalmena mugatua da eta balio bakarra baino ezin dezake gorde. Adibidez, zenbaki batzuen arteko eragiketak egin ondoren, aldagai eskalar batean gorde dezakegu emaitza, edo zazpi *string* kateatu eta guztia aldagai bakarrean utzi, baina aldagai eskalarrak betiere balio bakar bat izango du.

Batzuetan, hala ere, balio bat baino gehiago gorde nahi izango ditugu elkarrekin. Adibidez: asteko egunak, kurtsoko notak, taldekideen izenak, etab. Horrelako kasuetan, *array* edo bektoreak erabiliko ditugu. *Array*-ak oso erabiliak dira programazioan, balio eskalarrez osatutako zerrendak gordetzeko. Aldagai bat *array* motakoa dela adierazteko, `@` ikurra jarri behar zaio izenaren aurretik.

Array edo bektore-edukiontziek edozein motatako balio eskalarrak gorde ditzakete beren baitan, baina, oro har, elkarrekin erlazioen bat daukaten balioak biltzeko erabili ohi dira.

Array bat sortu eta bertan elementuak gordetzeko, komaz banatutako balio-zerrenda erabiliko dugu:

```
@lanegunak = ("Astelehena", "Asteartea", "Asteazkena",
              "Osteguna", "Ostirala");
@asteko_tenp = (17, 14, 20, 21, 19, 17, 15);
@koloreak = ("berdea", "gorria", "urdina");
```

Bektoreetan nahi adina elementu gorde ditzakegu, Perl-ek ez du inongo mugarik ezartzen. *Array*-etako osagaiak ordenaturik daude, eta indize baten bidez erreferentziatzen dira. *Array*-aren osagai jakin bat lortzeko, dagokion indizea kor-txete `[]` artean jarri behar dugu. Lehen osagaiaren indizea 0 da, bigarrenarena 1, eta horrela hurrenez hurren.

Programa sinple honek *array* bat definitu eta bere osagai indibidualak nola erabili erakusten du:

lanegunak.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# array-a definitu eta elementuak atzitu
my @lanegunak = ("Astelehena", "Asteartea", "Asteazkena",
                 "Osteguna", "Ostirala");
print("Lanegunak: $lanegunak[0], $lanegunak[1],
      $lanegunak[2], $lanegunak[3] eta $lanegunak[4] dira\n");
```

Programan, @lanegunak *array*-a erazagutu (my) eta asteko lanegunak gehitu dizkiogu komatxo arteko balio-zerrenda erabiliz. Segidan, *array*-ko elementu guztiak pantailaratu ditugu bakoitzaren indize edo posizioak erabiliz (0 posiziotik hasita). Detaile garrantzitsu bat: *array*-ko elementuak inprimatzerakoan \$lanegunak[] idatzi dugu eta ez @lanegunak[], *array* motako datu-egitura izan arren. Izan ere, bistaratzean ez diogu *array* osoari erreferentzia egin, soilik elementu bakar bati, eta *array*-aren elementu bakoitza eskalar motakoa da. Beste modu batera esanda, @ ikurrak plurala adierazten du, elementu guztiak, eta, \$ ikurrak, berriz, singularra, elementu bakarra. Beraz, *array* osoari erreferentzia egiteko @ erabiliko dugu, eta *array*-ko elementu bat bistaratu edo tratatzeko \$. Kontu izan, hemendik aurrera, ohiko programazio-errorea izaten da-eta biak nahastea.

Badago *array* osoa inprimatzerik, banan-banan elementuak erreferentziazten ibili beharrik gabe. Hona jarraian bi modu:

lanegunak2.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# array osoa bistaratu
my @lanegunak = ("Astelehena", "Asteartea", "Asteazkena",
                 "Osteguna", "Ostirala");
print("Lanegunak: @lanegunak\n");
print(@lanegunak);
```

Exekutatuz gero, hauxe da lortu beharko genukeen emaitza:

```
>perl lanegunak2.pl
Lanegunak: Astelehena Asteartea Asteazkena Osteguna Ostirala
AstelehenaAstearteaAsteazkenaOstegunaOstirala
>
```

Berriro ere, komatxo bikoitzak erabili edo ez, badago alderik. Erabiliz gero, inprimatzerakoan *array*-ko elementuen artean zuriune bat txertatuko du Perl-ek, bestela, elementu guztiak jarraian.

Bektoreekin lan egitean, sarritan gertatu ohi da beren elementu kopurua ez ezagutzea. Bektore baten azken elementua bistaratu nahi dugu, adibide bat jartzearen, baina azken elementu horren indizea zein den ez dakigu. Zer egin kasu horretan? Perl-ek eskaintzen dizkigu gutxienez bi aukera: lehena `$#array` aldagai berezia erabiltzea, zeinak `@array` bektorearen azken elementuaren indizea itzultzen duen. Bektoreak ez badauka elementurik (hutsik dago), aldagaiak `-1` balioa hartuko du. Bigarren aukera `scalar()` funtzioa erabiltzea litzateke. Funtzio horrek argumentu gisa pasatutako *array*-ko elementu kopurua itzultzen du. Bi sistemak erabil ditzakegu azken elementua atzitzeko, baina bien artean badago alderik: `$#array` aldagaiaren balioa, `scalar(@array)` funtzioak itzuliko diguna baino unitate bat txikiagoa izango da, indizea zerotik hasten delako kontatzen eta elementu kopurua, berriz, batetik. Hurrengo adibideak erakusten du hori:

```
my @bikoitiak = (2, 4, 6, 8, 10);
print $bikoitiak[2];           # 6
print $bikoitiak[4];           # 10
print $bikoitiak[$#bikoitiak]; # azken elementua, 10
print scalar(@bikoitiak);      # elementu kopurua, 5
print $#bikoitiak;             # azken indizea, 4
```

Bektore edo *array*-ko elementuen azpimultzo bat aukeratu eta horiekin *azpiarray*-a eratzea ere posible da:

```
my @bakoitiak = (1, 3, 5, 7, 9);
my @azpiBakoiti = @bakoitiak[2..4];
                    # 3.elementutik 5.elementura
```

`@azpiBakoiti` *azpiarray*-an `@bakoitiak` bektoreko azken 3 elementuak gorde: 5, 7 eta 9.

Badu Perl-ek eragile bat, tartea `".."`, *array*-ekin oso erabilgarria. Eragile horrek, bi elementuren arteko elementu-zerrenda itzultzen du. Adibidez, 1 eta 100 arteko zenbakiekin *array* bat osatu nahiko bagenu:

```
@zenbakiak = (1 .. 100);
```

Tarteak (`$x..$y`) formatua erabiliz adierazten dira, non `$x` hasierako balioa den eta `$y` amaierakoa. Balio batetik besterako pausoa lekoa da. Zenbakien ordez letrak ere jar genitzake:

```
@alfabetoa = ("a" .. "z"); # alfabetoko letra guztiak
```

Azkenik, indizeetan zenbaki negatiboak erabiliz gero, *array*-aren azken elementutik hasiko da kontatzen Perl. Adibidez:

```
@alfabetoa = ("a" .. "z");
print($alfabetoa[-1]); # array-ko azken elementua, "z"
print(@alfabetoa[-10 .. -1]);
# bistaratu azken 10 elementuen azpiarray-a
```

Hasi besterik ez dugu egin, eta *array* edo bektoreen benetako erabilgarritasuna ikustetik urrun gaude oraindik. Baina liburuan aurrera egin ahala, jabetuko zarete beraien erabilera eta garrantziaz.

2. kasu praktikoa: alfabetoarekin jolasean

Erabiltzaileari zenbaki bat eskatu, eta zenbaki horri dagokion alfabetoko letra bistaratzen duen programa idatzi (0 zenbakiari "a" letra dagokio eta 25 zenbakiari, berriz, "z").

Argigarri izan litekeelakoan, hona hemen programa-dei bat:

```
>perl alfabetoa.pl
Idatzi 0-25 arteko zenbaki bat: 15
15. letra hauxe da: p
>
```

Programak hiru pauso hauek betetzen dituela ziurtatu:

1. Osatu *array* bat alfabetoko letra guztiekin.
2. Erabiltzaileari zenbaki bat eskatu eta gorde aldagai batean.
3. Aldagaiak adierazten duen posizioako letra bistaratu.

Tarte bat hartu eta saia zaitez aurreko pausoei jarraituz programa idazten, ea zer moduz moldatzen zaren.



.....

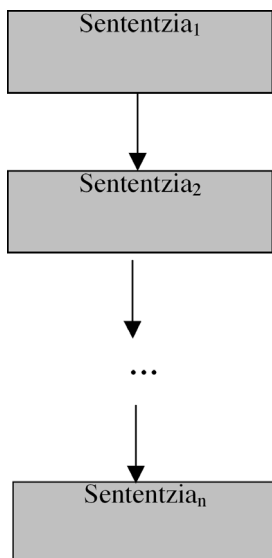
Orain konpara dezakezu gure soluzioarekin, baina ez ezabatu gero zuk egindakoa! Ia beti izango dira bide bat baino gehiago problema bera ebazteko (gogoratu Perl-en leloa), eta gure proposamenak ez du zertan egokiena edo argiena izan. Aurreko eskemari jarraitzen dion soluzio posible bat honakoa da:

alfabetoa.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my @alfabetoa = ("a" .. "z");
print ("Idatzi 0-25 arteko zenbaki bat: ");
# teklatutik datua jaso
my $pos = <STDIN>;
# bukaerako lerro-jauzia kendu
chomp($pos);
# bistaratu $pos posizioko letra
print (" $pos. letra hau da: $alfabetoa[$pos]\n");
```

3.4. KONTROL-EGITURAK

Orain arte idatzi ditugun programetan aginduak bata bestearen atzetik exekutatu dira, ordena sekuentzialean. Irudi bidez adierazi beharko bagenu, honela egin genezake:



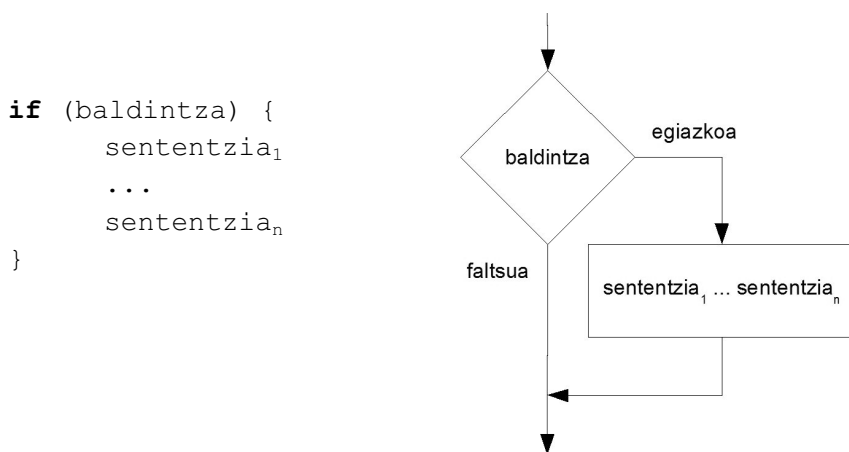
Atal honetan, kontrol-egituren bidez programaren ordena sekuentziala aldatzeko moduak aztertuko ditugu. Kontrol-egiturak programazio-lengoaiei oinarritzko elementuak dira, eta bi lan hauetarako erabiliko ditugu:

1. Balioak konparatu eta erabakiak hartzeko.
2. Agindu-blokeak errepikatzeko.

3.4.1. Baldintzazko egiturak

if egitura

Kontrol-egitura guztien artean erabiliena eta beharrezkoena **if baldintzazko egitura** izango da ziur asko. Egitura honek balioak konparatu eta erabakiak hartzeko balio du, erabakiaren arabera programa bide batetik edo bestetik eramanaz. Izan ere, beti portaera bera erakusten duen programa ez da oso erabilgarria. Programaren benetako potentziala sarrerako datuen arabera modu batera edo bestera erantzuteko ahalmenean datza. Baldintzazko **if** egiturarekin, **agindu-bloke** bat exekuta deza-kegu **baldintza** jakin bat betetzen denean soilik. Bi zati dauzka **if** egiturak: **baldintza**, betetzen den edo ez begiratu beharrekoa; eta **agindu-blokea**, giltza artean **{ }** idatzi eta baldintza betetzen denean soilik exekututuko dena.



Adibidez:

```

if ($tenperatura > 30) {
    print ("Uf, bero handiegia lanerako!\n");
}
  
```

Baldintzazko egiturak `$tenperatura` aldagaiaren balioa ebaluatu eta 30 baino handiagoa denean **soilik** exekututuko du giltza artean dagoen `print` funtzioa.

Baldintza inguratzen duten parentesiak (`$tenperatura > 30`) nahitaezkoak dira. Parentesi artekoak zenbaki edota *string*-ez osatutako espresio logikoa izan behar du, *Egia/Gezurra*, *Betetzen da/Ez da betetzen* motako erantzuna izango duena. Baldintzaren espresioan eragile hauek parte har dezakete:

Zenbakien artean	String-en artean	Esanahia
<	lt	Txikiago
>	gt	Handiago
<=	le	Txikiago edo berdin
>=	ge	Handiago edo berdin
==	eq	Berdin
!=	ne	Desberdin

Taulan ikus daitekeen bezala, zenbakiak eta karaktereak konparatzeko Perl-ek eragile ezberdinak erabiltzen ditu. Adibide pare batekin, erraz ulertuko dugu noiz eta nola erabili bakoitza.

Lehen adibidean, *Ezetz Asmatu!* motako joko sinple bat garatuko dugu, bide batez `if` egituraren funtzionamendua hobeto ulertzen lagunduko diguna. Jokoaren eskema oso sinplea da: zenbaki bat idatziko dugu ezkutuan, eta erabiltzaileari eskatuko diogu gure ezkutuko zenbakia asmatzen ahalegintzeko. Asmatuz gero, programak erabiltzailea zorionduko du.

Hona programaren eskema orokorra:

1. Aukeratu ezkutuko zenbakia eta gorde aldagai batean.
2. Erabiltzaileari zenbaki bat idazteko eskatu eta gorde aldagai batean.
3. Baldintzazko egitura erabiliz aurreko bi zenbakiak konparatu. Berdinak badira zoriondu erabiltzailea.

Aurreko eskema Perl lengoaiara itzuli ondoren:

zenbAsmatu.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# Zenbakia asmatu joko. Zenbaki sekretua 5 da
my $sekretua = 5;
print ("Idatzi ezazu zenbaki bat teklatu bidez: ");
my $zenb = <STDIN>;
chomp($zenb); # lerro-jauzia kendu
if ($zenb == $sekretua) {      # baldintza: berdina?
    print("Asmatu duzu, zorionak!\n");
}
```

Exekutatu programa eta aztertu bere funtzionamendua.

Programak erabiltzaileari teklatu bidez zenbaki bat idazteko eskatu eta ezkutuan daukanarekin konparatzen du. Berdinak baldin badira, erabiltzailea zorion-

duko du pantailan mezu bat bistaraziz. Kontuan izan bi zenbaki berdinak ote diren konparatzeko eragilea == dela (bi berdintza-ikur jarraian), eta ez = (berdintza-ikur bakarrarekin esleipena da). Ohiko programazio-errorea da aurreko biak nahastea.

```
($zenb == $sekretua) # konparaketa
($zenb = $sekretua)  # $zenb aldagaian gorde $sekretua
                    # aldagaiaren balioa
```

Zenbakiekin egin bezala, *string*-en arteko konparaketak ere egin ditzakegu baldintzazko espresioetan. Hurrengo programak *string* edo karaktere-kateak konparatzeko eq (*equal*, berdin) eragilea darabil, zeinak *egiazkoa* itzultzen duen espresioiko bi *string*-ak berdinak baldin badira, eta *faltsua* bestela.

izena.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# string-en arteko konparaketa
print ("Idatzi ezazu zure izena: ");
my $izena = <STDIN>;
chomp($izena);
if ($izena eq "Perl") {
    print("tokaio!\n");
}
```

Zer gertatuko litzateke bi *string* edo karaktere-kate "==" zenbakizko eragilearekin konparatuko bagenitu? Errorea emango luke exekutatzetarakoan? Ez, ez genuke errore-mezurik jasoko. Okerragoa dena, konparaketak beti itzuliko luke *egiazkoa*, *string*-ak zeinahi izanik ere. Ez nahastu, beraz, eta datu mota bakoitzarekin erabili dagokion eragilea.

Handiago eta txikiago eragileen erabilera begibistakoa da zenbakien kasuan.

```
7 < 21          # egiazkoa
3 > 200         # faltsua
4 <= (5-1)      # egiazkoa
```

String motako datuekin bagabiltza, konparaketak ordena alfabetikoz egiten dira:

```
"Aitor" lt "Zigor"      # txikiago: egiazkoa
"aaa" ge "aa"           # handiago edo berdin: egiazkoa
```


Komatxo artean jarrita, zenbakiak *string* gisa trata ditzakegu:

```
200 > 30           # egiazkoa. Zenbakien arteko konparaketa
"200" gt "30"      # faltsua. String arteko konparaketa
```

Azkenik, `if` egiturak badauka formatu laburtua, baldintza betetzen denean agindu bakarra exekutatzeke erabil dezakeguna:

```
# Inprimatu $i bikoitia baldin bada
print $i if ($i % 2 == 0);
# Agurtu bere izena Iñaki bada
print ("Kaixo $izena\n") if ($izena eq "Iñaki");
```

3. maila

Baldintza konplexuagoak ere eraiki ditzakegu taulako eragileak konbinatuz. Horretarako `&&` eta `||` eragile logikoak erabiliko ditugu.

- **&& eragile logikoa:** `esp1 && esp2` motako espresioak, non `esp1` **eta** `esp2` egiazkoak izan behar diren (biak), espresio osoa egiazkoa izan dadin.
- **|| eragile logikoa:** `esp1 || esp2` motako espresioak, non espresio osoa egiazkoa izango den, `esp1` **edo** `esp2` egiazkoa bada (bietako bat gutxienez).

Segidan datorren programak zenbaki bat eskatu eta idatzitakoa 10 eta 20 artekoa denean mezua bistaratzen du pantailan.

eragileLog.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# && eragile logikoa
print ("Idatzi zenbaki bat: ");
my $zenb = <STDIN>;
chomp($zenb);
if ($zenb > 10 && $zenb < 20) {
    # Bi baldintza: $zenb >10 eta $zenb < 20
    print("zenbakia 10 eta 20 artean dago\n");
}
```

Bigarren eragile logikoak, `||`, aukera adierazten du: hau edo hura. Ondoko programan ikus genezake nola erabili:

eragileLog2.pl

```
#!/usr/bin/perl
use strict;
use warnings;
print ("Idatzi zure izena:");
my $izena = <STDIN>;
chomp($izena);
# Agurtu Miren edo Edurne deituz gero
if ($izena eq "Miren" || $izena eq "Edurne") {
    print ("kaixo $izena!\n");
}
```

Baldintza berean *string* eta zenbakien arteko konparaketak konbina ditzakegu eragile logikoak erabilita.

eragileLog3.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# string-ak eta zenbakiak baldintza berean
print ("Nola duzu izena? ");
my $izena = <STDIN>;
chomp($izena);
print("Zenbat urte dituzu? ");
my $adina = <STDIN>;
chomp($adina);
if ($izena eq "Perl" && $adina > 20) {
    # Egiazkoa baldin Perl izena eta 20 urte baino gehiago
    print("Ni baino zaharragoa zara tokaio!\n");
}
```

Nahi bezain baldintza konplexuak eraiki daitezke, segidan datorren kode puskak erakusten duen moduan:

```
# Erabili parentesiak eragileen arteko ordena garbi uzteko
if (($zenb > -5 && $zenb < 0) || ($zenb == 7)) {
    print(" zenbakia -5 baino handiagoa eta zero baino
        txikiagoa, edo bestela 7 da\n");
}
```



Demagun zenbakiekin lan egiten duen programa bat garatzen ari garela, zeinak lehenbizi erabiltzaileari teklatu bidez zenbakiak idazteko eskatu eta ondoren haiekin lan egiten duen. Zenbakiak positiboak izan behar dute, oso garrantzitsua da baldintza hau, bestela programak errorea emango bailuke. Askotan izango ditugu era honetako murriztapenak gure programetan. Horretxegatik, Perl-ek badauka die izeneko funtzio berezi bat errore mezua pantailaratu eta programaren exekuzioa bertan behera uzten duena. Funtzio hori ll (edo bere baliokide den or) eragile logikoarekin batera, oso erabilgarria da baldintzak betetzen ez dituzten programen exekuzioa eteteko.

Zenbaki positiboak behar dituen programara itzul gaitezen. Honela idatziko genuke baldintza:

```
die("Zenbakiak positiboa izan behar du!\n") if ($zenb <= 0)
```

Edo bere baliokidea den:

```
die("Zenbakiak positiboa izan behar du!\n") unless ($zenb > 0)
```

Baldintza betetzen denean, \$zenb zero edo txikiagoa, die() funtzioa exekutatu, mezua pantailaratu eta programaren exekuzioa bertan behera geldituko da.

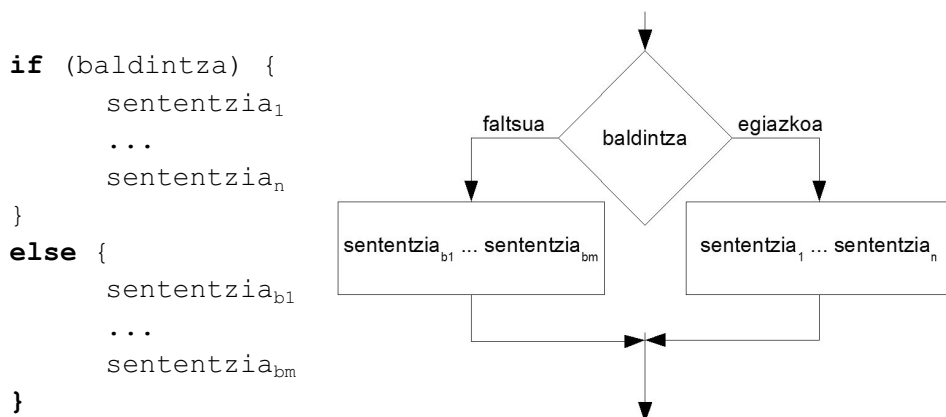
Ondoren datorren programak teklatu bidez bi zenbaki jaso eta bien arteko zatiketaren emaitza itzultzen du. Eragiketa egin aurretik zatitzailea zero ez dela ziurtatuko dugu or die egitura baliatuz.

zatiketa.pl

```
#!/usr/bin/perl
use strict;
use warnings;
print ("Idatzi zatikizuna:\n");
my $zenb1 = <STDIN>;
chomp ($zenb1);
print ("Idatzi zatitzailea:\n");
my $zenb2 = <STDIN>;
chomp ($zenb2);
die("Bigarren zenbakiak positiboa izan behar du!\n")
    if ($zenb2 <= 0);
my $zatiketa = $zenb1 / $zenb2;
print ("Zatiketaren emaitza: $zatiketa\n");
```

if-else egitura

Baldintza betetzen denean agindu-bloke bat exekutatzen den bezala, badago baldintza betetzen ez den kasuetan agindu-bloke ezberdin bat exekutatzeko aukera else adarra erabilia.



Hona hemen adibide bat:

```

if ($nota >= 5) {
    print ("Zorionak! Gainditu duzu: $nota\n");
}
else {
    print ("Sentitzen dut, ez duzu gainditu: $nota\n");
}

```

Notaren arabera (\$nota) gainditu dugun ala ez adieraziko digu goiko if-else egiturak. Kontuan izan else adarrak ez daukala baldintzarik, aurreko baldintzak huts egiten duenean aplikatzen den blokea da.

if-elsif egitura

Baldintzazko if egiturak badauka beste aldaera bat ere: if adarreko baldintza betetzen ez denean, badago beste baldintza bat edo batzuk betetzen ote diren galdetzeko aukera elsif klausula erabilita. Lehenbiziko baldintza (if adarra) betetzen ez denean, bigarrena ebaluatuko luke programak, horrek ere huts eginez gero, hurrengoa; eta horrela adarrez adar harik eta bukaerara iritsi edo betetzen den baldintza batekin topo egin arte. Aukera hau egokia da baldintza bati baino gehiagori begiratu behar diegun kasuetarako. Hona eskema:

```

if (baldintza1) {
    sententzia1
}
elsif (baldintza2)
{
    sententzia2
}

```

```

elsif (baldintza3)
{
    sententzia3
}
...
else {      # Aurreko baldintza guztiak bete ez badira

    sententzian
}

```

Lehenbiziko `if`-aren ondoren nahi adina `elsif` jar ditzakegu. Baldintza betetzen duen lehen adarrarekin topo egitean, dagokion agindu-blokea exekutatu, eta Perl-ek ez ditu gainerako adarrak ebaluatuko.

Adibide gisa, segidan datorren programak bi zenbaki jaso eta bietako handiena zein den erabakitzen du. Bi zenbakiak berdinak direnean, mezu bat bistartuko du hala adieraziz.

handiena.pl

```

#!/usr/bin/perl
use strict;
use warnings;
# bi zenbakiren arteko handiena itzuli
print("Idatzi 2 zenbaki, bakoitza lerro banatan:\n");
my $zenb1 = <>; # STDIN
my $zenb2 = <>;
chomp($zenb1); # bukaerako lerro-jauzi karakterea ezabatu
chomp($zenb2);
if ($zenb1 > $zenb2) {
    print (" $zenb1 handiagoa $zenb2 baino\n");
}
elsif ($zenb1 < $zenb2) {
    print (" $zenb2 handiagoa $zenb1 baino\n");
}
else {
    print("Biak berdinak dira!\n");
}
}

```

Hiru aukera posible baino ez daude: `$zenb1` handiagoa izatea `$zenb2` baino; edo `$zenb2` izatea `$zenb1` baino handiagoa; edo **bestela** biak berdinak izatea. Bukaerako `else` adarrak ez dauka baldintzarik, aurreko bi baldintzek huts egitean aplikatzen den sententzia da, eta beti izango da egiazkoa. Gure adibidean, aurreko bi baldintzek huts eginez gero, nahi eta nahi ez, bi zenbakiak berdinak izan behar dute.

3. kasu praktikoa: produktu-sailkatzailea

Produktuak prezioaren arabera sailkatuko dituen programa eskatu digute. Sailkapenak honakoa izan behar du: 120 € baino gehiago balio badu "garestia" etiketa izango du. 25 € baino gutxiago bada, "txollosa". 25-60 € artekoek "merkea" etiketa eraman behar dute, eta azkenik 60-120 € arteko produktuek "balekoa" etiketa.

Hau da:

Prezzoa < 25 € bada, "Txollosa"

25 € < *Prezzoa* < 60 € bada, "Merkea"

60 € < *Prezzoa* < 120 bada, "Balekoa"

Prezzoa > 120 € bada, "Garestia"

Saia zaitez sarrera gisa teklatutik prezioa jaso eta produktua behar bezala etiketatuko duen programa idazten.



.

Hona hemen gure proposamena:

prezioa.pl

```
#!/usr/bin/perl
use strict;
use warnings;
print("Idatzi produktuaren prezioa: ");
my $prezioa = <STDIN>;
chomp($prezioa);
if ($prezioa > 120) {
    print("garestia!\n");
}
elsif ($prezioa < 25) {
    print("txollosa!\n");
}
elsif ($prezioa < 60) { # prezioa 25-60 artean
    print("merkea\n");
}
else {                  # prezioa 60-120 artean
    print ("balekoa\n");
}
```

Exekutatu komando-lerrotik eta proba egin produktuaren prezio ezberdinekin.

Mota honetako baldintza-egiturak erabiltzean, hasierako *if*-aren ondoren nahi beste *elsif* baldintza jar ditzakegu, ez dago inongo mugarik. Bukaerako *else* adarra aukerakoa da, nahi edo behar izatekotan jarriko dugu, bestela ez, baina jartze-kotan bukaeran jarri behar da beti, aurreko baldintza guztiek huts egitean progra-mak egingo duena zehazteko.

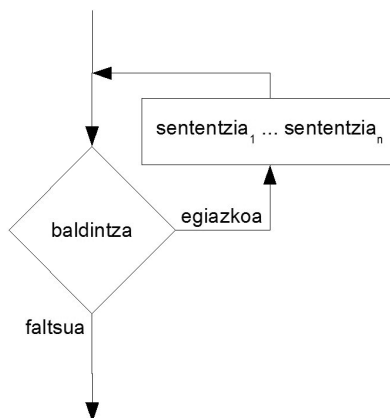
3.4.2. Begiztak

Konputagailuak erabiltzearen abantaila handienetakoa da lan bera behin eta berriz errepikatzeko duten ahalmena, neke edo asperdura-keinurik erakutsi gabe gainera. Programatzean, askotan errepikatu behar izaten dira zenbait eragiketa: egin hau erabiltzaileak teklatutik datuak idatzi bitartean; egin hori fitxategiko hitz bakoitzeko; egin hura *array*-ko elementu bakoitzari; etab. Atal honetan agindu-bloke bat nahi adina aldiz errepikatzeko Perl-ek eskaintzen dituen tresnak aztertuko ditugu, programazio-lengoaiei ingurunean *iterazio-egitura* edo begiztak deiturikoak.

Iterazio-egitura erabilienak, bai Perl-en baita gainerako programazio-lengoaia gehienetan ere, hiru hauek dira: *while*, *for* eta *foreach*. Hiruren egitekoa agindu-bloke bat behin eta berriz errepikatzea den arren, bakoitzak bere ezaugarri eta erabilera-eremu partikularrak dauzka, jarraian ikusiko dugun bezala.

while egitura

```
while (baldintza) {
    sententzia1;
    sententzia2;
    ...
    sententzian;
}
```



while egiturarekin, giltza arteko `{ }` agindu sorta behin eta berriz errepikatuko da (baldintza) betetzen den **bitartean**. Badauka ikusi berri dugun baldintzazko *if* egiturarekin antzekotasunik: bietan lehenbizi (baldintza) ebaluatzen da, eta betetzen denean, giltza arteko agindu-blokea exekutatu. Baina *while* egituran, agindu-blokea exekutatu ondoren berriro (baldintza) eba-

luatzen da, eta betez gero, berriro agindu-blokea exekutatu. Horrela behin eta berriz, harik eta baldintza betetzen ez den arte.

Adibide gisa, hona hemen atzeraka kontatzen duen programa `while` erabiliz:

`while.pl`

```
#!/usr/bin/perl
use warnings;
use strict;
my $atzeraka = 3;
while ($atzeraka >= 0) {
    print ("Atzera kontaketa: $atzeraka\n");
    $atzeraka = $atzeraka - 1;
}
print ("Amaitu da kontaketa!\n");
```

Programaren irteera:

```
>perl while.pl
Atzera kontaketa: 3
Atzera kontaketa: 2
Atzera kontaketa: 1
Atzera kontaketa: 0
Amaitu da kontaketa!
>
```

Programak jarraitzen duen bidea honakoa da: lehenbizi, `$atzeraka` aldagai eskalarra definitu eta 3 balioaz hasieratu dugu. Segidan, begiztan sartzean `while` egiturako baldintzak (`$atzeraka >= 0`), `$atzeraka` aldagaia 0 edo handiagoa den egiaztatuko du. Hala denez, giltza arteko `{ }` agindu-blokea exekutatu da: lehenbizi `$atzeraka` aldagaiaren balioa pantailan inprimatu, eta ondoren aldagaiari unitate bat kendu. Agindu-blokea amaitzean, begiztaren hasierara salto egingo du programak, eta berriro baldintza betetzen den ala ez egiaztatu. Bigarren itzuli honetan 2 balioa izango du `$atzeraka` aldagaiak, eta, ondorioz, berriz exekutatu da agindu-blokea. Behin eta berriz errepikatuko da prozesu bera harik eta laugarren itzuliaren ondoren, `$atzeraka` aldagaiak `-1` balioa hartu, eta baldintza beteko ez duenez (0 edo handiagoa izatea), programa begiztatik ateratzen den arte. Guztira, lau aldiz errepikatuko da begizta barruko agindu-blokea.

Kontu handiz erabili beharrekoak dira `while` egiturak. Ez berez arriskutsuak direlako, baina aurreko adibidean `while` egiturari sententzia bat kenduko bagenio:

while2.pl

```
#!/usr/bin/perl
use warnings;
use strict;
my $atzeraka = 3;
while ($atzeraka >= 0) {
    print ("Atzera kontaketa: $atzeraka\n");
}
print ("Amaitu da kontaketa!\n");
```

Exekutatu programa berria eta aztertu bere portaera. Zer gertatu da? Begiztaren gorputzean \$atzeraka aldagaia eguneratzen ez dugunez (lehen egiten genuen bezala unitate bat kenduz), while-ko baldintza beti beteko da (\$atzeraka beti izango da zero edo handiagoa) eta begizta amaierarik gabe errepikatuko da. *Begizta infinitua* deitzen zaio horri.

Programak amaierarik ez duela konbentzitzen garenean, komando-lerrora joan eta bere exekuzioa etengo dugu *Ctrl-c* teklak sakatuta (*Ctrl* tekla sakatu eta aldi berean *c*).

Gure programetan begizta bat erabiltzen dugunean, beti ziurtatu behar dugu begiztaren agindu-blokean baldintzako aldagaia egoki eguneratzen dugula, noizbait baldintza bete ez eta programak begiztatik ateratzeko modua izan dezan. Beste modu batera esanda, begiztan sartu bai, baina irteerarako atea zabalik utzi behar dugu beti.

Adibide gisa, erabiltzaileari bost zenbaki eskatu eta bosten batura itzultzen duen programa idatziko dugu *while* begizta erabiliz.

batuBost.pl

```
#!/usr/bin/perl
use warnings;
use strict;
my $kont = 5;           # kontagailua
my $zenb = 0;
my $batura = 0;
while ($kont > 0) {
    print ("Idatzi zenbaki bat:\n");
    $zenb = <STDIN>;    # teklatutik zenbakia jaso
    chomp ($zenb);
    $batura = $batura + $zenb;
    # gehitu uneko zenbakia aurrekoei
    $kont = $kont - 1;  # kontagailua eguneratu
}
print ("Zenbaki guztien batura: $batura\n");
```

Idatzi berri dugun `batuBost.pl` programa komando-lerrotik exekutatu dugu modu honetan:

```
>perl batuBost.pl
Idatzi zenbaki bat:
3
Idatzi zenbaki bat:
6
Idatzi zenbaki bat:
1
Idatzi zenbaki bat:
5
Idatzi zenbaki bat:
9
Zenbaki guztien batura: 24
>
```

Programaren muina begiztan dago: baldintzan kontagailu bat erabili dugu, eta honen balioa zero baino handiagoa den bitartean programa ez da begiztatik aterako. Begiztaren gorputzean, bi eragiketa: bata erabiltzaileari zenbaki bat eskatu, eta \$batura aldagaiari gehitzen diona. Bestea, \$kont kontagailuaren balioa unitate batean gutxitzen duena.

Programaren hasieran kontagailua, \$kont, 5 balioaz hasieratzen dugunez, 5 aldiz exekutatu da begizta. Ateratzean, erabiltzaileak teklatu bidez idatzitako zenbakien batura bistaratuko du.

4. kasu praktikoa: biderketa-taula

Idatzi programa bat teklatu bidez, zenbaki bat jaso eta zenbaki horren biderketa-taula pantailaratuko duena. Adibide gisa, hona hemen 7 zenbakiaren biderketa-taula itzultzen duen programa-deia:

```
>perl biderTaula.pl
Idatzi zenbaki bat eta bere biderketa taula kalkulatu
dut: 7
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
>
```

Hartu tarte bat eta ahalegindu programa zeure kabuz idazten.



.....

Zer moduz, atera al zaizu? Hemen duzu, nahi izanez gero, guk egindakoa:

biderTaula.pl

```
#!/usr/bin/perl
use warnings;
use strict;
print ("Idatzi zenbaki bat eta bere biderketa taula
      kalkulatu dut: ");
my $zenb = <STDIN>;
chomp($zenb);
my $kont = 1;
my $bider = 0;
while ($kont <= 10) {
    $bider = $zenb * $kont;
    print (" $zenb * $kont = $bider\n");
    $kont = $kont + 1;
}
```

foreach egitura

Egitura hau *array* edo bektoreekin lan egiteko oso erabilgarria da, elementuak banan-banan hartu eta tratatzeko bidea ematen baitu modu errazean.

```
foreach my $aldagai (@array) {
    sententzia1;
    ...
    sententzian;
}
```

Egitura honetan, *foreach* hitz erreserbatuaren ondoren jarri dugun *\$aldagai* aldagai eskalarra, *@array* bektoreko elementuen balioak hartuz joango da banan-banan; begiztaren gorputzean, *\$aldagai* elementuarekin nahi ditugun eragiketak egin ditzakegu.

Jarraian datorren adibideko programak *@frutak* bektoreko elementuak pantailaratzen ditu:

foreach.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my @frutak = ("sagarra", "udarea", "platanoa");
foreach my $fruta (@frutak) {
    print "$fruta oso goxoa dago\n";
}
```

Programa exekutatzean:

```
>perl foreach.pl
sagarra oso goxoa dago
udarea oso goxoa dago
platanoa oso goxoa dago
>
```

Portaera bereko programa erraz idatziko genuke, aurrez ikusitako `while` egitura erabiliz gero:

while.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my @frutak = ("sagarra", "udarea", "platanoa");
my $i=0;
# frutak array-aren azken indizea: $#frutak
while ($i <= $#frutak) {
    print "$frutak[$i] oso goxoa dago\n";
    $i = $i + 1;
}
```

`Array` egiturak azaltzean, tartea `..` eragilea ikusi dugu (bi elementuren arteko zerrenda itzultzen duena). Bada, tartea eragilea, `foreach` egituran ere erabil daiteke. Honako programa honek alfabetoko letrak pantailaratzen ditu banan-banan `foreach` egitura eta tartea `..` eragilea erabiliz.

foreach2.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# alfabetoa inprimatu foreach erabiliz
foreach my $letra ("a" .. "z") {
    print("$letra\n");
}
```

Zenbakiekin ere erabil ditzakegu tartea eragilea eta `foreach`. Adibidez, 1 eta 100 arteko zenbaki bakoitiak bistaratu nahiko bagenitu:

bakoitiak.pl

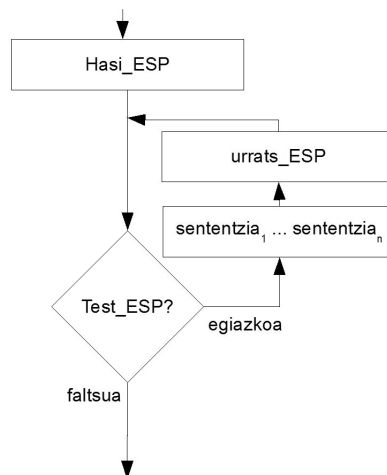
```
#!/usr/bin/perl
use strict;
use warnings;
# 1-100 arteko bakoitiak
foreach my $zenb (1 .. 100) {
    if ($zenb % 2 != 0) {
        # zati bi egin eta hondarra zero ez bada
        print("$zenb\n");
    }
}
```

3. maila

for egitura

Aldez aurretik agindu-blokea zenbat aldiz errepikatu nahi dugun dakigunean, `for` egitura egokiagoa izan daiteke `while` baino, `for`-ek iterazio kopurua kontrolatzen duen kontagailua baitarama berekin. Ordainetan, `for` egitura `while` begiztarena baino pixka bat konplexuagoa da. Hona hemen:

```
for (Hasi_ESP; Test_ESP;
Urrats_ESP) {
    sententzia1
    ...
    sententzian
}
```



Kontrol-egitura honek bete beharreko hiru eremu dauzka puntu eta komaz banatuak: kontrol-aldagaiak hartuko duen hasierako balioa (`Hasi_ESP`), kontrol-aldagaia bukaeraraino iritsi den ala ez jakiteko balio duen espresioa (`Test_ESP`), eta urrats bakoitzean kontrol-aldagaia eguneratzeko behar den espresioa (`Urrats_ESP`).

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

Adibide gisa, hona hemen `for` egitura erabiliz batetik ehunera kontatzen duen programa:

forKonta.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# letik 100era for egitura erabiliz
for (my $i = 1; $i <= 100; $i++) {
    print ("Kontaketa: $i\n");
}
```

Programaren irteera komando-lerrotik exekutatzean:

```
>perl forKonta.pl
Kontaketa: 1
Kontaketa: 2
Kontaketa: 3
...
Kontaketa: 100
>
```

Labur-labur azaldu dezagun adibideko `for` egituraren funtzionamendua: iterazioa hasi aurretik `$i` kontagailua hasieratzen da bat balioarekin. Segidan, muga espresioa (`$i <= 100`) ebaluatuko du `for`-ek, eta faltsua bada hortxe amaituko da iterazioa. Espresioa egiazkoa bada, ordea, giltza artean idatzitako agindu-blokea exekutatuko du programak. Azkenik `$i` kontagailua eguneratuko du `for`-ek `$i++` agindurekin (`$i++` agindua eta `$i = $i+1` baliokideak dira). Begiztan berriz sartu ahal izateko, muga espresioa ebaluatu beharko du berriro `for`-ek, harik eta kontagailuak 100eko muga gainditu eta begiztatik atera arte.

Hurrengo bi programek ere 1 eta 100 arteko zenbakiak pantailaratzen dituzte, baina soilik bikoitiak direnak. Lehenengoak, 1 eta 100 arteko zenbaki guztiak pasatuko ditu banan-banan `for` egitura erabiliz; begiztaren gorputzean, baldintzazko `if` egitura batek uneko zenbakia bikoitia den ala ez begiratu du.

3.5. ARRAY-EKIN LAN EGITEKO FUNTZIOAK

Array egiturak erruz erabiliko ditugu hemendik aurrera gure programetan. Perl-ek hainbat funtzio eskaintzen ditu beraiekin eragiketak egin ahal izateko. Atal honetan, haien artean garrantzitsuenak iruditu zaizkigunak aukeratu eta erakutsiko dizkizuegu, erabilgarri izango direlakoan.

Bektore-egitura bat izanda, elementuak gehitu edo kentzeko:

push(@array1, @array2)

@array1-i @array2-ko elementuak gehitzen dizkio bukaeran. Bigarren argumentua, @array2, bektore osoa edo elementu bakarra, eskalarra, izan daiteke. Adibidez:

```
@karak = ("a" .. "d");
@karak2 = ("f" .. "z");
$letra = "e";
push(@karak, $letra);
    # array-ari elementu bakarra gehitu bukaeran.
push(@karak, @karak2);
    # @karak array-ari erantsi bukaeran @karak2
    # @karak = ("a" .. "z");
```

pop(@array)

Argumentu bezala pasatutako *array*-aren azken elementua itzuli eta bektoretik ezabatzen du.

```
$letra = pop(@karak);
    # $letra aldagaiak "z" balioa hartuko du
    # karak == ("a" .. "y")
```

Array-aren bukaeran elementuak gehitu edo kendu beharrean *array*-aren hasieran egin nahi badugu, `shift()` eta `unshift()` funtzioak erabili behar ditugu:

shift(@array)

`pop()` funtzioaren baliokidea, baina honek *array*-aren lehen elementua itzultzen du.

```
$letra = shift(@karak);
    # $letra aldagaiak "a" balioa hartuko du
    # @karak == ("b" .. "y")
```


unshift(@array1, @array2)

@array2-ko elementuak @array1-en hasieran txertatzen ditu. `push()` funtzioak bezala, honek ere bigarren argumentutzat balio eskalarra har dezake.

```
@kirolak = ("txirrindularitza", "judoa", "eskubaloia");
@kirolak2 = ("eskalada", "futbola");
unshift(@kirolak, @kirolak2);
    # @kirolak2-ko elementuak gehituko ditu
    # @kirolak array-aren hasieran
```

sort(@array)

Argumentu bezala pasatutako @array bektorearen elementuak itzultzen ditu alfabetikoki ordenatuak. Funtzio honek ez du jatorrizko bektorea berrordenatzen, haren kopia bat itzultzen du alfabetikoki ordenatuta.

reverse(@array)

Funtzio honek argumentu bezala pasatutako *array*-a itzuliko digu atzekoz aurrera. `sort()` funtzioak bezala, honek ere ez dio jatorrizkoari eragiten.

Ondoko programak, kostaldeko herrien izenak bistaratzen ditu, nahasian lehenbizi eta alfabetikoki ordenatuta ondoren.

sort.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;
my @nahasi = ("Orio", "Baiona", "Bermeo", "Zarautz",
              "Mutriku", "Getaria", "Mundaka");
my @ordenatuta = sort(@nahasi);
print("Nahasian:\t@nahasi\n");
print("Ordenatuta:\t@ordenatuta\n");
```

>perl sort.pl

```
Nahasian:  Orio Baiona Bermeo Zarautz Mutriku Getaria Mundaka
Ordenatuta: Baiona Bermeo Getaria Mundaka Mutriku Orio Zarautz
>
```

Zenbakiekin, ordea, ondoko adibideak erakusten duen gisan, ez du behar bezala lan egiten:

sort2.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;
my @nahasi = (4, 2, 6, 23, 12, 1);
my @ordenatuta = sort(@nahasi);
print("Nahasian:\t@nahasi\n");
print("Ordenatuta:\t@ordenatuta\n");
}
```

```
>perl sort2.pl
Nahasian:    4 2 6 23 12 1
Ordenatuta:  1 12 2 23 4 6
>
```

12 zenbakia jarri du 1 eta 2-ren artean! Zergatik ote? Badu bere esplikazioa: herriekin egin bezala, alfabetikoki ordenatzen ahalegintzen da eta hortik dator emaitza ("12" ordena alfabetikoan "2"-ren aurretik doa). Zenbakiekin gabil-tzanez, zenbakiak konparatzeko eragilea da behar duguna: `<=>`. Perl-i espresuki zenbakien arteko konparaketa dela adieraziko diogu `{ $a <=> $b }` espresioaren bidez. Beste-rik adierazi ezean, berak `{ $a cmp $b }` *string*-en arteko konparaketa erabiltzen du.

Ondoko adibideak erakutsiko digu nola ordenatu zenbakizko bektore bat:

sort3.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;
my @nahasi = (4, 2, 6, 23, 12, 1);
my @ordenatuta = sort({ $a <=> $b } @nahasi);
print("Nahasian:\t@nahasi\n");
print("Ordenatuta:\t@ordenatuta\n");
```

```
>perl sort3.pl
Nahasian:    4 2 6 23 12 1
Ordenatuta:  1 2 4 6 12 23
>
```

Berrikuntza `sort()` funtzioan jarritako `{ $a <=> $b }` espresioan dago, zeinak konparaketa zenbakien artekoa dela adierazten dion Perl-i. Kontuz, `$a` eta `$b` aldagai global bereziak dira, eta ezin ditugu beste edozeinekin ordezkatu. Kontuan izan, halaber, konparaketa-espresioaren eta *array*-aren artean ez dela ko-marik jarri behar. Atzekoz aurrera ordenatu nahi badugu, `{ $b <=> $a }` idatziko

dugu zenbakiekin bagabiltza, eta `{ $b cmp $a }`, berriz, *string*-ekin (kasu honetan, `reverse()` funtzioa ere erabil dezakegu).

sort4.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;

my @nahasi = ("Orio", "Baiona", "Bermeo", "Zarautz",
              "Mutriku", "Getaria", "Mundaka");
my @ordena = sort(@nahasi);
my @atzekozAu = sort({$b cmp $a} @nahasi);

my @nahasiZenb = (4, 2, 6, 23, 12, 1);
my @ordenaZenb = sort({$a <=> $b} @nahasiZenb);
my @atzekozAuZenb = sort({$b <=> $a} @nahasiZenb);

print("Ordenatuta:\t\t@ordena\n");
print("Atzekoz aurrera:\t@atzekozAu\n");
print("Ordenatuta Zenb:\t@ordenaZenb\n");
print("Atzekoz aurrera Zenb:\t@atzekozAuZenb\n");
```

Exekutatzean, hauxe da pantailan agertuko zaiguna:

```
>perl sort4.pl
Ordenatuta:      Baiona Bermeo Getaria Mundaka Mutriku Orio
Zarautz
Atzekoz aurrera: Zarautz Orio Mutriku Mundaka Getaria
Bermeo Baiona
Ordenatuta Zenb: 1 2 4 6 12 23
Atzekoz aurrera Zenb: 23 12 6 4 2 1
>
```

1. maila



splice(@array, posizioa, zenbat, [@array2])

Funtzio hau nahiko konplexua da, baina ordainetan erabilera bat baino gehiago eskaintzen ditu. `splice()` funtzioak, *array*-etako elementuak ezabatu edo beste batzuekin ordezkatzeko balio duenak, **posizioa** indizetik hasita **zenbat** argumentuan adierazi adina elementu ezabatuko ditu *@array* bektorean, eta zerrenda gisa itzuli. Ezabatutako elementuak *@array2*-ko elementuekin ordezkatzeko ditu, baldin eta azken argumentu hau pasatzen badiogu. Adibidez:


```

print("Nola nahi duzu bistaratzea hitz-zerrenda?\n");
print("1 \t Alfabetikoki (a-z)\n");
print("2 \t Alfabetikoki atzekoz-aurrera (z-a)\n");
print("Egin aukera eta sakatu return\n");
$aukera = <STDIN>;
chomp($aukera);

if ($aukera == 1) {
    @ordenatuta = sort(@zerrenda);
}
elsif ($aukera == 2) {
    @ordenatuta = sort({$b cmp $a} @zerrenda);
}
else {
    die("Aukera okerra egin duzu...\n");
}

print("Zerrenda ordenatuta:\n");

foreach my $elem (@ordenatuta) {
    print("$elem\n");
}

```

6. kasu praktikoa: ezetz asmatu!

Ezetz asmatu! Jokoaren bertsio landuago bat proposatzen dizuegu oraingoan: programak zoriz 0 eta 100 arteko zenbaki bat aukeratuko du ezkutuan, eta erabiltzaileari eskatuko dio asmatzeko zein den ezkutuko zenbakia. Erabiltzaileak teklatu bidez zenbaki bat idazten duenean, programak ezkutukoarekin konparatuko du eta mezu hauetakoren batekin erantzun, kasuaren arabera: «*Arraioa, bai asmatu ere!*», edo «*... umm, nik daukadan zenbakia hori baino handiagoa da*», edo «*... umm, nik daukadan zenbakia hori baino txikiagoa da*». Ezkutuko zenbakia asmatu artean, programak behin eta berriz galdegingo du zenbaki sekretuagatik, harik eta erabiltzaileak asmatu arte. Bukaeran, erabiltzaileak zenbakia asmatzeko behar izan duen ahalegin kopurua ere pantailaratuko du programak.

Tira, badirudi programa idazten hasteko moduan gaudela. Prest?

Geldi, utzi teklatua alde batera eta itxaron pixka batean. Programa txikien kasuan ohikoa da zuzenean kodea idazten hastea, baina ez da inolaz ere prozedura egokiena ondo programatzen ikasteko. Ebatzi nahi ditugun atazen konplexutasun-maila handitu ahala, soluzio egoki batek planifikazioa eskatuko digu, hau da, programa idatzi aurretik pausoak ondo zehaztea eta egituratzea. Esku artean daukagun ataza oso konplikatu ez den arren, ohitu eta trebatu gaitezen ondo etorriko zaigu aurretik algoritmoa diseinatzea. Ez beldurtu gero algoritmo hitzarekin!

Algoritmoa errezeta da, oinarrizko osagaiak hartu eta haiekin nahi dugun platera sukaldatzeko zer pausori jarraitu zehazten duena.

Lehen begiratuan, hiru pausotan bana genezake programaren algoritmo orokorra:

1. Zoriz 0 eta 100 arteko zenbaki bat aukeratu eta gorde aldagai batean.
2. Erabiltzaileari zenbaki bat idazteko eskatu.
3. Erabiltzailearena eta gurea konparatu.

Lehenengo pausoan zenbaki bat aukeratu behar du zoriz programak. Baina nola aukera dezakegu zoriz zenbaki bat? Ez dugu guk egingo; izan ere, Perl-ek badauka espresuki zorizko zenbakiak sortzen dituen funtzioa: `rand()`. Funtzio horrek argumentu gisa `N` zenbaki bat jaso, eta zero eta `N` arteko zenbaki bat itzuliko digu zoriz. Adibidez:

```
$zenbaki = rand(100);

# 0-100 arteko zenbaki bat gorde $zenbaki-n
```

Baina `rand()` funtzioak zenbaki errealak itzultzen ditu (3.5 adibidez), eta guk jokoan zenbaki osoak behar ditugu. Zenbaki errealak biribildu eta oso bihurtzen dituen `int()` funtzioa erabiliko dugu konpontzeko.

```
$zenbaki = int(rand(100)); # 0-100 arteko zenbaki osoa
```

Ikusi berri ditugun bi funtzioekin, edozein bi zenbakiren artean bat zoriz aukeratzeak ez dauka inongo misteriorik. Jarrai dezagun.

Bigarren pausoan, erabiltzaileari teklatu bidez zenbaki bat idazteko eskatuko diogu. Aurretik behin baino gehiagotan egin dugun zerbait da eta ez du aparteko azalpenik behar.

Hirugarren eta azken pausoan, erabiltzailearen zenbakia programak ezkutuan aukeratutakoarekin konparatzeko baldintzazko egitura behar dugu. Biak berdinak badira, «*Arraioa, bai asmatu ere!*» mezua bistaratuko du programak. Bestela, baldin eta erabiltzaileak idatzitakoa txikiagoa bada ezkutukoa baino, «... *umm, nik daukadan zenbakia hori baino handiagoa da*» bistaratuko du. Azkenik, erabiltzailearena handiagoa denean ezkutukoa baino, «... *umm, nik daukadan zenbakia hori baino txikiagoa da*» mezua agertuko da pantailan.

Programaren eskema zehaztasun handiagoz berridazteko moduan gara:

1. Zoriz 0 eta 100 arteko zenbaki bat aukeratu eta aldagai batean gorde.
2. Erabiltzaileari zenbaki bat idazteko eskatu.
3. Erabiltzailearena eta gurea konparatu.

Baldin berdinak badira:

«Arraioa, bai asmatu ere!»

Bestela, baldin erabiltzailearena txikiagoa bada:

«... umm, nik daukadan zenbakia hori baino handiagoa da»

Bestela, baldin erabiltzailearena handiagoa bada:

«... umm, nik daukadan zenbakia hori baino txikiagoa da»

Algoritmoaren pausoei jarraituz gero, zoriz zenbaki bat aukeratu, erabiltzaileari zenbaki bat idazteko eskatu eta biak konparatuko genituzke. Baina behin bakarrik. Eta guk ezkutuko zenbakia asmatu artean erabiltzaileari behin eta berriz aukera emango dion programa nahi dugu. Igarriko zenion, irakurle, begizta-egitura behar duela gure programak. Begizta horretan, eskemako 2 eta 3 pausoak errepikatuko dira harik eta erabiltzaileak zenbakia asmatzen duen arte. Ikusi ditugun artean, `while` egitura da egokiena kasu honetan. Begizta barruan kontagailu bat jarriko dugu, iterazio edo ahalegin kopuruaren jarraipena egiteko.

Berridatz dezagun gure eskema begizta gehituz:

1. Zoriz 0 eta 100 arteko zenbaki bat aukeratu eta gorde.

2. `while` (ez asmatu).

2.1. Kontagailua inkrementatu.

2.2. Erabiltzaileari zenbaki bat idazteko eskatu.

2.3. Zenbakiak konparatu.

```
if (erabiltzailearena = gurea)
    "Arraioa, bai asmatu ere!"
if (erabiltzailearena < gurea)
    "... umm, nik daukadan zenbakia hori baino
    handiagoa da "
else
    "... umm, nik daukadan zenbakia hori
    baino txikiagoa da "
```

3. `print` (kontagailua).

Orain bai, ordenagailuaren aurrean jarri eta gure algoritmoa Perl lengoaiara itzultzeko moduan gara:

zenbAsmatu2.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my $aukera;
my $kop = 100;
# 0-100 arteko zenbaki bat aukeratu zoriz
my $zbk = int(rand(100));
print ("Zenbaki bat daukat gordeta, 0-100 artekoa. Zein ote
      da?\n");
my $jarraitu = 1;      # begiztan jarraitu ala ez
my $ahalegin = 0;      # ahalegin kopurua
while ($jarraitu != 0) {
    print "Egin zure aukera: ";
    $aukera = <STDIN>;
    chomp($aukera);
    $ahalegin++;      # $ahalegin = $ahalegin + 1;
    if ($aukera == $zbk) {
        print "Arraioa, bai asmatu ere!\n";
        $jarraitu = 0;
        # begiztatik aterako da hurrengo iterazioan
    } elsif ($zbk > $aukera) {
        print "... ummm, nik daukadan zenbakia $aukera baino
              handiagoa da \n";
    } else {
        # puntu honetan, beti gertatuko da $zbk < $aukera dela
        print "... ummm, nik daukadan zenbakia $aukera baino
              txikiagoa da \n";
    }
}
print (" $ahalegin ahalegin behar izan dituzu zenbakia
      asmatzeko.\n");
```

7. kasu praktikoa: pasahitza berreskuratzeko sistema

Oporretatik bueltan lanera iritsi, beti bezala ordenagailua piztu, eta bat-batean konturatu gara pasahitza ahaztu egin zaigula. Kaskarrari buelta eta buelta aritu arren, ez dugu gogoratzerik lortzen. Pista batzuk bai, badauzkagu: adibidez badakigu 4 letraz osatua dagoela, eta gogoratzen dugu, halaber, lehenbiziko eta hirugarren letrak kontsonanteak direla, eta bigarrena eta laugarrena, aldiz, bokalak. Memoriaren kapritxoak nolakoak diren, garbi gogoratzen dugu, baita ere, kontsonanteak eta bokalak ez direla errepikatzen. Baina hortik aurrera, ezer ere ez.

Informatika saileko arduradunarengana joan eta haren erretolika entzun baino nahiago hanka hutsik munduari buelta eman. Argialdi bat izan, eta ez si eta ez no, Perl-i buruz ikasitakoa aplikatzea erabaki dugu pasahitz posible guztien zerrenda atera eta geure kasa hitz ezkutua topatzeko.

Bero-bero zuzenean kodea idazten hastea ez da modurik zuzenena, ezta azkarrena ere, problema ebazteko. Lehenbizi programaren eskema edo algoritmoa idaztea komeni zaigu. Nondik hasi, ordea? Ataza honi aurre egiteko estrategia berri bat erabiliko dugu: problema sinplifikatzea. 4 letrako pasahitz guztien konbinazioak sortzea zaila denez, lehenbizi letra bateko pasahitzak kalkulatu ditugu. Ondoren bi letrakoak. Gero hiru, eta hortik aurrera segur aski programaren prozedura orokorra zein den deduzitzeko moduan izango gara. Has gaitezen.

Pasahitzak letra bakarra izango balu, hitz guztien zerrenda lortzea erraza litzateke:

```
1. foreach letra (letra_zerrenda)
    bistaratu (letra);
```

Demagun orain pasahitzak bi letra dauzkala, lehena kontsonantea eta bigarrena bokala. Pasahitz posible guztien zerrenda ateratzeko, kontsonante bakoitzeko bokal guztiak pasatu beharko genituzke banan-banan. Bi begizta habiaratu (bata bestearen barruan) behar dira horretarako. Hona eskema:

```
1. foreach $konts (@konts_zerrenda)
    1.1. foreach $bok (@bok_zerrenda)
        bistaratu ($konts-$bok)
```

Hurrengo pausoa, hiru letrako pasahitzak kalkulatzeko litzateke: *kontsonante-bokal-kontsonante* egiturari jarraitzen diotenak. Baina horrek murriztapen bat izango du: kontsonanteak ezin dira errepikatu. Begizta habiaratuak erabiliko ditugu berriro letren konbinazioak osatzeko, baina oraingoan baldintzazko egitura batekin lehen eta hirugarren kontsonanteak berdinak izan ez daitezen segurtatu behar dugu. Programaren eskema hauze litzateke:

```
1. foreach $konts1 (@konts_zerrenda)
    1.1. foreach $bok (@bok_zerrenda)
        1.1.1. foreach $konts2 (@konts_zerrenda)
            1.1.1.1. if ($konts1 != $konts2)
                bistaratu ($konts1-$bok-$konts2);
```

Aurrekoari laugarren eta azken letra gehitzea baino ez zaigu falta, bokala izango dena. Baina hemendik aurrera nahi beste letra gehitzeko moduan gara, dagoeneko badakigu pasahitzak osatzeko prozedura zein den eta.

Azken bokala gehitzean berriro ere kontuan izan behar dugu murriztapena: bi bokalak ezin dira berdinak izan. Baldintza guztiak betetzen dituzten hitzak bistaratzearekin batera, kontagailu batekin baliozko hitzen kopurua kontrolatuko du programak, bukaeran kopurua bistaratzuz.

```

1. foreach $konts1 (@konts_zerrenda)
  1.1. foreach $bok1 (@bok_zerrenda)
    1.1.1. foreach $konts2 (@konts_zerrenda)
      1.1.1.1. if ($konts1 != $konts2)
        1.1.1.1.1. foreach $bok2 (@bok_zerrenda)
          1.1.1.1.1. if ($bok1 != $bok2)
            bistaratu ($konts1-$bok1-$konts2-$bok2);
            $kontagailua++;
2. bistaratu ($kontagailua);

```

Lanaren alderdirik gogorrena egin dugu jada. Eskema aurrean hartu eta prest gaude kodea idazteko. Horretan hasi baino lehen, ohar bat: programak kontsonante eta bokalen zerrendak erabiltzen ditu. Aurretik ikusitako tartea eragileak ez digu balio kasu honetan letra-zerrendak osatzeko ("a".."z"). Eskuz idatzi beharko ditugu kontsonante eta bokalen *array* egiturak:

```

my @konts_zerrenda = ("b","c","d","f","g","h","j","k","l","m",
    "n","ñ","p","q","r","s","t","v","w","x","y","z");
my @bok_zerrenda = ("a","e","i","o","u");

```

pasahitzak.pl

```

#!/usr/bin/perl
use strict;
use warnings;
my @konts_zerrenda = ("b","c","d","f","g","h","j","k","l",
    "m","n","ñ","p","q","r","s","t","v","w","x","y","z");
my @bok_zerrenda = ("a","e","i","o","u");
my $kont = 0;
foreach my $konts1 (@konts_zerrenda) {
    foreach my $bok1 (@bok_zerrenda) {
        foreach my $konts2 (@konts_zerrenda) {
            # string-ak konparatzeko eragilea: ne
            if ($konts1 ne $konts2){
                foreach my $bok2 (@bok_zerrenda) {
                    if ($bok1 ne $bok2){
                        $kont++;
                        print("$konts1$bok1$konts2$bok2\n");
                    }
                }
            }
        }
    }
}
print("$kont konbinazio ezberdin!\n");

```

Komando-lerrotik programa abiatu eta hona bere irteera:

```
>perl pasahitzak.pl
bace
baci
baco
bacu
...
zuyo
9240 konbinazio ezberdin!
>
```

2. maila

8. kasu praktikoa: hitzen karaktere kopurua

Esperimentu baterako, teklatu bidez 5 hitz jaso eta honako kalkuluak egiten dituen programa eskatu digute: hitzik luzeena, motzena, batez besteko karaktere kopurua eta karaktere kopuru totala.

Ariketa egin ahal izateko, funtzio berri bat behar dugu lehenik: `length()`. Funtzio horrek argumentu bezala pasatutako *string*-aren karaktere kopurua itzultzen du. Adibidez:

```
$string = "hiru";
print(length($string)); # 4
```

Adibide gisa, hona `hitzLuzera.pl` programaren (izen hori jarri diogu) exekuzio-dei bat:

```
>perl hitzLuzera.pl
Idatzi 5 hitz banan-bana. Bakoitza idatzi ostean sakatu return
urtarrila
otsaila
martxoa
apirila
maiatza
Hitzik luzeena: urtarrila      Karaktere kopurua: 9
Hitzik laburrena: otsaila     Karaktere kopurua: 7
Karaktere kopurua guztira: 37
Batazbesteko karaktere kopurua hitzeko: 7.4
>
```

Ataza ebazteko moduari dagokionez, beti bezala aukera bat baino gehiago daude. Guk bat proposatuko dizuegu, baina kontuan izan gurea ez dela beti itsu-itsuan jarraitu beharreko erdua. Lehenbizi egin ahalegina ariketa zeuen kasa ebazten eta ondoren konparatu guk emandako soluzioarekin.



.

Lehenbizi programaren algoritmoa idatziko dugu, hau da, pausoz pauso zer egin behar duen. Hona hemen:

1. Errepikatu 5 aldiz:

- 1.1. Irakurri hitz bat.
- 1.2. Bere karaktere kopurua kalkulatu.
- 1.3. Begiratu ea handiagoa den aurreko hitz guztiak baino.
- 1.4. Begiratu ea txikiagoa den aurreko hitz guztiak baino.

2. Bistaratu handiena, txikiena, eta batez besteko zein karaktere kopuru totalak.

Lehen pausoko begizta inplementatzeko `while`, `foreach` edo `for` egitura erabil dezakegu, hiruretako edozein. Teklatu bidez jasotako hitzak ordenan gordetzeko egitura egokiena `array`-a litzateke, guk `@guztiak` izeneko erabiliko dugu. Baina aldagai gehiago beharko ditu programak: bat teklatutik jasotako hitzak gordetzeko, `$hitza`. Beste bat hitzen karaktere kopurua gordeko duena, `$karaktereak`. Bi aldagai gehiago karaktere kopuru handiena eta txikiena gordetzeko, `$handiena` eta `$txikiena`. Eta amaitzeko, karaktere kopuru totala gordetzeko erabiliko duguna, `$karakGuzti`;

Zehaztu dezagun eskema pixka bat gehiago:

1. Errepikatu 5 aldiz:

- 1.1. Irakurri teklatutik `$hitza` eta gorde `@guztiak` bektorean
- 1.2. Gorde bere karaktere kopurua `$karaktereak` aldagaian
- 1.3. `if ($karaktereak > $handiena) orduan`
`$handiena = $karaktereak`
- 1.4. `if ($karaktereak < $txikiena) orduan`
`$txikiena = $karaktereak`

2. Bistaratu `$handiena`, `$txikiena`, `$karakGuzti` eta batez besteko karaktere kopurua

Begiztaren gorputzean, teklatu bidez idatzitako hitzak `@guztiak` array-an gordeko ditugu banan-banan. Bakoitzaren karaktere kopurua kalkulatu eta aurrekoak baino handiagoa/txikiagoa den begiratuko dugu ondoren; modu horretan, `$handiena` eta `$txikiena` aldagaietan hitzik handienaren eta txikienaren karaktere kopurua izango dugu hurrenez hurren. Sistema honekin karaktere kopuruak gordetzen ditugu, baina bada arazo bat: programa-bukaeran ez genuke jakingo kopuru horien jatorrizko hitzak zein diren. Nolabait jakin behar dugu hitzik luzeena eta motzena `array`-ko zein posiziotan dauden gordeta. Bi indize erabil ditzakegu lan horretarako, `$indHandi` eta `$indTxiki`, hitzik luzeenaren eta laburrenaren indizeak gordetzeko.

Proposatzen dugun estrategiak badauka arazo praktiko bat: aldagai ugari erabiltzen ditugu eta beraien balioak hasieratu egin behar dira. Hau da, begiztaren lehen itzulian, `$handiena` eta `$txikiena` aldagaiek zer balio izango dute? Zein baliorekin hasieratu? Gure proposamena da begiztaren lehen iterazioan `$handiena` eta `$txikiena` aldagaiak lehen hitzaren karaktere kopuruarekin hasieratzea.

Aldagai asko aipatu ditugu eta baliteke horrek nahasmena eragin izana. Jar ditzagun guztiak ordenan:

- `@guztiak`: Teklatu bidez jasotako hitz guztiak gordetzeko.
- `$hitza`: Teklatu bidez jasotako uneko hitza.
- `$karaktereak`: Hitzen karaktere kopurua gordetzeko.
- `$handiena`: Hitzik luzeenaren karaktere kopurua.
- `$txikiena`: Hitzik laburrenaren karaktere kopurua.
- `$indHandi`: Hitzik luzeenaren indizea, `@guztiak` bektoreko posizioa.
- `$indTxiki`: Hitzik laburrenaren indizea, `@guztiak` bektoreko posizioa.
- `$karakGuzti`: Hitz guztien karaktere kopuru totala.

Orain arteko guztia biltzen duen algoritmoaren azken bertsioa:

1. `foreach $i (1..5):`

1.1. Teklatutik `$hitza` irakurri eta gorde. `$guztiak[$i] = $hitza`

1.2. Gorde bere karaktere kopurua `$karaktereak` aldagaian eta gehitu karaktere kopurua

1.2. `if` (lehen iterazioa) orduan hasieratu:

`$handiena = $txikiena = $karaktereak`

`$indHandi = $indTxiki = $i`

```
1.3.if ($karaktereak > $handiena) orduan
```

```
    $handiena = $karaktereak
```

```
    $indHandi = $i
```

```
1.4.if ($karaktereak < $txikiena) orduan
```

```
    $txikiena = $karaktereak
```

```
    $indTxiki = $i
```

2. Bistaratu \$handiena, \$txikiena, \$karakGuzti eta batez besteko karaktere kopurua

Programaren kodea:

hitzLuzera.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my ($hitza, @guztiak, $karaktereak, $handiena, $txikiena,
    $indHandi, $indTxiki, $karakGuzti);
print("Idatzi 5 hitz banan-banan. Bakoitza idatzi ostean
    sakatu return\n");

foreach my $i (1..5) {
    $hitza = <STDIN>;
    chomp($hitza);
    $karaktereak = length($hitza);
    $guztiak[$i] = $hitza; # edo push(@guztiak, $hitza);
    $karakGuzti += $karaktereak;
    # eragile berria: gehitu eta esleitu

    if ($i == 1) {
        $handiena = $karaktereak;
        $txikiena = $karaktereak;
        $indHandi = $i;
        $indTxiki = $i;
    }
    if ($karaktereak > $handiena) {
        $handiena = $karaktereak;
        $indHandi = $i;
    }
    if ($karaktereak < $txikiena) {
        $txikiena = $karaktereak;
        $indTxiki = $i;
    }
}
```

```
print("Hitzik luzeena: $guztiak[$indHandi]\tKaraktere
      kopurua: $handiena\n");
print("Hitzik laburrena: $guztiak[$indTxiki]\tKaraktere
      kopurua: $txikiena\n");
print("Karaktere kopurua guztira: $skarakGuzti\n");
print("Batazbesteko karaktere kopurua hitzeko: ");
print($skarakGuzti/5);
```

Eragile berri bat sartu dugu aurreko programan: +=. Berez ez da berri-berria ere, esleipen eta batuketaren forma laburtua baizik. Hurrengo bi sententziak baliokideak dira:

```
$karakGuzti += $karaktereak;  
$karakGuzti = $karakGuzti + $karaktereak;
```


4. Sarrera/Irteera

4.1. IRAKURRI ETA IDATZI

Orain arteko ahaleginak bideratu ditugu Perl-en oinarritzko egiturak ikasi eta horiek programa ezberdinetan aplikatzera. Esan liteke gure sukalde honetako tresnarik garrantzitsuenak ezagutzen ditugula, baita horiek nola erabili behar ditugun ere. Memento honetan, normala den bezala, benetako platerak kozinatzen hasteko irrikan gaude. Lanean hasteko lehengai egokiak baino ez zaizkigu falta. Izan ere, zaila da ganorazko ezer prestatzea lehengairik ezean. Gure sukalde berezi honetako lehengaiak datuak izango dira. Idatziko ditugun programen benetako erabilgarritasuna hein handi batean egongo da bide ezberdinetatik datuak jaso, prozesatu, eta ondoren emaitzak bueltatzeko gaitasunaren menpe. Horretaz arituko gara hain zuzen ere kapitulu honetan: datuak irakurri eta idazteko Perl-ek eskain-tzen dituen metodo nagusiak zein diren aztertuko dugu.

Programak ingurunearekin komunikatzeko, datuak irakurri eta idazteko, dauzkan bide ezberdinen multzoari **Sarrera/Irteera** (ingelesez *IO*, *Input/Output*) deitu ohi zaio. Finean, programa batek datuak jaso eta bueltan emaitzak itzultzeko hiru bide baino ez dauzka:

- **Sarrera/Irteera estandarra:** teklatura eta pantaila. Erabiltzaileak teklaturaren bidez programari datuak pasatu eta honek prozesatu ostean emaitzak pantailan bistaratuko ditu. Bide hau erabiliz, datuak exekuzio-garaian jasotzen ditu programak.
- **Komando-lerroa:** komando-lerrotik programa abiaraztean, programa-izenaren ondoren pasatu nahi dizkiogun datuak idaztea.
- **Fitxategi bidezko sarrera/irteera:** fitxategiak erabiltzea datuak irakurri eta idazteko.

Dagoeneko ezagutzen dugu **Sarrera/Irteera estandarra**. Badakigu teklatu bidez idatzitakoa jasotzen `<STDIN>` eragilea erabiliz, baita `print` funtzioarekin pantailan emaitzak nola bistaratu ere. Kapitulu honetan, aurrekoaz gain beste bideak ere aztertuko ditugu, bakoitzaren ezaugarri nabarmenenak azpimarratuz.

4.2. DESKRIBATZAILEAK

Gaian erabat murgildu aurretik, komeni zaigu kontzeptu berri bat azaltzea. Bai **Sarrera/Irteera estandarrekin** eta bai **fitxategiekin** lan egitean, Perl-ek **deskribatzaile** deituriko aldagai bereziak erabiltzen ditu. Deskribatzailea, sarrera edo irteera jakin baten helbide-izena da, Perl-ek nahitaez behar duena datuak nondik irakurri eta non idatzi jakiteko. Aurredefinitutako hiru deskribatzaile estandar dauzka Perl-ek: STDIN, STDOUT eta STDERR.

- **STDIN:** datu-sarrera estandarra, nondik irakurri erakusten duena. Guri dagokigunez, teklatura izango da sarrera estandarra.
- **STDOUT:** datu-irteera estandarra, irteerako datuak non idatzi erakusten duena. Guretzat pantaila izango da irteera estandarra.
- **STDERR:** errore-irteera estandarra, programako errore-mezuak idazteko erabiltzen dena.

Dagoeneko sarrera estandarretik datuak irakurtzen badakigu, `<>` eragilea erabiliz:

```
$lerro = <STDIN>; #sarrera estandarretik lerro bat irakurri
edo aurrekoaren baliokidea den:
```

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

Irteera estandarra da erabiltzaileari emaitzak erakusteko aukagun bidea. Nahiz eta horren jakitun izan ez, `print` funtzioa erabili den bakoitzean, irteera estandarra erabiltzen aritu gara.

```
print("kaixo, zer moduz?\n");    #irteera estandarrean idatzi
edo bere baliokidea den:
```

```
print(STDOUT "kaixo, zer moduz?\n");
```

Azken `print` funtzioan, STDOUT eta ondorengo *string* argumentua ez ditugu komaz banandu. Normalki, funtzio batek argumentu bat baino gehiago hartzen dituenean, komaz banatzen dira. Baina kasu honetan ez. Nahastu gabe beraz.

Hirugarren deskribatzaile estandarra, errore-irteerarena, ez dugu orain artean inoiz erabili; eta oraingoz bere horretan utziko dugu. Azalduko dugu aurrerago nola eta zertarako erabili.

Azaldu berri ditugun deskribatzaileak estandarrak dira, edozein programak erabil ditzake. Dena den, fitxategiekin lan egiteko, fitxategi bakoitzeko deskribatzaile berri bat sortu behar da lehenbizi. Deskribatzaile berriak nola sortu eta erabili fitxategiei eskainitako atalean azalduko dugu.

4.3. SARRERA/IRTEERA ESTANDARRA

Dagoeneko ondo dakigu Sarrera/Irteera estandarra nola erabili erabiltzaileak idatzitako datuak jasotzeko (<STDIN>), bai eta irteerako datuak pantailan bistaratzeko ere (print).

Ondoko programa honek lerroen karaktere kopurua bistaratuko du pantailan erabiltzaileak idatzi ahala. Amaitzeko nahikoa izango da lerro hutsa idaztearekin. *String* baten karaktere kopurua jakiteko, `length()` funtzioa erabil dezakegu, zeinak argumentu bezala pasatutako *string*-aren karaktere kopurua itzultzen duen.

input.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my $lerro;
my $kopuru;
print("Idatzitako lerroen karaktere kopurua kalkulatu-
    dut\n");
print("Amaitzeko, sakatu return\n");
while ($lerro = <STDIN>) {
    $kopuru = length($lerro);
    print (" $kopuru\n");
}
```

Programak dakarren berritasun bakarra `while` egituraren baldintzan dago: (`$lerro = <STDIN>`) . Sententzia honek teklatu bidez idatzitakoa gordetzen du `$lerro` aldagaian, eta hain zuzen ere aldagai horren edukia izango da baldintzak ebaluatuko duena. Aldagaia hutsik ez dagoen bitartean, `while`-aren baldintza beteko da (*egiazkoa*) eta begiztaren gorputzeko aginduak exekutatuko ditu. Aldagaia hutsik topatzen badu, ostera, baldintza ez da beteko (*false*) eta programa begiztatik aterako da.

Azken finean, `while ($lerro = <STDIN>)` egitura erabiltzearen ideia haxe da: segi datuak jasotzen erabiltzaileak lerro hutsa sartzen ez duen bitartean.

Idatzi programa editorean eta exekutatu. Aztertu arretaz bere portaera. Espero genuen bezalakoa da?

Ez. Programa begizta infinitu batean sartu da eta ezin da gelditu⁷. Detaile batek ihes egin digu, eta horrexek eragin du hain zuzen ere begizta infinitua: teklatu bidez idatzitako lerro guztiak *return* edo lerro-jauzi karaktereaz amaitzen dira, baita lerro hutsak ere. Beraz, lerro guztiek dute gutxien-gutxienez karaktere

7. Programaren exekuzioa eteteko, sakatu *Ctrl-c*

bat (bukaerako `\n`), `while`-aren baldintza beti egiazkoa eginez eta programa begiztatik atera ezinik utziz.

Hartu tarte bat pentsatzeko ea nola konpon dezakegun begizta infinituaren arazoa. Pista: gure soluzioak `length()` funtzioa erabiltzen du.



.

Lerro guztiek gutxien-gutxienez bukaerako lerro-jauzi karakterea izango dutenez, hona nola moldatu dugun `while`-aren baldintza:

Input2.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my $lerro;
print("Idatzitako lerroen karaktere kopurua kalkulatu-
    dut\n");
print("Amaitzeko, sakatu return\n");
while (length($lerro = <STDIN>) > 1) {
    print (length($lerro));
    print("\n");
}
```

Bigarren programa honek begiratzen du ea `while`-aren baldintzan karaktere kopurua bat baino handiagoa den. Lerro hutsak bukaerako lerro-jauzia soilik izango du eta ez du baldintza beteko, programa begiztatik ateraz.

Aurreko bi programak idatzi eta exekutatu badituzu konturatuko zinen bistaraten duten balioa espero genuena baino bat handiagoa dela beti. Horren erruduna, berrirori ere, lerro-jauzi karakterea da. Bukaerako lerro-jauzi karakterea kendu eta arazoa konpontzeko, erabili `chomp($lerro)` funtzioa `print`-aren aurretik. Zure esku uzten dugu sententzia berria programari gehitzea.

4.4. KOMANDO-LERROA

Komando-lerroaren bitartez programari sarrerako datuak pasatzea erraza da: exekutatzera koan, programaren izenaren ondoren idatzitako guztia, sarrera-datu gisa jasotzen du programak. Idatzitako guztia `@ARGV` array berezian gorde eta programari pasatuko dio Perl-ek. Adibide batekin hobeto ulertuko dugu:

```
>perl komandoLerroa.pl Karmele 7 Erreterria
```

Programa abiatzean, izenaren ondoren idatzitako hiru argumentuak `@ARGV` array-an gordeko dira: `@ARGV = ("Karmele", "7", "Erreterria");`

Honen guztiaren erabilgarritasuna ikusteko, hona hemen komando-lerrotik bi zenbaki jaso eta haien batura itzultzen duen programa:

batura.pl

```
#!/usr/bin/perl
use warnings;
use strict;
print $ARGV[0] + $ARGV[1];
```

@ARGV aldagai berezia da, eta aldagai arruntekin beti egin ohi dugun arren, ez dugu my erabilita erazagutu beharrik.

batura.pl exekutatzerakoan, programa-izenaren ondoren bi batugaiak idatziko ditugu:

```
>perl batura.pl 3 12
15
>
```

Programa oso sinplea da, komando-lerroan idatzitako bi argumentuak jaso eta haien batura itzultzen du.

Pauso bat urrunago joanaz, aurreko programa orokortzen ahaleginduko gara, batugai kopuruak mugarik izan ez dezan. Hau da, nahi adina zenbaki idatzi komando-lerroan eta programak guztien batura itzuli beharko du.

Nola egingo zenuke? Kontuan izan argumentu guztiak @ARGV array-an gordetzen direla.



.....

Asmatu duzu nola egin? Begizta egokia hautatzean dago gakoa.

batura2.pl

```
#!/usr/bin/perl
use warnings;
use strict;
my $batura = 0;
my $zenbaki;
foreach $zenbaki (@ARGV) {
    $batura = $batura + $zenbaki;
}
print ("Batura: $batura\n");
```

`foreach` egitura erabiliz oso erraza da soluzioa: `@ARGV`-ko elementuak hartu banan-banan, eta gehitu bakoitza `$batura` aldagaiari. Begiztatik ateratzean, argumentu guztien batura izango dugu `$batura` aldagaian.

Perl ez da mugak ezartzearen aldekoa, eta komando-lerrotik nahi adina argumentu pasa dakizkioke programari. Hala ere, bide hau argumentu gutxiko programetarako da batez ere egokia.

Datuak irakurri edo jasotzeko bi iturri izan ditugu orain arte: teklatua eta komando-lerroa. Datuak bistaratu edo itzultzeko, aldiz, bakarra: pantaila. Baina konturatuko zinetenez, zenbait programaren irteera bistartzeko komando-lerroaren pantaila txikiegia da. Datu kopuru handia itzuli behar dugunean, emaitzaren zati bat «galdu» egiten da pantailan. Kasu horietarako, oso erabilgarria da komando-lerroak irteera *birbideratzeko* eskaintzen duen aukera. Hona hemen nola erabil dezakegun:

```
>perl programa.pl > fitx.txt
```

Programa ohiko moduan exekutatu da, baina `print` funtzioarekin inprimatutako mezuak pantailan agertu ordez, `fitx.txt` fitxategira zuzenduko ditu Perl-ek. Aurrez `fitx.txt` izeneko fitxategirik baldin badugu, haren eduki guztiak ezabatu egingo dira.

Adibide gisa, teklatu bidez idatzitako guztia pantailan bistaratzen duen programa idatziko dugu, amaiera seinale bezala lerro-jauzia erabiliz.

input3.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my $lerro;
while (length($lerro = <STDIN>) > 1) {
    print $lerro;
}
```

Exekuta dezagun programa, irteera `fitx.txt` fitxategira birbideratuz. Go-goan izan programa abiatu ondoren idazten dugun guztia `fitx.txt` fitxategian gordeko dela.

```
>perl input3.pl > fitx.txt
kaixo!
Idazten dudana fitxategian gordeko da
lerroz-lerro. Amaitzeko nahikoa da return sakatzea
>
```

Idatzi ditugun lerroak pantailan bistaratu beharrea, `fitx.txt` fitxategian gorde ditu Perl-ek. Fitxategiaren edukia ikusteko, komando-lerrora joan eta editorearekin zabaldu, eta han agertuko zaigu idatzi dugun guztia.

Programaren irteera fitxategi batera birbideratzea oso baliagarria da emaitza pantailan bistartzeko baino luzeagoa denean, baita programaren irteera modu iraunkorrean gorde nahi dugunean ere. Batzuetan, baliteke gure programaren irteera fitxategi batean gorde nahi izatea, baina fitxategiaren aurretiazko edukia ezabatu gabe. Hau da, datu berriak bukaeran erantsiz. Horretarako nahikoa da gezi bikoitza erabiltzea birbideratzean:

```
>perl input3.pl >> fitx.txt
lerro hauek fitx.txt fitxategiari erantsiko dizkio
perl-ek aurreko edukia ezabatu gabe.
Agur!
>
```

Kontu handiz erabili beharrekoa da fitxategien birbideratzea. Nahi gabe eduki garrantzitsuko fitxategien gainean idatzi eta hauen edukia galtzeko arriskua baitago.

Zer gertatuko litzateke zure ustez honako agindu hau exekutatuz gero?

```
>perl input3.pl > input3.pl
```



.....

`input3.pl` programa abiarazi, eta teklatu bidez idatzitakoa `input3.pl` fitxategian gordeko lukeela, bere aurreko edukia ezabatuz. Hau da, gure programa ezabatuz.

4.5. FITXATEGIAK

Programari kanpotik datuak pasatzeko aztertu ditugun bi bideetan, erabiltzailea izan da programa sarrerako datuez hornitu duena. Baina datu kantitate handiak maneiatu behar ditugunean, datuak eskuz idazten aritzea ez da soluziorik onena. Kantitate handiekin lan egitean, ohikoena da datuak irakurri eta emaitzak itzultzeko fitxategiak erabiltzea. Atal honetan datuen sarrera/irteerarako fitxategiak nola erabili azalduko dugu.

Azalpenen lagungarri, zenbait programa garatuz joango gara, eta sarrerako fitxategi gisa `esaeraZaharrak.txt` fitxategia erabiliko dugu. Liburuarekin batera datorren CDan topatuko duzu; kopia handik `nerePerl` laneko direktoria.

Fitxategiak esaera zaharrak jasotzen ditu, lerro bakoitzeko bat, 458 esaera guztira. Ez da fitxategi handia, baina azalduko ditugun eragiketak ulertzeko nahikoa izango delakoan gaude. Noski, zeuen kabuz tamaina ezberdinetako fitxategiekin proba egitera animatzen zaituztegu.

Lanean hasi aurretik, testu-fitxategiei buruzko azalpen labur bat eman nahiko genuke: fitxategiak diskoan biltegitratutako datu multzoak dira. Guri dagokigunez, datu kantitate handiak irakurri eta idazteko erabiliko ditugu, eta fitxategietan gordeko dugun informazioa testu hutsezkoa izango da beti. Azken esaldi horrek badu bere garrantzia; izan ere, askotan testuarekin batera informazio gehigarria gorde ohi da fitxategietan: testuaren egiturari buruzko argibideak, hitzak eta esaldiak nola bistaratu, etab. Hori da, esaterako, Writer (*odt* formatua) edo Word (*doc* formatua) testu-prozesadoreek egin ohi dutena. Horregatik ez dira programaziorako egokiak. Fitxategiekin eragiketak egiteko Perl programak erabiltzean, aurretik fitxategi horiek testu hutsezkoak direla ziurtatu behar dugu (*txt* formatua normalki).

Gure programetan fitxategiekin lan egin ahal izateko, hiru pauso hauei jarraituko diegu beti:

1. Fitxategia ireki.
2. Datuak irakurri edo idatzi.
3. Fitxategia itxi.

Fitxategia irekitzean, lehenbiziko eginkizuna fitxategi-izena deskribatzaile batekin lotzea da. Aurrerantzean, deskribatzaile hori erabiliko du Perl-ek fitxategitik datuak irakurri edo bertan idazteko. Fitxategia zabaldu eta deskribatzailearekin lotzeko, `open()` funtzioa dauka Perl-ek:

```
open(FITX_DESKR, FITX_IZEN);
```

Bi argumentu dauzka funtzioak: lehena, fitxategi deskribatzailearen izena, `FITX_DESKR`. Perl-ek hartara behartzen ez gaituen arren, fitxategi deskribatzaileak letra larriz idazteko ohitura zabaldua dago. Bigarren argumentua, `FITX_IZEN`, zabaldu nahi dugun fitxategiaren izena. Horrekin batera, karaktere berezien bitartez adieraziko da fitxategia irakurtzeko, idazteko edo eransteko den.

Posible da, oso ohikoa ez den arren, aldagai bat erabiltzea fitxategi deskribatzaile gisa:

```
open($fitxDeskr, FITX_IZEN);
```

Adibidez, `nereFitx.txt` fitxategia irakurketarako zabaldu nahiko bagenu eta `FITX` deskribatzailearekin lotu, honela egingo genuke:

```
open(FITX, "nereFitx.txt");
```


Fitxategi-izenari karaktere berezirik gehitu ez diogunez, Perl-ek soilik irakurketarako dela interpretatuko du, eta ezin izango dugu fitxategi horretan ezer idatzi. Aurreko sententziaren baliokidea da honako hau:

```
open(FITX, "<nereFitx.txt");

# ireki nereFitx.txt irakurketarako
```

Zabaldut nahi dugun fitxategiaren izena zein den esan diogu Perl-i `open()` funtzioan, baina non bilatu behar du `nereFitx.txt`? Besterik adierazi ezean, Perl-ek uneko direktorioan bilatuko du fitxategia, programa abiarazi dugun direktorioan bertan alegia. Zabaldut nahi dugun fitxategia beste direktorio batean baldin badago, helbide osoa idatzi behar da modu honetan:

```
open(FITX, "c:/programazioa/nerePerl/nereFitx.txt");
```

Windows-en erabiltzailea bazara, konturatuko zinen helbidea idaztean `/"` barra jarri dugula ohiko `"\"` barraren orde. Windows-en notazioa erabiliz, horrela idatzi beharko genuke:

```
open(FITX, "c:\\programazioa\\nerePerl\\nereFitx.txt");
```

Baina orduan buruhaute franko izango genituzke. Izan ere `"\"` barrak esanahi berezia dauka Perl-en, eta ondo asko dakigun bezala komatxo arteko `\n` lerro-jauzi bezala interpretatuko luke Perl-ek, eta ez helbidearen puska bezala. Gure arazoak soluzio erraza dauka, Windows-ek bere barne-funtzionamendurako onartzen baitu aurrerako barra `/"`. Linux erabiltzaileek ez daukate inongo problemarik, aurrera begirako barra `/"` erabiltzen baita beti helbideak idazteko. Beraz, garbi dago zein izango den gure aholkua: helbideak idaztean beti `/"` barra erabili.

Fitxategiekin eragiketak egitean, oso erraza da akatsen bat egitea: fitxategiaren izena zuzen idatzi ez (`newreFitx.txt`), fitxategia ez dago guk uste genuen direktorioan, une horretan fitxategia beste norbaitek dauka irekita, etab. Horrelako akatsen bat gertatzen denean, Perl-ek ez du exekuzioa eten eta errore-mezurik bistartzen, baina programaren portaera ez da izango espero bezalakoa, eta arazoa zerk eragin duen asmatu ezinik ibiliko gara.

Horrelakorik gerta ez dadin, `open()` or `die()` egitura erabiliko dugu:

```
open(FITX, "c:/programazioa/nerePerl/nereFitx.txt") or
die("Errorea fitxategia zabaltzean\n");
```

Fitxategia zabaltzean huts egiten badu `open()` funtzioak, or egituraren bigarren zatia exekutatuko da, eta `die()` funtzioak «Errorea fitxategia zabaltzean» mezua pantailaratu eta programaren exekuzioa bertan behera utziko du. Fitxategia irekitzean errorerik gertatu ez bada, programak aurrera jarraituko du.

2. maila

Fitxategia irekitzeko aginduak huts egin badu, badago erreareari buruzko xehetasunak jakiterik \$! aldagai berezia erabilita. Perl-ek errearearen xehetasunak aldagai horretan gordetzen ditu. Ohiko bi errore-mezu hauek dira:

-`"No such file or directory"`: Ez da aurkitu izen horretako fitxategi edo direktoriorik.

-`"permission denied"`: Baimena ukatua.

Adibidez, existitzen ez den `nereFitx.txt` fitxategia zabaltzen ahalegintzen da hurrengo programa:

nereFitx.pl

```
#!/usr/bin/perl
use strict;
use warnings;
open(FITX, "nereFitx.txt") or die("Errorea: $!\n");
...
```

Mezu hau jasoko dugu exekutatzean:

```
>perl nereFitx.pl
Errorea: No such file or directory
>
```

Kasu batzuetan baliagarria da jakitea errorea zerk eragin duen.

Fitxategiekin lan egitean pauso berdinei jarraituko diegu beti: zabaldu, irakurketa/idazketa eragiketak egin, eta amaieran fitxategia itxi. Azken eragiketa horretarako, `close()` funtzioa erabiliko dugu. Funtzio horrek argumentu bakarra dauka, itxi nahi dugun fitxategiaren deskribatzailea.

```
open(FITX, "nereFitx.txt") or
    die("Errorea: ezin fitxategia zabaldu\n");
# irakurketa/idazketa eragiketak
close(FITX);
```

Badakigu fitxategiak zabaldu eta ixten. Hurrengo ataletan fitxategietatik datuak irakurtzeko bi modu ezberdin ikusiko ditugu: bata lerroz lerro, eta bestea fitxategiaren edukia osorik, pauso bakarrean, irakurtzen duena.

4.5.1. Lerroz lerro

Fitxategien edukia lerroz lerro irakurri ahal izateko, teklatutik lerro bat irakurtzeko erabili dugun <STDIN> egituraz baliatuko gara, baina STDIN-en lekuan zabaldu berri dugun fitxategi deskribatzailearen izena jarriz, <FITX>. Ondoko sententziak FITX deskribatzailearekin lotutako fitxategitik lerro bat irakurri eta \$lerro aldagaian utziko du:

```
$lerro = <FITX>;
```

Fitxategiko lerroak banan-banan irakurtzeko, while begizta erabiliko dugu <FITX> egiturarekin batera. Honako programa honek, komando-lerrotik pasatutako fitxategia zabaldu, eta lerroz lerro bere edukia pantailaratzen du dagokion lerro-zenbakiarekin batera:

fitxLerroz.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my $lerroa;
my $lerro_zenb = 0;
# argumentu bezala jasotako fitxategia ireki
open(FITX, $ARGV[0]) or
    die("Errorea! Ezin $ARGV[0] fitxategia zabaldu\n");
# fitxategia lerroz-lerro irakurri
while ($lerroa = <FITX>) {
    # lerro zenbakia eta $lerroa pantailaratu
    $lerro_zenb++; # $lerro_zenb = $lerro_zenb + 1;
    print ("$lerro_zenb: $lerroa");
}
# fitxategia itxi
close(FITX);
```

Idatzi berri dugun programa erabil genezake esaeraZaharrak.txt fitxategiaren edukia bistartzeko:

```
>perl fitxLerroz.pl esaeraZaharrak.txt
1: A zer pareta, karakola eta bareta.
2: Abadearen lapikoa, txikia baina gozoa
...
458: Zuri guztiak ez dira irin
>
```

Azter dezagun arretaz programaren egitura: fitxategi-izena komando-lerrotik jasotzen du programak eta \$ARGV[0]-n gorde. Programak beharko dituen aldagaiak definitu eta hasieratu ostean (\$lerro_zenb = 0), fitxategia zabaltzen ahalegintzen da, gerta litekeen edozein errore `open()` or `die()` egiturarekin egoki kudeatuz.

Programaren muina `while` egituran dago. Baldintzan jarritako \$lerroa = <FITX> sententziak fitxategitik lerro bat irakurri eta \$lerroa aldagaian uzteaz gain, *egiazkoa* itzuliko du fitxategian irakurtzeko lerrorik den artean. Baldintza betetzen denean (irakurtzeko lerrorik geratzen den artean), begiztaren gorputza exekutatu da, lehenbizi lerro-zenbakia eguneratuz eta ondoren pantailan lerro-zenbakia eta \$lerroa aldagaiaren edukia bistaratuz. Irakurtzeko lerro gehiagorik ez denean, `while`-aren baldintzak *faltsua* itzuliko du programa begiztatik ateraraziz.

Programaren azken sententziak fitxategia ixten du `close()` funtzioarekin.

Idatzi berri dugun programa, bere burua pantailan erakutsi dezan ere erabil genezake:

```
>perl fitxLerroz.pl fitxLerroz.pl
1: #!/usr/bin/perl
2: use strict;
3: use warnings;
4: my $lerroa;
5: my $lerro_zenb = 0;
6: # argumentu bezala jasotako fitxategia ireki
7: open(FITX, $ARGV[0]) or
    die("Errorea! Ezin $ARGV[0] fitxategia zabaldu\n");
8: # fitxategia lerroz-lerro irakurri
9: while ($lerroa = <FITX>) {
10:  # lerro zenbakia eta $lerroa pantailaratu
11:  $lerro_zenb++; # $lerro_zenb = $lerro_zenb + 1;
12:  print ("$lerro_zenb: $lerroa");
13: }
14: # fitxategia itxi
15: close(FITX);
>
```

Aurrerantzean, behin eta berriz erabiliko dugu fitxategia zabaldu eta lerroz lerro irakurtzeko egitura. Aldatuko dira begiztaren gorputzeko aginduak, ariketaren arabera, baina eskema orokorra bere horretan mantenduko da:

```
open(FITX, "adibide.txt") or
    die("Errorea fitxategia zabaltzean\n");
while ($lerroa = <FITX>) {
    sententziak;
}
close(FITX);
```

9. kasu praktikoa: bistaratu lehen N lerroak

Idatzi programa bat komando-lerrotik fitxategi-izena eta N zenbaki bat jaso eta fitxategiaren lehen N lerroak bistaratuko dituen.

lehenN.pl deituko dugu programa, eta adibide gisa, esaeraZaharrrak.txt fitxategiko lehen 4 lerroak bistartzeko erabili nahiko bagenu, haxe litzateke programa-deia eta bere irteera:

```
>perl lehenN.pl esaeraZaharrrak.txt 4
1: A zer pareta, karakola eta barea.
2: Abadearen lapikoa, txikia baina gozoa
3: Abendua jai huts eta gau huts.
4: Abenduko eguna, argitu orduko iluna
>
```

Fitxategiaren edukia bistartzeko ikusi berri dugun egituraz baliatuko gara, baina aldaera txiki batekin: fitxategia hasieratik bukaeraraino zeharkatu beharrean, lehen N lerroak baino ez ditugu bistaratu behar. Kasu, gerta liteke N zenbakia fitxategiaren lerro kopurua bera baino handiagoa izatea (edo biak berdinak). Kasu horretan, fitxategi osoa bistaratuko luke programak. while egituraren baldintza moldatuko dugu bi murriztapenak kontuan izan dituzan: jarraitu N. lerroa iritsi bitartean **eta** fitxategian lerrorik den artean.

Tarte bat hartu eta ahalegindu zaitez programa idazten. Ondoren konparatu zure soluzioa guk emandakoarekin.



.

lehenN.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my $n = $ARGV[1];
my $lerroa;
my $lerroKop = 0;
```

```
# argumentu bezala jasotako fitxategia ireki
open(FITX, $ARGV[0]) or
    die("Errorea! Ezin $ARGV[0] fitxategia zabaldu\n");
# baldintza: n. lerroa iritsi bitartean eta
# fitxategian irakurtzeko lerrorik den artean
while (($lerroKop < $n) && ($lerroa = <FITX>)) {
    $lerroKop++;
    print ("$lerroKop: $lerroa");
}
#fitxategia itxi
close(FITX);
```

4.5.2. Fitxategia osorik irakurri

Kasu batzuetan, fitxategia lerroz lerro irakurtzea baino egokiagoa izan daiteke osorik irakurtzea. Oso erraza da horretarako metodoa: `$lerroa = <FITX>` sententzia erabiltuta fitxategitik lerro bakar bat irakurtzen badugu aldiro, *array* motako aldagaia bat jarri eskalarraren lekuan, `@lerroa = <FITX>`, eta fitxategiko lerro guztiak gordeko ditu Perl-ek *array* egitura horretan.

Adibide gisa, hona hemen fitxategi-izena argumentu bezala jaso, bere edukia *array* egitura batean gorde eta pantailaratzen duen programa:

fitxLerroz2.pl

```
#!/usr/bin/perl
use strict;
use warnings;
open(FITX, $ARGV[0]) or
    die("Errorea! Ezin $ARGV[0] fitxategia zabaldu\n");
# edukia lerroz-lerro @lerroak array-an gorde
my @lerroak = <FITX>;
# array osoa bistaratu
print @lerroak;
close(FITX);
```

Fitxategiko lerro bakoitza `@lerroak` bektorearen elementu gisa gordeko du programak eta ondoren pantailan bistaratu.

Fitxategia osorik irakurtzeko metodoa oso sinplea den arren, fitxategi handiekin lan egitean metodo honek memoria-arazoak sor ditzake; izan ere, Perl-ek fitxategi osoko edukia gorde behar du *array* egitura batean (baita memorian ere), eta lerroz lerroko metodoarekin, aldiz, pauso edo iterazio bakoitzean lerro bakarra baino ez.

Dena den, fitxategi ez oso handiekin ari garenean, metodo hau erabilgarria izan liteke atazaren arabera. Adibidez, demagun erabiltzaileari zenbaki bat idazteko eskatu, N, eta pantailan N. esaera zaharra bistaratzen duen programa nahi dugula. Lehenbiziko metodoarekin, lerroz lerro irakurri beharko genuke fitxategia N. lerro-rra iritsi arte. Bigarrenarekin, aldiz, zuzenean bistara genezake N. esaera zaharra. Begira bestela:

esaerak.pl

```
#!/usr/bin/perl
use strict;
use warnings;
open(FITX, "esaeraZaharrrak.txt") or
    die("Errorea! Ezin fitxategia zabaldu\n");
my @lerroak = <FITX>;
print ("Idatzi 0 eta $#lerroak arteko zenbaki bat: ");
my $zenb = <STDIN>;
print $lerroak[$zenb];
close(FITX);
```

Fitxategien edukia ez ezik, teklatutik jasotakoa (STDIN) ere gorde liteke *array* egitura batean. Datuak irakurtzean, fitxategi deskribatzailearen orde, STDIN jarri besterik ez dugu egin behar. Sarrerako datuen amaiera adierazteko, erabiltzaileak *Ctrl-D* sakatu behar du Linux sistemetan, eta *Ctrl-Z*, aldiz, Windows erabiltzen ari bada. Programa honek erabiltzaileak teklatu bidez idatzitako guztia jasotzen du *array* egitura batean, eta ondoren pantailan bistaratzen du:

inputStd.pl

```
#!/usr/bin/perl
use strict;
use warnings;
print("Idatzi testua, amaitzeko sakatu Ctrl-D (Linux) edo
    Ctrl-Z (Windows)\n");
# sarrera estandarretik jasotakoa array-an gorde
my @lerroak = <STDIN>;
print @lerroak;
```

10. kasu praktikoa: bistaratu azken N lerroak

Aspaldi honetan fitxategiekin lanean gabiltza buru-belarri, eta konturatu gara lana asko erraztuko ligukeela honelako programa batek: argumentu gisa fitxategi-izena eta N zenbaki bat jaso, eta pantailan fitxategiaren **azken** N lerroak bistaratzten dituen.

Izena ere badugu, `azkenLerroak.pl`. Adibide gisa, `esaeraZahararak.txt` fitxategiaren azken 4 lerroak bistartzeko erabiliko bagenu, hona programa-deia eta lortuko genukeen emaitza:

```
>perl azkenLerroak.pl esaeraZahararak.txt 4
Azken 4 lerroak:
Zuhaitz ihartua, sutara.
Zuharrari ez eska gari
Zura berago, harra barrenago
Zuri guztiak ez dira irin
>
```

Pista: fitxategia kolpe batez irakurtzeko metodoa erabili. Gogoratu *array* bateko azken N elementuak bistartzeko modua zein den:

```
print @array[-3 .. -1];    # bistaratu azken 3 elementuak
```

Horra ariketa egiteko beharrezko (ia) azalpen guztiak. Hemendik aurrerakoak zeure esku.



.

Programaren kodea:

azkenLerroak.pl

```
#!/usr/bin/perl
use strict;
use warnings;
open(FITX, $ARGV[0]) or
    die("Errorea! Ezin $ARGV[0] fitxategia ireki\n");
my @lerroak = <FITX>;
print ("Azken $ARGV[1] lerroak:\n");
print @lerroak[-$ARGV[1] .. -1];
```

4.5.3. Fitxategietan idatzi

Fitxategietatik datuak irakurtzen badakigu, baina nahi duguna fitxategietan datuak idaztea bada, prozedura ez da asko aldatzen. Lehenbiziko lana, beti egin beharrekoa, fitxategia zabaltzea izango da, baina kasu honetan idazketarako dela zehaztuz. Fitxategi-izenari aurretik ">" eranstea baino ez dugu egin behar horretarako:

```
open(FITX, ">fitxIzen.txt");
```


Agindu horrekin, `fitxIzen.txt` izeneko fitxategi berri bat irekiko dugu idazketarako, edo aurrez izen bereko fitxategirik existitzen bada, hura zabalduko du Perl-ek eta haren edukia ezabatuko. Gure asmoa fitxategi baten edukia mantendu eta bukaeran datu berriak gehitzea bada, gezi bikoitza ">>" erantsi behar diogu aurretik izenari:

```
open(FITX, ">>fitxIzen.txt");
```

Behin fitxategia idazketarako zabaldu dugula, datuak idazteko `print` funtzioa erabiliko dugu fitxategi deskribatzailearekin batera:

```
print(FITX "fitxategian idatziko den testua.\n");
```

Adibide bezala, idatz dezagun fitxategiak kopiaitzen dituen programa. Komando-lerroaren bidez bi argumentu jasoko ditu, iturburu- eta helburu-fitxategiak, eta lehengoaren edukia kopiaituko du bigarrenean. Hona bere erabilera erakusten duen programa-dei posible bat:

```
>perl fitxKopi.pl fitx1.txt fitx2.txt
```

`fitx1.txt` fitxategiaren edukia kopiaituko du `fitx2.txt` fitxategian.

fitxKopi.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my $lerroa;
# iturburu fitxategia irakurketarako ireki
open(FITXIN, $ARGV[0]) or
    die("Ezin $ARGV[0] fitxategia zabaldu\n");
# helburu fitxategia idazketarako ireki
open(FITXOUT, ">$ARGV[1]") or
    die("Ezin $ARGV[1] fitxategia sortu\n");
while ($lerroa = <FITXIN>) {
    print (FITXOUT $lerroa);
}
# fitxategiak itxi
close(FITXIN);
close(FITXOUT);
```

Fitxategiei buruzko atalarekin amaitzeko, hona beraiekin egin ditzakegun eragiketak biltzen dituen taula:

<code>open(FITX, "lizardi.txt");</code> <code>open(FITX, "<lizardi.txt");</code>	Fitxategia irakurtzeko zabaldu
<code>open(FITX, ">lizardi.txt");</code>	Fitxategia idazketarako zabaldu
<code>open(FITX, ">>lizardi.txt");</code>	Idazketarako zabaldu datuak bukaeran erantsiz
<code>print("edozein testu\n");</code> <code>print(STDOUT "edozein testu\n");</code>	Idatzi testua irteera estandarrean, pantailan
<code>print(FITX "edozein testu\n");</code>	Idatzi testua FITX fitxategian

Ariketak:

- 1.- Sarrerako datu gisa jaso fitxategi baten izena eta idatzi bere lerroak alderantzizko ordenan pantailaratuko dituen programa.
- 2.- Sarrera gisa jaso fitxategi baten izena eta garatu haren karaktere kopurua itzultzen duen programa.
- 3.- Argumentu bezala bi fitxategi jaso, eta idatzi lehenengoaren edukia bigarrenari erantsiko dion *script* edo programa.

5. Geldiunea erdibidean

Honaino bidelagun izan bazaitugu irakurle, ezagutzen dituzu jada Perl lengoaiaren oinarriak. Geure neurri eta beharretara egokitzen diren benetako programak idatzi eta erabiltzeko prest gaude beraz. Baina horretan hasi aurretik, beharrezkoa iruditu zaigu geldione bat egitea gure ibilbidean, arnasa hartu eta gure ibiliari errepasoa egiteko. Izan ere, programa aski konplexuak idatzi ahal izateko puntura iritsi gara dagoeneko. Eta zenbat eta programa luzeagoak idatzi, orduan eta probabilitate handiagoa izaten da kodean erroreak izateko, eta horrekin batera, programen irakurgarritasun-mailak berriz behera egin ohi du nabarmen. Bi kontu horiek izango ditugu hizpide kapitulu honetan: ohiko errorearen tratamendua, alde batetik, eta programa argi eta ulerterrazak idazteko estrategiak beste aldetik.

5.1. OHIKO PROGRAMAZIO-ERROREAK

«Gauzak gaizki atera badaitezke, gaizki aterako dira». Hala dio Murphy-ren legeak, eta zeure kasa hemen proposatutako ariketak egiten ahalegindu bazara, ziur Murphy-rekin bat egiten duzula. Izan ere, oso erraza da akatsen bat egitea kodea idaztean. Aldiz, soluzioa topatzea ez da beti nahi bezain erraza izaten, eta askotan huskeria batengatik begiak erre beharrean aritu ohi gara pantailaren aurrean. Horregatik, eta gehienok harri bertsuekin tupust egiten dugula kontuan izanda, ohiko errorearen zerrenda prestatu dugu

Programazio-erroreak bi multzotan sailka ditzakegu: sintaktikoak eta semantikoak. Lehenbiziko multzoan sartzen dira lengoaiaren sintaxia oker erabiltzearen ondorioz sortzen direnak. Programa exekutatzean, konpiladoreak antzeman egiten ditu errore sintaktikoak eta exekuzioa bertan behera uzten du errore-mezua itzuliz. Errore semantikoa izango dugu, programak funtzionatu bai, baina egin behar lukeena egiten ez duenean.

Mota batekoa zein bestekoa, programak erroreren bat daukanean, aztertu puntuz puntu azpiko zerrenda, segur aski iturburua aurkitzen lagunduko baituzu. Hona hemen, errore sintaktiko eta semantikoen eragile ohikoenak:

Sintaxi-erroreak

- Sententzia amaieran puntu eta koma (;) jartzea ahaztu zaigu.

\$batura = 6 + 7

- Perl-eko hitz erreserbatua letra larriz idatzi dugu (`if` beharrean `If`, `while` beharrean `While`). Hitz erreserbatuak letra xehez idatzi behar dira.
- *String* amaieran (edo hasieran) komatxoak (") jartzea ahaztu zaigu.
- Zabaldutako parentesi (), giltza {} edo kortxeteak [] ez ditugu itxi.
- *Array*-ekin gabiltzala, elementu bakarra atzitu nahi eta @ ikurra erabili dugu \$ erabili beharrean:

```
$semea = @izenak[2];
```

Errore semantikoak

- Teklatu edo fitxategitik datuak irakurri eta `chomp()` funtzioa (bukaerako lerro-jauzia kentzen duena) erabiltzea ahaztu zaigu.
- Konparaketa egitean, *string* eragileak (`eq`, `ne`, `lt`, `le`, `gt`, `ge`) eta zenbakiak konparatzeko eragileak nahastu ditugu (`==`, `!=`, `<`, `<=`, `>`, `>=`).
- Konparaketan, = (esleipen-ikurra) jarri dugu `==` jarri beharrean.
- Aldagai bat erabiltzen ari gara aurretik baliorik esleitu gabe.
- Amaierarik gabeko begiztan sartu da programa, kontagailua behar bezala ez eguneratzeagatik.

5.2. PROGRAMAZIO-ESTILO EGOKIA

Zenbat eta programa luze eta konplexuagoak idatzi, orduan eta ageriagoan gertatzen da kode garbi eta irakurterraza idazteak daukan garrantzia. Perl ez da beste programazio-lengoaia batzuk bezain zorrotza, eta ia beti izan ohi dira ataza bera gauzatzeko bide bat baino gehiago. Norberari hobekien egokitzen zaion estiloan idaztea bilatzen du ezaugarri horrek, eta lengoaiaren ikasketa eta erabilera asko erraztu dezake. Baina kontuz ibili ezean, programa ilun eta ulertezinak idazteko bidea ere izan daiteke.

Atal honetan, egoki programatzeko zenbait gomendio emango ditugu. Programa egokia deituko diogu, bere programazio-eginkizuna betetzeaz gain, kode garbi eta irakurterraza daukanari. Garbi izan behar dugu, kodea ez dugula soilik idatzi behar Perl-ek interpreta eta exekuta dezan, garrantzitsua da, halaber, beste erabiltzaileek irakurri eta ulertzeko moduko programak idaztea. Programazioan murgiltzen diren guztiei egokituko zaie lehenago edo beranduago besteek idatzitako kode puskak eta programak irakurtzea (edo guk geuk aurretik egindakoa), eta benetan, alde ederra dago Perl-entzat bakarrik idatzitakotik erabiltzaileak kontuan izanda idatzitakora. Egoki idatzitako kodea errazagoa da irakurtzen, ulertzen, baita mantentzen ere.

Adibide gisa programa bat jarri eta haren inguruan egoki programatzeko aholkuak emango ditugu segidan.

Lerro-jauziak

Kodea irakurgarria izan dadin, ahal dela lerro bakoitzeko sententzia bat idaztea komeni da. Ondoren datorren programan, nahita desegoki erabili ditugu lerro-jauziak, sententziak erdiz erdi ebakiz. Perl-ek berdin-berdin irakurriko duen arren, gutako edozeinek lanak izango ditu programa deszifratzen.

bikoitiBakoiti.pl

```
#!/usr/bin/perl
use strict; use
warnings; print
("Idatzi zenbaki bat: \n");
my $k = <STDIN>; chomp
($k); my $z = 1; my $a = 0;
my $b = 0; while ($z <=
$k) {if (($z % 2) == 0)
{$a = $a + $z;} else {$b
= $b + $z;} $z++;}
print("bakoitien batura: $b\n");
print("bikoitien batura: $a\n");
```

Programak erabiltzaileari zenbaki bat eskatu, N, eta 1 eta N arteko zenbakien bi batura kalkulatzeko ditu: zenbaki bikoitiena alde batetik, eta bakoitiena bestetik. Amaieran, kalkulatuak bi baturak bistaratuko ditu pantailan.

Esan bezala, Perl-i bost axola dio lerro-jauziekin sententziak nola banatzen ditugun, berak berdin-berdin irakurtzen du kodea. Baina goiko programa noizbait beste pertsona baten eskutara iristen bada, ez dut uste oso erabilgarria izango zaionik.

Hona aurreko programa bera, baina oraingoan lerro bakoitzeko sententzia bakarra idatzita:

bikoitiBakoiti2.pl

```
#!/usr/bin/perl
use strict;
use warnings;
print("Idatzi zenbaki bat: \n");
my $k = <STDIN>;
chomp($k);
```

```

my $z = 1;
my $a = 0;
my $b = 0;
while ($z <= $k) {
  if (($z % 2) == 0) {
    $a = $a + $z;
  }
  else {
    $b = $b + $z;
  }
  $z++;
}
print("bakoitien batura: $b\n");
print("bikoitien batura: $a\n");

```

Bigarren bertsio honetan programaren sententziak zein diren garbi ikus daiteke. Hortik aurrera ezer gutxi, irakurterraza izatetik nahiko urrun baitago oraindik programa.

Aldagai-izenak

Programa egokiak aldagai-izen deskriptiboak izan behar ditu, hau da, beren erabilerari buruzko pistaren bat eman behar digute.

```

$b, @k, %m                                # ezegokiak
$batura, @karaktereak, %maiztasuna       # egokiak

```

Tarteka, onargarria (eta erosoagoa) da aldagai-izen laburrak erabiltzea beren esanahia nabaria denean. Adibidez, kontagailu bidez kontrolatutako begiztak:

```

my $i = 0;
while ($i < 10) {
    $i++;
}

```

Zenbaki bikoiti eta bakoitien batura kalkulatzeko duen programari helduko diogu ostera. Oraingoan, aldagai-izen deskriptiboagoak erabiliko ditugu:

bikoitiBakoiti2.pl

```

#!/usr/bin/perl
use strict;
use warnings;
print("Idatzi zenbaki bat: \n");
my $zenbaki = <STDIN>;
chomp($zenbaki);
my $bikoiti = 0;
my $bakoiti = 0;
my $kont = 1;

```

```
while ($kont <= $zenbaki) {  
  if (($kont % 2) == 0) {  
    $bikoiti = $bikoiti + $kont;  
  }  
  else {  
    $bakoiti = $bakoiti + $kont;  
    $kont++;  
  }  
  print("bakoitien batura: $bakoiti\n");  
  print("bikoitien batura: $bikoiti\n");  
}
```

Ari da poliki-poliki itxuratzen gure programa. Sententziak garbi idatzita daude, eta aldagai bakoitzaren funtzioa zein den ere nabaria da. Hala ere, programa bloke bakar bat bezala ikusten da oraindik, eta ez dago batere garbi zein den kontrol-egituren eremua.

Kontrol-egiturak eta tabulazioa

Oso garrantzitsua da programaren egitura zehazteko zuriuneak eta tabulazioa erabiltzea.

- Koskak: zuriune edo tabulazioak erabili kontrol-egituren eremua argi eta garbi mugatzeko.
- Zuriuneak: programa barneko bloke ezberdinak garbi mugatzeko zuriuneak utzi haien artean.

Programa berridatziko dugu, tabulazioa eta zuriuneak egoki erabiliz:

bikoitiBakoiti3.pl

```
#!/usr/bin/perl  
use strict;  
use warnings;  
  
print("Idatzi zenbaki bat: \n");  
my $zenbaki = <STDIN>;  
chomp($zenbaki);  
my $bikoiti = 0;  
my $bakoiti = 0;  
  
my $kont = 1;  
while ($kont <= $zenbaki) {  
  if (($kont % 2) == 0) {  
    $bikoiti = $bikoiti + $kont;  
  }  
  $kont++;  
}
```

```

        else {
            $bakoiti = $bakoiti + $kont;
        }
        $kont++;
    }

    print("bakoitien batura: $bakoiti\n");
    print("bikoitien batura: $bikoiti\n");

```

Lehen bertsioaren eta hirugarren honen artean badago aldea gero!

Azken ukitu bat baino ez zaio falta programa egokia izan dadin.

Iruzkinak

Programari buruzko argibideak emateko erabiltzen dira iruzkinak, bere portaeran inongo eraginik izan gabe. # ikurraren ondoren lerro-jauzia bitarte idatzitakoa Perl-ek ez du interpretatuko, eta azalpenak emateko erabil dezakegu tarte hori. Egoki idatzitako iruzkinak irakurleari honako galdera hauek erantzuten lagundu beharko liokete: *zer egiten du programa honek? Eta, egin behar duen hori, nola egiten du?*

Iruzkinak laburrak baina argigarriak izan behar dira. Ez erabili kodean nabariak diren gauzak errepikatzeko.

bikoitiBakoiti4.pl

```

#!/usr/bin/perl
use strict;
use warnings;

print("Idatzi zenbaki bat: \n");
# Teklatu bidez idatzitakoa jaso
my $zenbaki = <STDIN>;
# Bukaerako lerro jauzia kendu
chomp($zenbaki);
# Aldagaiak hasieratu
my $bikoiti = 0;
my $bakoiti = 0;
# Kontagailua hasieratu
my $kont = 1;
# Begizta. $kont aldagaiak 1 eta $zenbaki
# arteko balioak hartuko ditu

```



```

while ($kont <= $zenbaki) {
    # Bikoitia da?
    if (($kont % 2) == 0) {
        # Gehitu zenbakia bikoitien sakuan
        $bikoiti = $bikoiti + $kont;
    }
    else {      # Bakoitia da
        # Gehitu zenbakia bakoitien sakuan
        $bakoiti = $bakoiti + $kont;
    }
    # Eguneratu kontagailua
    $kont++;
}
# Bistaratu emaitzak
print("bakoitien batura: $bakoiti\n");
print("bikoitien batura: $bikoiti\n");

```

Puntu honetara iritsita, programak ez du azalpen gehigarriarik behar edozein Perl programatzailek uler dezan.

Sarreran esandakoa errepikatuz amaituko dugu atala: egoki idatzitako kodea errazago eta azkarrago irakurri eta ulertzen da. Errore gutxiago izango ditu eta hauek detektatzea samurragoa izango da.

5.2.1. Aldagai bereziak

Esperientziadun programatzaileek usu erabiltzen dituzte aldagai bereziak Perl lengoaian programatzean. Aurki ikusiko dugun bezala, oso erabilgarriak dira eta kode-lerro asko aurreztu ditzakegu beraiek erabilita. Hala ere, aldagai bereziekin programen irakurgarritasunak behera egiten du, eta mesede baino kalte gehiago egitea ere gerta liteke. Baina noiz edo noiz beste norbaitek idatzitako kodea irakurri behar izango dugunez, ia ziur, komeni da aldagai bereziak zer diren eta nola erabiltzen diren aurrez ezagutzea.

Perl-ek, eskalar, *array* eta *hash* motako aldagai berezi sorta handia dauka. Atal honetan bizpahiru erabilienak baino ez ditugu aipatuko.

Aldagai berezien artean erabiliena `$_` da, Perl funtzio gehienek aldagai lehenetsi gisa erabil dezaketena. Adibide baten bidez azalduko dugu bere erabilera. Programa honek, komando-lerrotik pasatutako fitxategia lerroz lerro pantailaratzen du dagokion lerro-zenbakiarekin batera.

aldagaiBereGabe.pl

```
#!/usr/bin/perl
use strict;
use warnings;

open(FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia ireki\n");

my $lerro;
my $lerroZenb = 0;

while ($lerro = <FITX>) {
    $lerroZenb++;
    print("$lerroZenb $lerro");
}
close(FITX);
```

Programak while begiztarekin fitxategia lerroz lerro irakurtzen du, uneko lerroa \$lerro aldagaian utziz. Begiztaren gorputzean, lerro-zenbakia eta lerroa bera pantailaratzen ditu print() funtzioak. Aldagai arruntak erabili ditugu eta programak ez dauka aparteko berezitasunik.

Aldagai bereziak erabiltzea erraza litzateke: begiztan, inongo aldagairik erabili izan ez bagenu, Perl-ek \$_ aldagai lehenetsia erabili nahi dugula ulertuko luke, segidan datorren programak erakusten duen moduan:

aldagaiBerezi.pl

```
#!/usr/bin/perl
use strict;
use warnings;

open(FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia ireki\n");

my $lerroZenb = 0;

while (<FITX>) {
    $lerroZenb++;
    print("$lerroZenb ");
    print("$_");
}
close(FITX);
```

Begiztaren baldintzan, fitxategitik lerro berri bat irakurtzen duen <FITX> aginduak ez dauka bere edukia non gorderik, eta \$_ aldagai bereziari esleituko dio. Bere edukia bistaratzeko, nahikoa da `print("$_");` sententzia idaztea ondoren.

Baina argumentu bakarreko funtzio gehienek, argumenturik zehaztu ezean, \$_ aldagaia hartuko dute argumentu lehenetsizat. Goiko programaren portaera berbera dauka ondoko honek:

aldagaiBerezi2.pl

```
#!/usr/bin/perl
use strict;
use warnings;

open(FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia ireki\n");

my $lerroZenb = 0;

while (<FITX>) {
    $lerroZenb++;
    print("$lerroZenb ");
    print();
}
close(FITX);
```

\$_ aldagai bereziak erabilera ugari dauzka: argumentu bakarreko funtzio (`print()`, `chomp()`, `etab.`) gehienen argumentu lehenetsia izateaz gain, `foreach` eta `while` begiztetan ere erabil daiteke ikusi dugun moduan.

Bada beste aldagai berezi bat, \$., fitxategiaren uneko lerro-zenbakia gordezten duena. Programa berridatz dezakegu, aldagai arruntik batere definitu gabe:

aldagaiBerezi3.pl

```
#!/usr/bin/perl
use strict;
use warnings;

open(FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia ireki\n");

while (<FITX>) {
    print ("$. $_");
}
close(FITX);
```

Kode-lerro dezente aurreztu dugu aldagai bereziak erabilia, baina trebatua ez dagoenarentzat, lehen begiratuan programa iluna edo kriptikoa gerta liteke. Programak irakurri eta idazten trebatu artean, erabili kontuz aldagai bereziak.

5.2.2. *Tresna berriak*

Sarrerako kapituluan aipamen labur bat egin ostean, orain arte ez diogu arretarik eskaini programen edizio-lanerako testu-editoreei. Badira merkatuan programaziorako editore bereziak lengoaiaren erabilera asko errazten dutenak. Sarrerako kapituluan ez dugu haien berri eman, arreta osoa Perl lengoaiaren jartzea nahi izan dugulako, eta ez gainerako tresnen instalazio eta erabileran. Baina orain ez gara jada hasiberriak, badakigu programak idatzi eta exekutatzeko, eta Perl-en oinarritzko elementuak ere ezagunak zaizkigu. Une egokia izan daiteke, beraz, tarte bat hartu, eta gure programatzaile-lana erraztuko duen editore bati begira jartzeko. Sarea pixka bat arakatzeko gero, programaziorako editore mordoa topatuko dugu. Guk ez ditugu denak probatu, baina ezagutzen ditugun artean **Notepad++** gomendatzen dugu Windows erabiltzaileentzat (Notepad erabili dugu orain arte eta Notepad++ hemendik aurrera!). Linux erabiltzaileek ez dute aplikazio berririk instalatu beharrik, sistemak berarekin dakarren **Emacs** programaziorako editorea oso ona baita.

Notepad++ programaren instalazioa

Windows sistemako Notepad edo ohar-blokarekin alderatuta, programa berri honek abantaila ugari eskaintzen dizkio programatzaileari. Lengoaia ezberdin asko identifikatzeko gai izateaz gain (horien artean Perl, noski), bakoitzaren sintaxia nabarmentzen du. Fitxategiak fitxa ezberdinetan zabaltzen ditu, eta programaren instantzia bakarrean nahi adina fitxategi zabalik eduki ditzakegu. Horrekin batera, fitxategi bakoitzaren lerroak zenbakiturik agertzen dira, eta konpilazio-erroreak gertatzean errazagoa da honela errorea non den identifikatzea.

Aurreko guztiaz gain, software librea da eta doakoa, eta sarri ateratzen dituzte programaren eguneraketak. Laburbilduz: Windows erabiltzailea bazara, merezi du tarte bat hartu eta Notepad++ instalatzea.

Hona eman beharreko pausoak.

1. Zabaldu nabigatzailea eta jo helbide honetara:

<http://notepad-plus.sourceforge.net/es/site.htm>

2. Azpiko orrialdea zabalduko zaizu. Zuzenean, programa jaisteko txokora abiatuko gara.



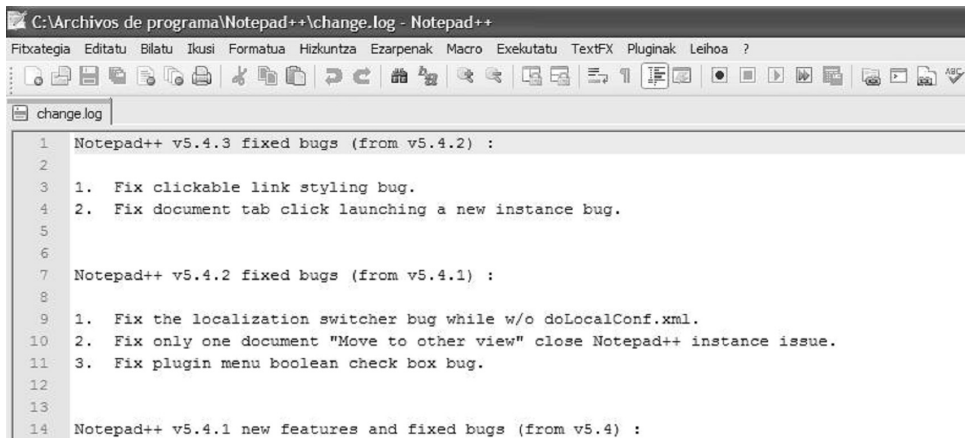
3. Orrialde honetan, programa jaisteko aukera ezberdinak ageri zaizkigu. Errazena instalatzaile exekutagarria jaistea denez, horixe izango da gure aukera.



4. Programaren azken bertsioa interesatzen zaigunez, **Installer.exe** testuaz amaitzen den zenbaki altueneko fitxategia aukeratu.

Package	Release	Filename
<u>notepad++ releases binary</u>		
Latest	npp 5.4.3 bin 	(2009-06-07 02:28)
		npp.5.4.3.bin.7z
		npp5.4.3.bin.zip
		npp.5.4.3.Installer.exe
		npp.5.4.3.release.md5
	npp 5.4.2 bin 	(2009-06-01 00:26)
	npp 5.4.1 bin 	(2009-05-27 23:43)
	npp 5.4 bin 	(2009-05-25 20:43)

5. Programa jaitsi, egin klik bikoitza bere gainean, eta jarraitu instalaziorako pausoei. Notepad++ programaren instalatzaileak zein hizkuntzatan instalatu nahi dugun galdetuko digu prozesuaren hasieran. Zorionez, programa euskaratuta dago eta posible da euskaraz instalatu eta erabiltzea. Instalazio-prozesua amaitzean, Notepad++ zabaldu eta honelako leiho bat agertuko zaigu:



```

1 Notepad++ v5.4.3 fixed bugs (from v5.4.2) :
2
3 1. Fix clickable link styling bug.
4 2. Fix document tab click launching a new instance bug.
5
6
7 Notepad++ v5.4.2 fixed bugs (from v5.4.1) :
8
9 1. Fix the localization switcher bug while w/o doLocalConf.xml.
10 2. Fix only one document "Move to other view" close Notepad++ instance issue.
11 3. Fix plugin menu boolean check box bug.
12
13
14 Notepad++ v5.4.1 new features and fixed bugs (from v5.4) :

```

Lehenbiziko lana, lengoaia lehenetsi bezala Perl ezartzea izango da. Horretarako: Ezarpenak-->Preferentziak... eta Dokumentu berria fitxa aukeratu. Hasierako hizkuntza jartzen duen lekuan aukeratu Perl. Hemendik aurrera, besterik zehaztu ezean, Notepad++ programarekin zabaldutako fitxategiak Perl lengoaian idatzitako programak direla ulertuko du Notepad++-ek, eta Perl-en sintaxia kolore bereizgarritz bistaratuko digu.

Orain artean komando-lerrotik editatzen genituen programak, honelatsu:

```
>notepad prog.pl
```

Notepad++ ez dugu komando-lerrotik abiaraziko, Windows-eko gainerako programak bezala, mahaigaineko laster-marka erabiliz edo programen zerrendan bere gainean klik eginez exekutatu beharko dugu aldiro. Gainerakoan, bere funtzionamendua ohiko editore batena da: fitxategiak zabaldu, idatzi, aldatu, gorde. Hori bai, ez ahaztu gero fitxategiak **p1** luzapenarekin gordetzea!

6. Adierazpen erregularrak

Programazio-lengoaia gehienak erabilera orokorrekoak dira, hau da, ataza edo zeregin ezberdinetarako erabil ditzakegu. Baina bakoitzaren ezaugarri propioek, eta erabiltzaileen komunitateak lengoaiaz egiten duen erabilerak, aplikazio-eremu jakin batzuetarako bereziki erabilgarri bihurtzen dute lengoaia, eta beste batzuetarako, aldiz, ez hainbeste. Perl-en ezaugarriak behinena, **adierazpen erregularren** bidez testuak tratatzeko eskaintzen duen erraztasuna da dudarik gabe. Gainera, Perl erabiltzaileen komunitatea bereziki aktiboa da, eta adierazpen erregularrak erabiliz testuak aztertu eta prozesatzen dituzten programa ugari ditu garatuak. Adierazpen erregularrak, labur-labur esanda, testuetan patroiak bilatzeko balio duten formulak dira. Kapitulu honetan, adierazpen erregularrak sortu eta erabiltzen ikasiko dugu, bai eta hauek hizkuntzaren azterketa eta prozesamendurako nola erabili ere.

Adierazpen erregularrak ez dira Perl-en ondare eskusiboa, eta beste lengoaia askotan ere topa ditzakegu: *python*, *C* edo *awk* bakar batzuk aipatzearren. Baina programazio-lengoaiak bakarrik ez, esaterako, *Word* edo *Writer* testu-prozesadoreek ere erabiltzen dituzte adierazpen erregularrak. Arazoa bateragarritasuna izaten da, batetik bestera sintaxia aldatu eta batean idatzitakoak bestean funtziona dezan aldaketak egin behar izaten baitira. Dena den, oinarrian dagoen tresna berdina da guztientzat.

Esan bezala, testuetan bilaketak egiteko erabiliko ditugu adierazpen erregularrak. Ideia zehatzagoa izan dezazuen, hona hemen zer-nolako eragiketak egin ahal izango ditugun:

- Bilaketa arruntak: hitz bat jaso eta haren agerpenak bilatu testuan.
- Bilaketa aurreratutak: hitz-bukaera jaso eta errimak aurkitu; karaktere berdinarekin hasi eta amaitzen diren hitzak bilatu; karaktere kopurua eman eta kopuru horretako hitzak bilatu (adb., 5 karakterekoak); etab.
- Fitxategi bat jaso eta edukia aztertu: hitz kopurua, bokal kopurua, hitz eta letra bakoitzaren maiztasuna, etab.

Eta askoz ere gehiago. Luze eta zabal arituko gara adierazpen erregularrei buruz kapitulu honetan, eta bidean testuen tratamendurako hainbat programa erabilgarri eraikiko ditugu. Kapitulu hauxe duzu, irakurle, liburuko mardulena orrialde kopuruari begiratuta, baita eduki aldetik ere. Baina ordainetan, orrialde hauetan azalduko da testu elektronikoak prozesatzeko Perl-en benetako potentziala.

Kapitulu honetan, behin eta berriz fitxategiak erabiliko ditugu programen sarrerako datu-iturri gisa. Espresuki horretarako prestatuak izan dira `esaeraLabur.txt` eta `xuxen.txt` fitxategiak. Lehenbizikoa esaera zaharren bilduma laburra da (57 esaera), eta bigarrena, aldiz, 80.000 hitz biltzen dituen hiztegi-fitxategia, bakoitza lerro banatan gordea. Liburuarekin batera datorren CDan topatuko dituzu; kopia handik `nerePerl` laneko direktoriora.

6.1. PATROIAK

Sarreran aipatu bezala, adierazpen erregularrak testuetan **patroiak** bilatzeko tresnak dira. Baina, zer dira zehazki **patroiak**? Gure testuinguruan, karaktere multzoari deituko diogu **patroia**. Beraz, testu elektronikoetan karaktere multzoak bilatzeko tresnak dira adierazpen erregularrak.

Oinarrizko patroiak eraiki eta erabiliz hasiko gara, eta pausoz pauso ezaugarri berriak gehitu eta patroia konplexuagoak nola idatzi ikusiko dugu kapituluan zehar.

Guretzat, izan daitekeenik eta patroirik sinpleena *hitza* da, edo modu orokoragoan esanda, *karaktere-segida*. Patroitzat hitz jakin bat hartu, eta *string* edo karaktere-kate batek hitz hori daukan ala ez galdegin dezakegu. Adibidez: `$string` aldagaiko testuak `beldur` karaktere-segida daukan ala ez jakiteko, ondoko adierazpen erregularra erabil dezakegu:

```
$string =~ m/beldur/;
```

Espresioak *true* (egiazkoa) itzuliko du baldin `$string` aldagaian `beldur` karaktere-katea topatzen badu, eta *false* (faltsua) bestela. Bilaketa-patroia aurrera begirako barra `/` (*slash*) artean idatzitakoa da. Aurretik daraman `m` karaktereak parekatzea (*match*) esan nahi du. Azkenik, `"=~"` eragileak bilaketa-patroia eta testua lotzen ditu, bilaketa zein testutan egin behar duen adieraziz Perl-i.

Hauxe da testu batean patroia jakin bat bilatzen duen adierazpen erregularren eskema orokorra:

```
$testua =~ m/patroia/;
```

Non `$testua` aldagaiak bilaketa-testua gordetzen duen, eta *patroia*, berriaz, bilatu nahi dugun hitz edo karaktere-katea den. Bilaketa-patroiaren aurretik jarritako `m` (*match*) aukerakoa da, eta jarriko ez bagenu ere Perl-ek berdin-berdin ulertuko luke bilaketa dela egin nahi duguna.

Parekatzea positiboa izan dadin, adierazpen erregularrak *egiazkoa* itzul dezan, bilaketa-patroiak karakterez karaktere egin behar du testuaren zati batekin, edo testu osoarekin bestela. Azaldutako guztia hobeto ulertze aldera, hona hemen bilaketak egiten dituzten zenbait adierazpen erregular:

```

"Ni neu naiz" =~ /neu/           # parekatze positiboa
"Ni neu naiz" =~ /Ni neu naiz/   # parekatze positiboa
"Ni neu naiz" =~ /u n/           # parekatze positiboa, zuriunea beste
                                # edozein karaktere bezala
"Ni neu naiz" =~ /i neu/         # positiboa

"Ni neu naiz" =~ /ue/;           # parekatze negatiboa
"Ni neu naiz" =~ /ni/;           # negatiboa, letra larriak
                                # eta xeheak bereizi egiten dira

```

Perl-ek letra larri eta xeheak ez bereiztea nahi badugu, bukaerako barraren ondoren *i* (*insensitive*) ikurra erantsiko diogu adierazpenari. Azken adibidera itzuliz:

```

"Ni neu naiz" =~ /ni/i;          #parekatze positiboa

```

Alderantzizko bilaketa ere posible da, hau da, hitz edo karaktere-kate jakin bat testuan **agertzen ez denean** adierazpenak *egiazkoa* (*true*) itzultzea, eta *faltsua* (*false*), berriz, topatzen duenean. Horretarako "**=~**" eragilearen kontrakoa den "**!~**" erabiliko dugu. Adibidez:

```

"Hartzeko poz emateko hotz" !~ m/u/;           # parekatze positiboa,
                                                # testuak ez dauka "u" letrarik
"Hartzeko poz emateko hotz" !~ m/Hartzeko/;    # parekatze negatiboa
"Hartzeko poz emateko hotz" !~ m/te/;          # parekatze negatiboa

```

Ondoko programak erabiltzaileari teklatu bidez bilaketa-patroia idazteko eskatuko dio, eta ondoren begiratuko du ea testuarekin parekatu eta bat egiten duen ala ez.

regExp.pl

```

#!/usr/bin/perl
use warnings;
use strict;

my $testua =
    "Besteen faltak aurreko aldean, geureak bizkarrean";
print ("Idatzi bilaketa-patroia: ");
my $patroia = <STDIN>;
chomp($patroia);

```

```

if ($testua =~ m/$patroia/) {
    print ("$patroia agertzen da testuan\n");
}
else {
    print ("$patroia ez da agertzen testuan\n");
}

```

Adierazpen erregularrak bilaketa egiteaz gain, *egiazkoa/faltsua* itzultzen duenez, baldintzazko egitura batean jar dezakegu, goiko programak erakusten duen bezala. Bestalde, bilaketa-patroiaren lekuan aldagai bat ere jar liteke (\$patroia), kasu horretan Perl-ek aldagai horren balioa erabiliko du patroia gisa bilaketa egi-terakoan. Idatzi berri dugun programa patroia ezberdinekin proba egiteko erabil dezakegu:

```

>perl regExp.pl
Idatzi bilaketa-patroia: faltak
faltak agertzen da testuan
>

```

```

>perl regExp.pl
Idatzi bilaketa-patroia: i
i agertzen da testuan
>

```

Proba egin zuek ere, asmatu bilaketa-patroiak eta idatzi. Aldatu nahi baduzue "=~" parekatze-eragilea eta jarri haren ordean "!~", bere portaera nolakoa den ikusteko (ez ahaztu gero amaieran berriro "=~" jartzea, hemendik aurrera ere programa erabili behar dugu eta!).

11. kasu praktikoa: bilaketak testuan

Testu-editore guztiek izan ohi duten *Bilatu* funtzioa inplementatuko dugu Perl erabiliz: programak `regExpOrok.pl` izena izango du, eta komando-lerrotik fitxategi-izena eta bilaketa-patroia jaso, eta patroia betetzen duten lerroak bistaratu ditu pantailan. Letra larrien eta xeheen arteko bereizketarik ez du egin behar programak.

Adibidez, `esaeraLabur.txt` fitxategian beti hitza daukaten esaeren bila bagabiltza:

```
>perl regExpOrok.pl esaeraLabur.txt beti
Beti on nahi duena, maiz gaizki
Nagia, beti lantsu
>
```

Zure txanda da orain. Hartu tarte bat eta ahalegindu bilaketa-programa zeure kasa idazten.



.....

Hemen duzu gure soluzioa ere, ea zer iruditzen zaizun:

regExpOrok.pl

```
#!/usr/bin/perl
use warnings;
use strict;
open(FITX, $ARGV[0]) or
    die("Ezin $ARGV[0] fitxategia ireki\n");
my $patroia = $ARGV[1];
my $lerroa;
while ($lerroa = <FITX>) {
    print ($lerroa) if ($lerroa =~ m/$patroia/i);
}
close(FITX);
```

Komando-lerroaren bitartez pasatutako argumentuak @ARGV *array* berezian gordeko ditu programak, fitxategi-izena \$ARGV[0]-n eta bilaketa-patroia \$ARGV[1]-en. Sarrera/Irteerako kapituluan ikusitako eskemari jarraituz, fitxategia zabaltzen ahaleginduko da Perl `open()` or `die()` funtzioa erabiliz. Erroreren bat gertatuz gero, mezua pantailaratu eta programaren exekuzioa bertan behera utziko du. Baina gauzak ondo bidean, fitxategia irakurketarako zabaldu eta aurrera jarraituko du exekuzioak, bigarren argumentu bezala jasotako bilaketa-patroia \$patroia aldagaian gordez. Segidan, `while` egitura baliatuz, fitxategia goitik behera zeharkatuko du, lerro bakoitza \$patroia patroiarekin parekatuz. Parekatzea positiboa den kasuetan, pantailan bistaratuko da \$lerroa aldagaia.

6.2. KARAKTERE BEREZIAK

Idatzi berri dugun `regExpOrok.pl` programarekin edozein *string* literalen bilaketa egin dezakegu. Baina ez da horretan geratzen adierazpen erregularrekin egin dezakeguna, ezta gutxiagorik ere! Oinarritzko bilaketa-mekanismoa besterik ez du-gu ikusi, eta orain, adierazpen erregular konplexuagoak eraikitzen hasteko moduan

gara. Horretarako, *metakaraktere* deiturikoak erabiliko ditugu. Metakarakterek beste karaktere batzuk deskribatzen dituzten karaktere bereziak dira. Hauek dira erabiliko ditugun karaktere bereziak:

`^ $ . * ? [ ] ( ) { } | \`

Esanahi berezidun karaktereak dira goiko horiek. Bakoitzaren esanahia eta erabilera gutxi barru aztertuko ditugu. Horietaz gain, beste karaktere guztiek ez dute esanahi berezirik izango Perl-entzat. Baliteke noizbait bilaketan goiko karaktereren bat erabili nahi izatea bere horretan, esanahi berezirik gabe. Adibidez, demagun galdera-marka daramaten lerroak bistaratu nahi ditugula. Kasu horretan, atzera begirako barra “\” jarriko diogu aurretik galdera-markari eta Perl-ek ez du karaktere berezi gisa interpretatuko:

```
$testua =~ m/\?/;      # ? Karaktere arrunta, galdera ikurra
$testua =~ m/?/;       # ? metakarakterea da
"6+2=8" =~ m/6+2/;     # parekatzerik ez, + metakarakterea da
"6+2=8" =~ m/6\+2/;    # parekatze positiboa, + karaktere arrunta
```

Hurrengo ataletan banan-banan aletuko ditugu metakarakterek, eta haiek erabiliz bilaketa-programa ahaltsuagoak eraiki ere bai.

6.2.1. Bilaketa mugatu

Orain arteko adierazpen erregularretan, parekatzea positiboa izan zedin, nahikoa zen patroiak nonbait testuaren zati batekin bat egitea. Hala ere, batzuetan bat-egiteak non gertatu behar duen zehaztea interesa dakiguke. Horretarako, bilaketa-mugatzaile metakarakterek erabil ditzakegu: `^8` eta `$`. Beren esanahia honakoa da: `^` mugatzailea patroiauren aurretik jartzen da eta parekatzeak lerro edo *string*-aren hasieran gertatu behar duela adierazten du. `$` mugatzailea, berriz, patroiauren ondoren doa, parekatzeak lerro edo *string*-aren amaieran gertatu behar duela zehaztuz. Hona hemen erabileraren adibideak:

```
"igeltseroa" =~ m/eroa/;      # parekatze positiboa
"igeltseroa" =~ m/^eroa/;     # parekatzerik ez
"igeltseroa" =~ m/^igel/;     # parekatze positiboa
"igeltseroa" =~ m/eroa$/;     # parekatze positiboa
```

Bigarren adierazpenak huts egin du, `^` mugatzailea jartzean *eroa* patroiauren bilaketa *string*-aren hasierara mugatu dugulako. Aldiz, azken adierazpenean parekatzea positiboa izan da, patroiak *string*-aren amaierarekin bat egitea exijitu dugulako.

8. `^` karakterea idazteko sakatu Maiuskula-`^` (biak batera) eta ondoren zuriune tekla.

Biak batera, `^` eta `$`, erabiltzen ditugunean adierazpenak bat egin behar du *string*-aren hasiera eta amaierarekin. Beste hitzetan esanda, patroiak *string* osoarekin bat egin behar du.

```
"igeltseroa" =~ m/^eltseroa$/;    # parekatze negatiboa
"igeltseroa" =~ m/^igeltseroa$/;  # parekatze positiboa
"" =~ m/^$/;                      # parekatze positiboa, string hutsa
```

Aurretik idatzi dugun `regExpOrok.pl` programa berreskuratu eta egin proba patroia ezberdinekin bilaketa-mugatzailleak erabiliz. Adibidez, `esaeraLabur.txt` fitxategian, zenbat esaera daude "a" letraz (larri edo xehe) hasten direnak?

```
>perl regExpOrok.pl esaeraLabur.txt "^a"
Adiskidegabeko bizitza, auzogabeko heriotza.
Aldi luzeak, guztia ahaztu
Amen, zu hor eta ni hemen.
Ardo gozoak lau begi eta oinik ez.
Asko baduk, asko beharko duk
Asto askok, lasto asko
Auziaren ondoren, galtzen duena larru gorrian eta irabazten
duena alkondara hutsean.
>
```

Komando-lerroan patroia komatxo artean idatzi dugu ("**a**"), bestela sistema eragileak kontrol-karakteretzat hartzen baitu `^` karakterea.

12. kasu praktikoa: perltsolaritza

Lagun baten ezkontza-ospakizuna dela-eta, bertso bat idaztea egokitu zaigu. Jarri gara lanean, eta bueltak eta bueltak eman ondoren, sekula aurretik inori bururatu gabeko (ejem) bukaera etorri zaigu burura: «elkarrekin zoriontsu/bizi izan zaitezte!». Orain gainerako puntuak osatu behar. Ekin diogu lanari, baina puf, ez da uste bezain erraza, errima falta gabiltza bertsoa osatzeko... Eskerrak Perl badakigun! Bat-batean otu zaigu fitxategi bat eta hitz-bukaera emanda, bukaera berdineko hitzak topatzea oso erraza dela!

Programa badaukagu, `regExpOrok.pl`, fitxategia behar dugu orain. Edozeinek balio dezakeen arren, gure lanerako egokiena `xuxen.txt` fitxategia da, 80.000 hitz baino gehiago dauzkan hiztegia, bakoitza lerro banatan.

Hauxe litzateke *ela* bukaerarekin errimatzen duten hitzak topatzeko modua:

```
>perl regExpOrok.pl xuxen.txt ela$
Adela
adela
aharkela
akuarela
albagela
aldagela
...
zoazkiotela
zoaztela
>
```

Holako errima-erauntsirik! Ez dira pantailan sartu ere egiten. Irteera fitxategi batera birbideratzea ideia ona litzateke, ondoren fitxategia editatu eta errimak lasai begiratzeko:

```
>perl regExpOrok.pl xuxen.txt ela$ > ela.txt
>
```

Tira, ezin izango esan dugu behintzat errima faltagatik utzi dugunik bertsoa osatu gabe!

6.2.2. *Karaktere-klaseak*

Hitz edo karaktere-kate konkretuak bilatzen ikasi dugu, eta ikasitakoa oso erabilgarria da, baldin eta zehatz baldin badakigu zer bilatu nahi dugun. Baina zer egin, adibidez, bokalez hasten diren hitzak topatu nahi baditugu? Edo, hiru karaktereko hitzen bila baldin bagabiltza? Aurreko galderei erantzun eta gure bilaketa-esparrua zabaltzeko **karaktere-klaseak** zer diren ikusiko dugu atal honetan. Karaktere-klasea, kortxete "[]" artean jarritako karaktere multzoak osatzen du, bat-egitea kortxete artean jarritako edozein karaktererekin izan daitekeela adieraziz. Multzoko karaktere bakarrak bat egitea nahikoa da parekatzea positiboa izan dadin. Adibidez:

```
/zai/;                # "zai" katea bilatuko du
/[zgb]ai/;           # bilaketak: "zai", "gai" eta "bai"
"zai nago" =~ /[aeiou]/; # Bokalen bilaketa. Bat egitea
                        # testuko lehen bokalak eragingo du, "a".
/[zZ][aA][iI]/      # "zai" larri/xehe bereizketarik gabe:
                        # Zai, zai, zAi...
/zai/i;              # aurrekoaren berdina
```


Kortxete artean erabiliz gero, "-" karaktere bereziak tarte (*range*) eragilearen funtzioa betetzen du, zenbait karaktere-klaseren adierazpena asko sinplifikatuz. Adibidez:

```
/[0123456789]/;           # edozein digitu
/[0-9]/;                   # aurrekoaren baliokidea
/[abcdefghijklmnopqrstuvwxyz]/; # a eta n arteko edozelan karaktere
/[a-n]/                    # aurrekoaren baliokidea
```

Erabileraren adibideak:

```
/[0-9][0-9][0-9]/;
    # hiru digituko edozein zenbaki: 123, 333, 754...
/[A-Z]/; # edozein letra larri
```

Batzuetan, bilatu nahi dugun hura baino errazagoa da bere kontrakoa zehaztea, hau da, topatu nahi ez dugun hura adieraztea. Adibidez: u letrarik gabeko lerroak, a letraz hasten ez direnak, etab. Horretarako ^ karaktere berezia erabiliko dugu kortxete artean, adibidez [^a], adierazteko "a" izan ezik edozein karaktere. Parekatze positiboa itzuliko du baldin eta kortxete arteko adierazpenarekin bat-egiterik gertatzen ez bada, hau da, "a" ez den edozein karaktererekin bat egingo du. Adibide gehiago:

```
[^a-zA-Z]/;           # letra ez beste edozein karaktere
/[^0-9]/;              # digitua ez den edozein karaktere
/[^e]/;                # edozein karaktere "ev izan ezik
```

Gure esaera zaharren bildumatxoa berreskuratu, eta "ango" katearen aurretik "z"-rik ez daramaten karaktere-kateak lortzen ahaleginduko gara oraingoan. Hau da, *txango*, *tango*, *joango*, eta abar, baina inola ere ez *zango*.

```
>perl regExpOrok.pl esaeraLabur.txt "[^z]ango"
Egintzak lo eta jango dek mehe, izan arren bi neskame.
Ez eukitxa ere, maite zaitxunak emango dizu.
>
```

Kontuz! ^ sinboloak bi adiera ezberdin ditu testuinguruaren arabera: patroiaaren aurretik jarrita bilaketa-mugatzaila da (ikus 6.2.1.); eta kortxete artean [] erabiliz gero, karaktere-klasearen kontrakoa adierazten du. Ez nahastu beraz.

Egin ditzagun bilaketa gehiago gure esaera zaharren bilduman.

Zeintzuk dira a letraz hasten ez diren esaerak?

```
>perl regExpOrok.pl esaeraLabur.txt "^[^a]"
```

Honako murriztapenak betetzen dituzten esaerak bistaratu: "err" katearen ondoren "i" eta "a" karaktererik ez daukatenak. Adibidez: *berregin*, *errota*, *erreka*; baina ez *herri*, *gerra* edo *Eguberri*.

```
>perl regExpOrok.pl esaeraLabur.txt "err[^ai]"
```

Zenbait karaktere-klase behin eta berriz agertuko dira gure adierazpen erregularrarretan. Hori dela-eta, Perl-ek horien laburdurak eskaintzen ditu. Hona hemen erabilienak:

Laburdura	Karaktere-klasea	Azalpena
\d	[0-9]	Digitua
\s	[\t\n\r]	Zuriunea, lerro-jauzia, tabulazioa
\w	[0-9a-zA-Z]	Karaktere alfanumerikoa
\D	[^0-9]	Digitua ez den edozein karaktere
\S	[^\t\n\r]	Zuriunea ez den karakterea
\W	[^0-9a-zA-Z]	Alfanumerikoa ez den karakterea
.	[w\W]	Edozein karaktere (lerro-jauzia izan ezik)

Laburdura hauekin denbora aurrezteaz gain, adierazpen erregular irakurterezagoak idatziko ditugu. Hona hemen zenbait adibide:

```
/\d\d:\d\d/;      #bi digitu:bi digitu, adb: 19:37, 04:45
/\w\w\s\w\w/;     #bi karaktere alfanumeriko, jarraian zuriunea
                  #eta ondoren beste bi karaktere alfanumeriko
/..ela/;           #edozein bi karaktere, eta jarraian "ela"
```

Esperimentu baterako, 2 letrako hitzak topatuko dituen programa behar dugu. Di-da batean, betiko `regExpOrok.pl` programa hartu eta patroia honekin abiarazi dugu bi aldiz pentsatu gabe:

```
>perl regExpOrok.pl esaeraLabur.txt "\w\w"
```

Programaren irteera espero genuena al da? Bi letrako hitzak soilik topatu ditu?



.....

Programak `esaeraLabur.txt` fitxategia osorik bistaratu digu pantailan. Lerro guztiek bat egin dute patroiarekin. Baina lerro guztiek ez daukate bi letrako

hitzik. Zer gertatu da orduan? Errore semantiko bat egin dugu, bilaketa-patroia ez dugulako behar bezala zehaztu. Izan ere, `\w\w` jarrita, Perl-ek gutxienez bi karaktere elkarren jarraian dituen edozein *string* balekotzat jo du. Bi letra besterik ez dituzten hitzak bilatzeko, ondokoa egokiagoa litzateke:

```
>perl regExpOrok.pl esaeraLabur.txt "\s\w\w\s"
Amen, zu hor eta ni hemen.
Bat eman eta bi hartu, gure etxean ez berriz sartu
...
>
```

Oraingoan parekatze positiboen kopurua txikiagoa izan da. Hala ere, ez gaude guztiz conforme gure programarekin. Badauzka oraindik leundu beharreko ertzak:

1. Ez ditu bi letrako hitz guztiak aurkitu, aurretik eta ondoren zuriunea dararmatenak bakarrik. Lerro-hasieran, puntuazio-marka baten aurretik edo ondoren daudenak ihes egin diote gure bilaketa-espresioari.
2. Parekatzea positiboa den kasuetan lerro osoa bistaratzen dugu, gure benetakoa interesa bi letrako hitzetan soilik dagoenean. Nola bistaratu bat egin duten karaktere-kateak soilik?

Lehen puntuari dagokionez, Perl-ek badauka hitzen arteko mugak detektatzen dituen karaktere berezia: `\b`. Metakaraktere horrek `\w` (hitz-karakterea) eta `\W` (hitz-karakterea ez dena) arteko mugak antzematen ditu, hau da, karakteretik ez-karaktererako jauziak eta alderantziz. Adibidez:

```
"gordeta dago eta" =~ /eta/
# "gordeta" hitzeko "eta"-rekin bat egitea
"gordeta dago eta" =~ /\beta\b/
# bat egitea bukaerako "eta" hitzarekin
"gordeta dago eta" =~ /\bgorde\b/ # negatiboa
```

Perl-ek *string* hasiera eta bukaera hitz-mugatzat hartzen ditu.

Metakaraktere berri honekin 2 letrako hitzen gure bilaketa findu dezakegu:

```
>perl regExpOrok.pl esaeraLabur.txt "\b\w\w\b"
```

Hobetu beharreko bigarren puntuak emaitzak bistaratzeko moduarekin dauka zerikusia. Izan ere, parekatzea positiboa denean lerro osoa bistaratzen dugu pantailan. Baina nola jakin espresioak zehazki testuko zein zatirekin bat egin duen? Dударik gabe, interesgarria litzateke parekatze positiboa eragin duen testu puska soilik bistaratu ahal izatea. Trankil, Perl-ek badauka-eta soluzioa gure problematzat, honentzat bai behintzat. Adierazpen erregularrak erabiltzen ditugun aldiro, Perl-ek parekatzeari buruzko informazioa gordetzen du `$&`, `$`` eta `$'` aldagai

berezietan. Parekatzearen ondoren, hauxe da aldagai bakoitzak izango duen informazioa:

\$& Patroiarekin bat egin duen *string* puska

\$` Bat egin aurreko *string* puska

\$' Bat-egitearen ondorengo *string* puska

Aldagai bereziek beren balioa mantenduko dute hurrengo patro-i-parekatzea gertatu bitartean.

Adibidez:

```
"zorrak zor lepoa lodi" =~ /\b\w\w\w\b/;
```

Hona aldagai berezien edukia parekatzearen ondoren:

```
#$` = "zorrak "      $& = "zor"      $' = " lepoa lodi"
```

Beste adibide bat:

```
"begiratu" =~ b/i.a/;
```

```
#$` = "beg"      $& = "ira"      $' = "tu"
```

Bigarren adibideak erakusten duen moduan, \$& aldagaiak ez du beti hitz osoa gordetzen, patroiarekin bat egin duen *string* puska zehatza baizik.

Nahikoa da agindu bakar bat aldatzea gure `regExpOrok.pl` programan, horrek patroiarekin bat egiten duten *string* zatiak bakarrik bistara ditzan: `print($lerroa); jartzen duen lekuan print("$&\n"); jarri eta kito.` Aldaketak gorde `regExpZehatz.pl` izenarekin.

Hartu tarte bat eta jolastu programarekin luzera ezberdinetakoa hitzak bilatuz. Ondoren, erantzun galdera honi: Perl-ek patroiarekin bat egiten duten hitz guztiak itzultzen al ditu?

Probak egiteko tarte hartu baduzu, konturatuko zinen lerro berean 3 hitzek bat egiten badute espresioarekin, Perl-ek lehenbizikoa soilik itzultzen duela. Aurrerago aztertuko dugu nola behartu Perl, lerroko bat-egite guztiak itzul ditzan.

13. kasu praktikoa: gurutzegramak

Ona omen da garunari herdoila kentzeko gurutzegramak egitea. Gurea gogortu xamarra dago honezkero, nahiz eta gogotsu hasi, nekez amaitu ohi ditugu eta. Bukaera aldera beti gelditzen zaigu asmatu ezineko baten bat. Ez aurrera eta ez atzera, holakoetan etxeko ohiko hiztegiek apenas laguntzen diguten.

Nola bilatu hiztegia, adibidez, lehenengo letra "e" eta azkena "o" daukaten 3 letrako hitzak? Edo "ia" kateaz amaitzen diren 5 karaktereko hitzak? Era honetako bilaketak egiteko hiztegi elektronikoa beharko genuke, eta bilaketak zehazteko lengoaia egokia ere bai.

Ez daukagu urrutira joan beharrik. Hiztegi elektronikoa eskura daukagu, `xuxen.txt`, eta zer hoberik adierazpen erregularrak baino bilaketa-terminoak zehazteko?

Herdoildutako kaskoari buelta batzuk eman, eta onena neurrira egindako Perl programa idaztea litzatekeela erabaki dugu, programaren sarrera/irteera honela zehaztuz: sarrerako datu gisa bi argumentu, lehena hiztegi-fitxategiaren izena, eta bigarrena bilaketa-terminoa edo -patroia. Irteera, bilaketa-patroiarekin bat egiten duten hiztegiko hitzen zerrenda izango da.

Programaren gakoa bilaketa-patroian dago. Nola adierazi gure murriztape-nak? Hona modu bat: karaktere ezagunak bere horretan idatzi eta ezezagunak puntu karaktere bidez adierazi. Adibidez: 6 karaktereko hitza "ra" katearekin bukaeran, `"....ra"`. Edo, "e" letraz hasi eta "a" letraz amaitzen den 4 karaktereko hitza, `"e..a"`. Puntu karakterea ez dugu besterik gabe aukeratu karaktere ezezagunak adierazteko, ezta? Ez, noski, puntu karakterea berezia da adierazpen erregularretan, "edozein karaktere" adierazten duen metakarakterea. Horretxegatik aukeratu dugu hain zuzen ere karaktere ezezagunak adierazteko.

Hor duzu ariketaren abiapuntua. Saia zaitez eskema orokorra garatzen kodea idazten hasi aurretik.



.

Egia esan, programa honen eskema orokorra eta `regExpOrok.pl` programarenak berdinak dira ia. Berrikuntza gisa, bilaketa-terminoa teklatu bidez idazteko eskatuko diogu erabiltzaileari. Hona algoritmo edo eskema orokorra:

1. Komando-lerrotik jasotako fitxategia ireki
2. Teklatu bidez bilaketa-terminoa jaso, `$bilaketa`
3. Fitxategiaren `$lerro` lerro bakoitzeko

3.1. Baldin (`$lerro =~ /^$bilaketa$/i`) orduan

`bistaratu $lerro`

4. Fitxategia itxi

Eskema Perl lengoiara itzultzeak ez digu lan handirik emango:

gurutzegramak.pl

```
#!/usr/bin/perl
use warnings;
use strict;
print("Idatzi bilaketarako terminoa: ");
my $bilaketa = <STDIN>;
chomp($bilaketa);
open(FITX, $ARGV[0]) or
    die("Ezin $ARGV[0] fitxategia ireki!\n");
my $lerro;
while ($lerro = <FITX>) {
    # patroiak osorik bat egin behar du $lerro-ko
    # edukiarekin
    if ($lerro =~ /^$bilaketa$/i) {
        print("$lerro");
    }
}
close(FITX);
```

Proba dezagun ea zer moduz dabilen:

```
>perl gurutzegramak.pl xuxen.txt
Idatzi bilaketarako terminoa: a.t.a
Aitza
altxa
...
autza
aztia
>
```

Ez dago batere gaizki, baina ia programa guztietan bezala, bada zer hobetarik. Gure programak exekuzio bakoitzeko bilaketa bakarra egiten du, eta bilaketa berri bat egin nahi dugun aldiro, programa berririo abiarazi behar dugu. Ez al litza-teke erosoagoa exekuzio bakoitzean nahi adina bilaketa egitea? Bai noski. Horretarako bi aldaketa behar ditu aurreko programak. Lehenbizikoa, bilaketak behin eta berriz egiteko aukera ematea, harik eta erabiltzaileak nahikoa duela erabaki arte. Begizta berri batekin konpon genezake hori, erabiltzaileak "q" letra sakatu artean iteratzen jarraituko duena. Bigarren aldaketa fitxategiaren irakurketarekin dago lotuta. `while ($lerro = <FITX>)` begiztarekin fitxategia lerroz lerro irakurtzen dugu, bilaketarekin bat egiten duten hitzak bistaratuz. Bukaerara iristean, bilaketa berri bat egin ahal izateko espresuki adierazi behar diogu Perl-i berririo fitxategi-hasierara joateko `seek(FITX, 0, 0)` funtzioarekin, bestela fitxategi-bukaeran geldituko da ainguratuta eta gure bilaketek ez dute emaitzarik lortuko.

Bigarren algoritmo honek aipatutako hobekuntzak biltzen ditu:

1. Komando-lerrotik jasotako fitxategia ireki

2. Teklatu bidez bilaketa-terminoa jaso

```
$bilaketa = <STDIN>
```

2.1.while (\$bilaketa ne "q")

2.1.1.while (\$lerro = <FITX>)

2.1.1.1.if (\$lerro = ~/^\$bilaketa\$/i)

```
print $lerro
```

2.1.2. Fitxategia hasieratu: seek(FITX, 0, 0)

2.1.3. Bilaketa-termino berria eskatu, \$bilaketa = <STDIN>

4. Fitxategia itxi

Programaren kodea:

gurutzegramak2.pl

```
#!/usr/bin/perl
use warnings;
use strict;
print("Idatzi bilaketarako terminoa. Amaitzeko, idatzi q\n");
my $bilaketa = <STDIN>;
chomp($bilaketa);
open(FITX, $ARGV[0]) or
    die("Ezin $ARGV[0] fitxategia ireki!\n");
my $lerro;
while ($bilaketa ne "q") {
    while ($lerro = <FITX>) {
        # patroiak osorik bat egin behar du $lerro-ko
        # edukiarekin
        if ($lerro =~ /^$bilaketa$/i) {
            print("$lerro");
        }
    }
    # amaitu da bilaketa
    # fitxategiaren hasierara mugitu
    seek(FITX, 0, 0);
    print("\n\nIdatzi bilaketarako terminoa. Amaitzeko,
        idatzi q\n");
    $bilaketa = <STDIN>;
    chomp($bilaketa);
}
close(FITX);
```

Bigarren programa honekin nahi beste bilaketa egin ahal izango ditugu exekuzio-aldi bakoitzean.

14. kasu praktikoa: hitz-egiturak

Egitura konkretuko hitzak bilatuko dituen programa eskatu digute esperimentu linguistiko baterako. Adibidez: K B K B idatziz gero, *Kontsonantea-Bokala-Kontsonantea-Bokala* egitura duten hitzak bilatu beharko lituzke programak: pala, sute, pila, etab.

Sarrerako datu gisa, hitz-egitura eta bilaketa-fitxategia behar ditugu, horiek programan jasotzeko bi bide ezberdin erabiliko ditugarik: patroia-egitura komando-lerroaren bitartez, letra larriz eta letren artean hutsune bat utziz, modu honetan: B K B (*Bokala-Kontsonantea-Bokala*). Bilaketa-fitxategia, berriz, programak berak eskatuko du abiatzean.

Programaren dei posible bat haxe litzateke:

```
>perl egitura.pl K B K K B
Zein fitxategitan egingo dut bilaketa?
xuxen.txt
bable
bafle
balbe
...
zurtu
Zutza
zuzki
>
```

Egin beharrekoa oso konplikatua ez den arren, aurretik programaren eskema zehazteak lana erraztuko digu. Programak bi egiteko nagusi dauzka:

1. Komando-lerrotik jasotako egiturarekin bilaketa-patroia eraiki.
2. Sarrera estandarretik jasotako fitxategian bilaketak egin.

Aurrera segi aurretik, saia zaitez zure aldetik programaren bi egitekoak zehazten edo fintzen.



.....

Ataza edo egiteko bakoitza honelaxe garatu dugu guk:

1. Komando-lerrotik jasotako egiturarekin patroia eraiki
 - 1.1. Jasotako \$arg argumentu bakoitzeko
 - 1.1.1. Baldin \$arg berdin "K" orduan
\$patroia patroiar kontsonantea gehitu
 - 1.1.2. Bestela, baldin \$arg berdin "B" orduan
\$patroia patroiar bokala gehitu
 - 1.1.2. Bestela bistaratu "argumentu okerra"
2. Sarrera estandarretik jasotako fitxategian bilaketak egin
 - 2.1. Teklatutik jasotako fitxategia ireki
 - 2.2. Fitxategiaren \$lerro lerro bakoitzeko
 - 2.2.1. Baldin (\$lerro =~ /\$patroia/i) orduan
bistaratu \$lerro
 - 2.3. Fitxategia itxi



.....

Eskema aurrean daukazula, idatz ezazu dagokion kodea Perl-en eta gorde `egitura.pl` izenarekin.

egitura.pl

```
#!/usr/bin/perl
use warnings;
use strict;

my $konts = "[bcd fghjklmnñpqrstvwxyz]"; # kontsonanteak
my $bok = "[aeiou]"; # bokalak
my $patroia = "\b"; # hasieratu patroia hitz-mugarekin
foreach my $arg (@ARGV) {
    if ($arg eq "B") {
        # kateatu bokala patroiar
        $patroia = $patroia . $bok;
    }
    elsif ($arg eq "K") {
        # kateatu kontsonantea patroiar
        $patroia = $patroia . $konts;
    }
    else {
        die("Patroi egitura okerra\n");
    }
}
```

```

$patroia = $patroia . "\b"; # hitz-muga kateatu bukaeran
print("Zein fitxategitan egingo dut bilaketa?\n");
my $fitx = <STDIN>;
open(FITX, $fitx) or
    die("Ezin $fitx fitxategia zabaldu!\n");
my $lerroa;
while ($lerroa = <FITX>) {
    # i modifikatzailea: letra larri/xehe bereizketarik ez
    if ($lerroa =~ /$patroia/i) {
        # bat egiten duen string puska soilik bistaratu
        print("$&\n");
    }
}
close(FITX);

```

Programan, betiko hiru lerroen ostean, kontsonante eta bokalen karaktere-klaseak definitu ditugu beste ezer baino lehen. Segidan, `$patroia` aldagaia `"\b"` (hitz-muga) katearekin hasieratu ondotik, komando-lerrotik idatzitako argumentuak jasotzen ditu programak banan-banan `foreach` egitura baliatuz. Argumentua `"K"` karakterea bada, kontsonanteen karaktere-klasea kateatzen zaio `$patroia` aldagaiari. Bestela, argumentua `"B"` karakterea bada, orduan bokalen karaktere-klasea izango da `$patroia` aldagaiari kateatuko zaiona. Beste edozein karaktere jasoz gero, programak errore-mezua bistaratu eta exekuzioa bertan behera utziko du. Argumentu guztiak jaso eta bilaketa-patroiari gehitu ondoren, bukaeran berriro `"\b"` hitz-muga kateatuko diogu `$patroia` aldagaiari, egitura betetzen duten **hitzak** bakarrik interesatzen zaizkigulako.

Behin bilaketa-patroia eraikita, programak bilaketa-fitxategiaren izena eskatzen dio erabiltzaileari. Fitxategia zabaldu, `while` egiturarekin lerroz lerro irakurri, eta patroiarekin bat egiten duten hitzak pantailan bistartzeko.

Berriro ere, arretaz begiratzen badugu, gure programaren eraginkortasuna mugatua dela konturatuko gara. Izan ere, fitxategia lerroz lerro bilaketa-patroiarekin parekatu, eta bat egiten duen lehen *string* puska edo hitza itzultzen du programak, bat-egiterik den kasuan noski. Lerro berean espresioa betetzen duten 2, 3 edo 4 *string* edo hitz badira, lehenbizikoa baino ez du bistaratuko. Aurrerago aztertuko dugu nola bistaratu bat-egite guztiak.

6.2.3. Aukerak: *hau, hori edo hura*

Baliteke gure bilaketetan tarteka era honetako parekatzeak egin nahi izatea: «hitz hau *edo* beste hura daukaten esaldiak». Nola adierazi *edo* adierazpen erregularrean? Bada, `"|"` metakarakterea erabiliz.

```
/kafea|tea/;
# "kafea" edo "tea" string-en agerpenekin bat egingo du
/a|e|i|o|u/;
# bokalak /[aeiou]/-ren baliokidea

"itsasoa eta mendia, biak ditut gustuko" =~ /mendia|itsasoa/;
```

Konturatuko zinen honezkero Perl-ek ahalik azkarren burutu nahi izaten duela bat-egitea. Horrela lan egiten du bere parekatze-mekanismoak: testuko karaktere bakoitzeko, Perl-ek espresioko lehen aukerarekin egingo du proba lehen-bizi, mendia, eta bat egiten ez badu bigarrenarekin ahaleginduko da, itsasoa. Biek huts eginez gero, hurrengo karakterera mugitu eta berriz ahaleginduko da, harik eta bat egiten duen *string* puskarekin topo egin edo testuaren amaierara iristen den arte.

Kasu honetan, bat-egitea itsasoa *string*-arekin gertatuko da, zeren, espresioan bigarren aukera bezala agertu arren, testuan lehenbiziko bat-egitea berak eragiten baitu.

Demagun fitxategi bat eman eta bertan agertzen diren diptongoak nahi ditugula bueltan. `regExpOrok.pl` programa eta "|" metakarakterearekin lan erraza da hori:

```
>perl regExpOrok.pl esaeraLabur.txt "ai|ei|oi|ui|au|eu|ou"
```

6.2.4. Espresioak bildu

Biltze-metakarakterearak "()" parentesi artean idatzitako adierazpen erregularra unitate edo patroiz hartzea ahalbidetzen du.

Adibidez:

```
/(g|t)alai/;
# "galai" edo "talai" string-ekin bat egingo du
# /galai|talai/ espresioaren baliokidea
/bai(mena|ta|)/;
# "bai" eta ondoren parentesi arteko aukerak:
# "mena" edo "ta" edo "". "baimena", "baita" eta "bai"
```

6.2.5. Atzera begirako erreferentziak

Adierazpen erregularren puskak parentesi artean biltzeak badauka beste abantaila bat ere: parekatze positiboetan, parentesi artean bildutako espresioak zer gordetzen duen jakin liteke. Aldagai bereziak dauzka zeregin horretarako Perl-ek, eta parentesi arteko espresio bakoitza zenbakitutako aldagai batean gordetzen du: lehenbizikoa \$1-en, bigarrena \$2-n eta horrela hurrenez hurren. Modu honetan,

atzera begirako erreferentzi aldagaiak (backreferences) erabiliz, bat-egitea eragin duen *string*-aren puska ezberdinak aztertu eta erabil ditzakegu.

Hona zenbait adibide:

```
$testua =~ /[aeiou](\w)[aeiou]/;
    # bokala, edozein karaktere, bokala
print($1);
    # bi bokalen arteko karakterea bistaratu
$testua =~ /\b([aeiou])\w\w\w\b/; # bokalez hasten diren
                                # lau karakteretako hitzak
print($1);                      # zein bokalekin hasten da?
```

Aurreko \$1, \$2, ... aldagai bereziekin lotura estua duten \1, \2, ... aldagaiak ere badira. Biek ala biek informazio bera gordetzen dute, beren arteko diferentzia bakarra haxe izanik: \ karakterea aurretik daramatenak espresio erregularrean bertan soilik erabil daitezke, eta \$ zeinua daukatenak, aldiz, espresio erregularretik kanpo baino ezin daitezke kontsultatu. Bien erabilera nahasteak uste-kabeko emaitzak eragin ditzake programan. Aldagai bereziak erabiltzean, kontuan izan beharrekoa da irakurketarako aldagaiak direla soilik, hau da, beraien balioa kontsulta dezakegu baina ez aldatu.

Adibide berri bat jartzearen, demagun bokal berdinarekin hasi eta amaitzen diren lau karaktereko hitzak bildu nahi ditugula. Adibidez: aita, otso, etab. Patroiak testatzeko `regExpOrok.pl` programarekin honela egingo genuke:

```
>perl regExpOrok.pl xuxen.txt "\b([aeiou])\w\w\1\b"
```

Bilaketa-patroian, atzera begirako \1 erreferentziak parentesi arteko espresioaren ([aeiou]) balioa hartuko du, eta bokal berdinarekin hasi eta amaitzen diren hitzak itzuliko ditu aurreko programa-deiak.

Atzera begirako erreferentziak erabiliz, "tira tira" edo "ezin ezin" bezalako hitz-errepikapenak ere topa ditzakegu. Espresio honek lau karaktereko hitz-errepikapenak bistaratuko ditu:

```
>perl regExpOrok.pl xuxen.txt "\b(\w\w\w\w)\s\1\b"
```

Parentesi artean bildutako espresioa \1 aldagaian gordetzen du Perl-ek. Adierazpen erregularrak bat egingo du lau karakterez osatutako *string*-a, zuriunea, eta ondoren berriro aurreko lau karaktereak dituen *string* puskarekin.

2. maila

Hona beste ariketa bat: eraiki lau karaktereko palindromoak bilatzen dituen adierazpen erregularra. Ondoren, proba egiteko erabili `regExpOrok.pl` programa eta `xuxen.txt` fitxategia.

Hona gure soluzioa:

```
>perl regExpOrok.pl xuxen.txt "\b(.) (.) \2\1\b"
```

Adierazpen erregularrak lau karaktereko hitzak bilatzen ditu murriztapen honekin: lehen biak edozein karaktere izan daitezke (puntu metakarakterek), baina hirugarrenak (`\2`) bigarrenaren berdina izan behar du, eta laugarrenak (`\1`) lehenengoaren berdina.

Gure bilaketa-esparrua hitzetara mugatu nahiko bagenu, zehatzagoa litzateke adierazpen erregular hau erabiltzea: `/\b([a-zA-Z])([a-zA-Z])\2\1\b/`

Ikasitakoa praktikan jarritz, egin dezagun bilaketa berri bat: karaktere-errepikapenez osatutako hitzak bilatuko ditugu, *borbor*, *zozo* edo *marmar*, adibidez. Lehenbizi, bi karaktereren errepikapenak diren hitzekin hasiko gara:

```
>perl regExpOrok.pl xuxen.txt "\b(\w\w)\1\b"
```

Orokor dezagun adierazpena, bi, hiru edo lau karaktereren errepikapenez osatutako hitzak bila ditzan aldi berean:

```
>perl regExpOrok.pl xuxen.txt
"\b(\w\w|\w\w\w|\w\w\w\w)\1\b"
arenaren
baba
bobo
borbor
...
zozo
Zuazua
```

Posible da parentesi bidezko espresio-bilketak habiaratzea ere, hau da, parentesiak bata bestearen barnean erabiltzea. Horrelakoetan, ezkerretik hasita lehenbizi zabaltze-parentesiari 1 aldagai-zenbakia esleituko dio Perl-ek, 2 hurrengo zabaltze-parentesiari, eta horrela hurrenez hurren.

Hona adibide bat:

```
"erabilgarritasun" =~ /(e(ra(bil(ga(rri(tasun|ago))))))/;
print ("$1\n");    # "erabilgarritasun" bistaratuko du
print ("$2\n");    # "rabilgarritasun" bistaratuko du
print ("$3\n");    # "bilgarritasun" bistaratuko du
print ("$4\n");    # "garritasun" bistaratuko du
print ("$5\n");    # "rritasun" bistaratuko du
print ("$6\n");    # "tasun" bistaratuko du
```

6.2.6. Errepikapenak: zenbat aldiz?

Gure lehenbiziko adierazpen erregularrek hitz edo karaktere-kate konketuak bilatzen zituzten. Ondoren, metakarakterek ezagutu ahala, patroio orokorrangoak eraikitzen ikasi dugu, baina betiere karaktere kopuru jakineko bilaketak eginez: bokalez hasi eta amaitzen diren lau karaktereko hitzak, hiru karaktereko hitzak, etab. Atal honetan, gure bilaketa-esparrua zabalduko dugu edozein luzeratako hitzak edo *string*-ak jasotzeko, adibidez: gutxienez bi kontsonante dituzten hitzak, bokalez hasi eta amaitzen diren hitz guztiak, etab.

Zeregin horretarako metakarakterek *kuantifikatzaileak* erabiliko ditugu: `?`, `*`, `+` eta `{}`. Horiekin, aukeratutako patroia nahi adina aldiz errepika dezakegu. Azter ditzagun bakoitzaren esanahia eta erabilera:

? Aurreko patroio elementua zero aldiz edo behin. Aukera bezala interpretatu daiteke, badago edo ez dago.

```
/\s?/;          # zuriunea behin edo zero aldiz
/[A-Z]?w\w/;    # letra larria eta bi karaktere, edo
                  # bi karaktere soilik
/di(tu)?gu/;     # "ditugu" edo "digu"
/oh?i/;          # "oi" edo "ohi"
```

+ Aurreko patroio elementua behin edo gehiagotan.

```
/\s+/;          # zuriune bat edo gehiago
/e.+o/;         # "e" eta "o" artean karaktere bat edo
                  # gehiago dutenak (ero, erro, enbido...)
/[1-7]+/;       # 1-7 arteko digituekin osatutako
                  # zenbakiak: 2, 3, 34, 456...
/\b\w+\b/;      # edozein luzerako hitza
/./+;           # karaktere bat edo gehiagoko edozein kate
/(\w+)\s+1/;    # edozein luzerako hitz errepikapenak:
                  # "bai bai" , "bero bero"...
```

* Aurreko patroia elementua zero aldiz edo gehiagotan.

```
/\s*/;          # zuriunea zero aldiz edo gehiagotan
/at*a/;         # "a" ondoren zero edo gehiagotan "t"
                # eta bukaeran "a" (aa, ata, atta, attta...)
/\s*\w+/;
    # Edozein luzerako zuriunea, zero barne, eta ondoren hitza
/./;

    # Edozein string-ekin bat egiten du (string hutsa barne)
```

{n,m} Aurreko patroia elementua gutxienez n aldiz eta gehienez m. Errepikapen kopuru zehatzagoak nahi ditugunean erabiltzeko.

```
/\b\w{2,4}\b/;      # 2 eta 4 karaktere arteko hitzak
```

{n,} Aurreko patroia elementua gutxienez n aldiz.

```
/\b\w{4,}\b/;       # gutxienez 4 karaktereko hitzak
```

{n} Bere aurreko patroia zehazki n aldiz.

```
/\b\w{5}\b/        # 5 karaktereko hitzak
```

Hona hemen zenbait adibide:

```
/ba?i/;           # "bai" eta "bi"
/ba+i/;           # "bai", "baai", "baaa"i...
/ba*i/;           # "bi", "bai", "baai", "baaa"i...
/ba{2,3}i/;       # "baai", "baaa"i
/ba{1}i/;         # "bai"
```

Laburpen taula:

Perl	Azalpena
<i>Karaktere bat adierazteko espresioak</i>	
a	"a" karakterea
[^a]	"a" ez den beste edozein karaktere
[a-z]	letra xeheen multzoa
[a-zñ]	letra xeheen multzoa ñ barne dagoela
[A-Z]	letra larrien multzoa (Ñ ez dago)
[a-zA-Z]	letra guztien multzoa (ñ ez dago)
[a-zA-ZñÑ]	letra guztien multzoa, ñ letra barne
[aeiou]	bokalak

[0-9]	zenbakiak
[^aeiou]	bokal ez den edozein karaktere
[^0-9]	edozein karaktere zenbakiak izan ezik
.	edozein karaktere
<i>Kokapena adierazteko espresioak</i>	
^	<i>string</i> hasiera
\$	<i>string</i> bukaera
\b	hitz-hasiera edo hitz-bukaera
<i>Errepikapen kopurua adierazteko espresioak</i>	
*	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
<i>Bestelakoak</i>	
.*	edozein karaktere-kate
xly	x edo y
(...)	Parentesi arteko espresioa elkartu (memoriaratzen da)
<i>Laburpenak</i>	
\.	puntu karakterea (berdin karaktere bereziekin)
\+	plus karakterea
\d	edozein digitu
\D	digitu ez den beste edozein karaktere
\w	edozein karaktere alfanumeriko
\W	alfanumeriko ez den karakterea
\s	zuriune bat, tabuladorea, lerro-bukaerakoa
\S	aurrekoaren kontrakoa

15. kasu praktikoa: nola hasi hala amaitu

Karaktere berdinarekin hasi eta amaitzen diren lerroak topatu nahi ditugu fitxategi batean. Erabili ikasitako adierazpen erregularrak horretarako.



.....

Hona hemen soluzio posible bat:

hasiBuka.pl

```
#!/usr/bin/perl
use warnings;
use strict;

my $fitx = $ARGV[0];
my $lerro,
open(FITX, $fitx) or
    die("Ezin izan dut $fitx fitxategia zabaldu!\n");
while ($lerro = <FITX>) {
    chomp($lerro);
    if ($lerro =~ /^(.)*\1$|^.$/) {
        # karaktere bakarreko kasua ere aintzat
        print ("$lerro\n");
    }
}
```

Programak ez dauka aparteko misteriorik: lerroz lerro fitxategia irakurri eta espresio erregularrarekin parekatzen du. Bi kasu bereizi ditugu espresioan, eta "|" metakarakterearen bidez bata edo bestea aukeran jarri. Lehen aukera: karaktere berdinez hasi eta bukatzen den *string*-a, tartean zero edo karaktere gehiago egon daitezkeelarik. Aukera honek gutxienez 2 karaktereko *string*-ekin bat egingo du. Bigarren aukera karaktere bakarreko *string*-a da, hau ere karaktere berdinez hasi eta amaitzen dena.

6.3. ADIERAZPEN ERREGULARREN FUNTZIONAMENDUA

Adierazpen erregularren oinarritzko sintaxia ezagutzen dugu jada. Badakigu sintaxi hori nola erabili gure beharretara egokitzen diren patroiak eraikitzeko, baita gero patroi horiek nola parekatu teklatu edo fitxategi bidez jasotako datu multzoekin ere. Baina aurrera jarraitu aurretik, une egokia da geldialditxo bat egin, eta patxadaz Perl-ek sintaxi hori nola aplikatzen duen ikusteko. Adierazpen erregularren parekatze-mekanismoak nola funtzionatzen duen aztertzeko, alegia.

Adibide batzuk jarriko ditugu, eta haiei buruzko galdera-erantzunekin aztertuko ditugu adierazpen erregularren tripak

Galdera: zer gordeko du \$1 aldagai bereziak parekatzearen ondoren?

```
$testua = "3, kolkua bete diru";
$testua =~ /([a-z]+)/;
```

Adierazpen erregularrak *string*-aren hasieratik hasten du parekatzea, eta patroiak *string*-aren zatiren batekin bat egin bezain azkar, karakterez karaktere bat egiten duten karaktere guztiak irentsiko ditu. Bat egiten ez duen lehen karakterearekin topo egitean, gelditu eta ez du aurrera jarraituko. Adibideko espresioak letra xehez osatutako hitz edo karaktere-kateak bilatzen ditu. Ezkerretik hasten da begira, eta bat egiten duen lehen karakterea "k" da. Karaktereak irentsiz jarraitzen du harik eta baldintza betetzen ez duen zuriunearekin topo egiten duen arte. Bilaketa bertan behera geldituko da memento horretan, eta \$1 aldagaiak "kolkea" hitza izango du.

Adierazpen erregularra azpiespresio batek baino gehiagok osatzen badute, parekatzea positiboa izan dadin, ezkerretik hasita bata bestearen ondoren guztiek bat egin behar dute testuarekin.

Galdera: zein balio izango dute parekatzearen ostean \$1 eta \$3 aldagaiek?

```
$testua = "Eginahalak egin, beti gelditzen da zer egin";
$testua =~ /\b(\w{2,4}) (\s) (\w+)/;
```

Erantzuna: \$1-ek "beti" karaktere-katea izango du eta \$3-k "gelditzen".

Adierazpena honela irakur daiteke: 2-4 karaktere arteko hitza, segidan zuriunea, eta ondoren edozein luzeratako hitza. Lehenengo azpiespresioa "egin" kateak betetzen du, baina ez dauka segidan zuriunerik eta espresio osoak ez du bat egiten. Aurrera segi, eta "beti" da lehen azpiespresioa betetzen duen hurrengo katea. Segidan zuriunea dauka, eta edozein luzeratako hitza "gelditzen" katea izango da. Adierazpen osoak bat egiten du. Beraz, \$1-ek "beti" hitza izango du eta \$3 aldagaiak "gelditzen".

Galdera: nolako *string*-ak bilatzen ditu ondoko adierazpenak? Zein balio izango dute \$1, \$2 eta \$3 aldagaiek?

```
$testua = "Ai, Maria! Ortzik ez dunak ogi mamia";
$testua =~ /^(a.*) (ortzik) (.*a)$/i;
```

Adierazpen erregularrak "a" karaktereaz hasi eta amaitzen den *string*-arekin bat egingo du, baldin eta tartean "ortzik" karaktere-katea baldin badauka. Aldagai bereziek izango dutena parekatzearen ostean:

```
$1 = "Ai, Maria! ";
$2 = "Ortzik";
$3 = " ez dunak ogi mamia";
```

Metakaraktere kuantifikatzaileak jatun aseezinak dira, eta ahalik eta *string* puskarik handiena irentsiko dute beti, adierazpen erregular osoaren bat-egitea mantentzen den bitartean, jakina.

Galdera: kuantifikatzaileen joera irenslea ezagututa, zein balio izango dute \$1, \$2 eta \$3 aldagai bereziek honako parekatzearen ostean?

```
$testua = "matematiketan bat zenbakia agertzen da gehienbat";
$testua =~ /(.*) (at) (.*)/;
```

Espresioak hau bilatuko du: "at" katea, aurretik eta atzetik edozein karaktere kopuru duelarik.

Badirudi lehen begiratuan "matematiketan" hitzeko "at" katearekin bat egin behar duela (at) azpiespresioak. Baina kuantifikatzaileak jatun amorratuak dira, eta adierazpen osoaren bat-egitea kolokan jartzen ez duen artean, karaktereak irensten jarraituko du lehenbiziko (.) azpiespresioak. Kasu honetan *string* osoa irentsiko du azken bi karaktereak izan ezik: "at". Bi karaktere horiek bigarren azpiespresioaren bat-egitea eragingo dute, eta hirugarrenak kate hutsarekin konformatu behar. Beraz, hauxe izango da aldagaien edukia parekatzearen ostean:

```
$1 = "matematiketan bat zenbakia agertzen da gehienb";
$2 = "at";
$3 = "";      # karaktere-kate hutsa
```

Metakarakterek kuantifikatzaileak aseeginak dira, ahalik eta karaktere-kate puskarik handiena irensten dute beti, besteei uzten dietenaz arduratu gabe. Gogoan izan behar dugu adierazpen erregular osoaren bat-egitea kolokan jartzen ez duen puskarik handiena jango dutela beti.

Joera irensle hori ez da beti desiragarria, ordea. Batzuetan nahiago genuke gure kuantifikatzaileek ahalik eta *string* puskarik txikienarekin bat egingo balute. Bada, kasu horietan, nahikoa da kuantifikatzaileei ? ikurra gehitzea atzean: ??, *?, +?, eta {}?. Aurretik *jatuna* zen kuantifikatzailea *mizkina* izango da aurrerantzean.

Har dezagun berriro aurreko adibidea:

```
$testua =
    "matematiketan bat zenbakia agertzen da gehienbat";
$testua =~ /(.*) (at) (.*)/;
```

Susmorik bai aldagai berezien balioei buruz?

Kuantifikatzaile mizkinak erabili ditugunez, lehenbiziko azpiespresioa "at" katearen lehen agerpenean geldituko da, aseta. Hirugarrenak ere espresioa betetzen duen puskarik txikiena irentsiko du, *string* hutsa kasu honetan.

```
$1 = "m";
$2 = "at";
$3 = "";
```

Galdera: ondoren datorren adierazpen erregularrean, bi aukerek bat egiten dute testuarekin. Zein aukeratuko du Perl-ek?

```
$testua = "Beharrak begiak gorri";
$testua =~ /(\w|\w+)/i;
```

Adierazpen erregularrak aukera bat baino gehiago baldin badauzka eskura, pentsatu gabe lehenbizikoa hartuko du beti. Nahiz eta bigarren aukerak bat-egite luzeagoa eskaini.

Beraz, adibideko espresioak lehenbiziko aukera `(\w)` hartuko du, eta `$1` aldagaian "B" karakterea gorde.

Bestelako ariketa bat proposatuko dizuegu oraingoan: programa eman eta haren portaera antzematen ahalegindu beharko zarete. Azpiko `bokalak.pl` programak fitxategi baten lerroz lerroko irakurketa eta adierazpen erregularren bidezko parekatzea konbinatzen ditu programa berean.



.....

bokalak.pl

```
#!/usr/bin/perl
use warnings;
use strict;

open(FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia ireki\n");
my $lerro;
my $kont = 0;
while ($lerro = <FITX>) {
    while ($lerro =~ m/[aeiou]/i) {
        $kont++;
    }
}
print $kont;
```

Ariketa ona da programa exekutatu aurretik kodea irakurtzeko tarte bat hartu eta bere portaera nolakoa izango den antzematen ahalegintzea. Programa geuk idatzitakoa bada, egindako akatsak topatzen lagunduko digu. Beste norbaitek idatzitako programa bada, aldiz, kasu honetan bezala, kodea irakurri ahala programaren asmoa igartzen trebatuko gara, bai eta egitura eta estrategia ezberdinen erabileran ere.

Tira, eta? Zer egingo du programak?

Zati handi bat ezaguna zaigu, behin baino gehiagotan erabili baitugu: komando-lerrotik pasatutako fitxategia ireki eta `while` begiztarekin lerroz lerro irakurtzen duena, hain zuzen ere. Berritasuna habiaratutako bigarren `while` begiztan dator, `$lerro =~ m/[aeiou]/` adierazpen erregularra daukana begiztako baldintza gisa. Badakigu ezen adierazpen horrek `$lerro` aldagaiko lehenengo bokalarekin bat egingo duela. Baina `while` egiturako baldintza gisa jartzean, zer egingo du? Badirudi begiztak biraka jarraitu behar duela `$lerro` aldagaian bokalik den artean. Begiztaren gorputzeko `$kont` aldagaia aldi bakoitzean eguneratuko da topatutako bokalen kontua eramanez.

Laburbilduz, programak fitxategi bat jaso eta bere bokal kopurua bistaratuko du pantailan. Ados al zaude?

Bada, exekutatu programa eta egin proba!

Zer gertatu da? Pantailan mezurik bistaratu ez, eta begizta infinitu batean bueltaka gelditu da programa. Gogoratu, gelditzeko *Ctrl-c*.

Uste genuen bezala, `while ($lerro =~ m/[aeiou]/)` sententziak iteratzen edo begiztan biraka mantenduko du programa lerroan bokalik den artean. Baina usteak erdia ustel, `$lerro` inon eguneratzen ez dugunez, beti bokal berdinarekin topo egingo du eta ez da inoiz begiztatik aterako! Hobeto uler dadin, demagun fitxategiaren lehen lerroa "Adiskidegabeko bizitza, auzogabeko heriotza." *string*-a dela. Lehen begiztak `$lerro` aldagaian gordeko du esaera. Bigarrenak, baldintzako adierazpen erregularra ebaluatzean, esaerako lehen bokalarekin topo egin "A") eta `$kont` aldagaiaren balioari bat gehituko dio. Berrito adierazpen erregularra ebaluatu, berrito "A" bokala topatu eta berrito `$kont` eguneratuko du. Eta horrela behin eta berriz amaiera jakinik gabe.

Gogoratu, `while` egitura azaltzean, nola azpimarratu genuen baldintza eguneratzearen garrantzia: begiztako baldintza **beti** eguneratu behar dugu, noizbait baldintza bete ez eta begiztatik ateratzeko modua izan dezan programak.

Bi soluzio posible proposatuko ditugu jarraian:

1.- Begiztaren iterazio bakoitzean `$lerro` aldagaia eguneratu tratatutako bokalak bazter utzi edo ezabatuz.

```
while ($lerro =~ m/[aeiou]/i) {  
    $lerro = $';  
    $kont++;  
}
```

Parekatzearen ondoren, Perl-ek `$&`, `$`` eta `$'` aldagai berezietan bat egin duen *string*-a, bat-egitearen aurrekoa eta bat-egitearen ostekoa gordetzen ditu hurrenez hurren. Gure kasuan, begiztaren gorputzean `$lerro` aldagaia `$'`-ren edukiarekin eguneratzen badugu (`$lerro = $'`), parekatzearen osteko *string* puskarekin alegia, begiztak iterazio bakoitzean bokal berri bat bilatuko duela ziurtatuko dugu.

2.- Adierazpen erregularrari nolabait pauso edo iterazio bakoitzean bokalen instantzia ezberdinak bilatu behar dituela agindu.

```
while ($lerro =~ m/[aeiou]/ig) {
    $kont++;
}
```

Adierazpenari *g* (*global*) *flag*-a gehitzea nahikoa da, espresioa ebaluatzen duen aldi bakoitzean Perl-ek patroiaren bat-egite berriak bila ditzan.

Horra beraz gure arazoari irtenbidea emango dioten bi proposamen. Inplementatu biak, aztertu beraien funtzionamendua, eta aukeratu gehien gustatzen zaizuna. Gure aldetik bakar-bakarrik esan bigarren aukera ezagunagoa dela, eta kodeari begiratzuz gero, apur bat sinpleagoa ere bai. Baina besterik ez, biak dira baliozkoak.

Askotan errepikatu dugun ariketa mota da fitxategia lerroz lerro irakurri eta bilaketa-patroiarekin parekatzearena. Orain artean, gehienez ere lerro bakoitzeko parekatze positibo bat bistartzeko baino ez ginen gai. Aurrerantzean, lerroko parekatze positibo **guztiak** bistaratu nahi ditugunean, bi proposamen hauetako edozein erabili ahal izango dugu. Gogoratu:

- Adierazpen erregularrari *g* *flag*-a gehitu:

```
while ($lerro =~ m/patroia/g) {}
```

- Iterazio bakoitzean aldagaia (`$lerro`) eguneratu:

```
while ($lerro =~ m/patroia/) {
    $lerro = $';
}
```

16. kasu praktikoa: N. bat-egitea aurkitu

Bilaketa aurreratuagoak egiteko gai garenez, bururatu zaigu erabilgarria litzatekeela patroia baten *N*. agerpena edo **bat-egitea** topatzen duen programa. Sarrera gisa, komando-lerrotik fitxategi-izena, bilaketa-patroia eta *N* zenbaki bat jaso eta irteera, patroia *N*. agerpena eragin duen lerroa izango da. Hona adibide gisa, `esaeraLabur.txt` fitxategian "ez" patroia *N*. agerpena bilatzen duen programa deia:

```
>perl agerpenN.pl esaeraLabur.txt ez 2
topatu dut!
Bat eman eta bi hartu, gure etxean ez berriz sartu
```

Saia zaitez programa idazten, eta ondoren, nahi baduzu, begiratu gure soluzioari.



.....

agerpenN.pl

```
#!/usr/bin/perl
use warnings;
use strict;
my $fitx = $ARGV[0];
my $patroi = $ARGV[1];
my $zenb = $ARGV[2];
my $kop = 0;
my $lerro,
open(FITX, "$fitx") ||
    die("Ezin $fitx fitxategia zabaldu!\n");
while ($lerro = <FITX>) {
    while ($lerro =~ /$patroi/gi) {
        $kop++;
        if ($kop == $zenb) {
            print ("topatu dut!\n");
            print ($lerro);
        }
    }
}
```

Programak patroiaren instantziak bilatzen ditu lerroz lerro. Adierazpenak *g* *flag*-a daukanez, lerro bereko agerpen guztiak kontatuko ditu.

17. kasu praktikoa: esku bakarrarekin idatzitako hitzak

Jolaserako tartea ere hartu behar da noizean behin, dena ez da izango azalpena eta ariketa. Hona hemen, jolasa eta ariketa uztartzen dituen proposamen bat: gure ordenagailuetako ohiko teklatua erabiliz (QWERTY), eta mekanografiako arauak aplikatuz, zein da esku bakarra erabiliz idatz dezakezun hitzik luzeena?

Nahasmenik izan ez dadin, hauek dira mekanografiako arauen arabera ezkerreko eskuarekin idatz daitezkeen karaktereak: q w e r t a s d f g z x c v b. Hitzak: *zerbeza*, *baztertze*, *area*. Eskuineko eskuari dagozkionak, aldiz: y u i o p h j k l ñ n m. Hitzak: *mihi*, *ukuilu*.

Ea nork topatzen duen luzeena. Gurea "*berraztertze*" da, 12 karakterekoa.

Laguntza gisa, Perl *script* bat idatz dezakezue, teklatu bidez hitz bat jaso eta ezkerreko eskuaz, eskuinez edo biak erabilia idatzi dugun esango diguna. Adierazpen erregularrei buruz dakiguna jakinda, programa idazteak ez liguke buruhauste handiegirik sortu beharko.



.....

Hona hemen gure proposamena:

qwerty.pl

```
#!/usr/bin/perl
# Sarrera: teklatu bidez idatzitako hitza
# Irteera: ezkerreko eskuaz, eskubikoaz edo biak
# erabilia idatzi den
use warnings;
use strict;
print("Idatzi hitz bat eta ondoren sakatu return\n");
my $lerro = <STDIN>;
chomp($lerro);
my $luzera = length($lerro);
# ezkerreko eskuarekin soilik?
if ($lerro =~ /^[qwertasdfgxcvb]+$/i) {
    print("Ezkerraz idatzitakoa. Luzera: $luzera\n");
}
# eskubiko eskuarekin soilik?
elsif ($lerro =~ /^[yuiophjklñm]+$/i) {
    print("Eskubiaz idatzitakoa. Luzera: $luzera\n");
}
else {
    print("Bi eskuak erabili dituzu!");
}
```

Topatu al duzu gurea baino hitz luzeagorik? Ea bada, oraingoan gu geu hitzak asmatzen aritu beharrean, sarrera gisa fitxategi bat pasatuko diogu programari eta berak bistaratu ditzala ezkerreko zein eskuineko eskuekin idatzitakoak. Baita alde bakoitzeko hitzik luzeenak ere.

Programa hau ez da aurrekoa bezala di-da batean idaztekoa, eta komeni da aurretik algoritmoa ondo zehaztea. Saiatu zaitez fitxategiko hitz guztiak aztertuko dituen programaren pausoak idazten.



.....

Algoritmoa:

1. Fitxategia ireki
2. Fitxategi amaiera ez den bitartean, irakurri \$lerro
 - 2.1. \$lerro *string*-ean aurkitzen duen hitz bakoitzeko
 - 2.1.1. Gorde hitza \$hitza aldagaian
 - 2.1.1. Kalkulatu karaktere kopurua eta gorde:


```
$karKop = length($hitza);
```
 - 2.1.2. Baldin \$hitza ezkerrez idatzitakoa, bistaratu mezua
 - 2.1.2.1. Hitzik luzeena da?


```
if ($karKop > $sezkerLuze) orduan
    $sezkerLuze= $karKop;
    $sezkerHitz = $hitza;
```
 - 2.1.3. Bestela, baldin \$hitza eskubiaz idatzia, bistaratu mezua
 - 2.1.3.1. Hitzik luzeena da?


```
if ($karKop > $eskuinLuze) orduan
    $eskuinLuze = $karKop;
    $eskuinHitz = $hitza;
```
3. Bistaratu \$sezkerHitz, \$sezkerLuze eta \$eskuinHitz, \$eskuinLuze

Algoritmoan agertzen ez diren detaile ugari jaso behar ditu kodeak. Honez-kero ondo asko dakizu akats txiki batek nolako eragina izan dezakeen programaren portaera orokorrean. Baina akatsez jabetzeko, ez dago kodea norberak idaztea bezalakorik. Saia zaitetz beraz eskemari jarraitu (zurea edo guk proposatua) eta programa idazten.



.....

qwerty2.pl

```
#!/usr/bin/perl
# Sarrera: komando lerrotik fitxategi-izena
# Irteera: bistaratu ezkerreko edo eskubiko eskuaz soilik
# idatzitako hitzak, # eta hitzik luzeenak
use warnings;
use strict;
```

```

open(Fitx, $ARGV[0]) ||
    die("Ezin izan dut $ARGV[0] fitxategia zabaldu!\n");
my $lerro;
my $hitza;
my $karKop;
my $sezkerLuzer = 0;
my $eskuinLuzer = 0;
my $sezkerHitz;
my $eskuinHitz;

while($lerro = <Fitx>) {
    chomp($lerro);
    # baldintza: lerroko hitz bakoitzeko
    while ($lerro =~ /\w+/g) {
        $hitza = $1;    # hitza gorde
        $karKop = length($hitza);
        # ezkerrekin?
        if ($hitza =~ /^[qwertasdfgxcvb]+$ /i) {
            print("Ezkerrekin: \t$hitza\n");
            # ezkerrekin idatzitako luzer?
            if ($karKop > $sezkerLuzer) {
                $sezkerLuzer = $karKop;
                $sezkerHitz = $hitza;
            }
        }
        # eskuinarekin?
        if ($hitza =~ /^[yuiophjklñnm]+$ /i) {
            print("Eskuinarekin: \t$hitza\n");
            # eskuinarekin idatzitako luzer?
            if ($karKop > $eskuinLuzer) {
                $eskuinLuzer = $karKop;
                $eskuinHitz = $hitza;
            }
        }
    }
}

print ("Ezkerrekin idatzitako luzer: $sezkerHitz.
      Karaktereak: $sezkerLuzer\n");
print ("Eskuinarekin idatzitako luzer: $eskuinHitz.
      Karaktereak: $eskuinLuzer\n");
close(Fitx);

```

Idatzi programa eta exekutatu. Proba egin fitxategi ezberdinekin. Esku bakarrez idatzitako hitz asko aurkitu al ditu? Badirudi askoz gehiago direla ezkerrez idatzitakoak, ezta? Moldatu ezazu programa ezker zein eskuin eskuez

idatzitako hitzen kopurua zenbatu dezan. Hitz asko topatzen dituenean, pantaila ez da oso eroso emaitza guztiak ikusteko. Aldatu programa ezkerreko eskuaz idatzitako hitzak `ezker.txt` fitxategian gorde ditzan eta eskuinez idatziak, berriz, `eskubi.txt` fitxategian.

6.4. ORDEZKAPENAK

Orain artean *string*-etan bilaketak egitera mugatu gara adierazpen erregularretan `m//` parekatze funtzioa erabiliz. Baina horrez gain, Perl-ek *string* bat *bilatu* eta beste batekin *ordezkatzeko* aukera ere eskaintzen du `s///` funtzioaren bitartez. Funtzioaren erabilera orokorra honakoa da:

```
s/patroia/ordezkapena/modifikatzaileak
```

Non *patroia* testuan bilaketak egiteko erabiliko den espresioa den, orain arte erabili dugun bilaketa-patroia alegia. Parekatze positiboa gertatzen denean, Perl-ek patroiarekin bat egindako *string* puska *ordezkapena* argumentuarekin ordezkatuko du testuan. Bigarren argumentu hau, *ordezkapena*, *string*-a izan daiteke edo baita aldagaia ere. Kasu honetan, Perl-ek *ordezkapena* burutzean aldagaiaaren balioarekin ordezkatuko du bat egin duen *string* puska. Amaitzeko, *ordezkapen-funtzioarekin* ere erabil daitezke *flag*-ak, aurretik ikusitako berberak.

Adibidea:

```
$testua =~ s/1/bat/;
```

Adierazpenak "1" digituaren lehen agerpena "bat" karaktere-katearekin ordezkatuko du. Berriro ere, testua eta adierazpen erregularra lotzen dituen eragilea `=~` da. Ordezkapena "1"-en agerpen guztietarako egin nahi izanez gero, `g` (global) *flag* edo modifikatzailea gehitu behar diogu espresioari bukaeran. Parekatze positiborik izan bada, *ordezkapena* burutzeaz gain, `s///` funtzioak egindako *ordezkapen* kopurua itzuliko digu.

Hona hemen azalpena ilustratzen duen adibidea:

```
$testua = "Bat, bat, bat, bart parrandan ibili...";
$testua =~ s/bat/1/ig; # ordezkatu agerpen guztiak
print("$testua\n");    # "bat,bat,bat, bart parrandan ibili..."
print ($testua =~ s/1/bat/g);
                                # ordezkatu eta kopurua bistaratu: "3"
print("$testua\n");    # "bat, bat, bat, bart parrandan ibili..."
```

18. kasu praktikoa: bilatu eta ordeztu

Testu-editoreetan horren erabilia den *bilatu eta ordeztu* funtzioa inplementatuko dugu adierazpen erregularrak erabiliz. Programak sarrerako hiru argumentu izango ditu ondoko programa-deiak erakusten duen bezala:

```
>perl ordezkatu.pl esaeraLabur.txt kaixo iepa
```

Programak, `esaeraLabur.txt` fitxategiko `kaixo` hitzaren agerpen guztiak ordezkatuko ditu `iepa` hitzarekin. Horrekin batera, egindako ordezkapen kopurua pantailaratu behar du baita.

ordezkatu.pl

```
#!/usr/bin/perl
use warnings;
use strict;

# Egiaztatu argumentu kopurua egokia dela
die("programak hiru argumentu behar ditu!\n")
    if ($#ARGV != 2);

open (FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia zabaldu\n");
my $lerro;
my $agerpenak;
while ($lerro = <FITX>) {
    $agerpenak += ($lerro =~ s/$ARGV[1]/$ARGV[2]/ig);
    print $lerro;
}
print ("\n\nOrdezkapen kopurua: $agerpenak\n");
```

Programak lerroz lerro irakurtzen du fitxategia ohiko egitura erabiliz. Lerro bakoitzean, ordezkapenak egin eta zenbatzen dituen sententzia hauxe da:

```
$agerpenak += ($lerro =~ s/$ARGV[1]/$ARGV[2]/ig);
```

Sententziak bi zati berezi dauzka. Bata, eskuinaldeko zatia, `($lerro =~ s/$ARGV[1]/$ARGV[2]/ig)`, zeinak `$ARGV[1]` argumentuaren agerpen guztiak ordezkatu dituen `$ARGV[2]`-rekin, eta horrekin batera burututako ordezkapen kopurua itzuli. Beste zatiak, `$agerpenak +=`, ordezkapen kopurua hori jaso eta `$agerpenak` aldagaiari gehituko dio.

Idatzi dugun programa, ohiko *bilatu eta ordeztu* funtzioa baino askoz ahaltsuagoa da, adierazpen erregularrak onartzen baititu bilaketa-patroi gisa. Demagun, adibide bat jartzearen, fitxategi bateko bokal guztiak ordezkatu nahi ditugula "*" jarritz euren lekuan:

```
>perl ordezkatu.pl esaeraLabur.txt [aeiou] *
*d*sk*d*g*b*k* b*z*tz*, **z*g*b*k* h*r**tz*.
*ld* l*z**k, g*zt** *h*zt*
...
Z*h*rr*r* *z *sk* g*r*

Ordezkapen kopurua: 833
>
```

Bestalde, gure asmoa "a" karakterearen agerpen kopurua jakitea bada besterik gabe, honela erabil genezake programa:

```
>perl ordezkatu.pl esaeraLabur.txt a a
```

Ordezkapenak ez du eraginik izango testuan, eta bukaeran "a"-ren agerpen kopurua itzuliko digu programak.

Begiratu ondoko ordezkapena eta erantzun: zer gertatuko litzateke ordezkapen hau egiten ahaleginduko bagina?

```
$testua =~ s/$hitza//ig;
```

Kasu horretan, \$hitza patroiaaren agerpen bakoitza *string* hutsarekin ordezkatuko luke Perl-ek. Beste hitzetan esanda, patroiaaren agerpen guztiak ezabatuko lituzke testuan.

Adibide gisa, esaeraLabur.txt fitxategiko bokal guztiak ezabatu nahiko bagenitu:

```
>perl ordezkatu.pl esaeraLabur.txt [aeiou] ""
dskdgbk bztz, zgbk hrtz.
ld lzk, gzt hzt
mn, z hr t n hmn.
...
Zhrrrr z sk gr
>
```

2. maila



19. kasu praktikoa: lehen eta azken hitzak trukatu

Teklatu bidez esaldi bat jaso, eta esaldiko lehen eta azken hitzak trukatzen dituen programa idatzi.

Funtzioaren bi argumentuek ez daukate zertan luzera berekoak izan. Adibide gisa, jar dezagun bigarren argumentua lehenengoa baino luzeagoa daukan espresioa:

```
$testua =~ tr/012/abcde/;
```

Funtzioak "0", "1" eta "2"-ren agerpenak "a", "b" eta "c" letraz ordezkatu ditu hurrenez hurren; "d" eta "e", berriz, ez ditu aintzat hartuko.

Kontrakoa ere gerta liteke, bigarrena lehena baino motzagoa izatea:

```
$testua =~ tr/0123/ab/;
```

Oraingoan, "0" ordezkatu du "a" letrarekin, eta "1", berriz, "b"-rekin. Hortik aurrerako guztiak, "2" eta "3", "b"-rekin ordezkatu ditu. Bigarren zerrendan parekide gabe gelditzen diren lehen zerrendako elementuak, bigarreneko azken elementuarekin ordezkatu dira.

Kontuan izan behar da, orobat, `tr///` ahalik eta ordezkapen gehien egiten ahalegintzen dela, bukaeran `g flag`-a jarri beharrik izan gabe. Ordezkapenak egiteaz gain, itzulera balio gisa zenbat ordezkapen egin dituen ere itzultzen du `tr///` funtzioak. Begiratu ondoko adibideari:

```
$bokalak = ($testua =~ tr/aeiou//);
```

Lehenbiziko argumentutzat bokalak hartzen ditu adibideko `tr///` funtzioak, baina bigarren argumenturik ez du, hutsik dago. Zerekin ordezkatu ez duenez, funtzioak ez du ordezkapenik egingo, baina itzulera-balio gisa testuko bokalen agerpen kopurua bai itzuliko digu. Balio hori, `$bokalak` aldagaian gordetzen du adibideko sententziak.

Ikusi bezala, ordezkapenik egin gabe, karaktere zerrenda baten agerpen kopurua zenbatu nahi dugunean ere erabilgarria da `tr///` funtzioa.

Baina ordezkapen-funtzioa `d flag`-arekin erabiliz gero, bigarrenean parekiderik ez daukaten lehen argumentuko elementuak ezabatu egingo ditu. Adibidez:

```
$testua =~ tr/0123/ab/d;
```

Funtzioak "0" ordezkatu du "a" letrarekin, "1", berriz, "b"-rekin, eta "2" eta "3"-ren agerpen guztiak ezabatu egingo ditu.

Azken ezaugarri honek badauzka bere aplikazioak. Esate baterako, *string* batean zuriuneak ezabatu nahi izanez gero:

```
$testua =~ tr/ //d;
```

Testuan agertzen diren zenbaki guztiak ezabatzeko, berriz:

```
$testua =~ tr/0-9//d;
```

Beste espresio honek, *string*-ak izan ditzakeen puntuazio-marka guztiak ezabatuko ditu:

```
$testua =~ tr/\.\?!;:,-//d;
```

20. kasu praktikoa: rot13 zifratze-metodoa

Bere ibili gorabeheratsuan, gizakiak mezuak isilpean bidali eta jasotzeko makina bat kode berezi asmatu du. Bada sistema bat, Zesarren zifratze-metodoa deiturikoa (pentsa noizkoa den!). Metodo horretan, mezua zifratzeko letra guztiak **n** posizio mugitzen dira eskuinerantz; **n** igorleak eta jasotzaileak aldeez aurretik adosturiko zenbakia izanik.

Zesarren metodoa orokorra da, eta **rot13** haren kasu berezia, non **n** zenbakia 13 den. Hau da, mezua zifratzeko, jatorrizkoaren letra guztiak 13 posizio mugitzen dira eskuinerantz. Helburura iristean, jatorrizko mezua irakurtzeko alderantzizko eragiketa aplikatu (13 posizio ezkererantz mugitu) eta listo.

Gara dezagun teklatu bidez jasotako testua **rot13** zifratze-algoritmoarekin kodetu eta ondoren pantailan bistaratzen duen programa. Erabiltzaileak nahi beste lerro idatzi ahal izango ditu teklatu bidez, eta amaitzeko lerro hutsa sartuko du. Hona hemen bere erabileraren adibidea:

```
>perl rot13.pl
Idatzi testua lerroz-lerro. Amaitzeko sartu lerro hutsa
ari naizela ari naizela
hor ikusten det Txirrita

Testua zifratuta:
nev anvmryn nev anvmryn
ube vxhfgra qrg Gkveevgn
>
```

Ariketak, honako eragiketak eskatzen ditu: lerroak irakurri, aldagai batean gorde, karakterez karaktere ordezkapena gauzatu `tr///` funtzioarekin eta pantailan bistaratu.

Tarte bat hartu eta erabaki nola egin programa.



.

Programaren eskema orokorra:

1. Lerro hutsa ez den bitartean, irakurri teklatutik \$lerroa
 - 1.1. Kateatu \$lerroa *string*-a \$testua-ri
2. \$testua aldagaiko karaktereak desplazatu 13 posizio eskuinerantz:


```
$testua =~ tr/a-zA-Z/n-za-mN-ZA-M/;
```
3. Bistaratu \$testua

Eta hona eskema programa bihurtua:

rot13.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;
my $lerroa;
my $testua = "";
print "Idatzi testua lerroz-lerro. Amaitzeko sartu lerro
      hutsa\n";
while (length($lerroa = <STDIN>) > 1) {
    $testua .= $lerroa; # $testua = $testua . $lerroa;
}
# Letrak 13 posizio eskuinerantz
$testua =~ tr/a-zA-Z/n-za-mN-ZA-M/;
print("Testua zifratuta:\n");
print ($testua);
```

6.6. NOLA IDATZI ADIERAZPEN ERREGULARRAK

Dударик gabe, adierazpen erregularrak potentzial handiko tresnak dira. Alabaina, ataza bakoitzeko adierazpen egokiak sortzea ez da beti lan erraza izaten. Nahi baino maizago gertatu ohi da adierazpena idatzi eta exekutatzean bere funtzioa ondo ez betetzea, hau da, nahi/espero genuen testu puskarekin bat egin ez eta ustekabeko emaitzak itzultzea. Izan ere, metakarakterek tresna ahaltsuak dira, eta haien arteko konbinaketek izango duten portaera aurreikusteak buru-ariketa handia eskatzen du.

Une bakoitzeko adierazpen egokiak sortzeko makinarik ez daukagu tamalez. Hemen ere, ibilian-ibilian ikasiko dugu oinez. Baina laguntza gisa, hona hemen adierazpen erregular konplexuak idazterakoan jarraitu beharreko pausoak:

1. Adierazpen erregularra idazten hasi baino lehen, garbi izan behar dugu patroiak testuaren zein puskarekin bat egin behar duen. Zehaztu ahalik eta gehien **bat-egite** positibo eta negatiboen kasuak.

2. Ataza konplikatua bada, banatu problema soluzio errazagoko puska txikietan.
3. Puska txiki bakoitzeko adierazpen erregularra idatzi.
4. Bildu adierazpen erregularrak bakar batean.

6.7. *split()* ETA *join()*: **BANATU ETA BILDU**

Testuekin lan egitean badira bi funtzio bereziki erabilgarriak direnak, `split()` eta `join()`. Lehenak *string* edo karaktere-kateak puskatan banatzen ditu; eta bigarrenak, aldiz, kontrako lana, *string* puskek bakarrean bildu. Bi funtzioek adierazpen erregularrekin lan egiten dute.

Demagun, adibidez, \$mendiak aldagai eskalarrak honakoa gordetzen duela:

```
$mendiak = "Aizkorri Toloño Lakora Soila";
```

Edukia *array* batean gorde nahiko bagenu, mendi bakoitza *array*-ko posizio batean, `split()` funtzioa eta adierazpen erregularrak erabiliz erraza litzateke:

```
@mendiak = split(/\s+/, $mendiak);
```

Orain mendi guztiak @mendiak *array*-an daude gordeta, *array*-aren posizio bakoitzeko bat. Beraz, Lakora inprimatu nahiko bagenu:

```
print($mendiak[2]);          # Lakora
```

Adibidearen ondoren, ikus dezagun `split()` funtzioaren sintaxi orokorra:

```
@array = split(/espReg/, $string);
```

Bi argumentu dauzka, lehena adierazpen erregularra eta bigarrena *string*-a. `split()`-ek lehen argumentuko patroia erabiltzen du bigarren argumentuko *string*-a puskatan banatzeko. Patroiak testuarekin bat egiten duen aldiro `split()`-ek *string*-ari puska bat kenduko dio, puska guztiak *array* edo zerrenda gisa itzuliz.

Lehenbiziko adibidean, zuriunea (`/\s+/`) erabili dugu banatzaile gisa, eta funtzioak zuriunearen agerpen bakoitzeko puska bat moztu dio *string*-ari. Hala ere, guk nahi dugun adierazpena erabil dezakegu lan horretarako:

```
$testua = "tomatea, letxuga, porruak, tipula";
@barazkiak = split(/,/, $testua); # koma ", " banatzaile gisa
```

Adierazpen erregularrak testuarekin bat egiten duen aldiro *string*-a zatikatuko du Perl-ek, banatzailea bera alboratuz. Banatzaile gisa adierazpen erregular hutsa (`//`) erabiliz gero, *string*-a karakterez karaktere banatuko du `split()` funtzioak. Adibidez:

```
$testua = "Gabikagogeaskoa";
@karaktereak = split(//, $testua);
print($karaktereak[0]);           # G
print($karaktereak[14]);          # a
```

Testuko karaktere bakoitzari *array*-ko posizio bat esleitu dio.

Testua puskatan banatu nahi badugu ondoren zati bakoitzarekin lan egiteko, `split()` da gure funtzioa.

21. kasu praktikoa: testua hitzetan banatu

Sarrera gisa komando-lerrotik fitxategi-izena jaso, eta bere edukia hitzetan banatzen duen programa idatziko dugu. Hitzak *array* egitura batean gordeko ditu, ondoren beraien gainean edozein eragiketa aplikatu ahal izateko moduan. Guk, besterik gabe pantailan bistaratuko ditugu banan-banan, bukaeran fitxategiaren hitz kopurua bistaratzuz.

hitzak.pl

```
#!/usr/bin/perl
use warnings;
use strict;

open(FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia zabaldu!\n");
my $lerro;
my $hitzKop = 0;
my @hitzLerro;
my @hitzak;

while ($lerro = <FITX>) {
    chomp($lerro);
    $lerro =~ tr/\.\?!;,:-//d; # puntuazio-ikurrak ezabatu
    # hitz banatzailea: zuriunea
    @hitzLerro = split(/\s+/, $lerro);
    # lerroko hitzak zerrendara gehitu
    push(@hitzak, @hitzLerro);
}

foreach my $hitz(@hitzak) {
    $hitzKop++;
    print("$hitz\n");
}

print("Hitz kopurua: $hitzKop\n");
```

Programak komando-lerrotik jasotako fitxategia zabaldu eta lerroz lerro irakurtzen du dagoeneko ezaguna zaigun egitura erabiliz. Fitxategiko lerro bakoitzaren gainean hiru eragiketa hauek aplikatzen dira: lehenbizi puntuazio-marka eta marratxoak ezabatzen ditu `tr///` funtzioak. Ondoren, `split()` funtzioak \$lerro *string*-a hitzetan banatzen du zuriunea erabiliz banatzaile bezala. Azkenik, hitzok `@hitzak` bektoreari gehitzen dizkio `push()` funtzioak.

Programaren bigarren zatiak hitzak banan-banan pantailaratzea besterik ez du egin behar, kontagailuak hitz kopurua zenbatzen duen bitartean. Programaren amaiera, hitz kopuru totala bistaratzen duen mezuarekin iritsiko da.

22. kasu praktikoa: N karaktereko hitzak

Argumentu gisa fitxategi-izena eta N zenbaki bat jaso eta, fitxategi horretan dauden N karaktereko hitz guztiak pantailaratzen dituen programa nahi dugu oraingoan. Bukaera, topatutako hitz kopurua ere bistaratu behar du programak.

Ez gara sekula nekatuko esaten kodea idatzi aurretik algoritmoa edo eskema garatzea zeinen garrantzitsua den.



.

Algoritmoa:

1. Argumentuak irakurri
 - 1.1. Egiaztatu argumentu kopurua zuzena dela
 - 1.2. Fitxategia zabaldu
2. Irakurri fitxategia lerroz lerro
 - 2.1. Ezabatu puntuazio-markak
 - 2.2. Lerroa hitzetan banatu
 - 2.3. Hitz bakoitzeko:
 - 2.3.1. Baldin karaktere kopurua == N

Pantailan bistaratuko
 Kontagailua++
3. Bistaratu N luzerako hitz kopurua

Algoritmoa Perl lengoaiara itzuli ondoren:

```
#!/usr/bin/perl
use warnings;
use strict;

my $lerro;
my $kopurua = 0;
my $luzera = $ARGV[1];
if ($#ARGV == 1) {
    open(FITX, $ARGV[0]) ||
        die ("Ezin $ARGV[0] fitxategia zabaldu\n");
}
else {
    die("Bi argumentu behar dira: fitxategi-izena eta hitz-
        luzera");
}
while($lerro = <FITX>) {
    chomp($lerro);
    $lerro =~tr/;:,.!?!-/d;
    foreach my $hitza (split(/\s+/, $lerro)) {
        if (length($hitza) == $luzera) {
            print ("$hitza\n");
            $kopurua++;
        }
    }
}
print ("\n$luzera karakteretako $kopurua hitz daude
    fitxategian\n");
```

Egitura aldetik proposamen berririk agertu ez arren, badira bi ezaugarri aipatzeko modukoak: lehenbizikoak besterik egin aurretik programak sarrerako datuak egokiak direla egiaztatzen du. Hau da, erabiltzaileak komando-lerroan bi argumentu idatzi dituela, lehena fitxategi-izena eta bigarrena zenbakia. Gure programetan mota honetako egiaztapenak egitea oso gomendagarria da, erabiltzaileak programa abiatzerakoan egin ditzakeen erroreak aurreikusiz. Bigarren ezaugarri aipagarria, `foreach` egituran ohi bezala *array* edo zerrenda erabili beharrean funtzio bat jarri izana da: `foreach my $hitza (split(/\s+/, $lerro))`. Hala jarri dugu `split()` funtzioak *array* edo zerrenda itzultzen duelako, eta horrenbestez, `$hitza` aldagaiak zerrenda horretako elementuak hartuko dituelako banan-banan.

3. maila

String-ak puskatan zatitzeko `split()` funtzioa oso erabilgarria da, eta gainera, banatzaile gisa adierazpen erregularrak erabiltzeak malgutasun handia ematen dio funtzioari. Ondoko adibideak ere testu bat hartu eta hitzetan banatzen du `split()` erabiliz. Puntuazio-marka, marratxo eta zuriuneak dira hitzen arteko banatzaileak

```
# $lerro string-a hitzetan banatu
@hitzak = split(/[ \s\.\?!;:,-]/, $lerro);
```

Topatutako hitzak `@hitzak` array-an gordeko ditu aurreko adierazpenak. String-a zatitzeko erabilitako banatzaileak berriz baztertu egiten ditu funtzioak. Hau da, `@hitzak` array-tik abiatuta haserako testua osatu nahiko bagenu, arazoak izango genituzke puntuazio-marka guztiak ezabatu ditugulako.

Baina espresio banatzailea parentesi artean jarritz gero, Perl-ek banatzailea ere gorde nahi dugula ulertuko du:

```
@hitzak = split(/([\s\.\?!;:,-])/, $lerro);
```

Beraz, ikur banatzaileak mantentzea interesatzen zaigunean, parentesi artean jarri behar dugu patroia banatzailea. Perl-ek gainerako elementuak bezala gordeko du zerrendan.

Espero izatekoa zen moduan, `split()` funtzioak badauka bere bikotea, `join()` izenekoa. Horrek, *string* edo karaktere-kateak hartu eta elkarrekin lotzen ditu *string* bakar batean. Bi argumentu dauzka `join()` funtzioak: lehena *string*-a, eta bigarrena *array*-a. Funtzioak, lehen argumentu bezala pasatutako *string*-a erabiliko du *array*-ko elementuak elkarrekin itsasteko. Hona hemen adibide bat:

```
print(join("-", a-z)); # a-b-c-d-e-...-z pantailaratuko du
$lerro = join(" ", @hitzak);
# array-ko hitzak elkarrekin itsatsi
# zuriunea jarritz hitzen artean
```

`join()` funtzioak ez ditu adierazpen erregularrak erabiltzen `split()`-ek bezala. Array-ko elementuak elkarrekin itsasteko edozein karaktere-kate erabil dezakegu.

Hemendik aurrera maiz joko dugu `split()` eta `join()` funtzioen laguntza bila. Elkarrekin lan ederra egin ohi dute testua zatitu, prozesatu eta ondoren berriro bildu behar denetan. Baina ez ahaztu gero `join()` funtzioaren lehen argumentuak *string*-a izan behar duela, eta ez adierazpen erregularra.

Hona hemen adierazpen erregularrak erabiltzen dituzten funtzioak denak batean:

<code>m//</code>	Bilaketa-funtzioa
<code>s///</code>	Ordezkapena
<code>tr///</code>	Ordezkapena karakterez karaktere
<code>split(//, string)</code>	<i>String</i> -a zatitu adierazpen erregularra erabiliaz banatzaile bezala

Honenbestez egin ditugu kapitulu honekikoak ere. Behin puntu honetara iritsita, badakigu adierazpen erregularren lengoaiak nolako potentziala daukan testuetan bilaketak eta aldaketak egiteko, eta baita potentzialtasun hori gure helburuetarako nola aplikatu ere. Bidea ez da hemen amaitzen, ordea, eta hemendik aurrerako kapituluetan ere presente izango dira adierazpen erregularrak.

Bi kasu praktiko ikusiz itxiko dugu kapitulua, adierazpen erregularrak erabiltzen dituzten funtzioak dantzan jarriko dituztenak.

23. kasu praktikoa: palindromoak

Teklatu bidez hitz edo esaldi bat jaso eta palindromoa den ala ez adierazten duen programa idatzi. Palindromoa da, ezkerretik eskuinera zein eskuinetik ezkerredera berdin irakurtzen dena. Adibidez: "ama", edo askoz luzeagoa den "nik enara neraman amaren aranekin".

Programak zuriune eta puntuazio-markak ez ditu aintzat hartu behar.

Hartu behar duzun denbora tarte eta idatzi programaren portaera deskribatuko duen algoritmoa.



.

Ataza nagusia hiru azpiataza edo egitekotan banatuko dugu:

1. Letra edo zenbakia ez den guztia ezabatu: zuriuneak, puntuazio-markak, etab.
2. Letra larriak xehe bihurtu
3. Karaktere-katea palindromoa den ala ez ebatzi

Lehen bi pausoei sarrerako testua prestatzea dute zeregin, ondoren prozesatu ahal izateko. Programaren mamia hirugarrenean dago. Nola jakin karaktere-kate bat palindromoa den ala ez? Erabakitzeko modu bat litzateke karakterez karaktere *array* batean gorde eta ondoren bere alderantzizkoarekin konparatzea. Katea palindromoa izango da bi *array*-ak (jatorrizkoa eta alderantzikatua) berdinak direnean. Kontuz, kasu berezi bezala karaktere bakarreko katea izango genuke, palindromoa dena baita.

Zehaztu dezagun 3. pausoa pixka bat gehiago:

3. Karaktere-katea palindromoa den ala ez ebatzi

3.1. Testua karakterez karaktere *array* batean gorde

3.2. *Array*-a atzekoz aurrera jarri

3.2. Konparatu aurreko biak

Baldin (karaktere kop == 1) bistaratu

"Letra bakarreko palindromoa!"

Baldin (\$testua eq \$AtzekozAurrera) bistaratu

"Palindromoa da"

Bestela bistaratu

"Ez da palindromoa"

Programaren portaera zehaztu ondoren, prest gaude kodea idazteko. Hona gure algoritmoari jarraitzen dion programa:

palindromo.pl

```
#!/usr/bin/perl
# sarrera: teklatu bidez idatzitako esaldia
# irteera: idatzitakoa palindromoa den ala ez
use warnings;
use strict;

print("Idatzi esaldi edo hitz bat:\n");
my $testua = <STDIN>;
chomp($testua);

# hitz karaktere ez direnak ezabatu
$testua =~ s/\W//g;
# Letra larriak xehe bihurtu
$testua =~ tr/A-Z/a-z/;
# karakterez-karaktere array-an gorde
my @karak = split (//, $testua);
# array-a atzekoz aurrera jarri eta karaktereak
# elkarrekin itsatsi
my $atzekozAu = join ("", reverse(@karak));

# palindromoa den konprobatu
if (@karak == 1) {
    print "Letra bakarreko palindromoa!\n";
}
elsif ($atzekozAu eq $testua) {
    print "Palindromoa da\n";
}
else {
    print "Ez da palindromoa\n";
}
```


*1. maila***24. kasu praktikoa: urkatuaren jokoa**

Urkatua izeneko jokoa ohiko denbora-pasa izaten zen eskola garaian. Jolaskidetako batek ezkutuko hitz bat aukeratu, eta besteak hitz hori zein den asmatu behar du, aldiro letra bat aukeratuz. Hitz ezkutuak aukeratutako letra baldin badauka, letra hori zenbat aldiz eta zein posiziotan azaltzen den adierazi beharko zaio jolaskideari. Huts eginez gero, urkamendirako pauso bat emango da gorputzeko zati bat marraztuz. Jokoa amaitzen da hitzaren letra guztiak asmatzean, edo gorputz osoa irudikatu ondoren urkatua gelditzen denean.

Ariketa honetan ordenagailua izango dugu aurkari. Ez du urkarik marraztuko, baina 7 aukera emango dizkigu ezkutuko hitza asmatzeko. Horretaz gain, pauso bakoitzean ahalegin kopurua, asmatutako letrak eta zein letra erabili ditugun adierazi beharko digu.

Programak argumentu bat jasoko du komando-lerrotik: hitz ezkutuak aukeratzeko erabiliko duen hiztegi-fitxategiaren izena. Lanak erraztearren, lerro bakoitzeko hitz bat gordetzen duen fitxategia erabiliko dugu, `xuxen.txt`.

Hona hemen `urkatua.pl` programaren erabileraren adibidea:

```
>perl urkatua.pl xuxen.txt

_ _ _ _ _
7 aukera gelditzen zaizkizu
Idatzi karaktere bat (erabilitakoak ): a

Ez dauka a karaktererik

_ _ _ _ _
6 aukera gelditzen zaizkizu
Idatzi karaktere bat (erabilitakoak a): e

Badauka e karakterea
_ e _ _ _ _ e _
6 aukera gelditzen zaizkizu
Idatzi karaktere bat (erabilitakoak a e):
...
}
```

Hartu behar duzun denbora eta saia zaitetz lehenbizi programaren algoritmoa edo portaera zehazten.



.

Hona gure proposamena:

1. Zabaldu fitxategia
2. Hiztegi fitxategitik hitz bat aukeratu ausaz.
3. Gorde hitza karakterez karaktere @hitza array-an
4. Begizta: 7 akats egin *edo* hitza asmatu bitartean
 - 4.1. Erabiltzaileari letra bat idazteko eskatu \$karak
 - 4.2. Konprobatu ezkutuko hitzak \$karak letra daukan ala ez


```
foreach $karHitz (@hitza) {
    baldin ($karHitz == $karak) {
        Hitzak badu $karak letra
    }
}
```
 - 4.3. Huts egin badu... akats kopurua eguneratu eta mezua bistaratu
 - 4.4. Asmatu badu... letraren posizioak pantailan erakutsi
5. Hitzaren letra guztiak asmatu diren ala ez egiaztatu
 - 5.1. Asmatu badu... zorionak!
 - 5.2. Ez badu asmatu... hitz ezkutua erakutsi

Honela itzuli dugu aurreko algoritmoa Perl lengoaiara:

urkatua.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;

# Komando-lerrotik hiztegi-fitxategiaren izena jaso
my $fitx = $ARGV[0];
# Fitxategia zabaldu
open (FITX, $fitx) or
    die("Ezin $fitx fitxategia zabaldu\n");
# Gorde fitxategia @hiztegia array-an
my @hiztegia = <FITX>;
# Hiztegiko hitz bat aukeratu zoriz
# $#hiztegia: @hiztegia-ren azken indizea
# rand($#hiztegia): 0 eta $#hiztegia arteko zorizko
# zenbaki bat
my $hitza = $hiztegia[rand($#hiztegia)];
chomp($hitza);
# hitza karaktere zerrenda bihurtu
my @hitza = split(//, $hitza);
```

```
# Jokoa erabilitako karaktereak gordetzeko array-a
my @ibilitakoak;
# Hitza asmatzeko aukera kopurua
my $aukerak = 7;
# Asmatutako letra kopurua
my $asmatuak = 0;
# 1 asmatu, 0 huts
my $igarri = 0;
# Teklatutik karaktere bat jasotzeko
my $karak;
# Asmatutako letrak
my @aurkituak;
# @aurkituak array-a hasieratu
for (my $i=0; $i<=$#hitza; $i++) {
    $aurkituak[$i] = "_";
}

# Jarraitu jokoa aukera guztiak erre edo hitza
# asmatu bitartean
while (($aukerak > 0) && ($asmatuak < length($hitza))) {
    print("@aurkituak\n");
    print("$aukerak aukera gelditzen zaizkizu\n");
    print("Idatzi karaktere bat (erabilitakoak
        @ibilitakoak): ");
    $karak = <STDIN>;
    chomp($karak);
    # Gorde karakterea erabilitakoen zerrendan
    push(@ibilitakoak, $karak);
    $igarri = 0;
    # Konprobatu ezkutuko hitzak $karak karakterea daukan
    # Karaktere berbera behin baino gehiagotan izan
    # dezake ezkutuko hitzak
    for (my $i = 0; $i <= $#hitza; $i++){
        if ($hitza[$i] eq $karak) {
            # Asmatutako karaktere kopurua gehitu
            $asmatuak++;
            # Asmatutako karaktereen array-ean dagokion
            # posizioan idatzi
            $aurkituak[$i] = $karak;
            # Jatorrizko hitzetik karakterea ezabatu
            $hitza[$i] = "_";
            $igarri = 1;
        }
    }
}
```

```
    if ($igarri) {
        print("\nBadauka $karak karakterea\n\n");
    }

    else {
        print("\nEz dauka $karak karaktererik\n\n");
        $aukerak--;
    }
} # while

if ($asmatuak == length($hitza)) {
    print("ZORIONAK! $hitza zen gure hitz sekretua\n");
}
else {
    print("Ez duzu asmatu: $hitza zen gure hitz sekretua\n");
}

close(FITX);
```



7. Hash egitura

Orain artean bi aldagai-klase baino ez ditugu erabili, eskalarrak eta *array*-ak. Dakigun bezala, eskalarrak balio bakunak gordetzen dituzten aldagaiak dira, eta *array*-ak, berriz, aldagai eskalarrez osatutako zerrendak. Baina bi horiekin batera bada hirugarren aldagai mota bat ere, **hash** deiturikoa, eta kapitulu labur hau berari eskainiko diogu.

Hash motako aldagaiak eskalar motako bi elementuz osatuak daude: **gakoa** eta **balioa**. *Hash* motakoa dela adierazteko, izenaren aurretik % ikurra jarri behar zaio aldagaiari. Aurretik ikusitako aldagai-izenei buruzko arauak bere horretan aplikatzen dira hemen ere.

Adibide gisa, hona gure lagunen izenak eta beraien jaioterria gordetzen dituen *hash*-a:

```
my %lagunak = ("Aitor" => "Leitza",  
              "Eneko" => "Ondarroa"  
              "Eneritz" => "Lasarte");
```

Ikus daitekeen moduan, *hash*-a balio-bikotez osatutako datu mota da. Geziaren ezker aldekoari **gakoa** deritzo, eta eskuinekoari, berriz, **balioa**. Gezia bera, bi balioak lotzen dituen eragilea da. *Hash* egiturako balio jakin bat kontsultatzeko haren gakoa erabiliko dugu:

```
print($lagunak{Eneritz});      # "Lasarte" inprimatuko du
```

Adibide gisa jarritako *hash* egitura irudi bidez adierazi beharko bagenu, honela egin genezake:

	% lagunak
Aitor	Leitza
Eneko	Ondarroa
Eneritz	Lasarte

Sarritan *array*-ekin konparatu ohi diren arren —batak zein besteak elkarrekin erlazionatutako balioak biltzen dituztelako—, badira ezberdintasunak beraien artean. Lehenbizikoa, begibistikoa dena, *array* motako aldagaiek @ ikurra daramate izenaren aurretik, *hash* motakoek, % ikurra. Bigarrena, *array* batean elementuak ordenaturik daude, beren indizearen arabera atzitu ditzakegaririk; *hash* egituretan, aldiz, elementu-bikoteen artean ez dago ezarritako ordenarik, ez dago lehen, bigarren edo hirugarren elementurik, denak zaku berean daude. Elementu jakin bat atzitu ahal izateko, haren gakoa erabili behar da.

```
$karak = $letrak[0];      # @letrak array-ko lehen elementua
$jaioterria = $lagunak{"Aitor"}; # $jaioterria == "leitza"
```

Hirugarren diferentzia, elementu indibidualak adierazteko moduari dagokio. *Array*-ekin indizea kortxete [] artean jarri behar den bezala, *hash* aldagaiekin gabiltzanean gakoa giltza {} artean jarriko dugu.

Hash motako aldagaiari **gako-balio** elementu-bikote berri bat gehitzeko, gakoari dagokion balioa esleitzea besterik ez dugu egin behar:

```
$lagunak{"Leire"} = "Laudio";
```

Hash-aren gakoak ezin dira errepikatu. Hau da, gako berdinarekin balio bat baino gehiago ezin ditugu gorde. Lehendik existitzen den gakoari balio berri bat gehitzen ahalegintzen bagara, aurretik zeukan balioa galduko du:

```
$lagunak{"Eneko"} = "Erreenteria";

# Enekor i jaioterria aldatu diogu,

# Ondarroa ezabatu eta Erreenteria jarritz
```

Hash-aren gakoak eta balioak aldagaiak izan daitezke. Kasu horretan, Perl-ek aldagaiaren lekuan dagokion balioa jarriko du. Segidan datorren programak erakusten du nola egin hori. Programak, teklatu bidez idatzitako lagunen izenak eta beren urtebetetze-datak gordetzen ditu *hash* egitura batean. Begiztan sartuta, sarrera berriak eskatuko ditu harik eta amaitzeko "q" letra sakatzen dugun arte.

urtebetetze.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;

my %urtebetetze;
my $data;
my $izena;
print("Idatzi lagunaren izena(amaitzeko sakatu q): ");
```

```

$izena = <STDIN>;
chomp($izena);
while ($izena ne "q") {
    print("$izena, urtebetetze data? HH-EE formatuan: ");
    $data = <STDIN>;
    chomp($data);
    $urtebetetze{$izena} = $data;
    print("Izena: $izena\tUrtebetetzea:
          $urtebetetze{$izena}\n");
    print("Idatzi lagunaren izena. Amaitzeko sakatu q\n");
    $izena = <STDIN>;
    chomp($izena);
}

```

Programaren lehen zatian, betiko lehen hiru lerroen ondoren, %urtebetetze *hash* aldagai nagusia eta bi aldagai eskalar laguntzaileak definitu ditugu. Segidan erabiltzaileari teklatu bidez lagun baten izena idazteko eskatzen diogu, \$izena aldagaian gorde eta bukaerako lerro-jauzi karakterea kenduz.

Programaren muina `while` begiztan dago. Erabiltzaileak "q" (amaitu) idazten ez duen bitartean, programa begiztan bueltaka arituko da. Iterazio bakoitzean, idatzi berri dugun izenari dagokion urtebetetze-datatik galdegingo du programak, hura sarrera estandarretik jaso eta \$data aldagaian gordez. Gakobalio bikote berria *hash*-ari gehitu, eta pantailan bistaratuko du ondoren. Berririo begiztako baldintza ebaluatu aurretik, izen berri bat idazteko eskatu digu programak, eta \$izena aldagaian gorde.

Array-ekin bezala, *hash*-ekin ere lan egiteko funtzio sorta eskaintzen du Perl-ek. Hurrengo ataletan erabilienak aztertuko ditugu.

7.1. delete()

Hash egituratik gako-balio bikotea ezabatzen duen funtzioa.

```

delete ($urtebetetze{"Aitor"});

# "Aitor" gakoa eta bere balioa ezabatu

```

Dagoeneko existitzen ez den sarrera bat atzitzen ahalegintzen bagara, Perl-ek errore-mezu bat bistaratuko du:

```

print($urtebetetze{"Aitor"});    # errorea!

```

Errore-mezuak saihesteko, gako-balio bikote jakin bat definituta dagoen ala ez galdegin dezakegu `exists()` funtzioa erabiliz.

7.2. *exists()*

Funtzio honek, gakoa argumentu bezala jaso eta *hash* egituraren definituta baldin badago, egiazkoa (*true*) itzultzen du, edo faltsua (*false*) bestela. Adibidez:

```
exists($urtebetetze("Aitor"));    # false
```

Molda dezagun aurretik idatzitako `urtebetetze.pl` programa teklatu bidez idatzitako izenak *hash* egiturari gehitu aurretik aurrez existitzen diren edo ez konproba dezan.

urtebetetze2.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;

my %urtebetetze;
my $data;
my $izena;
print("Idatzi lagunaren izena(amaitzeko sakatu q): ");
$izena = <STDIN>;
chomp($izena);
while ($izena ne "q") {
    if (exists($urtebetetze{$izena})) {
        print("Ez da sarrera berria!\n");
    }
    else {
        print("$izena, urtebetetze data? HH-EE formatuan: ");
        $data = <STDIN>;
        chomp($data);
        $urtebetetze{$izena} = $data;
        print("Izena: $izena\tUrtebetetzea:
              $urtebetetze{$izena}\n");
    }
    print("Idatzi lagunaren izena.Amaitzeko sakatu q\n");
    $izena = <STDIN>;
    chomp($izena);
}
```

Teklatu bidez jasotako izena aurretik existitzen bada `%urtebetetze` egituraren, mezu bat pantailaratuko du abisu emanaz, bestela urtemuga-dataz galdetu eta sarrera berria gehituko du lehen egiten zuen moduan.

7.3. *keys()*

Funtzio hau oso erabilgarria da *hash*-aren sarrera guztiak bisitatu nahi ditugunean. Argumentu bezala *hash* aldagai bat jaso eta bere gako guztien zerrenda itzultzen du *array* moduan. Adibidez, `keys(%urtebetetze)` idatzita, `%urtebetetze` egiturako gako guztien zerrenda itzuliko liguke funtzioak.

Moldaketa berri bat egingo diogu `urtebetetzeak` kudeatzeko programari: amaieran, *hash* egituraren sarrera guztiak pantailaratuko ditu alfabetikoki ordenatuta.

urtebetetze3.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;

my %urtebetetze;
my $data;
my $izena;
print("Idatzi lagunaren izena(amaitzeko sakatu q): ");
$Izena = <STDIN>;
chomp($izena);
while ($izena ne "q") {
    if (exists($urtebetetze{$izena})) {
        print("Ez da sarrera berria!\n");
    }
    else {
        print("$izena, urtebetetze data? HH-EE formatuan: ");
        $data = <STDIN>;
        chomp($data);
        $urtebetetze{$izena} = $data;
    }
    print("Idatzi lagunaren izena. Amaitzeko sakatu q\n");
    $izena = <STDIN>;
    chomp($izena);
}

foreach my $gako (sort(keys(%urtebetetze))) {
    print("Izena: $gako\tUrtebetetzea:
        $urtebetetze{$gako}\n");
}
```

Lehenbiziko zatian ez dago inolako aldaketarik. Berritasuna `while` begiztatik ateratzean dator, non programak `%urtebetetze` egiturako sarrera guztiak pantailaratzen dituen alfabetikoki ordenatuta. Horretarako `foreach` egitura eta bi funtzio, `sort()` eta `keys()`, erabili ditugu. `keys()` funtzioak *hash*-eko gako guztiak itzultzen ditu zerrenda batean, ordenarik gabe. Zerrenda hau `sort()` funtzioak jasotzen du eta alfabetikoki ordenaturik itzuli:

```
sort(keys(%urtebetetze))
```

Azkenik, `foreach` egiturak \$gako aldagaiarekin *array* ordenatuko balioak hartuko ditu banan-banan, eta `print()` funtzioa erabilia formatu egokiarekin pantailaratuko.

7.4. values()

Argumentu bezala *hash* egitura jaso, eta haren balio guztien zerrenda itzultzen du. Adibidez:

```
my %adina = ("Jon" => 26,
            "Kepa" => 41,
            "Itziar" => 26,
            "Nekane" => 17);
```

```
@urteak = values(%adina);    # @urteak == (26, 41, 17)
```

Funtzio hau ez da `keys()` bezain erabilgarria. Izan ere, gakoa izanda dago-kion balioa lortzerik badugu, baina kontrakoa ez da horren erraza. Adibidez: "26" balioa izanda, nola bereizi "Jon" eta "Itziar" gakoak?

25. kasu praktikoa: hitzen agerpen kopurua testuan

Ariketa interesgarria da hurrengo hau: sarrera bezala testu bat jaso, eta testuko hitzak bistaratuko ditugu banan-banan, hitz bakoitzak testuan duen agerpen kopuruarekin batera. Programak komando-lerrotik argumentu bakarra jasoko du: fitxategi-izena. Eraitza fitxategiko hitz ezberdinen zerrenda izango da, ondoan bakoitzaren agerpen kopurua agertuko delarik.

Adibide moduan, hona `esaeraLabur.txt` fitxategiko hitzen agerpenak kalkulatzeko dituen deia eta programaren irteera:

```
>perl fitxHitzMaiztasun.pl esaeraLabur.txt
"Adiskidegabeko" hitzaren agerpen kopurua: 1
"Aldi" hitzaren agerpen kopurua: 1
"Amen" hitzaren agerpen kopurua: 1
...
"zuzentzen" hitzaren agerpen kopurua: 1
" " hitzaren agerpen kopurua: 1
>
```

Programa:

fitxHitzMaiztasun.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my ($fitx, $lerro, @hitzak, %Maiz);
$fitx = $ARGV[0];
open (FI, $fitx) ||
    die("Ezin $fitx fitxategia zabaldu!\n");

while ($lerro = <FI>) {
    chomp($lerro);
    # Banatu hitzak. Banatzaileak koma, puntu eta koma,
    # puntua, galdera- eta harridura-ikurrak eta zuriuneak dira
    @hitzak = split(/[\\,\\.\\s\\?!;]+/, $lerro);
    foreach my $hitz (@hitzak) {
        $Maiz{$hitz}++;
    }
}

# Hash-eko hitzak ordenatu eta agerpen-kopurua idatzi
foreach my $hitz (sort(keys(%Maiz))) {
    print "'$hitz' hitzaren agerpen kopurua: $Maiz{$hitz}\n";
}
print "\n";
```

Lehenbiziko lerroetan berrikuntzarik ez: fitxategia lerroz lerro irakurri, bukaerako lerro-jauzi karakterea kendu, `split()` funtzioarekin lerroa hitzetan banatu eta `@hitzak` *array*-an gorde. Ondoren, *array*-ko hitz bakoitza indibidualki tratatuko du programak `foreach` begizta erabiliz.

Berrikuntza hitz bakoitzaren prozesamenduan dator:

```
$Maiz{$hitz}++;
```

Sententzia bakar hori oso garrantzitsua da. Ez dugu aldagai eskalarrik erabili hitzen agerpen kopurua zenbatzeko. Izan ere, hitz ezberdin bakoitzeko aldagai berri bat beharko genuke. Egokiagoa da kasu honetan *hash* egitura erabiltzea, gakoak testuko hitzak eta balioak bakoitzaren agerpen kopurua izanik.

Honela interpretatuko du Perl-ek `$Maiz{$hitz}++;` sententzia (gogoratu `++` eragileak aurreko aldagaiari bat gehitzen diola): *hash*-aren `$hitz` gakoa berria bada, esleitu bat balioa. Gakoa aurretik existitzen bada, oster, bere aurreko

balioari bat gehitu. Sententzia bakar honekin, testuko hitz guztiak gordeko ditugu *hash* egituran, baita bakoitzaren agerpen kopurua ere.

Azkenik, emaitza bistaratzeko, `sort(keys(%Maiz))` funtzioen konbinazioak *hash*-eko gako guztiak itzuliko dizkigu ordenaturik *array* batean, eta `foreach` egiturak banan-banan hartu eta inprimatuko ditu pantailan.

Exekutatu programa eta proba egin fitxategi ezberdinekin. Berehala konturatuko zarete akats bat (gutxienez) badaukala: letra larriz eta xehez hasten diren hitzak bereizi egiten ditu. Zer egin horrelako bereizketarik egin ez dezan? Zure esku uzten dugu ariketa.

7.5. *each()*

Hash egitura baten elementu guztiak aztertzeko `each()` funtzioa ere erabil dezakegu. Funtzio honek argumentu gisa *hash* egitura jaso eta gako-balio bikotea itzultzen du bi elementuko *array* batean. Funtzioaren ondoz ondoko deiek *hash*-aren ondoz ondoko gako-balio bikoteak itzuliko dizkigute. Sarrera gehiagorik ez denean, zerrenda hutsa itzuliko digu funtzioak.

Praktikan, `keys()` funtzioa `foreach` begiztarekin erabiltzen den bezala, `each()` eta `while` elkarrekin erabili behar dira *hash* baten sarrera guztiak aztertzeko:

```
while (my($gako, $balio) = each(%hash)) {
    print("$gako => $balio\n");
}
```

Adibidean, `each()` funtzioak *%hash* egiturako gako-balio bikotea itzultzen du dei bakoitzean, bikotea `$gako` eta `$balio` aldagaietan gordez. Begiztaren iterazio bakoitzean, balio-bikote berri bat itzuliko digu funtzioak, harik eta *hash*-eko sarrerak amaitu eta zerrenda hutsa itzultzen duen arte. Programa begiztatik aterako da orduan.

Azken finean, aurretik `keys()` eta `foreach` erabilita egin dugun gauza berbera da. Baina kontuan izan `each()` funtzioa ezin dela `foreach` egiturarekin erabili.

Kapitulu honetan *hash* motako aldagaiak izan ditugu aztergai. Izenaren aurretik `%` ikurra daramate *hash* aldagaiak, eta sarrera bakoitza bi elementuz osatua dago: gakoa eta balioa. *Hash* egitura edo aldagai bat izanda, gako-balio bikoteak gehitzeko `"=>"` eragilea erabil dezakegu:

```
my %hash = ("gako1" => "balio1", "gako2" => balio2);
```

`%hash` egiturako `gako2` gakoaren balioa atzitu nahi izanez gero, giltza artean jarri behar dugu gako izena: `$hash{gako2}`. *Hash* egiturako elementuen artean ez dago ezarritako ordenarik, denak multzo berean daude. Elementu jakin bat atzitu ahal izateko bere gakoa erabili behar da.

Hash aldagaia oso erabilgarria da egitura berean elkarrekin erlazionatutako elementu-bikoteak gorde nahi ditugunerako. Adibidez: hitzak eta beren agerpen kopuruak testuan, hiztegiko sarrerek eta definizioak, NA eta izen-abizenak, etab.

8. Azpiprogramak, erreferentziak eta moduluak

8.1. AZPIPROGRAMAK

Programazioan esperientzia irabazi ahala, konturatuko gara zenbait prozesu behin eta berriz errepikatzen ditugula: testua hitzetan banatu, hitz bat jaso eta bere bokal kopurua zenbatu, etab. Behin eta berriz kode bera idazten aritu beharrean, badago modu bat errepikatzen den kode zatia programaren gainerako puskatik aparte jarri, eta noiznahi komando sinple baten bidez eskuragarri uzteko. Horixe da hain zuzen ere azpiprogramek egiten dutena.

Dagoeneko ikusi eta erabili ditugu Perl-ek dauzkan zenbait funtzio: `print()`, `sort()`, `chomp()`, etab. Bada, atal honetan geure neurriko funtzioak nola egin ikasiko dugu. Azpiprograma eta funtzio terminoak baliokideak izango dira guretzat, Perl-ek ez baitu beste lengoaia batzuek bezala beraien artean bereizketarik egiten.

Azpiprogramak definitu eta erabiltzeak bi abantaila nagusi dauzka: bata, kodea behin idatzi eta nahi adina aldiz erabil dezakegula ondoren gure programan. Kodeak aldagetak bat beharko balu, nahikoa litzateke azpiprograma definitu den lekuan aldatzearekin. Beste abantaila programatzeko metodologiarekin erlazionatua dago. Ataza orokorra azpiataza edo puska txikiagotan bana genezake, eta puska horietako bakoitza azpiprograma bidez ebatzi. «Zatitu eta garaitu» deitzen zaio problemak ebazteko estrategia honi, eta ataza konplexuak ebazteko oso erabilia da.

Datozen azpiataletan funtzioen ezaugarri garrantzitsuenak erakusten saiatuko gara, betiko moduan azalpenak eta adibideak tartekatuz.

8.1.1. Definizioa

Azpiprograma bat definitzeko, `sub` hitz erreserbatua idatzi behar da lehenbizi, segidan azpiprogramaren izena, eta ondoren giltza artean `{}` agindu-blokea, funtzioaren gorputza deiturikoa. Azpiprogramari izena jartzeko, letra, digitu eta azpimarra karaktereak erabil daitezke, baina lehen karakterea ezin da digitua izan (aldagaiekin bezala). Perl-eko hitz erreserbatuak ere (`while`, `print`, `for`...) ezin dira funtzio izen gisa erabili. Gomendio gisa esango dizuegu ohitura ona dela izenaren bidez azpiprogramak zer egiten duen deskribatzea, programaren irakurketa asko erraztuko baitu.

Adibide gisa, demagun garatzen ari garen programak behin eta berriz erabiltzen duela honako sententzia hau:

```
print("Idatzi duzun zenbakia ez da egokia.\n");
```

Lana aurrezteko, funtzio gisa definituko dugu:

```
sub ezegokia {
    print("Idatzi duzun zenbakia ez da egokia.\n");
}
```

Funtzioa programako edozein puntutan defini genezake; guk, hala ere, hasieran edo bukaeran egitea gomendatzen dizuegu, programa nagusitik bereizteko moduan.

8.1.2. Funtzio-deiak

Definitu berri dugun funtzioa beste edozein agindu gisa erabil dezakegu programan, bere beharra dugun bakoitzean `ezegokia()`; sententzia idatziz.

Ondoko programak zenbaki bat eskatzen dio erabiltzaileari, eta idatzitakoa `egokia` ez bada `ezegokia()` funtzioari dei egingo dio.

azpiprog.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;
print("Idatzi 1 eta 4 arteko zenbaki bat: ");
my $zenb = <STDIN>;
if ($zenb < 1){
    ezegokia();
}
elsif ($zenb > 4) {
    ezegokia();
}

# funtzioaren definizioa
sub ezegokia {
    print("Idatzi duzun zenbakia ez da egokia.\n");
}
```

Nahiz eta logikak agindu funtzioaren definizioak erabileraren aurretik joan beharko lukeela, adibideak erakusten duen moduan, ez da zertan horrela izan. Hau da, `ezegokia()` funtzioa programa-bukaeran definitu arren, programan nonahi erabil daiteke.

Ikusi berri dugun adibidea sinplea zen oso, erabilgarritasun handirik gabekoa.

Laster izango gara, ordea, azpiprograma konplexuagoak idazteko moduan. Baina aurretik, komeni zaigu azpiprogramen atalei erreparatzea.

8.1.3. Argumentuak

Azpiprograma edo funtzioari kanpotik datuak pasatu nahi dizkiogunean erabiliko ditugu argumentuak. Funtzio-deia egitean parentesi artean idazten dira eta Perl-ek automatikoki `@_ array` berezian gordeko ditu idatzitako argumentu guztiak. Azpiprogramak datuok erabili behar dituenean `@_` bektorean izango ditu eskuragarri.

Ondoko programak `handiena()` azpiprograma erabiltzen du, zeinak argumentu gisa bi zenbaki jaso eta haien arteko `handiena` zein den erabakitzen duen.

azpHandiena.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;

# funtzio deiak
handiena(3, 7);
handiena(13, 5);
handiena(12, 12);

# definizioa
sub handiena {
    # argumentuak gorde aldagaietan
    my ($zenb1, $zenb2) = @_;
    if ($zenb1 > $zenb2) {
        print("$zenb1 handiagoa $zenb2 baino\n");
    }
    elsif ($zenb2 > $zenb1) {
        print("$zenb2 handiagoa $zenb1 baino\n");
    }
    else {
        print("Biak berdinak dira!\n");
    }
}
```

Adibidean ikusten den bezala, funtzioari dei egitean argumentuak parentesi artean eta komaz bananduak idatzi behar dira. Azpiprogramari dei egiten zaion bakoitzean, Perl-ek automatikoki `@_` aldagai berezian gordetzen ditu parentesi artean idatzitako argumentuak, azpiprogramak hortik har ditzan. Gure kasuan, lehen argumentua `$_[0]` posizioan izango dugu eta bigarrena, berriz, `$_[1]`-en. Programan, `my ($zenb1, $zenb2) = @_;` sententzia erabiliz bi argumentuak pauso bakarrean `$zenb1` eta `$zenb2` aldagaietan gordeko ditugu.

Array berezia da `@_`, **aldagai anonimo** deiturikoa, eta horrek esan nahi du zuzenean erreferentziatu gabe ere erabil dezakegula azpiprograman:

```
my $zenb1 = shift();
my $zenb2 = shift();
```

`shift()` funtzioari argumenturik pasatu ez diogunez, Perl-ek `@_` aldagai anonimoa hartuko du argumentu gisa, `$zenb1` aldagaian lehen elementua utziz eta `$zenb2` aldagaian bigarrena. Oso ohikoa da `@_ array`-a modu anonimoan erabiltzea.

8.1.4. Funtzioaren gorputza

Funtzioaren gorputza giltza artean `{ }` idatzitako blokea da, kalkuluak egiten dituen zatia. Programa batean egin dezakegun edozer egin dezakegu hemen ere: eragiketa matematikoak, *string*-ak prozesatu adierazpen erregularrekin, beste funtzio bati deitu, etab.

8.1.5. return agindua

Funtzio batzuek emaitza itzuli ohi dute. Adibidez, `length()` funtzioak argumentu gisa pasatutako *string*-aren karaktere kopurua itzultzen du, eta `sort()`-ek, aldiz, *array*-a alfabetikoki ordenatuta.

Guk definitutako azpiprogramak emaitza itzultzea nahi dugunean, `return` sententzia erabiliko dugu. Adibide gisa aztertu ondoko programa, zeinak `bokalKop()` funtzioa definitu eta erabiltzen duen argumentu gisa pasatutako *string*-aren bokal kopurua ezagutzeko.

azpBokalak.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;

print("Idatzi esaldi bat eta amaitzeko sakatu return\n");
my $testua = <STDIN>;
chomp($testua);
# funtzio deia. Emaitza $kop-en gorde
my $kop = bokalKop($testua);
print("Esaldiak $kop bokal ditu\n");

# Azpiprogramaren definizioa
sub bokalKop {
    my $esaldi = shift();
    my $bokalak = ($esaldi =~ tr/aeiou//);
    return $bokalak;
}
```

Azpiprogramak 3 pausotan gauzatzen du bere egitekoa: argumentu gisa pasatutako *string*-a \$esaldi aldagaian gordetzen du lehenbizi; ondoren, `tr///` funtzioaren bitartez bokal kopurua kalkulatu eta \$bokalak aldagaian uzten du (`tr///` funtzioak bigarren argumenturik ez duenez ez du ordezkapenik gauzatuko, baina itzulera-balio gisa bokalen agerpen kopurua itzuliko du). Amaieran, `return $bokalak`; sententziaren bidez emaitza bidaltzen dio programa nagusiarri.

Azpiprograma berean `return` sententzia bat baino gehiago jar ditzakegu. Kasu horietan `return` bakoitzak funtzioaren emaitzaren mota bereko adierazpena itzuli behar du. Hau da, ezin du `return` batek zenbakia itzuli besteak *string*-a itzultzen duen bitartean.

Funtzioak emaitza itzuliko badu, beharrezkoa da mementoren batean `return` sententzia exekututzea. Funtzioaren emaitza zehazteko `return` sententziaren ondoren dagoen adierazpena ebaluatuko da, eta emaitza itzuliz amaituko azpiprogramaren exekuzioa.

Zenbait azpiprogramak, bestalde, ez dute inongo baliorik itzuliko, eta betebeharrak jakin bat beteko dute soilik. 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 prozeduren eta funtzioen artean desberdintzeko inongo sintaxi berezirik ezartzen. Azpiprogramaren semantikak esango digu funtzioa edo prozedura den.

Ondoko programak erabiltzaileari zenbaki bat eskatu, *N*, eta pantailan *N* izartxo bistaratuko ditu. Horretarako, `izartxoakIdatzi()` azpiprograma erabiliko du, *N* zenbaki bat jaso eta *N* izartxo bistaratzen dituen.

izartxoak.pl

```
#!/usr/bin/perl
use warnings;
use strict;

print("Idatzi zenbaki bat: ");
my $zenb = <STDIN>;
chomp($zenb);
izartxoakIdatzi($zenb); # azpiprogramari deia

sub izartxoakIdatzi { # azpiprogramaren definizioa
    my ($izar_Kop) = @_ ;
    my $kont = 0; # Kontrol-aldagaia
    while ($kont < $izar_Kop) {
        print "*";
        $kont++;
    }
}
```

`izartxoakIdatzi()` azpiprogramak ez du `return` bidez baliorik itzultzen. Zuzenean, pantailan `N` izartxo bistaratzen dituen *prozedura* da.

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

Aztertu hurrengo kode zatia:

```
# programa nagusia
my $i = 5;
funtzioTxoro();
funtzioTxoro();
print($i);

# azpiprograma
sub funtzioTxoro {
    $i++;
}
```

Zure ustez, zein balio bistaratu behar luke programa nagusiko `print($i);` sententziak?



.....

Tranparik gabeko galdera zen, zazpi pantailaratuko luke. Azpiprogramak arazorik gabe atzitu eta alda ditzake programa nagusiko aldagaiak, goiko adibidean bezala. Tentuz ibili honekin, nahaste eta buruhauste asko eragin ditzake eta. Ahal dela azpiprogramako eta programa nagusiko aldagaiak ez ditugu nahastu behar. Horretarako ez dugu aparteko lanik hartu beharrik, nahikoa da orain artean bezala aldagai bat erabili aurretik `my` gako-hitzarekin definitzea.

Hona hemen aurreko azpiprograma baina zertxobait aldatuta:

```
sub funtzioTxoro {
    my $i++;
}
```

Zein izango da orain `print($i);` sententziak bistaratuko duen balioa?

Bost. Programa nagusiko `$i` aldagaia eta azpiprogramakoa ezberdinak direlako, nahiz eta izen berdinak eduki. Azpiprogramak `my` gako-hitza erabiliz definitu dituen aldagaiak ez daude atzigarri azpiprogramatik kanpo.

8.2. ERREFERENTZIAK

Hiru dira Perl-ek ezagutzen dituen aldagai motak: eskalarrak, *array*-ak eta *hash*-ak. Hiruren artean oinarritzkoena eskalarra da, eta beste biak, *array* eta *hash*-ak, osagai eskalarrez osatuta daude. Orain artean gure zereginetarako ondo moldatu gara hiru hauekin. Baina zenbat eta gehiago programatu, orduan eta ataza konplexuagoetan murgilduko gara, eta zenbaitetan Perl lengoaiaren mugekin tope egin ere bai. Oso urrutira jo gabe, hona hemen horren adibide bat: sarrera gisa fitxategi bat eman, eta fitxategi horretako hitzak karaktere kopuruaren arabera sailkatuko dituen programa nahi dugu. Programaren irteerak honelako zerbait behar luke:

```
2 karaktereko hitzak: ez,bi,ni,da
3 karaktereko hitzak: hor,eta,hau,zer
4 karaktereko hitzak: etxe,laua,hiru,maiz
...
```

Ataza ebazteko modurik zuzenena *hash* egitura erabiltzea litzateke, non gakoa karaktere kopurua litzatekeen, eta balioa, berriz, karaktere kopuru horretako hitzen zerrenda. Programak, ohiko egiturak erabiliz, fitxategia lerroz lerro irakurri eta hitzetan banatuko luke. Hitz bakoitzaren karaktere kopurua zenbatu, eta gako gisa erabiliz, hitz berria kopuru horretako hitzen zerrendari gehituko lioke. Karaktere kopuru horretako sarrerarik ez balego *hash*-ean, sarrera berria definituko litzateke. Fitxategiko hitz guztiak irakurri ostean, *hash*-eko elementu guztiak bistaratuko genituzke ordenan.

Lehen begiratuan egokia dirudien arren gure hurbilpenak, badu akats bat: *hash*-aren balio gisa ezin dugu *array* bat erabili. Gako eta balioak eskalar motakoak izan behar dira nahitaez. Beraz, karaktere kopuru jakin bateko hitz guztiak eskalar motako *string* bakarrean kateatu beharko ditugu. Honek —hitz-zerrenda osoa *string* bakarrean izateak— asko zailtzen du ondoren manipulatu ahal izatea; adibidez, bukaeran gako bakoitzaren hitz kopurua jakin nahiko bagenu, hitz guztien *string*-a zerrenda bihurtu beharko genuke lehenbizi, hitzak zenbatu, eta ondoren berriro *string* bakarrean kateatu hitz guztiak. Prozesua asko konplikatu zaigu eta aski nahasgarria da edozeinentzat.

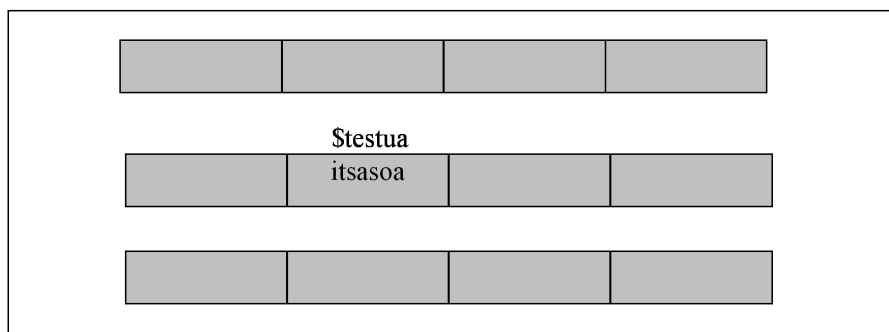
Datu-egitura konplexuagoak eraiki nahi izatean, kasu honetan bezala, Perl lengoaiaren mugekin topo egin dezakegu. Ez da posible *hash* baten barruan *array* bat erabiltzea edo *array*-ez osatutako *array* bat eraikitzea. Ez da posible zuzenean, baina badago modu bat egitura horiek eraiki eta maneiatzeko: erreferentziak erabiliz.

Aurrera segi eta erreferentziekin buru-belarri murgildu baino lehen, geldialdi bat egingo dugu gure bidean. Izan ere, erreferentziak behar bezala ulertu ahal izateko, aldagaien balioak memorian nola gordetzen diren ezagutzea komeni zaigu lehenbizi. Jar dezagun adibide erraz bat:

```
my $testua = "itsasoa";
```

Aurreko sententzia irakurtzean, Perl-ek memoriako posizio bat esleitzen dio `$testua` aldagaiari, eta posizio horretan gorde «itsasoa» testua.

Memoria



Memorian zehazki non dagoen jakiteko helbide bat izango du `$testua` aldagaiak. Aurreko sententziaren atzetik, berri hau idatziko bagenu:

```
testua = "hondartza";
```

Esleipen berriak `$testua` aldagaiaren memoria helbidean zegoena ezabatuko luke eta "hondartza" idatzi haren lekuan.

`$testua`

hondartza

Azalpen labur honen ondoren, itzul gaitezen abiapuntura erreferentziak zer diren ikusteko. Perl-en, erreferentzia memoriako posizio jakin bat seinalatzen duen balio eskalarra besterik ez da. Aldagai bat memorian non dagoen identifikatzeko balio digu. Aldagai hori eskalarra, *array*-a edo *hash* motakoa izan daiteke, berdin digu, erreferentziak memorian non dagoen besterik ez digu adieraziko. Konparazio batera, erreferentzia *ilargia seinalatzen duen atzamarra* dela esan dezakegu. Gure kasuan, atzamarra memorian dagoen aldagai bat seinalatuko du. Baina kontuz! Ez gero nahastu atzamarra eta ilargia. Erreferentzia eta honek seinalatzen duen aldagaia bi gauza erabat ezberdin dira.

Aldagai baten erreferentzia sortzeko, nahikoa da aldagaiaren aurretik "\" era-

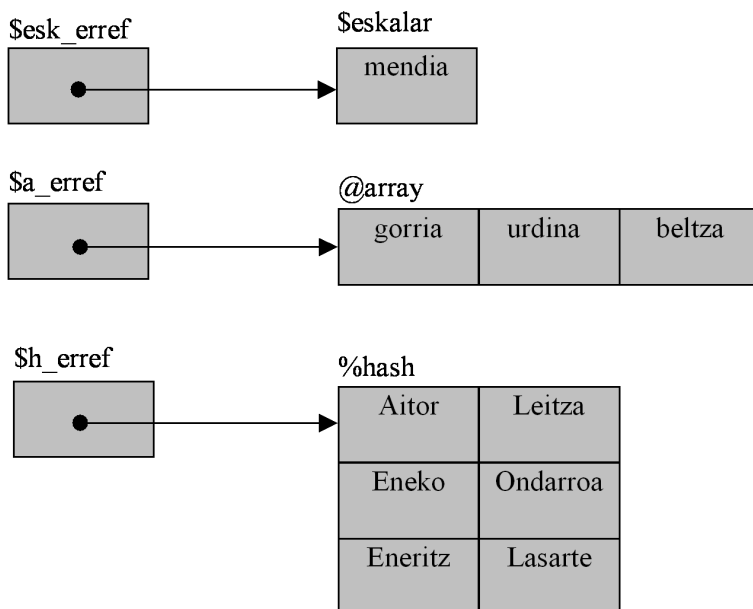
gilea jartzea. Berdin digu zer-nolako aldagaia den, bere erreferentzia eskalar mota-koa izango da beti. Hurrengo adibideak erakusten du nola sortu aldagai mota ezberdinen erreferentziak:

```
my $eskalar = "mendia";
my $esk_erref = \ $eskalar;    # $eskalar aldagaiaren erreferentzia
                                # $esk_erref aldagaian gorde

my @array = ("gorria", "urdina", "beltza");
my $a_erref = \@array;         # @array aldagaiaren erreferentzia
                                # $a_erref-en gorde

my %hash = ( "Aitor" => "Leitza",
             "Eneko" => "Ondarroa",
             "Eneritz" => "Lasarte");
my $h_erref = \%hash;          # %hash aldagaiaren erreferentzia
                                # $h_erref-en gorde
```

Azpiko irudiak deskribatzen ditu adibideko erreferentzien eta hauek seinaltatzen dituzten aldagaien arteko loturak.



Erreferentziak bi eginkizunetarako erabiliko ditugu: batetik datu mota konplexuak edo habiaratuak osatzeko, eta bestetik azpiprogramei argumentu gisa *array* eta *hash* egiturak pasatu ahal izateko.

Aurreko adibideak erakutsi bezala, eskalar, *array* eta *hash*-en erreferentziak sor ditzakegu. Horietatik, lehenengoak ez du ia erabilera praktikorik; bai, ordea, bigarren eta hirugarrenak. Hori horrela izanik, ondoko azpiataletan azken biek hasiko gara, eta eskalarren erreferentziak azkenerako utziko ditugu.

8.2.1. Array-en erreferentziak

Azter ditzagun honako kode-lerro hauek:

```
my @A = (0,0);    # Array bat
my $aRef;         # Eskalar baten erazagupena
$aRef = \@A;      # $aRef @A array-aren erref. da

$a[0] = 1;        # @A == (1,0);
$aRef->[1] = 2;    # @A == (1,2)
push (@{$aRef}, 3); # @A == (1,2,3)
```

Lehenengo bi lerroetan bi aldagai definitu ditugu, @A bektorea eta \$aRef eskalarra. Azter dezagun hurrengo agindua:

```
$aRef = \@A;      # $aRef @A bektorearen erref. Da
```

Hemen, \$aRef aldagaiari @A bektorearen *helbidea* esleitzen diogu "\" eragilearen bidez. Hortik aurrera, \$aRef aldagaiak @A bektorea erreferentziatuko du, alegia, \$aRef aldagaia @A array-ko osagaiak atzitzeko (eta aldatzeko) erabili ahal izango da. Hurrengo bi aginduek @A array-ko osagaien balioak ezartzen dituzte, bata *array*-a zuzenean atzitzuz, eta besteak \$aRef erreferentziaren bidez:

```
$a[0] = 1;        # @A == (1,0);
$aRef->[1] = 2;    # @A == (1,2)
```

Ohar zaitezte sintaxi-aldaketan: \$aRef bidez @A bektoreko osagaiak atzitzeko, “->” eragilea behar da.

Programak, azkenik, @A bektoreari elementu berri bat esleitzen dio *push* funtzioarekin, berriro ere \$aRef erreferentzia erabiliz.

```
push (@{$aRef}, 3); # @A == (1,2,3)
```

push funtzioaren lehenengo parametroak *array* bat izan behar duenez, eta ez *array* baten erreferentzia, *array* osoa eskuratu behar da \$aRef erreferentziatik abiatuz. Horretarako, @{\$aRef} forma erabiltzen da.

Bestalde, erreferentziek ez dute jadanik existitzen den bektore bat erreferentziatu behar. Izan ere, erreferentzien bidez izengabeko *array*-ak ere sor baitaitezke:


```

my $aRef = [];          # $aRef erreferentziak izengabeko bektore
                        # bat erreferentziazten du
push (@{$aRef},1);      # @{$aRef} == (1)
$aRef->[1] = 2;          # @{$aRef} == (1,2)

```

Laburbilduz:

Adibidea	Azalpena
<code>\$erref = \@Array</code>	Erreferentziari bektore baten helbidea eman
<code>\$erref = []</code>	Erreferentziari izengabeko bektore baten helbidea eman
<code>\$erref->[i]</code>	<code>\$erref</code> bidez erreferentziaztatutako bektorearen <code>i+1</code> -garren elementua atzitu
<code>@{\$erref}</code>	<code>\$erref</code> bidez erreferentziaztatutako <i>array</i> -a

8.2.2. Hash-en erreferentziak

Bektoreak bezalaxe, *hash*-ak ere erreferentzia daitezke. Ikus dezagun adibide bat:

```

my %H = (izena => "Nekane",
         adina => 33);
my $hRef = \%H;          # $hRef %H hash-aren erreferentzia da

$hRef->{izena} = $hRef->{izena}. " Agirre";
$hRef->{adina}++;
# Orain %H == (izena => "Nekane Agirre",
#             adina => 34);
my @hGakoak = keys(%{$hRef});
# @hGakoak == ("izena", "adina")

```

Lehenengo lerroan `%H` *hash*-a definitzen dugu. Bigarrenean, berriz, `$hRef` eskalarra definitu eta `%H` *hash*-aren helbidea esleitzen diogu, berriro ere, `"\"` eragilearen bidez. Hortik aurrera, `$hRef` aldagai eskalarrean `%H` *hash*-aren erreferentzia izango dugu. Hurrengo lerroek, `%H` *hash*-aren edukia aldatuko dute `$hRef` erreferentzia erabiliz:

```

$hRef->{izena} = $hRef->{izena}." Agirre";
$hRef->{adina}++;

```

Lehenengo lerroak, `"izena"` gakoari dagokion balioari `" Agirre"` karaktere-segida eranstean dio kateaketa eragilearen bidez `(".")`. Hurrengo aginduk, berriz, `"adina"` gakoaren balioari bat gehitzen dio.

Azken aginduak, %H *hash*-aren gakoak bektore batean gordetzen ditu \$hRef erreferentziatik abiatuta:

```
my @hGakoak = keys(%{$hRef});
```

Array-ekin bezala, izengabeko *hash*-ak ere sor daitezke:

```
my $hRef = {};      # $hRef izengabeko hash baten erref. da
$hRef->{izena} = "Koldo";
$hRef->{adina} = 36;
```

Laburbilduz:

Adibidea	Azalpena
\$erref = \%Hash	Erreferentziari <i>hash</i> baten helbidea eman
\$erref = {}	Erreferentziari izengabeko <i>hash</i> baten helbidea eman
f->{gako} = balio \$erref	\$erref bidez erreferentziatutako <i>hash</i> -aren "gako" gakoari "balio" balioa erantsi.
%{\$erref}	\$erref bidez erreferentziatutako <i>hash</i> -a

8.2.3. Eskalarren erreferentziak

Aldagai eskalarrek ere erreferentziatu ditzakegu, honako eran:

```
my $ald = "Kaixo";
my $sRef = \$ald;
${$sRef} .= " mundua !";
# orain $ald == "Kaixo mundua!"
```

Aurreko adibideetako pauso berei jarraituz, lehendabizi \$ald aldagai eskalarra definitu dugu eta ondoren bere erreferentzia sortu, "\" eragilearen bidez. Hortik aurrera, \$sRef erreferentziak \$ald aldagaia erreferentziatuko du. Jatorrizko aldagaia erreferentziaren bidez aldatzeko, \${\$sRef} eragilea erabili behar da.

Laburbilduz:

Adibidea	Azalpena
\$erref = \\$scalar	Erreferentziari eskalar baten helbidea eman
\${\$erref}	\$erref bidez erreferentziatutako eskalarra

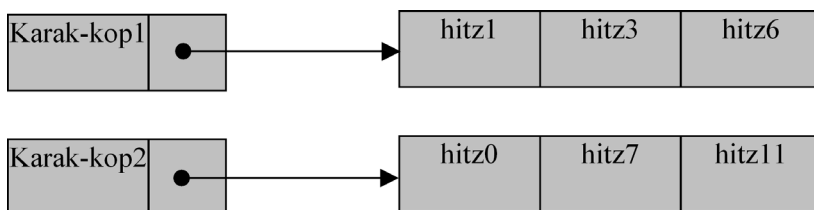
8.2.4. Datu-egitura habiaratuak

Aldagai-klase bakoitzaren erreferentzia nola sortu eta erabili ikusi ostean, orain, zertarako erabil ditzakegun azaltzeko unea iritsi zaigu. Atal honen hasieran jarritako adibideari helduko diogu ostera:

26. kasu praktikoa: sailkatu hitzak luzeraren arabera

Sarrera-datu gisa testu bat jaso eta bertako hitzak karaktere kopuruaren arabera sailkatuko dituen programa nahi dugu.

Erreferentziak ikusi ondoren, erabaki dugu datuak gordetzeko *hash* egitura erabiltzea. Honela: gako bezala karaktere kopurua, eta bere balioa *array* baterako erreferentzia. Erreferentziatutako *array* honek gakoan adierazitako karaktere kopurua duten hitzen zerrenda gordeko du.



Aipatutako datu-egitura erabiliz, hona idatzi dugun programa. Baina kodeari begiratu baino lehen, saia zaitez zeure kasa problema ebazten, eta konparatu ondoren guk proposatutakoarekin.



.....

karKopFitx.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my ($fitx, $lerro, @hitzak, $karKop, %Zerrenda);
$fitx = $ARGV[0];
open (FI, $fitx) ||
    die("Ezin $fitx fitxategia zabaldu!\n");

while ($lerro = <FI>) {
    chomp($lerro);
    # Banatu lerroa hitzetan.
    # Banatzaileak: koma, puntu eta koma, puntua,
    # galdera eta haridura ikurrak eta zuriuneak
```

```

@hitzak = split(/[\\.\s\?!;]+/, $lerro);
# Lerroko hitz bakoitzeko
foreach my $hitz (@hitzak) {
    # Hitzaren karaktere kopurua gorde
    $karKop = length($hitz);
    # %Zerrenda hash-ean $karKop gakoak existitzen
    # ez bada, sortu gako horrentzako
    # izengabeko array baten erref.
    $Zerrenda{$karKop} = []
        if (!exists($Zerrenda{$karKop}));
    # Gehitu hitza erreferentziatutako array-aren
    # amaieran
    push(@{$Zerrenda{$karKop}}, $hitz);
}# foreach
}# while

# %Zerrenda egiturako gakoak ordenatuta lortu
# eta begizta bidez banan-banan pantailaratu
foreach my $luz (sort({$a <=> $b} keys(%Zerrenda))) {
    print("$luz karakteretako hitzak: ");
    # $luz luzerako hitzen array-a atzitu eta @hitzak
    # array-an gorde
    @hitzak = @{$Zerrenda{$luz}};
    # @hitzak ordenatu, string bakarrean bildu eta
    # pantailan bistaratu
    print(join(" ", sort(@hitzak)));
    print("\n");
}

```

Azter dezagun patxadaz. Programak bi zati nagusi dauzka:

- 1.- **while** begizta. Fitxategia lerroz lerro irakurri eta datuak *hash* egituran gordetzen ditu.
- 2.- **foreach** begizta. *Hash* egituran gordetako datuak ordenatu eta pantailan bistaratzen ditu.

Lehen zatian, fitxategiko lerro bakoitza irakurri eta hitzetan banatuko du programak `split` funtzioa erabiliz. Hitz bakoitzeko, karaktere kopurua kalkulatu (`$karKop`) eta kopuru horretako gakorik baden begiratu du `%Zerrenda` hash egituran. Ez badago, izengabeko *array* huts bat sortu, eta bere erreferentzia gordeko du gako berriaren balio gisa. Azken eragiketa, sententzia honek egiten du:

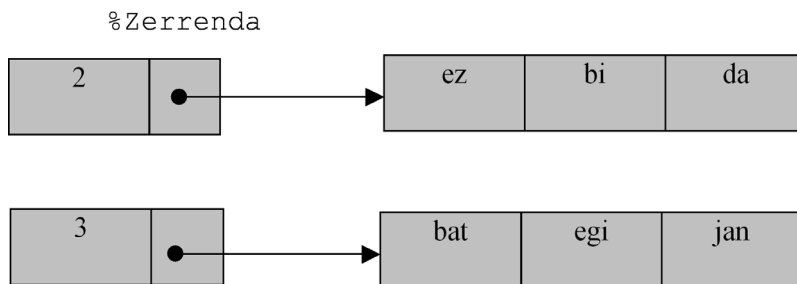
```
$Zerrenda{$karKop} = [] if (!exists($Zerrenda{$karKop}));
```

Hurrengo sententziak, `$hitz` hitza hartu eta bere karaktere kopuruari dagokion *array*-an gehitzen du `push` funtzioarekin.

```
push(@{$Zerrenda{$karKop}}, $hitz);
```

Funtzio honek *array* bat behar du lehen parametro gisa, eta `$karKop` gakotik abiatuta, `$Zerrenda{$karKop}` luzera horretako hitzen *array* erreferentzia da, eta `@{$Zerrenda{$karKop}}`, berriez, *array* osoa.

Programa begiztatik ateratzean, `%Zerrenda` egitura hauxe izango dugu: gako gisa karaktere kopuruak, eta haren balio gisa hitzez osatutako *array*-etarako erreferentziak. Honako irudi honek erakusten duen moduan:



Programaren bigarren zatia, lehenbizikoak elikatutako `%Zerrenda` datu-egitura bistaratzeaz arduratzen da. Hash egiturekin egin ohi dugun moduan, bere gakoak lortu, ordenatu (gogoratu zenbakizko zerrendak ordenatzeko `{ $a <=> $b }` jarri behar dela `sort` funtzioan), eta banan-banan bisitatuko ditugu `foreach` egitura erabiliz. `$luz` gakoa izanda *hash*-aren balioa lortzeko `$Zerrenda{$luz}` idaztea besterik ez dugu. Baina balioak *array* erreferentziak dira, eta beraz, *array* osoa lortzeko `@{$Zerrenda{$luz}}` idatzi behar da. Ondoko sententziak `$luz` gakoari dagokion *array*-a atzitu eta `@hitzak` *array*-an gordetzen du:

```
@hitzak = @{$Zerrenda{$luz}};
```

Amaitzeko, hitzen *array*-a ordenatu, *string* bakarrean bildu `join`-ekin, eta pantailan bistaratuko du programak.

8.2.5. Parametroak erreferentziaz

Datu-egitura konplexuak sortzeko baliagarri izateaz gain, erreferentziek badute beste zeregin garrantzitsu bat ere: azpiprogramei argumentu gisa *array* eta *hash* egiturak pasatu ahal izateko beharrezko dira. Dakigun bezala, azpiprogramei pasatutako argumentu guztiak `@_array` berezian biltzen ditu Perl-ek. Hori dela-eta, ez da posible argumentu bezala bi *array* pasatu eta beraien jatorrizko identitatea mantentzerik.

Hona hemen adibide bat azaldutakoaren argigarri:

mendiHerri.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @mendiak = ("Anboto", "Gorbeia", "Aizkorri");
my @herriak = ("Bastida", "Lizarra", "Elgoibar");

# Deia azpiprogramari
mendiak_eta_herriak(@mendiak, @herriak);

# Azpiprogramaren definizioa
sub mendiak_eta_herriak {
    my (@mendiak, @herriak) = @_;

    foreach my $mendi (@mendiak) {
        print("mendia:\t$mendi\n");
    }
    foreach my $herri (@herriak) {
        print("herria:\t$herri\n");
    }
}
```

Aztertu arretaz kodea eta ahalegindu programaren portaera aurreikusten. Garbi ikustean, idatzi kodea editorea erabiliz eta exekutatu.

Uste zenuen portaera izan al du?

```
>perl mendiHerri.pl
mendia:    Anboto
mendia:    Gorbeia
mendia:    Aizkorri
mendia:    Bastida
mendia:    Lizarra
mendia:    Elgoibar
>
```

Argumentu gisa pasatutako bi *array*-ak "mendi" gisa bistaratu ditu programak; izan ere, lehen aipatu bezala, azpiprogramaren argumentu guztiak *@_array*-an biltzen ditu Perl-ek baina gero ez da gauza azpiprograma barruan berriro banatzeko sententzia hau irakurtzean:

```
my (@mendiak, @herriak) = @_;
```

Perl-ek ez daki non amaitzen den `@mendiak` *array*-a eta non hasten den `@herriak`, eta horregatik esleitzen dio `@_` *array* osoa `@mendiak` *array*-ari.

Array bakoitza bere aldetik nahi/behar badugu, erreferentziak erabiltzea beste aukerarik ez zaigu gelditzen. Aurreko programa berridatziko dugu, oraingoan baina argumentuak erreferentzia bidez jasotzen dituen azpiprograma idatziz.

mendiHerri2.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @mendiak = ("Anboto", "Gorbeia", "Aizkorri");
my @herriak = ("Bastida", "Lizarra", "Elgoibar");
# Deia azpiprogramari, argumentuak erreferentziak
erref_mendiak_eta_herriak(\@mendiak, \@herriak);

# Azpiprogramaren definizioa
sub erref_mendiak_eta_herriak {
    my ($mendiak, $herriak) = @_;

    foreach my $mendi (@$mendiak) {
        print("mendia:\t$mendi\n");
    }
    foreach my $herri (@$herriak) {
        print("herria:\t$herri\n");
    }
}
```

Bigarren bertsioa exekutatuz gero:

```
>perl mendiHerri2.pl
mendia:    Anboto
mendia:    Gorbeia
mendia:    Aizkorri
herria:    Bastida
herria:    Lizarra
herria:    Elgoibar
>
```

Oraingoan herriak eta mendiak bereizi ditu, guk nahi bezala. Kontuan izan behar da argumentuak erreferentzia bidez pasatuz gero, azpiprogramak erreferentziatutako datu-egituraren gaineko kontrol osoa izango duela. Hau da, datu-egituretan egindako edozein aldaketak hortxe jarraituko du azpiprograma exekutatatu ondoren ere.

8.3. MODULUAK

Azpiprogramak erabiliz programazio-lanak erraztu egiten dira. Izan ere, zenbait ataza espezifikoren kodea funtzio edo prozedura desberdinetan «estali» daiteke, arestian aipatu dugun legez. Funtzio edo prozedurak erabiltzeak bi abantaila nagusi ditu programazio-lanetan dihardugunean, eta bereziki, programa handi eta konplexuak egin nahi ditugunean:

- Errepikatzen diren kode-lerroak azpiprograma batean koka ditzakegu. Horrela, bada, kode-lerro horiek ez ditugu errepikatu behar izango programa nagusian behin eta berriz. Horrek 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 txikiagotan banatzen lagunduko digute, horrelako zati bakoitzaren kodeak azpiprograma batean joan beharko lukeelarik.

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 izaten duten prozeduren eta datu-egituren multzoak dira, edozein programatzailek erabili ahal izateko diseinatuak. Hau da, azpiprograma edo kode zati jakin bat zenbait programak erabili nahi dutenean, guztientzat eskuragarri dagoen leku jakin batean gordetzea. Moduluetan alegia.

Perl milaka moduluz hornitua dago, ia edozein zeregin betetzeko balioko dutenak. Nahi dugula Perl-en bidez posta elektronikoak automatikoki bidali? Badugu modulu bat hori egiteko. Web zerbitzari batera konektatu behar dugula hango orriak deskargatzeko? Baita. Testuen prozesamendurako moduluak ere, ezin konta ahala daude.

Moduluak Interneten atzigarriak dira, CPAN deituriko (www.cpan.org) moduluen biltegi erraldoian. Bertan 15.000 modulu inguru dago, beste horrenbeste eginkizun gauzatzeko eginak. Gainera, bertako moduluak Perl-eko adituak garatuak dira maiz, eta beraz, moduluak erabiliz mundu mailako programatzaile azkarren lanaz —moduluen sortzaileak, hain zuzen— baliatuko gara.

Atal honetan, bada, moduluen erabilera aztertuko dugu. Lehendabizi, gure instalazioan modulu zehatz bat dugun ala ez jakiteko zenbait komando ikusiko ditugu, eta, ondoren, modulu berriak nola instalatu (bai Windows-en bai Linux-en). Gero, modulua bera erabiltzen ikasiko dugu, aurretik moduluaren dokumentazioa nola arakutzen den erakutsiz.

Horretarako, adibide batean oinarrituko gara. Adibideak fitxategi bateko Ngramak kontatuko ditu. Baina zer dira Ngramak? Zertarako balio dezakete?

8.3.1. Ngramak kontatzen

Sekuentzia bat izanik —adibidez, esaldi bat—, ondoz ondoko N elementuz osatutako azpisekuentziei Ngramak deitu ohi zaie. Elementu kopuruaren arabera (N), Ngramek izen desberdinak hartzen dituzte: elementu bakarraz osatutako Ngramei unigrama deritze; 2 elementuz osatutako Ngramei, berriz, bigrama, eta 3 elementuz osatutakoei, trigramak. Hortik aurrera, 4-gramak, 5-gramak eta abar izendatu ohi dira.

Adibidez, "Etxekoak" karaktere-sekuentzia izanik, 6garren mailarainoko Ngramak osa daitezke:

Unigramak	Bigramak	Trigramak	4-gramak	5-gramak	6-gramak
E	Et	Etx	Etxe	Etxek	Etxeko
t	tx	txe	txek	txeko	txekoa
x	xe	eko	xeko	xekoa	xekoak
e	ek	koa	ekoa	ekoak	
k	ko	oak	koak		
o	oa				
a	ak				
k					

Taula honetan "Etxekoak" hitzaren 6garren maila arteko Ngramak ikus daitezke.

Ngramen kalkulua oso erabilgarria da hizkuntzaren tratamendu automatikoko hainbat eginkizunetan. Esaterako, Ngramak erabil daitezke euskarazko hitzen aurizki zein atzizki usuenak identifikatzeko. Izan ere, testu handi batean atzizki zein aurizkiak diren letra-sekuentzien maiztasuna altuagoa da, ausaz osatutako letra-sekuentzien maiztasunarekin alderatuta. Alegia, "eko", "ren" edo "tik" atzizkien maiztasuna handia izango da, "tre" edo "jiu" ausaz osatutako sekuentzien maiztasunarekin konparatuta. Horrela, testu baten gainean Ngrama guztiak zenbatzen baditugu, oso litekeena da usuen azaltzen direnak benetako aurizki edo atzizkiak izatea. Egin dezagun froga!

Jo dezagun fitxategi bat eskatu eta bertan azaltzen diren trigrama guztiak kalkulatu nahi ditugula, eta maizen agertzen direnak pantailan idatzi. Esan bezala, oso litekeena da maizen agertutako sekuentziak aurrezki zein atzizkiak izatea.

Trigramak kalkulatzeko dituen programa asmatzea neketsua izan daiteke. Zorionez, Perl-ek badu Ngramak kontatzeko modulu berezi bat, `Text::Ngrams` deiturikoa⁹. Hortaz, modulu horretaz baliatuko gara gure eginkizuna betetzeko.

9. Testu hau idaztean bat ez, 3 dira CPAN biltegiak eskaintzen dituen moduluak Ngramak osatzeko: `Text::Ngram`, `Text::Ngrams` eta `Algorithm::Ngram`.

8.3.2. Moduluak instalatzen

Lehenik eta behin, jakin behar dugu `Text::Ngrams` modulua jadanik instalatua dagoen ala ez. Izan ere, Perl banaketetan hainbat modulu instalatuta datoz, eta litekeena da `Text::Ngrams` horien artean egotea. Hori jakiteko, komando-lerroko pantailan gaudela honako hau idatzi behar da:

```
> perl -e "use Text::Ngrams"
```

Aurreko komandoa exekutatzean pantailan ezer agertzen **ez** bada, `Text::Ngrams` modulua instalatuta dugun seinale izango da. Aitzitik, errore-mezu hau azaltzen bazaigu:

```
Can't locate Text\Ngrams.pm in @INC (@INC contains:...) at
-e line 1.
```

```
BEGIN failed--compilation aborted at -e line 1.
```

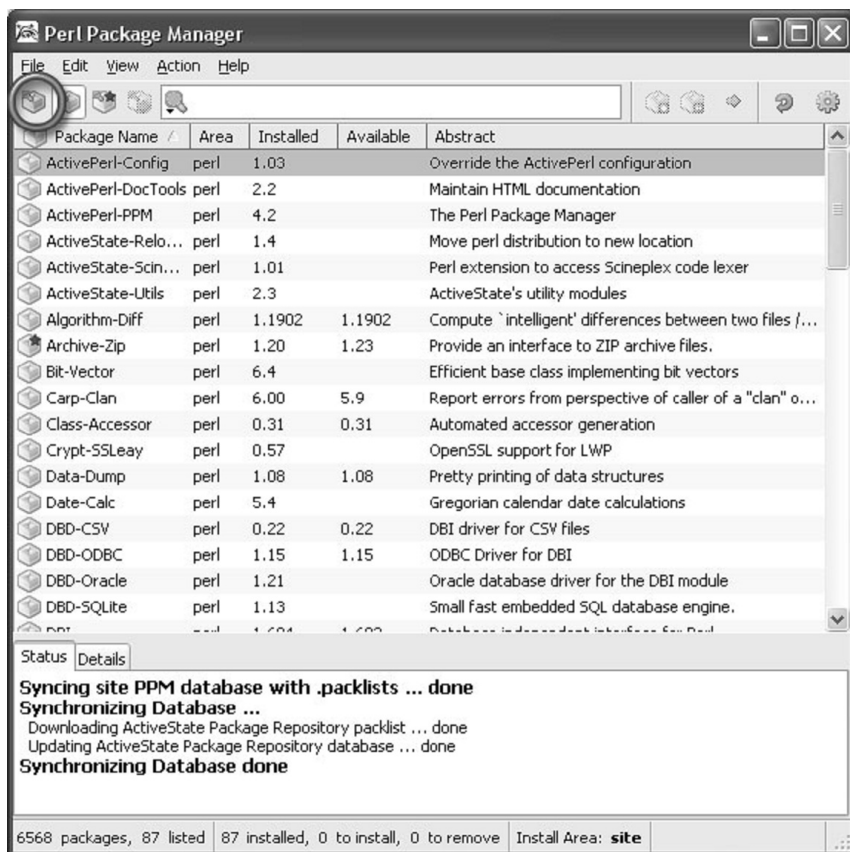
orduan modulua ez dago instalatua eta guk geuk instalatu beharko dugu. Egin proba zure ordenagailuan, seguru asko ez duzu modulua instalatua izango eta. Hala bada, `Text::Ngrams` moduluaren instalazioari ekin beharko diogu!

Ikus dezagun zeintzuk diren egin beharreko urratsak modulu berriak instalatzeko, bai Windows-en baita Linux-en ere. Kontuan izan moduluak instalatzeko beharrezkoa dela Internetera konektatua egotea, orain ikusiko ditugun modulu-kudeatzaileak —bai Windows-ekoak baita Linux-ekoak ere— zuzenean konektatzen baitira sarera moduluen bila.

Moduluak instalatzen: Windows

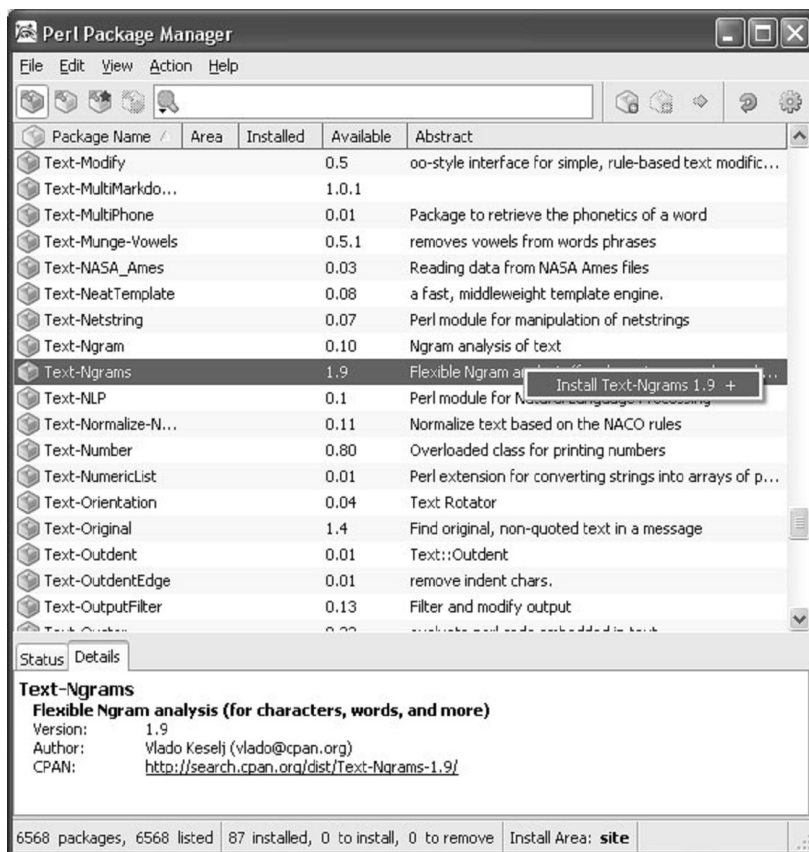
Windows-eko ActivePerl banaketa moduluak kudeatzeko aplikazio batekin dator, Perl Package Manager izenekoa. Windows-eko *Hasiera* menuan eta *Programak* pean *ActivePerl* izeneko multzoan dago paketeen kudeatzailea. Bestela, komando-lerrotik exekutatu *ppm* agindua, kudeatzailea abiarazteko.

Kudeatzaileak honako itxura hau du:



Hasiera batean kudeatzaileak dagoeneko instalatuta dauden moduluak erakusten ditu. Modulu guztiak ikusteko, berriz, goi-ezkerreko botoiari sakatu behar zaio, irudian agertzen den bezala.

Modulu guztien zerrenda ikusita, instalatu nahi dugun modulua aurkitu behar da. Gure kasuan, `Text::Ngrams` modulua instalatu nahi dugunez, `Text-Ngrams` paketea bilatu behar dugu. Ohar zaitez moduluaren izena aldatu dela, alegia, `Text::Ngrams` bi puntuen bikotea `Text-Ngrams` marra batez ordezkatu dugula. Ondoko irudian aurkitu dugu modulua, eta lerroan bertan saguaren eskuineko botoia sakatuz, instalatzeko agindua eman diogu.



Azkenik, kudeatzaileari esan behar zaio eskatutakoa (modulua instalatzea, alegia) egikaritzeko. Horretarako, menu-barratik *File/Run marked actions* sakatu. Agindu horrekin, modulua instalatu eta Perl-etik zuzenean erabili ahal izango dugu.

Moduluak instalatzen: Linux

Linux-en are eta errazagoa da moduluak instalatzea. Besterik gabe, eta komando-lerroan gaudela, `cpan`¹⁰ komandoa exekutatu behar da. Komando hori exekutatzeke, baina, *root* supererabiltzailearen baimenak behar dira, aldaketek sistema osoan izango baitute eragina. Hortaz, komando-lerroan gaudela, `Text::Ngrams` modulua instalatzeko exekutatu honakoa:

```
$ sudo cpan Text::Ngrams
```

¹⁰. `cpan` aplikazioa lehenengo aldiz exekutatzean bere burua hasieratzen du. Horretarako, galdera bat egingo digu, horrelako testu batekin:

```
Are you ready for manual configuration? [yes]
```

Galderari **ezet**z erantzun (no) eta aplikazioa automatikoki konfiguratuko da.

Lehenengo `sudo` aginduak ondoren datorrena supererabiltzaile bezala exekutatzeko balio du, baina lehenago erabiltzailearen pasahitza eskatuko du.

Dena ondo joan bada, komandoa zuzenean CPANera konektatu, modulua deskargatu eta instalatzeaz arduratuko da sisteman. Komandoaren irteeraren azkeneko lerroek honako itxura izan beharko lukete:

```
Writing /usr/local/lib/perl.8.8/auto/Text/Ngrams/.packlist
Appending installation info to
/usr/local/lib/perl/5.8.8/perllocal.pod
/usr/bin/make install - OK
```

8.3.3. `Text::Ngrams` *modulua erabiltzen*

Orain badakigu modulua instalatua dagoela, baina ez dakigu nola erabiltzen den, ezta zein funtzionalitate eskaintzen duen ere. Horretarako behar-beharrezkoa da moduluaren dokumentazioa irakurtzea, komando-lerrotik honako komando hau exekutatuz:

```
> perldoc11 Text::Ngrams
```

Berehala azalduko zaigu moduluaren dokumentazioa pantailan, eta bertan azalduko da moduluarekin erabil daitezkeen funtzioak, hauen parametroak zein diren, eta zertarako balio duten. Gure kasuan, sinopsi txiki bat ere azaltzen zaigu, `Text::Ngrams` modulua nola erabili azaltzen diguna. Honela dio testuak:

SYNOPSIS

```
For default character n-gram analysis of string:
    use Text::Ngrams;
    my $ng3 = Text::Ngrams->new;
    $ng3->process_text("abcdefghijklmnop");
    print $ng3->to_string;
```

Hor azaltzen den kodea Perl programa txiki bat da, modulua erabiltzen duen programa sinple bat. Azter ditzagun kodearen lerroak, banan-banan, beren esanahia argitze aldera. Lehenengo lerroak honela dio:

```
use Text::Ngrams;
```

Agindu honen bidez adierazten da programak `Text::Ngrams` modulua erabili nahi duela. Modulua instalatzeke egongo balitz, programa une honetan etengo litzateke errore-mezu bat inprimatuz.

```
my $ng3 = Text::Ngrams->new;
```

¹¹. `perldoc` komandoari eta bere erabilera buruzko informazio zehatza 10.1. atalean aurkituko duzue.

Lerro hau bereziki garrantzitsua da. Honekin `Text::Ngrams` moduluko *objektu* berri bat sortuko dugu, eta `$ng3` aldagaiari esleitu. Objektuak egitura bereziak dira, liburu honetan oraindik azaldu ez direnak. Labur azaltzeko, objektuak aldagai arruntak bezalakoak dira, alegia, balioak gordetzen dituzte. Horretaz gain, eta aldagai arruntak ez bezala, objektuek *metodoak* eskaintzen dituzte. Metodoak funtzioak dira, objektuei atxikita daudenak. Horrela, objektu baten metodoari deitzean, objektuak zeregin zehatz bat gauzatuko du, une horretan gordeta duen informazioaz baliatuz. Litekeena da, gainera, metodo baten deiak objektuaren egoera —hots, gordeta duen informazioa— aldatzea. Perl-eko modulu gehienek objektuak sortzeko funtzio berezi bat, *new* izenekoa izan ohi dute, modulu horretako objektuak sortzeko. Honako espresio hau erabili behar da:

Modulua->*new*(*parametroak*)

Itzul gaitezen, bada, aztertzen ari garen koderaz:

```
my $ng3 = Text::Ngrams->new;
```

Esan bezala, `Text::Ngrams` moduluko objektu berria sortu du aurreko sententziak, eta `$ng3` aldagaian gorde. `$ng3` aldagaiak testu baten gainera Ngramak kalkulatzen dituen objektu bat gordeko du, eta objektu horren bidez egin ahal izango dira Ngramen kalkulurako egin beharreko guztiak. Horretarako, baina, objektua testuarekin elikatu behar da lehenbizi, eta horixe bera egiten da hurrengo lerroan, `process_text` metodoari dei eginez:

```
$ng3->process_text("abcdefg1235678hijklmnop");
```

Lerro honetan `$ng3` objektuaren `process_text` metodoari deia egiten zaio. Ohar zaitezte metodo bati deitzeko syntaxian: *\$obj*->*metodoa*(*parametroak*) syntaxia erabili behar dela, non *\$obj* objektua gordetzen duen aldagaia den eta *parametroak* metodoaren parametroak. Esan bezala, goiko adibidean `$ng3` objektuari `process_text` metodoari deitu diogu, "abcdefg1235678hijklmnop" karaktere-katea parametro gisa pasatuz. Finean, `$ng3` objektuari testu bat prozesatzeko eskaera egin diogu, testu horretan dauden Ngramak kalkulatzeko.

Azken kode-lerroak honela dio:

```
print $ng3->to_string;
```

Hots, `$ng3` objektuko `to_string` metodoari deia (parametrorik gabe), zeinak objektuak kalkulatuak dituen Ngramak itzuliko dituen, karaktere-kate moduan. Funtzioak itzultzen duen karaktere-katea pantailan inprimatuko da, `print` dela medio.

Dokumentazioak erakusten duen adibide txikia ulertu dugularik, ekin diezaio-gun esku artean dugun ariketari, alegia, *corpus.txt* fitxategiko trigramak atera, eta

maiztasun handienekoak inprimatzeari. Hona hemen programaren hasierako lerroak

```
#!/usr/bin/perl
use strict;
use warnings;
use Text::Ngrams;
my $trig = Text::Ngrams->new(windowsize => 3,
                             type => "utf8");
```

Betiko 3 lerroen ondoren, modulua erabili nahi dugula adierazten dugu `use Text::Ngrams;` aginduaren bidez. Ondoren, `Text::Ngrams` objektu berri bat sortzen dugu, eta `$trig` aldagaian gorde. Oraingo honetan zenbait parametro¹² pasatu dizkiogu moduluko `new` eraikitzaileari, `$trig` objektuaren izaera zehazteko helburuarekin. `windowsize` parametroari 3 balioa eman diogu, 3 elementuko Ngramak (trigramak) nahi ditugula adierazteko. `type` parametroan, berriz, elementu bakoitza karaktere bat izango dela (eta ez hitz bat, adibidez), eta karakterearen kodeketa `utf8` izango dela zehaztuko dugu.

Ondoren datoz kode-lerro garrantzitsuenak:

```
$trig->process_files("corpus.txt");
print $trig->to_string(orderby => "frequency",
                      onlyfirst => "20",
                      spartan => 1);
```

Lehenengo lerroak kontaketa guztiak egiten ditu. Bertan, `$trig` objektuari *corpus.txt* fitxategia kargatzeko agintzen diogu, eta bertako trigrama guztiak kontatzeko `process_files` metodoaren bidez. Ondoren, `to_string` metodoa deitzen dugu, kalkulatu dituen trigramak itzul ditzan, karaktere-kate gisa. Parametroak pasatu dizkiogu `to_string` metodoari: `"orderby=>frequency"`, emaitzak maiztasunaren arabera ordenatuak nahi ditugula adierazteko, `"onlyfirst =>20"` hasierako 20 elementuak soilik itzul ditzan, eta azkenik `"spartan=>1"` ...

Ez, bada, ez dugu esango "spartan" parametroak zertarako balio duen. Izan ere, programa garatzeko moduluaren dokumentazioan aztarka ibili gara, metodoak eta hauen parametroak ezagutzeko. Baina informazio guztia zuk zeuk ere eskura duzu, "perldoc Text::Ngrams" idatzi orduko komando-lerroan. Beraz, ariketa bezala planteatuko dizuegu: aztertu moduluaren dokumentazioa eta "spartan" parametroak zertarako balio duen esan.

12. Eta nola jakin dugu `new` metodoak zein parametro har ditzakeen eta zertarako balio duten? Moduluaren dokumentazioa irakurritz, jakina!

Kontuak kontu, hona hemen programak itzultitako emaitza:

```
> perl bigramak.pl corpus.txt
e n \n 16583
r e n 12155
a r e 11088
a k \n 10545
k o \n 10478
i k \n 9378
a n \n 8991
t z e 8672
a r r 7890
e t a 7679
a r i 6884
t z a 6772
e r a 6369
r i k 5998
n t z 5930
t i k 5844
i k o 5722
a k o 5438
r r e 5422
a t u 5396
```

Ohar zaitezte irteerako ia trigrama guztiak direla aurrizki edo atzizkiak ("*n*" karaktereak esan nahi du hitz-eten karakterea azaldu dela; horrela, gehien azaldu den trigrama, "*en**n*", alegia, *en* bukaerako hitzek osatu dute). Euskararen atzizki/aurrezkiak automatikoki antzemateko programa garatu dugu, corpus baten gaineko azterketa estatistikoa erabiliz. Kontuan izan ez dugula inolako hiztegi edo gramatikarik erabili aurrezki/atzizkiak identifikatzeko. Eta hori dena Perl-eko modulu simple bat erabiliz. Hau da hau pagotxa!

Atal honekin amaitzeko, beste adibide bat erakutsiko dizuegu, moduluak erabiltzen dituen hau ere. Oraingoan moduluaren azalpen laburra baino ez dugu emango, eta zuen ardura izango da modulu instalatu eta bere dokumentazioa aztertzea. Adibide honetan erabiliko dugun moduluak *LevenshteinXS* du izena, eta aurrekoarekiko zertxobait desberdina da. Izan ere, *LevenshteinXS* moduluak ez baititu objektuak sortzen; horren orde, moduluak zenbait funtzio zehatz eskaintzen ditu.

27. kasu praktikoa: zuzentzaile ortografikoa

Moduluen laguntzaz, zuzentzaile ortografiko baten lehen bertsioa landuko

dugu ariketa honetan. Gure zuzentzaile ortografikoak, hitz bat jaso eta zuzen ala oker idatzitakoa den esango digu. Oker idatzitako hitza bada, programak harentzako alternatiba zuzenak proposatu behar ditu. Laguntza gisa, programak hiztegi-fitxategi bat jasoko du (`xuxen.txt`), fitxategi horretan agertzen ez diren hitzak okertzat jo eta proposamenak aurkeztuko dituelarik.

Idaztean egin ohi ditugun 3 oinarrizko errore mota antzemango ditu programak:

- 1.- Karaktere bat ezabatu. Adibidez: "eztia" idatzi beharrean "etia".
- 2.- Behar baino karaktere bat gehiago idatzi. Adibidez: "edo" idatzi beharrean "edos".
- 3.- Transposizioa: hitzeko 2 karaktere posizioz nahastea. Adibidez: "idatzi" beharrean "idazti".

Hona hemen programaren exekuzio-adibidea:

```
>perl xuxentzaile.pl xuxen.txt
Sartu hitz bat:
zihur
*****
Aukerak
*****
0 bihur
1 mihur
2 zigur
3 zimur
4 ziur
5 zizur
6 zuhur
>
```

Aurretik esan bezala, Perl-ek eskaintzen duen `LevenshteinXS` modulua erabiliko dugu ariketa honetan. Modulu honek bi *string*-en arteko distantzia kalkulatzeko duen `distance()` funtzioa dakar. *String*-en arteko distantzia, *Levenshtein distantzia* deiturikoa, *string* batetik bestera iristeko beharrezko den karaktere ordezkapen, ezabaketa edo gehitze kopurua izango da. Adibidez:

```
print (distance("gaur", "haur")); # 1 pantailaratuko du
print (distance("bare", "gore")); # 2 pantailaratuko du
```

Programaren ideia orokorra hau da: teklatu bidez hitz bat jaso eta `xuxen.txt` hiztegia arakatuko du programak. Hiztegian hitz berbera topatuz gero, sarrerakoa zuzena dela adieraziz mezu bat bistaratuko du pantailan. Bestela,

hiztegiko hitzik antzekoenak aukeratuko dira zuzenketarako proposamen gisa (antzekoenak, gehienez sarrerako hitzarekiko bateko distantziara daudenak).

Hona ideia orokorra algoritmo gisa idatzita:

1. Teklatutik hitza jaso: \$hitza
2. Hiztegi-fitxategia zabaldu
3. Hiztegiko \$xuxen sarrera bakoitzeko
 - 3.1. Baldin (\$hitza == \$xuxen)

bistaratu "hitza zuzena da"
 - 3.2. Bestela,

baldin (distance(\$hitza, \$xuxen) <= 1)

Gehitu \$hitza proposamen-zerrendan
4. Bistaratu proposamen-zerrenda

Hona hemen programa:

xuxentzaile.pl

```
#!/usr/bin/perl
use warnings;
use strict;
use Text::LevenshteinXS;

my ($xuxen, $hitza, $ondo, @aukerak);

print("Sartu hitz bat:\n");
$hitza = <STDIN>;
chomp($hitza);
$ondo = 0;

open(FITX, $ARGV[0]) or die("Ezin fitxategia zabaldu!");

while (($ondo != 1) && ($xuxen = <FITX>)) {
    chomp($xuxen);
    if (distance($hitza, $xuxen) == 0) {
        $ondo = 1;
    }
}
```

```

        elseif (distance($hitza, $xuxen) <= 1) {
            push(@aukerak, $xuxen);
        }
    }
    print("*****\n");
    print("Aukerak\n");
    print("*****\n");

    if ($ondo) {
        print("hitza zuzena\n");
    }
    else {
        for ($i=0; $i <= $#aukerak; $i++) {
            print("$i $aukerak[$i]\n");
        }
    }
}
close(FITX);

```

while begiztan baldintza konposatua jarri dugu: \$ondo aldagaiak 1 balioa ez duen bitartean **eta** fitxategian lerrorik den artean jarraitu, ((\$ondo != 1) && (\$xuxen = <FITX>)). Zergatik? Programak lehenago buka dezan. Izan ere, sarrerako hitza hiztegia topatuz gero, ez daukagu bilaketarekin aurrera jarraitu beharrik, hitza zuzen idatzia dagoelako. Kasu horretan, \$ondo aldagaiari bat balioa eman eta hurrengo iterazioan begiztatik aterako da, "hitza zuzena" mezua bistaratu. Bestela, sarrerako hitza hiztegia topatzen ez duen bitartean aurrera jarraituko du, hitzik antzekoenak aukeratuz. Jar dezagun adibide konkretu bat. Demagun programa abiarazi eta "abelazkuntza" idazten dugula, ondo edo gaizki dagoen jakin nahirik. Gure zuzentzailea xuxen.txt hiztegia hitzez hitz arakatzen hasiko zaigu, eta 122. sarreran abelazkuntza sarrerarekin topo egin. \$ondo aldagaiari bat balioa eman eta hurrengo iterazioan begiztatik aterako da. Baina begiztaren baldintzan \$ondo aldagaia begiratu ez bagenu, fitxategi osoa zeharkatuko luke beti programak (hitza topatu edo ez), eta gure hiztegiak 80.778 sarrera dauzka!

9. Kasu praktikoak

Kapitulu honetan hizkuntzaren azterketarako interesgarriak izan daitezkeen programak topatuko dituzu. Behin puntu honetara iritsita, amaitu dira ezaugarri berriak eta azalpen luzeak; esan beharrekoak esan ditugu jada. Baina zer hoberik ikusitakoa barneratzeko praktikatzea baino. Horixe da kapitulu honetan egingo duguna: ezagunak diren osagaiak hartu eta errezeta ezberdinak probatu haiekin. Batzuk zure lanerako erabilgarriak izan daitezke dauden-daudenean. Edo bestela, aldaketa gutxi batzuk eginda zeurera ekarri ahal izango dituzu. Izan daitezela gutxienez Perl-ekin egin ditzakegun lanen erakusgarri eta argigarri.

Irakurri, aztertu, probatu eta moldatu zuen gogora. Ez etsi guk emandako soluzioekin, gogoratu, beti daude problema bera ebazteko modu bat baino gehiago.

Orain artekoek gain, CDan topatuko dituzuen 2 fitxategi berri ere erabiliko ditugu kapitulu honetan: `xenpelar.txt` eta `anabitarteDonostia.txt`. Biak ala biak Klasikoak¹³ bildumaren webgunetik jaitsi ditugu eta testu-fitxategi gisa gorde, beraiekin lan egin ahal izateko.

9.1. AZPIPROGRAMAK

3. maila

28. kasu praktikoa: datuak grafikoki bistaratu

Askotan, datu sorta bat aurkezten digutenean, errazagoa izan ohi da datu horiek interpretatzea grafikoki adieraziz gero. Esaterako, hauteskunderen berri ematean, jasotako boto kopurua bera baino adierazgarriagoa izaten da alderdi bakoitzak lortu duen boto-portzentajea. Gauzak horrela, emaitzak barra bidezko grafiko batez bistaratuko dituen azpiprograma garatuko dugu, datu sorta bat aurkeztu behar dugunean erabili ahal izan dezagun.

Hona hemen, adibide gisa, komando-lerrotik datu sorta pasatu eta grafikoki bistaratzen dituen `bistaratu.pl` programaren irteera:

13. www.klasikoak.com webgunean euskal egileen testu-bilduma ederra topatuko duzue eskuragarri.

```
>perl bistaratu.pl 50 10 26 28
DATUAK PORTZENTAIA GRAFIKOA
104 %100
*****
10 9.62 ****
16 15.38 *****
28 26.92 *****
50 48.08 *****
>
```

Lerro hauen ondoren duzue `bistaratu.pl` programaren kodea, zeinak `grafikoa()` azpiprograma erabiltzen duen komando-lerrotik jasotako datuak bistartzeko. Programak ez du mugarik ezartzen, eta nahi adina datu pasa dakizkioke komando-lerrotik. Aztertu kodea eta exekutatu programa sarrerako datu ezberdinekin.

`bistaratu.pl`

```
#!/usr/local/bin/perl
use warnings;
use strict;

# Funtzio deia. Argumentua komando-lerrotik jasotako datuak
grafikoa(@ARGV);

# Funtzioaren definizioa
sub grafikoa {
    # Datuen array-a jaso argumentu gisa
    my @datuak = @_;
    my $batura = 0;
    my $portzentaia;
    my $zenbatIzar;
    # %100 kalkulatu: balio guztien batura
    foreach my $balio (@ARGV) {
        $batura = $batura + $balio;
        # edo $batura += $balio;
    }
    # Grafikoen goiburukoa
    print ("DATUAK\tPORTZENTAIA\tGRAFIKOA\n");
    print (" $batura \t %100\t\t");

    # Datuen %100 adierazteko 50 izartxo
    print ("*" x 50);
    print ("\n\n");
}
```

```
# Datuak ordenatu eta grafikoki bistaratu
foreach my $balio (sort ({ $a <=> $b } @datuak)) {
    # Datu bakoitzaren portzentaia
    $portzentaia = ($balio / $batura) * 100;
    # Bistaratzeko formatua
    printf("%s \t %4.2f \t\t", $balio, $portzentaia);
    # Portzentaiari dagozkion izartxoak
    $zenbatIzar = int($portzentaia * 0.5);
    print ("*" x $zenbatIzar);
    print("\n");
}
}
```

Programaren muina `grafikoa()` azpiprograman dago. Horrek, argumentu gisa zenbakizko datuen *array* bat jaso eta datuok barra bidezko grafiko bidez adierazten ditu pauso hauei jarraituz: lehenbizi, *array*-ko elementu guztien batura kalkulatzeko du `$batura` aldagaian, hau da, datuen % 100 adieraziko duen kopurua. Segidan datozen lerroek grafikoa goiburua bistaratzen dute, grafikoa interpretatzeko lagungarri izango direnak. Datuen % 100 adierazteko 50 izartxo erabili ditugu (100 jarritz gero, emaitza ez baita pantailan lerro bakarrean sartzen). Segidan, datuak txikienetik handienara ordenatu eta `foreach` begizta erabiliz banan-banan dagozkien portzentajeak eta izar-kopurua bistaratzen dira. Uneko `$balio` datuari dagozkion ehunekoa kalkulatzeko formula sinplea da:

```
$portzentaia = ($balio / $batura) * 100;
```

Perl-ek zehaztasun handiz egiten ditu kalkuluak, eta emaitza zenbaki erreala denean, koma ondoren digitu mordoa agertuko zaigu. Hainbestearinoko zehaztasunik behar ez dugunez, digitu kopurua kontrolatzeko `printf` komandoa erabiliko dugu, `print`-en aldaera dena:

```
printf("%s \t %4.2f \t\t", $balio, $portzentaia);
```

Bi argumentu dauzka komando berriak. Lehenbizikoan datuak bistartzeko formatua zehazten da, eta bigarrenetan formatu horretan aurkeztu behar diren datuak. Gure kasuan, formatuan `"%s \t %4.2f \t\t"` idatzi dugu. `%s` datua bere horretan idatzi behar dela adierazteko, eta `%4.2f`, berriz, gehienez 4 zifra eta koma ondoren bi digitu dituen zenbaki erreala dela adierazteko. Azken argumentu honekin `$portzentaia` aldagaiaren balioak biribilduko ditugu, koma ondoko zenbakiak bira murriztuz eta pantailan bistartzeko formatu egokiagoa emanez.

Ehunekoa kalkulatu ostean izartxo kopurua kalkulatzeko, nahikoa da ehunekoa zati bi egitearekin (izan ere, % 100ari 50 izartxo dagozkio). Kasu honetan emaitza osoa izatea interesatzen zaigu, eta hargatik erabili dugu `int()` funtzioa, argumentu gisa jasotako funtzioaren zati osoa itzultzen duena. Kopurua kalkulatu


```

while ($lerro = <FI>){
    chomp($lerro);
    # $hitza-ren eta bere hurrengoaren agerpenak
    # bilatu lerroan
    while ($lerro =~ /\s+$hitza\s+(\w+)/gi) {
        # Adierazpen erregularraren aukerak:
        # g globala (agerpen guztiak)
        # i letra larri/xeheak ez bereizi

        # $1 aldagaia $hitza-ren hurrengoa
        print "$hitza $1\n";
        $zenbat++; # $zenbat = $zenbat + 1;
    }
}
print("Guztira $zenbat agerpen\n");
close FI;

```

Programak lerroz lerro aztertzen du testua. Lerro bakoitzeko, bilaketa-hitzaren eta bere hurrengoaren agerpen guztiak bilatzen ditu, adierazpen erregularra begiztan jarritz:

```
while ($lerro =~ /\s+$hitza\s+(\w+)/gi)
```

Baldintzako adierazpenak honako bilaketa hau egingo du lerroan: bilaketa-hitza (\$hitza) ondoren zuriune bat (edo gehiago) eta jarraian hitz bat daukaten kateak. Hitz laguntzailea \$1 aldagaian gordeko du espresioa parentesi artean jarria dagoelako. `print` funtzioak \$hitza eta \$1 bere laguntzailearen agerpen guztiak bistaratuko ditu.

30. kasu praktikoa: karaktere erabili

Programa honek komando-lerroaren bidez fitxategi-izena eta bi karaktere jasoko ditu, eta lerroz lerro karaktere horien agerpenak zenbatuz joango da. Bu-kaeran bi karaktere horietatik usuen zein agertu den adieraziko du, agerpen kopuru berbera izan badute hala adieraziz.

Exekuzio-adibidea:

```

>perl zeinKarGehiago.pl esaeraZaharrak.txt a o
a (2322) karakterea o (763) baino gehiagotan azaltzen da

```

Aurreko adibideak, `esaeraZaharrak.txt` fitxategiko "a" eta "o" karaktereen agerpen kopuruak zenbatzen ditu.

Programaren kodea:

zeinKarGehiago.pl

```
#!/usr/bin/perl
use strict;
use warnings;
# Fitxategia zabaldu
open (FI, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia zabaldu!\n");
# Bilaketa karaktereak jaso
my $karBat = $ARGV[1];
my $karBi = $ARGV[2];
my $agerpenBat = 0;
my $agerpenBi = 0;
my $lerro;
# Fitxategia lerroz-lerro irakurri
while ($lerro = <FI>){
    chomp($lerro);
    # Lerroa karakterez-karaktere irakurri
    while ($lerro =~ /($karBat|$karBi)/gi) {
        # Adierazpen erregularraren aukerak:
        # g globala (agerpen guztiak)
        # i letra larri/xeheak ez bereizi

        # bat egitea $karBat edo $karBi-k eragin du?
        if ($1 eq $karBat) {
            $agerpenBat++;
        }
        if ($1 eq $karBi) {
            $agerpenBi++;
        }
    }
}
if ($agerpenBat > $agerpenBi) {
    print("$karBat ($agerpenBat) karakterea $karBi
        ($agerpenBi) baino gehiagotan azaltzen da\n");
}
elsif ($agerpenBat < $agerpenBi) {
    print("$karBi ($agerpenBi) karakterea $karBat
        ($agerpenBat) baino gehiagotan azaltzen da\n");
}
else {
    print("$karBat eta $karBi karaktereen agerpen kopurua
        berbera da fitxategian: $agerpenBat\n");
}
close (FI);
```

Komando-lerrotik fitxategi-izena eta bi karaktere jaso, eta fitxategia lerroz lerro zeharkatuko du programak. Lerro bakoitzeko, \$karBat edo \$karBi karaktereen agerpenekin bat egingo du adierazpen erregularrak, aukera "l" metakarakterea erabili baitugu. Bat-egitea gertatzean, baldintza egitura bidez \$karBat edo \$karBi-k eragin duen egiaztatu eta dagokion kontagailuari bat gehituko zaio.

Karaktereekin egin berri duguna hitzekin egin nahiko bagenu, programa bera erabil genezake aldaera txiki batekin: bilaketa-patroiei hitz-mugak jarri beharko genizkieke aurretik eta atzetik, hitzek osorik bat egin dezaten. Honela:

```
$lerro =~ /\b($hitzBat|hitzBi)\b/
```

Egin proba!

3. maila

31. kasu praktikoa: hitza bere testuinguruan

Bilaketa-programa berri bat garatuko dugu, zeinak bilaketa-hitza bere testuinguruan bistaratuko digun. Programak argumentu bezala komando-lerrotik fitxategi-izena, bilaketa-patroia eta N zenbaki bat jasoko ditu, eta bilaketa-hitza duten lerro guztiak bistaratuko, aurreko eta hurrengo N lerroekin batera testuingurua hobeto ikustearren.

Adibide gisa bila dezagun "amorio" hitza xenpelar.txt fitxategian, aurreko eta segidako 2 lerroekin batera:

```
>perl aurreHurre.pl xenpelar.txt amorio 2
dote eta arrio,
ez dutela balio
amorio fiñarentzat,
zergatik izan biarko duben
il arteraño beretzat.
>
```

Hona hitza bere testuinguruan bistaratzen duen programaren kodea:

aurreHurre.pl

```
#!/usr/bin/perl
use strict;
use warnings;
my $hitza = $ARGV[1]; # bilaketa hitza
my $zenb = $ARGV[2]; # aurretik/ondoren zenbat lerro?
open(FITX, $ARGV[0]) ||
    die("Errorea! Ezin $ARGV[0] fitxategia ireki\n");
```



```

open (FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia zabaldu\n");
my @hitzak;
my $lerro;
while ($lerro = <FITX>) {
    chomp($lerro);
    # Banatu lerroa hitzetan.
    # Banatzaileak: . , ; : ? ! eta zuriunea
    @hitzak = split(/\.\.?!\;:\s]+/, $lerro);
    # Lerroko hitz bakoitzeko
    foreach my $h (@hitzak) {
        # Bistaratu bi karaktere berdin baditu jarraian
        print ("$h\n") if ($h =~ /([a-z])\1/);
    }
}
close(FITX);

```

Programak sarrerako fitxategia ireki eta hitzez hitz prozesatzen du. Hitz bakoitzaren tratamentuan, bi karaktere berdinen sekuentziak aurkitzeko, atzera begirako erreferentziak erabiltzen ditu. Parentesi artean [a-z] karaktere-klasea jarri eta segidan \1 aldagai berezia, aurretik parentesi arteko espresioak jasotako karakterearen berdina izan behar duela adierazteko (\1 aldagai bereziak lehenbiziko parentesi arteko espresioaren balioa gordeko du).

Aldaketa gutxi batzuk eginez, karaktere bikoiztuen ordeztu hitz bikoiztuak topatzen dituen programa idatz genezake. Honela:

hitzBikoiztuak.pl

```

#!/usr/bin/perl
use strict;
use warnings;

open (FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia zabaldu\n");
my @hitzak;
my $lerro;
while ($lerro = <FITX>) {
    chomp($lerro);
    while ($lerro =~ /\b(\w)+\b\s+\b\1\b/g) {
        print ("Bikoiztutako hitza: $1\n");
        print ("$lerro\n");
    }
}
close(FITX);

```

Programak ez ditu hitz bikoiztu bezala kontatzen *bi bide*, *gero gerokoak* eta abar. Hitz bikoiztu bezala onartua izateko, hitz osoak bat egin behar du, adib.: *ez ez*, *tira tira*.

9.3. ORDEZKAPENAK

33. kasu praktikoa: testua esalditan banatu

Testu bat jaso eta bere edukia lerroz lerro irakurtzen badakigu, baita hitzez hitz ere. Interesgarria litzateke baita, testua esaldiz esaldi irakurtzerik izango bage-nu. Horixe da hain zuzen ere ariketa honetan egingo duguna, sarrera gisa testu fitxategi bat jaso eta esalditan banatu.

Adibide gisa, hona anabitarteDonostia.txt fitxategia esalditan banatzen duen deia eta programaren irteera:

```
>perl esalditan.pl  anabitarteDonostia.txt
DONOSTIA  Augustin Anabitarte    I Donostia!

Azkeneko gerratea bukatu zan.

Txomin donostiarrak Ernani-ko aldapan gora zaldi-gañean
zeturten.

An, atzean, gelditu zan Ernani ta Ernani-ko kartzela,
Txomin ain luzaroan egondakoa.
...
>
```

Ataza orokorra hiru azpiataza edo egitekotan bana genezake:

1. Sarrera-fitxategiaren eduki osoa \$testua *string* motako aldagaian gorde
Horretarako:

1.1. Fitxategia zabaldu

1.2. Lerroz lerro irakurri

1.2.1. Lerro bakoitzean bukaerako lerro-jauzia zuriunearekin ordezkatu

1.2.2. Kateatu lerroa \$testua *string*-ari

2. Banatu \$testua esalditan `split()` erabiliz

3. Bistaratu esaldiak banan-banan

Hona hemen, aurreko pausoei jarraitzen dien programa:

esalditan.pl

```
#!/usr/bin/perl
use warnings;
use strict;
open(FITX, $ARGV[0]) or
    die("Ezin $ARGV[0] fitxategia zabaldu!\n");
# hasieratu testu hutsarekin
my $testua = "";
my $lerro;
while ($lerro = <FITX>) {
    # Ordezkatu lerro jauziak zuriuneengatik
    $lerro =~ s/\n$/ /g;
    # Kateatu lerroa $testua aldagaiari
    $testua .= $lerro;
}
# $testua string-a esalditan banatu
my @esaldiak = split(/([\.\?!\+)]\s+/, $testua);
# Esaldiak banan-banan bistaratu
foreach my $esaldi (@esaldiak) {
    # bistaratu esaldia
    print ("$esaldi");
    # Esaldi bukaerako ikurra denean 2 lerro jauzi
    if ($esaldi =~ /([\.\?!\+)]+/) {
        print ("\n\n");
    }
}
close(FITX);
```

Konturatuko zineten moduan, \$testua-ren edukia esalditan banatzean, `split()` funtzioaren patroia banatzailea parentesi artean jarri dugu. Izan ere, horrela eginez gero, `split()`-ek testu puskak ez ezik, banatzaileak ere gordeko ditu @esaldiak *array*-an, eta ondoren, bistaratzerakoan, esaldi bakoitzari dagokion bukaerako ikurra eransteko moduan izango gara. `foreach` egiturak banan-banan hartzen ditu @esaldiak *array*-ko elementuak, eta pantailan bistaratu `print()` funtzioa erabilita. Baldintzazko `if` egiturak uneko elementuak puntuazio-marka duen ala ez begiratzen du, hala den kasuetan bi lerro-jauzi erantsiz.

3. maila

34. kasu praktikoa: ordezkapen anitz

Ordezkapenak egiten dituen beste programa bat da hau. Honen berezitasuna, sarrerako testuan aldiberean nahi adina ordezkapen egin ahal izatea da. Sarrera-

datu gisa, testu-fitxategiaren izena eta bilatu eta ordezkatu nahi ditugun hitzak (binaka) jasoko ditu programak komando-lerrotik.

Adibidez:

```
>perl ordezkapenak.pl testua.txt haundi handi
```

testua.txt fitxategiko "haundi" hitzaren agerpen guztiak ordezkatu "handi" hitzarekin.

```
>perl ordezkapenak.pl testua.txt 1 bat 2 bi 3 hiru
```

Hiru ordezkapen aldi berean: "1", "2" eta "3" digituen agerpenak, "bat" "bi" eta "hiru" *string*-ekin ordezkatzeko ditu hurrenez hurren.

Hona programaren kodea. Irakurri arretaz, azalpen guztiak iruzkinen bidez eman nahi izan ditugu-eta bertan.

ordezkapenak.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;

my ($fitx, $argluz, $lerro);
$fitx = $ARGV[0];
open (FITX, $fitx) or
    die("Ezin $fitx fitxategia zabaldu\n");

# Egiaztatu argumentu kopurua egokia dela (bakoitia):
# $ARGV[0] -> fitxategi-izena
# $ARGV[1] eta $ARGV[2] , $ARGV[3] eta $ARGV[4] ... bilatu
# eta ordeztu nahi ditugun hitzak, binaka idatzi behar
# dira.
# Beraz, @ARGV-ren azken indizeak bakoitia izan behar du
$argluz = @ARGV;
if (($argluz % 2) == 0) {
    # Argumentu kopuru okerra
    die ("Hitzak bikoteka idatzi behar dira!\n");
}
```



```
while($lerro = <FITX>){
    #lerro bakoitzeko ordezkapen guztiak burutu
    for(my $i = 1; $i < $argluz; $i += 2){
        # Argumentuak binaka hartu
        # i. argumentua bilatu eta (i+1). argumentuagatik
        # ordezkatu
        $lerro =~ s/$ARGV[$i]/$ARGV[$i+1]/g;
    }
    print $lerro;
}
```

9.4. ADIERAZPEN ERREGULARRAK ETA HASH EGITURAK

35. kasu praktikoa: karaktereen maiztasunak testuan

Testu elektronikoak aztertzen diharduen lagun batek honako enkargua eman digu: sarrera gisa testu bat jaso eta letra bakoitzaren agerpen kopurua zenbatzen duen programa behar du. Gainera, emaitza bi modutan ordenatuta bistaratzea nahiko luke: lehenbizi karaktere-zerrenda alfabetikoki, bakoitza bere agerpen kopuruarekin; eta segidan agerpen-maiztasunaren arabera ordenatuta, handienetik txikienera.

Kasu gehienetan emaitza pantailan ikusteko baino handiagoa izango denez, egokia litzateke bigarren argumentu bezala fitxategi-izen bat eman eta bertan idatzi ditzala irteerako datuak. Modu honetan:

```
>perl letraMaiz.pl esaeraZaharrak.txt irteera.txt
```

Aurreko programa-deiak esaeraZaharrak.txt fitxategiko letren maiztasuna kalkulatu eta irteera.txt fitxategian gordeko ditu. Exekuzioaren ostean, irteera.txt fitxategiaren edukia editorearekin bistaratuz gero:

irteera.txt

```

-----
Alfabetikoki ordenatuta:
-----
_ 4
a 2386
b 413
d 425
...
-----
Orain maiztasunaren arabera ordenatuta:
-----
a      2386
e      1633
i      1179
r      1124
...
>

```

Hona hemen, azkenik, karaktere-maiztasunak kalkulatzeko dituen programa:

letraMaiz.pl

```

#!/usr/bin/perl
use strict;use warnings;
my ($fitxIn, $fitxOut, $lerro, $kar, %frek, $gako, $balio);

$fitxIn = $ARGV[0];
$fitxOut = $ARGV[1];

# Iturburu eta helburu fitxategiak ireki
open(FITXIN, $fitxIn) ||
    die ("Ezin $fitxIn fitxategia zabaldu\n");
open(FITXOUT, ">$fitxOut") ||
    die ("Ezin $fitxOut fitxategia zabaldu\n");

while ($lerro = <FITXIN>) {
    chomp($lerro);
    $lerro =~ tr/A-Z/a-z/; # Letra larriak xehe bihurtu
    while ($lerro =~ /(\w)/gi) {
        # Espresio erregularraren aukerak: g globala
        # i maiuskula/minuskula ez bereizi
        $kar = $1;      # parekatu-berria den karakterea
        $frek{$kar}++;
    }
}

```

```
# Iturburu fitxategia itxi
close(FITXIN);

print (FITXOUT "-----\n");
print (FITXOUT "Alfabetikoki ordenatuta:\n");
print (FITXOUT "-----\n");
foreach $kar (sort (keys(%frek))) {
    print (FITXOUT "$kar $frek{$kar} \n");
}

print (FITXOUT "-----\n");
print (FITXOUT "Orain maiztasunaren arabera
              ordenatuta:\n");
print (FITXOUT "-----\n");
# Hurrengo agindua konplexu-samarra da.
# Hash baten gakoak atxikitua duten balioen arabera
# ordenatu, handienetik txikienera
foreach $kar
    (sort ({ $frek{$b} <=> $frek{$a} } (keys %frek))) {
    print (FITXOUT "$kar\t$frek{$kar}\n");
}
close(FITXOUT);
```

Programaren egitura orokorrak ez dakar berritasunik: komando-lerrotik jasotako bi fitxategiak zabaldu, bata irakurketarako eta bestea idazketarako, eta irakurketarako fitxategia lerroz lerro zeharkatzen du `while` kontrol-egiturarekin. Begiztaren gorputzean, honako eragiketak egiten ditu programak: lerro bakoitzeko bukaerako lerro-jauzia kendu, letra larriak xehe bihurtu, eta, azkenik, bigarren `while` begizta batek lerroko karaktereak banan-banan jasoko ditu ondoren prozesatu ahal izateko. Karaktere bakoitzaren tratamendua honakoa da:

```
$frek{$kar}++;
```

`$kar` karakterea gako berria bada *hash*-ean, adierazpenak bat balioa esleitu-ko dio. Aldiz, karakterea aurretik azaldu bada, bat gehitu dio bere aurreko balioari. Begizta amaitzean, `%frek` egiturak honakoa gordeko du: gako bezala sarrerako fitxategiaren karaktere ezberdin guztiak, eta balio bezala gako bakoitzaren agerpen kopurua testuan.

Programaren bigarren zatia `%frek` *hash*-aren balioak ordenatu eta irteerako fitxategian gordetzeaz arduratzen da. Bi `foreach` begizta erabiltzen dira

horretarako, lehenak alfabetikoki ordenatuta gordeko ditu letren agerpenak, eta bigarrenak, berriz, agerpen kopuruaren arabera ordenatuta.

36. kasu praktikoa: hitzen maiztasunak testuan

Aurreko ariketan karaktereekin egindakoa, hitzekin egingo du oraingo programak: sarrera-datu gisa fitxategi-izena jaso eta bertako hitz bakoitzaren agerpen kopurua kontatu. Emaitzak agerpen kopuruaren arabera ordenatuta bistaratuko ditu, handienetik txikienera, hitz bakoitzeko honako informazioa erakutsiz: hitza bera, bere agerpen kopurua edo maiztasuna, eta maiztasun erlatiboa. Hitz baten maiztasun erlatiboa, hitz horren agerpen kopurua hitz kopuru orokorraz zatituta lortzen den koefizientea da.

Programa `esaeraZaharrak.txt` fitxategiko hitzen maiztasunak ezagutzeko erabiliz gero, hauxe da lortuko genukeena:

```
>perl hitzMaizErlat.pl esaeraZaharrak.txt
hitza: "ez"           Maiztasuna: 103   Maiz. Erlatiboa: 4.1382
hitza: "eta"          Maiztasuna: 90    Maiz. Erlatiboa: 3.6159
hitza: "da"           Maiztasuna: 50    Maiz. Erlatiboa: 2.0088
hitza: "du"           Maiztasuna: 24    Maiz. Erlatiboa: 0.9642
hitza: "baino"        Maiztasuna: 19    Maiz. Erlatiboa: 0.7634
...
>
```

Berriro ere, emaitza handiegia da pantailan osorik ikusi ahal izateko. Irteera-fitxategi batera birbideratzea izan daiteke soluzio egokia, edo baita irteerako fitxategi bat erabiltzea datuak bertan idazteko ere.

Programaren kodea:

hitzMaizErlat.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my ($l,@hitzak, $hitzakguztira, $h, %Maiz);
open (FI, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia zabaldu\n");
$hitzakguztira = 0;
while ($l = <FI>) {
    chomp($l);
    # Banatu lerroa hitzetan
    # Banatzaileak: , ; . : ! ? eta zuriunea
```

```

@hitzak = split(/[\.?!,:;\s]+/, $1);
foreach $h (@hitzak) {
    # $h hitzaren agerpen kop. eguneratu
    $Maiz{$h}++;
    # Hitz kopuru totala
    $hitzakguztira = $hitzakguztira + 1;
}
my $maiztasuna;
# %Maiz hash-aren gakoak atxikitako balioaren arabera
# ordenatu, handienetik txikienera
my @gakoOrdenaduak =
    sort ({ $Maiz{$b} <=> $Maiz{$a} } keys(%Maiz));
foreach $h (@gakoOrdenaduak) {
    # $h hitzaren maiztasun erlatiboa:
    # bere agerpen kopurua zati hitz kopuru totala
    # (ehunekotan adierazia)
    $maiztasuna = 100 * ($Maiz{$h} / $hitzakguztira) ;
    printf ("hitza: '%s' \t\t maiztasuna(ehuneko):
           %4.4f \n" , $h, $maiztasuna);
}

```

2. maila

37. kasu praktikoa: bigrama-ereduak

Testuak elementu-sekuentzia gisa har daitezke: karaktereak, hitzak, esaldiak, etab. Elementu-sekuentzia horien azterketa ohiko prozedura da hizkuntzaren tratamendu automatikoan, eta horretarako erabiltzen dira Ngrama-ereduak. Moduluen gaia lantzean, Ngramak kalkulatzeko modulua instalatu eta erabili genuen (text::Ngrams). Orduan, adibide gisa, testu bat hartu eta bere trigramak kalkulatzuten programa garatu genuen modulua erabiliz. Oraingoan, bigramak kalkulatuko ditugu, baina modulurik erabili gabe, guk geuk garatuko dugu osorik karaktere-bikoteen bigrama-eredua itzultzen duen programa (gogoratu, "bigrama" hitzaren bigrama-eredua *bi-ig-gr-ra-am-ma* da).

Programa bera, bigrama-eredua, interesgarria izateaz gain, uste dugu garatzeko erabilitako estrategia ezagutzeak ere merezi duela. Beste aplikazio batzuetarako ideiak eman diezazkiguke.

Gure programak, komando-lerrotik fitxategi-izena jasoko du sarrera gisa, eta irteeran fitxategi horetan karaktere-bikote bakoitza zenbatetan azaltzen den bistaratuko. Bigrama-maiztasunak bi modutara bistaratuko ditu, lehenbizi testuan agertu diren ordenari jarraituz, eta, ondoren, maiztasunaren arabera ordenatuta.

Hona programaren erabileraren adibidea. Irteera-fitxategi batera birbideratu dugu pantailan osorik ikusteko handiegia zelako:

```
>perl bigramaMaiz.pl esaeraZaharrak.txt > irteeraBigrama.txt
```

Fitxategiaren edukia bistaratuz gero:

irteeraBigrama.txt

```
sp bikotea 2 aldiz agertu da
ne bikotea 161 aldiz agertu da
eb bikotea 11 aldiz agertu da
...
ij bikotea 4 aldiz agertu da
kk bikotea 2 aldiz agertu da
-----
Maiztasunaren arabera ordenatuta:
-----
an bikotea 314 aldiz agertu da
ar bikotea 307 aldiz agertu da
en bikotea 278 aldiz agertu da
ak bikotea 272 aldiz agertu da
...
zm bikotea 1 aldiz agertu da
th bikotea 1 aldiz agertu da
>
```

Hona programaren funtsa: karaktere-bikoteen bigrama-eredua eraiki eta beren agerpen kopurua gordetzeko, *hash* egitura erabiliko dugu. Bigramak eraikitzeke, sarrerako testua karakterez karaktere zeharkatuko dugu, pauso bakoitzean uneko karakterea *t* eta bere aurrekoa kateatu eta bigrama osatuz. Adibidez: *\$kar* erabil dezakegu uneko karakterea gordetzeko, eta *\$aurrekokar* aldagaia bere aurrekoa gordetzeko. Bigrama, biak kateatuz osatuko da: *\$aurrekokar . \$kar*

Bigrama eratu berria *hash*-aren gakoa izango da, eta balioa, berriz, bere agerpen kopurua. Bigrama bakoitzeko, honela eguneratuko dugu balioa:

```
$frek{$aurrekokar . $kar}++;
```

Non *%frek* bigrama-eredua gordetzen duen *hash* egituraren izena den.

Hona hemen *bigramaMaiz.pl* programaren kodea:

bigramaMaiz.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my ($lerro, $kar, $aurrekokar, $fIzena, %frek,
    $gako, $balio);
$fIzena = $ARGV[0];
open(FITX, $fIzena) ||
    die ("Ezin $fIzena fitxategia zabaldu\n");

while ($lerro = <FITX>) {
    chomp($lerro);
    # Letra larriak xehe bihurtu
    $lerro =~ tr/A-Z/a-z/;
    $kar = " ";
    $aurrekokar = " ";
    while ($lerro =~ /(\w)/gi) {
        # g globala
        # i maiuskula/minuskula ez bereizi
        $aurrekokar = $kar;
        $kar = $1; # parekatu-berria den karakterea
        # Karaktere-bikotearen lehen agerpena bada
        # hasieratu, bestela 1 gehitu aurreko balioari
        $frek{$aurrekokar . $kar}++;
    }
}
close(FITX); # Fitxategia itxi
# Bistaratu testuko ordena jarraituz
while (($gako, $balio) = each(%frek)) {
    print "$gako bikotea $balio aldiz agertu da\n";
}
print "-----\n";
# Bistaratu agerpen kopuruaren arabera ordenatuta,
# handienetik txikienera.
# Lehenbizi hasharen gakoak ardenatu balioen arabera.
# Zenbakizko ordenazioa, handienetik txikienera
my @gakoOrdenaduak =
    sort ({ $frek{$b} <=> $frek{$a} } keys(%frek));
my $gk;
foreach $gk (@gakoOrdenaduak) {
    print "$gk bikotea $frek{$gk} aldiz agertu da\n";
}
```

Aldaketa gutxi batzuekin, **hitz-bigramak** kontatzen dituen programa lor dezakegu. Horretarako *Hash*-aren gakoak bi hitzez osatutako *string*-a izan beharko du, eta ez bi karakterez osatutakoa. Gainerakoan, programa ia berbera da. Diferentzia da sarrerako testua karakterez karaktere irakurri beharrean, honako honetan hitzez hitz irakurri behar izatea. Programa laburtzearren, hitz-bigramak maiztasunaren arabera ordenatuta soilik bistaratuko ditu.

Hona exekuzio-adibidea:

```
>perl bigramaHitz.pl esaeraZaharrak.txt > irteeraBigramHitz.txt
```

Eta irteeraBigramHitz.txt-en edukia:

```
-----
Maiztasunaren arabera ordenatuta:
-----
```

```
"ez da" bikotea 16 aldiz agertu da
" ez" bikotea 11 aldiz agertu da
"ez du" bikotea 9 aldiz agertu da
" eguberri" bikotea 7 aldiz agertu da
"hobe da" bikotea 6 aldiz agertu da
"nahi ez" bikotea 6 aldiz agertu da
...
>
```

Azkenik, programaren kodea:

bigramaHitz.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my ($lerro, $hitz, $aurrekoHitz, $fIzena, @hitzak, %frek,
    $gako, $balio);
$fIzena = $ARGV[0];
open(FITX, $fIzena) ||
    die ("Ezin $fIzena fitxategia zabaldu\n");

while ($lerro = <FITX>) {
    chomp($lerro);
    # Letra larriak xehe bihurtu
    $lerro =~ tr/A-Z/a-z/;
    # Hasieratu uneko eta bere aurreko
    # hitzak zuriunearekin
    $hitz = " ";
```



```
$aurrekoHitz = " ";
# Banatu lerroa hitzetan. Banatzaileak: . , ; : ? !
# eta zuriunea
@hitzak = split(/[\.?!;:,\\s]+/, $lerro);
# Lerroko hitz bakoitzeko
foreach my $h (@hitzak) {
    $aurrekoHitz = $hitz;
    $hitz = $h;
    # Bi hitzen artean kateatu zuriunea
    $fрек{$aurrekoHitz . " " . $hitz}++;
}
}
close(FITX); # Fitxategia itxi
print ("-----\n");
print ("Maiztasunaren arabera ordenatuta:\n");
print ("-----\n");
# Bistaratu agerpen kopuruaren arabera ordenatuta,
# handienetik txikienera.
# Lehenbizi hash-aren gakoak ordenatu balioen arabera.
# Zenbakizko ordenazioa, handienetik txikienera
my @gakoOrdenaduak =
    sort ({ $fрек{$b} <=> $fрек{$a} } keys(%fрек));
my $gk;
foreach $gk (@gakoOrdenaduak) {
    print "'$gk' bikotea $fрек{$gk} aldiz agertu da\n";
}
}
```



2. maila



38. kasu praktikoa: unigrama, bigrama eta trigramak

Hizkuntza ezberdinetako testu-corpusak dauzkagu, eta hizkuntza bakoitzaren ezaugarri nagusiak atera nahi ditugu corpus horietatik abiatuta. Honako ezaugarriak dira batez ere interesatzen zaizkigunak:

- Unigrama, bigrama eta trigramarik erabilienak.
- Gehien errepikatzen den letra eta bere maiztasun erlatiboa.
- Bokalen maiztasun erlatiboa.

Hona hemen, adibide gisa, programak egindako bi azterketa: lehena, euskarazko `euskCorp.txt` corpora erabilia sarrerako fitxategi bezala, eta, bigarrena, berriz, `engCorpus.txt` ingelesezko corpora erabilia:

```
>perl lengoaiaAzt.pl euskCorp.txt
```

```
Unigramak Bigramak Trigramak
```

```
a          en          ren
```

```
e          ar          eta
```

```
i          er          tze
```

```
r          re          are
```

```
n          ta          era
```

```
t          ra          zen
```

```
o          at          arr
```

```
z          te          ber
```

```
k          tz          ere
```

```
u          an          rre
```

```
Letra ohikoena: a  Maiztasuna: 13.2717
```

```
Bokal maiztasuna: 41.3115
```

```
>
```

```
>perl lengoaiaAzt.pl engCorp.txt
```

```
Unigramak Bigramak Trigramak
```

```
e          in          ing
```

```
i          er          ion
```

```
s          es          ati
```

```
a          ti          tio
```

```
r          re          ers
```

```
n          on          ter
```

```
t          te          ent
```

```
o          ng          ate
```

```
l          ed          ess
```

```
c          at          tin
```

```
Letra ohizkoena: e  Maiztasuna: 11.7390
```

```
Bokal maiztasuna: 37.5109
```

```
>
```

Horra hor hizkuntza ezberdinen ezaugarri orokorrak konparatzeko tresna. Berrez, programak ez du analisi berririk egiten; aitzitik, aurreko ariketetan erabilitako tresnak biltzen ditu programa bakarrean. Berrikuntza bakarra haxe da: programa honek zuriuneak eta lerro-jauziak ez dituela aintzat hartzen Ngrama-ereduak eraikitzerakoan.

Hona programaren kodea:

lengoaiaAzt.pl

```
#!/usr/bin/perl
use strict;
use warnings;

open(FITX, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia zabaldu\n");

my ($lerro, $hitz, @karak, $kar,
    $aurrekokar, $aurre2kar, $karkop, $bokalkop);
my (%unigram, %bigram, %trigram);
my (@unigramOrd, @bigramOrd, @trigramOrd);

$karkop = 0;
$bokalkop = 0;

while($lerro = <FITX>){
    chomp($lerro);
    # Lerro hasierako eta bukaerako zuriuneak kendu
    $lerro =~ s/^\s*//;
    $lerro =~ s/\s*$//;
    # Letra larriak xehe bihurtu
    $lerro =~ tr/A-Z/a-z/;
    # Letra ez diren ikurrak ezabatu
    $lerro =~ tr/.,;!?"(){}//d;
    # Lerroa karakteretan banatu
    @karak = split(//, $lerro);
    # Hasieratu aldagaiak zuriunearekin
    $kar = " ";
    $aurrekokar = " ";
    $aurre2kar = " ";
    foreach my $i (@karak){
        $karkop++;
        # Aurrekoaren aurreko karakterea gorde
        # $aurre2kar-en
        $aurre2kar = $aurrekokar;
        # Aurreko karakterea gorde $aurrekokar-en
        $aurrekokar = $kar;
        # Uneko karakterea gorde $kar aldagaian
        $kar = $i;
        # Bokalak kontatu
        $bokalkop++ if ($kar =~ /[aeiou]/);
        # unigrama, bigrama eta trigramak eraiki
        # Zuriunerik bada baztertu
```


3. maila

39. kasu praktikoa: fitxategien edukiak konparatu

Hurrengo programak fitxategien edukia konparatzeko balio du. Argumentu gisa bi fitxategi jaso, eta lerro komunak bistaritzen ditu. Programak ez du edukia hitzez hitz edo esaldiz esaldi konparatzen, lerroak hartzen ditu aintzat. Lerroen ordenak ez du zertan bi fitxategietan berbera izanik programak detekta ditzan.

Adibide gisa, demagun `fitx1.txt` eta `fitx2.txt` fitxategiak konparatu nahi ditugula, beraien edukia honakoa izanik:

<i>fitx1.txt</i>	<i>fitx2.txt</i>
arbela	belarria
belarria	haizea
danba	danba
filma	filma
gogorra	
haizea	

Beraien edukia konparatzeko `edukiKonp.pl` programa erabiliz gero, hauxe da itzuliko ligukeena:

```
>perl edukiKonp.pl fitx1.txt fitx2.txt
>fitx2.txt 1 lerroan fitx1.txt 2 lerroan:
belarria
>fitx2.txt 3 lerroan fitx1.txt 3 lerroan:
danba
>fitx2.txt 4 lerroan fitx1.txt 4 lerroan:
filma
>fitx2.txt 2 lerroan fitx1.txt 6 lerroan:
haizea
>
```

Bi fitxategietan errepikatzen diren lerroak soilik bistaritzen ditu, aurretik fitxategi bakoitzean zenbatgarren lerroan agertzen diren ere adieraziz.

Hona hemen `edukiKonp.pl` programaren kodea:

`edukiKonp.pl`

```
#!/usr/local/bin/perl
# Komando-lerrotik bi fitxategi jaso eta lerro komunak
pantailaratu
use warnings;
use strict;
```

```

my ($fitxBat, $fitxBi, $lerro1, $lerro2, $l1, $l2,
    %fitxHash);
$fitxBat = $ARGV[0],
$fitxBi = $ARGV[1];

open(FITX1, $fitxBat) ||
    die("Ezin $fitxBat fitxategia zabaldu\n");
$l1 = 0;
$l2 = 0;
# gorde $fitxBat hash egituran
while ( $lerro1 = <FITX1> ) {
    chomp($lerro1);
    $l1++;
    # gakoa: hitza
    # balioa: agerpenaren lerro zenbakia
    $fitxHash{$lerro1} = $l1;
}

close(FITX1);

open(FITX2, $fitxBi) ||
    die("Ezin $fitxBi fitxategia zabaldu\n");

# $fitxBi lerroz-lerro zeharkatu eta
# lerro bakoitza %fitxHash-en aurrez
# existitzen den begiratu
while( $lerro2 = <FITX2> ) {
    chomp($lerro2);
    $l2++;
    if( exists $fitxHash{$lerro2} ) {
        # lerro errepikatua
        print (">$fitxBat $fitxHash{$lerro2}
            lerroan\t$fitxBi $l2 lerroan:\n");
        print (" $lerro2\n");
    }
}
close(FITX2);

```



2. maila



40. kasu praktikoa: kopia-detektatzailea

Kopiak detektatzen dituen programa garatuko dugu oraingo honetan. Aurreko ariketan bezala, sarrera gisa 2 fitxategi jaso eta haien edukia konparatuko du pro-

gramak. Baina oraingoak ez du konparaketa lerroz lerro gauzatuko. Iruditzen zai-
gu kopiak detektatzeko fitxategien edukia lerroz lerro edo hitzez hitz konparatzea
baino egokiagoa litzatekeela esaldiz esaldi konparatzea. Beraz, konparaziorako
elementu gisa esaldia erabiliko dugu.

Programak bi fitxategi jasoko ditu komando-lerrotik, eta irteeran alfabetikoki
ordenatutako esaldi-zerrenda itzuliko du honako sinboloekin batera:

- **A:** esaldia lehen fitxategian bakarrik agertu da.
- **B:** esaldia bigarren fitxategian bakarrik agertu da.
- **AB:** esaldia bi fitxategietan agertu da.

Adibide gisa, eta programaren funtzionamendua hobeto uler dadin, bi
fitxategi hauen edukia konparatuko dugu:

fitx1.txt

```
Azkeneko gerratea bukatu zan. Txomin donostiarra Ernani-ko  
aldapan gora zaldi-gañean zetorren. An, atzean, gelditu zan  
Ernani ta Ernani-ko kartzela, Txomin ain luzaroan  
egondakoa. Pakearekin askatasuna eman zioten eta Txomin eta  
zaldia alkarrekin ixilik zetozen.
```

fitx2.txt

```
An, atzean, gelditu zan Ernani ta Ernani-ko kartzela,  
Txomin ain luzaroan egondakoa. Pakearekin askatasuna eman  
zioten eta Txomin eta zaldia alkarrekin ixilik zetozen.  
Zaldiak eup egin zuan; nekez bazan ere, aldapa gaitza  
menderatu zuan. Zaldunak eta zaldiak atsedean artu.
```

Testu beraren bi puska dira, eta bi esaldi berdín dituzte. Ea zer dioen gure
programak:

```
>perl kopiaDetekt.pl fitx1.txt fitx2.txt
*****
An, atzean, gelditu zan Ernani ta Ernani-ko kartzela,
Txomin ain luzaroan egondakoa AB

Azkeneko gerratea bukatu zan A

Pakearekin askatasuna eman zioten eta Txomin eta zaldia
alkarrekin ixilik zetozen AB

Txomin donostiarra Ernani-ko aldapan gora zaldi- gañean
zetorren A
```

```
Zaldiak eup egin zuan; nekez ba-zan ere, aldapa gaitza
menderatu zuan B
```

```
Zaldunak eta zaldiak atsedean artu. B
```

```
*****
```

```
>
```

Portaera ulertu ostean, has gaitzen programa nola eraiki pentsatzen. Datu-egiturek pisu handia izango dute ariketa honetan. Gure proposamenak hiru *hash* egitura erabiltzen ditu: fitxategi bakoitzeko esaldiak gordetzeko bi, eta hirugarrena, berriz, esaldi guztiak gordeko dituen:

- `%hash1`: lehen fitxategiko esaldiak gordetzeko.
- `%hash2`: bigarren fitxategiko esaldiak gordetzeko.
- `%hashOrok`: esaldi guztiak, lehen eta bigarren fitxategikoak.

Programaren prozedura orokorra honakoa izango da: lehenbizi sarrerako fitxategiak zabaldu, esalditan banatu eta gorde hauek dagozkien *hash*-etan. Fitxategia zabaldu eta esalditan banatzeko prozesua birritan errepikatu beharko denez, azpiprograma bat sortuko dugu zeregin horretarako: `esalditanBanatu()`. Azpiprograma horrek argumentu bezala fitxategi-izena jaso, eta bere edukia esalditan banatu ondoren, *array* gisa itzuliko du alfabetikoki ordenatuta.

Fitxategietako edukia esalditan banatu eta *hash* egituretan gorde ostean, esaldi guzti-guztiak biltzen dituen `%hashOrok` egiturako elementuak (esaldiak) banan-banan hartu eta honela etiketatuko ditugu:

- Esaldia `%hash1`-en badago? Baiezkoan jarri **A** etiketa.
- Esaldia `%hash2`-n badago? Baiezkoan jarri **B** etiketa.

Esaldia bi testuetan agertzen bada, **AB** etiketa eramango du, eta bestela, dagokion testuarena.

Hona azaldu berri dugun prozedura kode bihurtua:

kopiaDetekt.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;
my (%hash1, %hash2, %hashOrok, @esaldiak1, @esaldiak2);

# lehen fitxategia esalditan banatu
@esaldiak1 = esalditanBanatu($ARGV[0]);
```



```

# Esaldi bakoitza prozesatu
foreach my $esaldi (@esaldiak1) {
    # Lehen fitxategiko esaldiak biltzen dituen hash-a
    $hash1{$esaldi} = 1;
    # Esaldi guztiak biltzen dituen hash-a
    $hashOrok{$esaldi} = 1;
}

# bigarren fitxategia esalditan banatu
@esaldiak2 = esalditanBanatu($ARGV[1]);
# Esaldi bakoitza prozesatu
foreach my $esaldi (@esaldiak2) {
    # Bigarren fitxategiko esaldiak biltzen dituen hash-a
    $hash2{$esaldi} = 1;
    # Esaldi guztiak biltzen dituen hash-a
    $hashOrok{$esaldi} = 1;
}
# Esaldi guztiak alfabetikoki ordenatu
# banan-banan tratatzeko
print ("*****\n");
foreach my $elem (sort(keys(%hashOrok))) {
    print ("$elem\t");
    # $elem agertu da lehen lerroan?
    if(exists($hash1{$elem})) {
        print("A");
    }
    # $elem agertu da bigarren lerroan?
    if(exists($hash2{$elem})) {
        print("B");
    }
    print ("\n\n");
}
print ("*****\n");

sub esalditanBanatu {
    my $fitxIzen = shift(); # argumentua: fitxategi-izena
    open(FITX, $fitxIzen) ||
        die("Ezin $fitxIzen fitxategia zabaldu\n");
    my $testua = "";
    my $lerro;
    while ($lerro = <FITX>) {
        # Ordezkatu lerro jauziak zuriuneengatik
        $lerro =~ s/\n$/ /g;
        # Kateatu lerroa $testua aldagaiari
        $testua.= $lerro;
    }
}

```

```
close(FITX);
# $testua string-a esalditan banatu
my @esaldiak = split(/\.\.?!\]+\s+/, $testua);
return @esaldiak;
}
```



2. maila



41. kasu praktikoa: hitz propioak eta komunak

Aurreko ariketaren haritik tiraka, aplikazio berri hau bururatu zaigu: komando-lerrotik bi fitxategi jaso eta fitxategi bakoitzaren hitz propioak bistaratzen dituen programa. Eta zeintzuk dira hitz propioak? Fitxategi horretan soilik agertzen direnak, komunak ez direnak.

Hobeto uler dadin, demagun `egaña.txt` fitxategian Andoni Egaña bertsolariak amodioaren inguruan kantatutako bertso guztiak dauzkagula, eta `maia.txt` fitxategian Jon Maiarenak. Gure nahia, maitasunari buruz kantatzean bertsolari bakoitzak erabiltzen dituen hitz propioak topatzea da, hau da, gai horri buruz kantatzean berak bakarrik erabili dituen hitzak.

Hitz propioak topatzeko gure programarekin erraza litzateke aurreko ataza egitea:

```
>perl hitzPropio.pl egaña.txt maia.txt
egaña.txt-ren hitz propioak:
amorio
moñoña
...
maia.txt-ren hitz propioak:
muxutxue
...
>
```

Programak 2 fitxategien edukia hitzez hitz konparatuko du, eta 2 fitxategietan errepikatzen ez diren haiek soilik itzuli. Modu horretan gai beraren inguruan bertsolari bakoitzak, berak bakarrik, erabili dituen hitzak zeintzuk diren jakin ahalko dugu.

Programaren prozedura orokorra aurrekoaren antzekoa da, bi ezberdintasunekin: programa honek ez du testua esaldi esaldi prozesatzen, hitzez hitz baizik. *Hash*-aren gakoak oaringo honetan hitzak izango dira. Bigarren ezberdintasuna datuak bistaratzean ageri da: aurreko programan esaldi guztiak bistartzen genituen, lehenengo fitxategian (A), bigarreanean (B) edo bietan (AB) agertzen zen zehaztuz. Oraingoan, fitxategi bakoitzaren hitz propioak soilik bistartu nahi ditugunez, A etiketa daramatenak eta B etiketa daramatenak bistartuko ditugu, hitz komunak (AB etiketa daramatenak) baztertuz.

Hona programa:

hitzPropioak.pl

```
#!/usr/bin/perl
use strict; use warnings;

my ($lerro,%hash1, %hash2, %hashOrok, @hitzak1, @hitzak2);
my (@lehen, @bigarren);

open (FITX1, $ARGV[0]) ||
    die("Ezin $ARGV[0] fitxategia ireki\n");

while ($lerro = <FITX1>) {
    chomp($lerro);
    # Lerroa hitzetan banatu
    @hitzak1 = split (/[\s\.,;:\?!\+]/, $lerro);

    foreach my $hitz (@hitzak1) { # Hitz bakoitza prozesatu
        # Lehen lerroko hitzak biltzen dituen hash-a
        $hash1{$hitz} = 1;
        # Hitz guztiak biltzen dituen hash-a
        $hashOrok{$hitz} = "A";
    }
}
close(FITX1);
open (FITX2, $ARGV[1]) || die("Ezin $ARGV[1] ireki\n");
while ($lerro = <FITX2>) {
    chomp($lerro);
    @hitzak2 = split (/[\s\.,;:\?!\+]/, $lerro);
    foreach my $hitz (@hitzak2) {
        # Bigarren lerroko hitzak dituen hash-a
        $hash2{$hitz} = 1;
        if(exists($hashOrok{$hitz})) {
            $hashOrok{$hitz} = "AB"; }
        else { $hashOrok{$hitz} = "B"; }
    }
}
```

```

close(FITX2);

foreach my $elem (sort(keys(%hashOrok))) {
    if ($hashOrok{$elem} eq "A") {
        push(@lehen, $elem);
    }
    elsif ($hashOrok{$elem} eq "B") {
        push(@bigarren, $elem);
    }
}

print ("*****\n");
print("$ARGV[0]-ren hitz propioak:\n");
foreach my $elem (@lehen) {
    print("$elem\n");
}
print ("*****\n\n");
print("$ARGV[1]-ren hitz propioak :\n");
foreach my $elem (@bigarren) {
    print("$elem\n");
}
print ("*****\n");

```

9.5. ERREFERENTZIAK

1. maila

42. kasu praktikoa: indize onomastikoa

Ikerketa-lan batean gabilta buru-belarri. Bertan, besteak beste, liburuetako pertsonaiak aztertu behar ditugu: beraien izena, zenbat aldiz agertzen diren liburuan, eta beren aipamenak non agertzen diren. Lana errazteko asmoz, indize onomastikoak egiteko Perl programa bat garatzea bururatu zaigu.

Hona programaren portaera erakusten duen exekuzio-adibidea:

```
>perl indOnomastikoa.pl anabitarteDonostia.txt
...
Anita
61, 67, 248, 747, 766, 1112
Anton
106, 118, 127, 129, 129, 130, 132, 146, 151, 158, 160, 214,
220, 242, 251, 453, 517, 677, 714, 716, 718, 1109, 1114,
1115, 1144, 1144, 1156, 1198, 1255
Antton
138, 220, 221, 222, 438, 529, 670, 670, 670, 672, 677, 710,
737, 740, 1023, 1111, 1123, 1124, 1124, 1125, 1135, 1135,
1135, 1141, 1142, 1142, 1142, 1143, 1145, 1145, 1146, 1174,
1200, 1200, 1210, 1256
...
>
```

Pertsonaia bakoitzaren agerpenak kontatzeko, egokiena *hash* egitura erabil-tzea dela erabaki dugu: gakoak izen bereziak izango dira, eta balioak, berriz, izen horiek agertzen diren lerro-zenbakiak. Izan berbera behin baino gehiagotan ager liteke testuan, eta gure egiturak agerpen guztiak gorde behar ditu bata bestearen atzetik. Komeni zaigu agerpen guztiak *array* batean gordetzea, ondoren kontsul-tatu, ordenatu, kontatu, etab. nahi badugu ere. Erreferentzia erabiliz egin dezakegu hori: *hash* egituraren gako gisa, izen bereziak erabiliko ditugu, eta balio gisa, berriz, *array* baterako erreferentzia. Erreferentziatutako *array*-ak izen bereziaren agerpen guztien zerrenda gordeko du.

Baina programak badauka beste koska bat ere: nola bereizi izen bereziak gainerakoetatik? Bai, izen bereziak aurretik puntuazio-markarik izan ez eta letra larriz hasten diren hitzak dira. Ados, irizpide hori erabiliko dugu, nahiz erabat zu-zena izan ez; esaldi-hasierako hitz bereziak baztertu egingo ditugu gure hurbilpe-narekin. Adib.: «Jonek ez zuen aurpegi onik jarri». Horrez gain, arazo praktiko bat ere planteatzen du, fitxategiak lerroz lerro prozesatu ohi ditugunez, lerroko lehen hitza letra larriz hasten bada, nola jakin hitz berezia den ala ez? Aurreko lerroko azken karakterea puntuazio-marka den ala ez begiratu beharko genuke, eta horrelakorik ez dugu orain artean egin.

Hona gure proposamena: aldagai berezi bat erabiliko dugu, *\$puntuAurretik*, letra larriz hasten den hitz batekin topo egiten dugunean, bere aurretik puntuazio-markarik izan den ala ez jakiteko. Hau da, puntuazio-marka aurkitzen dugun aldiro *\$puntuAurretik* aldagaiari bat balioa esleituko diogu, ondoren etorriko den hitza letra larriz hasi arren, ez dela hitz berezia jakin dezagun.

Gure ideia Perl lengoaiara itzulita:

indOnomastikoa.pl

```
#!/usr/bin/perl
use warnings;
use strict;

open(FITX, $ARGV[0]) or
    die("Ezin $ARGV[0] fitxategia zabaldu!\n");

my $lerro;
my %indizea;
my $puntAurretik = 1;
my @agerpen;

while ($lerro = <FITX>) {
    while ($lerro =~ /([A-Z]\w+|[\.\?!\])/g){
        # bat egitea larriz hasitako hitzak?
        If ($1 =~ /([A-Z]\w+)/) {
            if ($puntAurretik == 0) {
                # %indizea hash-ean $1 gakoa existitzen ez
                # bada, sortu gako horrentzako
                # izengabeko array baten erref.
                $indizea{$1} = []
                if (!exists($indizea{$1}));
                # Gehitu lerro zenbakia
                # erreferentziatutako array-aren amaieran
                push(@{$indizea{$1}}, $.);
                # $. ->uneko lerro zenbakia
            }
            $puntAurretik = 0;
        }
        else { # bat egitea puntuazio-ikurrak
            $puntAurretik = 1;
        }
    }
}

close(FITX);
```

```
# %indizea egiturako gakoak ordenatuta lortu
# eta begizta bidez banan-banan pantailaratu
foreach my $izen (sort(keys(%indizea))) {
    print("$izen\n");
    # $izen pertsonaiaren array-a atzitu eta @agerpen
    # array-an gorde
    @agerpen = @{$indizea{$izen}};
    # @agerpen string-ean bildu eta pantailan bistaratu
    print(join(" ", @agerpen));
    print("\n");
}
```

9.6. ARIKETAK HIZTEGIAREKIN

Atal honetan, hiztegiekin lan egiten duten programak garatuko ditugu. *EH_hiztek.txt* fitxategian Ibon Sarasolaren Euskal Hiztegiko zatitxo bat daukagu. Lerro bakoitzak hiztegiko sarrera baten definizioa dauka, tabuladorez banatutako honako lau eremuekin: sarrera-hitza, kategoria, adiera-zenbakia eta definizioa.

Hona hiztegiaren puska bat adibide gisa:

a	iz	H1.	Euskal alfabetoko lehen letra; letra horren izena. A larria.
a	interj	H2.	Oi.
aba	iz		Elizgizonen titulua, izen-deituren ondoan ezartzen dena.
ababor	iz		Ontziaren ezkerraldea, brankara begiratzuz.
abade	iz	A2.	Gizonezkoentzako monasterio bateko burua, apaizteko esku duena.
abadegai	iz		Apaizgaia.
abadesa	iz		Emakumezkoentzako monasterio bateko burua.

Ariketa guztiak *EH_hiztek.txt* hiztegia oinarri hartuta egingo ditugu: hitz bat eman eta definizioak topatu hiztegian, kategoria jakin bateko sarrerak bistaratu, azpikate jakin bat daukaten sarrerak, etab.

43. kasu praktikoa: definizioa topatu hiztegian

Programa honek, sarrera gisa hitz bat eman eta bere definizioa topatzen du *EH_hiztek.txt* hiztegian. Bilaketa berri bat egin nahi dugun bakoitzean programa abiarazten ibil ez gaitezen, nahi adina bilaketa egin ahal izango ditugu, programa amaitzeko "q" letra sakatzea nahikoa izango delarik.

Hona programaren exekuzio-adibidea:

```
>perl EhDefinizio.pl
Idatzi bilaketa terminoa. Amaitzeko sakatu q: abade
abade:      Gizonezkoentzako monasterio bateko burua,
apaizteko esku duena.

Idatzi bilaketa terminoa. Amaitzeko sakatu q: ai
ai:      Atsekabea, oinazea... adierazteko erabiltzen den
hitza.

Idatzi bilaketa terminoa. Amaitzeko sakatu q: q
>
```

Programak ez du sarrerako datu gisa fitxategi-izena jasotzen, orain artean bezala. Zuzenean `EH_hiztek.txt` hiztegia zabaltzen du. Ustekabeko errorerik izan ez dadin, programa exekutatzen dugun direktorio edo karpeta berean izan behar dugu hiztegia ere.

Programaren eskema orokorra:

1. `Eh_hiztek.txt` zabaldu eta gorde `@hiztegia` array-an
2. Bilaketa-terminoa jaso teklatutik: `$bilatu`
3. `while ($bilatu ne "q")`
 - 3.1. Banan-banan hiztegiko sarrerak bisitatu


```
foreach $sarrera (@hiztegia)
  3.1.1. Sarrerako eremuak gorde array batean: @eremuak
  3.1.2. Baldin ($bilatu eq $eremuak[0])
    Bistaratu definizioa: $eremuak[3]
```
 - 3.2. Bilaketa-terminoa jaso teklatutik: `$bilatu`

Eta hona jarraian aurreko eskemari jarraitzen dion programa:

EhDefinizio.pl

```
#!/usr/bin/perl
# Sarerra: bilaketa terminoa
# Irteera: bere defizioak
use strict;
use warnings;

my (@hiztegia, @eremuak, $lerro, $bilatu);
open(FI, "EH_hiztek.txt") ||
  die ("Ezin EH_hiztek.txt fitxategia zabaldu!\n");
```



```

@hiztegia = <FI>;
close (FI);

print("Idatzi bilaketa terminoa. Amaitzeko sakatu q: ");
$bilatu = <STDIN>;
chomp($bilatu);

while ($bilatu ne "q") {
    # hiztegia irakurri
    foreach my $sarrera (@hiztegia) {
        # sarrerari dagozkion eremuak gorde array-an
        @eremuak = split(/\t/, $sarrera);
        if ($eremuak[0] =~ /^$bilatu$/) {
            # Bistaratu sarrera hitza eta bere definizioa
            print "$eremuak[0]:      ";
            print "$eremuak[3]\n";
        }
    }
    print("Idatzi bilaketa terminoa. Amaitzeko sakatu q: ");
    $bilatu = <STDIN>;
    chomp($bilatu);
}

```

44. kasu praktikoa: atzizki bidezko bilaketa

Aurreko programa moldatuko dugu, oraingo honek bilaketa-termino gisa atzizki bat eskatuko digu, eta atzizki hori duten hiztegiko sarrerak itzuli beren definizioekin batera. Bilaketa bakoitzaren amaieran, topatutako sarrera kopurua ere bistaratuko du programak.

Hona hemen, adibide gisa, exekuzio-dei bat:

```

>perl EhAtzizki.pl
Idatzi bilaketarako atzizkia. Amaitzeko sakatu q: tasun
Hirutasun:      Kristau dotrinan, Jainko bakarraren hiru
pertsonek batasunaren dogma eta ezkutukia; hiru pertsonako
Jainkoa bakar hori.

abadetasun:      Abadetza, abadearen kargua eta egitekoa.

...
zuzentasun:      Zuzena denaren nolakotasuna.
914 sarrera tasun atzizkiarekin bat egiten dutenak
Idatzi bilaketarako atzizkia. Amaitzeko sakatu q: q
>

```

Programaren kodea:

EhAtzizki.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my (@hiztegia, @eremuak, $lerro, $atzizki, $kont);

open(FI, "EH_hiztek.txt") ||
    die ("Ezin EH_hiztek.txt fitxategia zabaldu!\n");
@hiztegia = <FI>;
close (FI);

print("Idatzi atzizkia. Amaitzeko sakatu q: ");
$atzizki = <STDIN>;
chomp($atzizki);

while ($atzizki ne "q") {
    $kont = 0;
    # Sarrerari dagozkion eremuak gorde array-an
    foreach my $sarrera (@hiztegia) {
        @eremuak = split(/\t/, $sarrera);
        if ($eremuak[0] =~ /$atzizki$/) {
            $kont++;
            # Bistaratu sarrera hitza eta bere definizioa
            print "$eremuak[0]:      ";
            print "$eremuak[3]\n";
        }
    }

    print("$kont sarrera $atzizki atzizkiarekin bat
          egiten dutenak\n");
    print("Idatzi atzizkia. Amaitzeko sakatu q: ");
    $atzizki = <STDIN>;
    chomp($atzizki);
}
```

45. kasu praktikoa: atzizkidun definizioak aztertzen

Atzizkiak hiztegian nola dauden definituak aztertzen gabiltza. Zehazki, atzizkiak definitzerakoan gehien erabiltzen diren hitzak zein diren jakin nahi dugu. Horretarako, honako programa garatzea otu zaigu: erabiltzaileari atzizki bat idazteko eskatu, eta zehaztutako atzizkia duten hiztegiko sarrera guztien definizioak itzultzen dituen. Baina definizioan erabilitako hitzak maiztasunaren arabera

ordenatuta bistaratuko ditu programak. Hau da, atzizki bat eman eta atzizki hori duten sarreraren definizioetan erabili diren hitzak itzuliko ditu programak maiztasunaren arabera ordenaturik.

Honako honetan, exekuzio-aldi bakoitzean bilaketa bakarra egingo du programak. Horretaz gain, programaren irteera pantailan osorik ikusteko baino handiagoa izango denez kasu askotan, atzizkia komando-lerroaren bitartez pasatuko diogu programari. Modu honetan programaren irteera fitxategi batera birbidera dezakegu eta ondoren bistaratu.

Adibidez:

```
>perl EhAtzizkiMaiz.pl tasun > tasun.txt
```

Programaren exekuzioak, “tasun” atzizkia daukaten sarrerak topatuko ditu, eta beren definizioetan erabilitako hitzak maiztasunaren arabera ordenatuta itzuliko `tasun.txt` fitxategian. Hona exekuzioaren osteko edukia:

tasun.txt

```
nolakotasuna    491
denaren          406
edo             110
egoera           87
dagoenaren      44
eza              31
...
ditzakeena      1
baitara         1
```

`EhAtzizkiMaiz.pl` programaren kodea:

EhAtzizkiMaiz.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my (@eremuak, $lerro, $atzizki, $kategoria, $hitz, %maiz);

open(FI, "EH_hiztek.txt") ||
    die ("Ezin fitxategia zabaldu!\n");

# irakurri atzizkia
$atzizki = $ARGV[0];
```

```

while ($lerro = <FI>) {
    chomp($lerro);
    # Sarrerari dagozkion eremuak gorde array-an
    @eremuak = split(/\t/, $lerro);
    # $atzizki atzizkia eta $kategoria kategoriako
    # sarrerak hautatu
    if ($eremuak[0] =~ /$atzizki$/) {
        # definizio eremua hitzetan banatu
        foreach $hitz (split(/\W+/, $eremuak[3])) {
            # hitz bakoitzaren maiztasuna gorde
            $maiz{$hitz}++;
        }
    }
}
close (FI);
# %maiz egiturako hitzak ordenatu
# maiztasunaren arabera, handienetik txikienera
# ondoren hitz bakoitza eta bere agerpen kop. bistaratu
foreach $hitz (sort({$maiz{$b} <=> $maiz{$a}} (keys
%maiz))) {
    print (" $hitz\t$maiz{$hitz}\n");
}

```

46. kasu praktikoa: definizio zirkularrak

Definizio zirkularra deitu ohi zaio definitu nahi den terminoa definizioan bertan erabiltzen denean. `EhZirk.pl` programak hiztegia lerroz lerro irakurri, eta definizio zirkularrik topatzean, dagokion sarrera-hitza eta definizioa bera bistaratu tuko ditu.

Ba ote gure hiztegian definizio zirkularrik?

```
>perl EhZirk.pl
```

Egin proba, eta zeuek esan! Programaren kodea:

EhZirk.pl

```

#!/usr/bin/perl
use strict;
use warnings;
# Hiztegia, zenbat definiziook dute definitzen ari den
# hitza ?
my (@a, $l);
open(FI, "EH_hiztek.txt") ||
    die ("Ezin fitxategia zabaldu!\n");

```

```

while ($1 = <FI>) {
    chomp($1);
    # eremu banatzaileak tabuladoreak dira
    @a = split(/\t/, $1);
    # begiratu sarrera hitza definizioan ere ageri den.
    # Sarrerako hitza aurrizki gisa erabiliko da
    # bilaketan:
    # "ama" hitzak ez dut bat egingo "narama" hitzarekin,
    # bai ordea "amaren" edo "amatxo" hitzekin. Dena dela,
    # modu honetan "ama" hitza ez du aurkituko "hainbeste
    # amen etxetan" testuan. Horretarako lematizazioa
    # behar da!
    if ($a[3] =~ /\b$a[0]/) {
        print "$1\n";
    }
}
close (FI);

```

9.7. INTERNET CORPUS GISA

Orain arteko gure programek kanpotik datuak jasotzeko bi bide izan dituzte: bata, teklatu bidez erabiltzaileak datuak zuzenean sartzea, datu kopurua txikia denean erabili izan duguna. Bestea, berriz, prozesatu beharreko datu kopurua handia denean, fitxategiak erabiltzea programak elikatzeke. Hala ere, dudarik ez da gaur egun arlo askotan informazio-iturririk aberatsena eta eguneratuena Internet dela. Gure programek sarea atzitu ahal izango balute, *corpus* zoragarria izango genuke esperimendu eta datu-bilketetarako. Horixe da hain zuzen ere hurrengo ariketetan landuko duguna: gure programetatik nola atzitu dezakegun Internet, eta nola erabili bertako informazioa gure helburuetarako. Gai potoloa da inondik ere, eta oinarri-oinarrizko bi kasu praktiko baino ez ditugu azalduko, egin daitekeenaren erakusgarri txiki bat.

47. kasu praktikoa: web orriak atzitu

Datozen adibideetan modulu berri bat erabiliko dugu gure programetan: `LWP::Simple` modulua. Eskaintzen duen `get()` funtzioa oso erabilgarria da: webgune baten helbidea eman eta bere edukia itzultzen du.

Ondoren datorren programak web helbide bat eskatu, eta helbide horren edukia pantailaratuko du `get()` funtzioa erabiliz. Web orriak *html* formatuan idatzirik daude, eta halaxe pantailaratuko ditugu.

Hona programaren erabilera-adibidea:

```
>perl url.pl
Idatzi URL-a: www.berria.info
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="eu"
xml:lang="eu">
<head>
<title>Berria.info</title>
....
>
```

Irakurtezina, ezta? Izan ere, ez dago guk ulertzeko idatzia, nabigatzaileak ulertu dezan idatzia dago, HTML lengoaian. Interneten argitaratzen diren dokumentuei itxura emateko erabiltzen den lengoia da HTML. Etiketetan oinarritutako lengoia da, eta dokumentu ezberdinak ondo egituratuta, eta web orri batetik besterako estekekin argitaratzea posible egiten du.

Hona programaren kodea:

url.pl

```
#!/usr/local/bin/perl
use warnings;
use strict;
# Web orriak atzitzeko behar dugun modulua
use LWP::Simple;
my ($URL, $edukia);
print ("Idatzi URL-a: ");
$URL = <STDIN>;
chomp($URL);
# Teklatu bidez idatzitako helbideari gehitu protokoloa
$URL = "http://" . $URL;
# Web orria eskuratu
$edukia = get($URL);
# Pantailan bistaratu edukia
print ($edukia);
```

1. maila

48. kasu praktikoa: bilaketak sarean

Komando-lerroaren bitartez bi bilaketa-patroi eman, eta patroik bakoitzaren agerpen kopurua kalkulatzen du segidan datorren programak. Aurretik ere egin ditugu antzekoak, baina programa honen berritasuna fitxategi bat miatu beharrean teklatu bidez helbidea idatzi eta web eta bertatik atzitu daitezkeenak miatzean datza.

Hona hemen, programaren erabilera posible bat:

```
>perl agerpenWeb.pl a e
Ze helbidetan begiratu behar dut?
www.erabili.com
a agerpen kopurua: 13807
e agerpen kopurua: 11015
>
```

Programa honek erabilera ugari ditu, besteak beste hiztegi edo kontsultarako corpus gisa erabil genezake sarea. Adibidez, nola idatzi behar da : "handi" edo "haundi"?

```
>perl agerpenWeb.pl haundi handi
Ze helbidetan begiratu behar dut?
www.berria.info
haundi agerpen kopurua: 0
handi agerpen kopurua: 28
>
```

Garbi asko dago kasu honetan zein den zuzena.

Programaren prozedura orokorra honakoa da: teklatu bidez adierazitako helbidea jaso eta web orri horretatik abiatuko du bere bilaketa. Bi *array* erabiliko ditu: bata oraindik bisitatu gabeko web helbideak gordeko dituen, eta bestea dagoe-neko bisitatutako web helbideak gordeko dituen. Bigarren *array* horren funtzioa web orri bera behin bakarrik bisitatuko dugula ziurtatzea da.

Adierazitako helbidetik abiatu eta 2 pausoko prozesua aplikatuko du programak: lehenbizikoa loturak ateratzea; web orriek beste orrialdeetarako loturak eduki ohi dituzte. Horiek *array* batean gordeko ditugu aurrerago aztertu ahal izateko. Bigarrena, web orriaren testua aztertu eta komando-lerro bidez pasatutako patroien bat-egiteak zenbatzea.

Aurreko prozesua begizta batean sartuko dugu, iterazio bakoitzean bisitatu gabeko web orri bat prozesatu ahal izateko. Web orri baten edukia jaitsi baino lehen, aurretik bisitatua izan den ala ez egiaztatu behar da. Azpiprograma bat erabiliko dugu zeregin horretarako.

Web orri bat jaitsi, estekak atera eta testua aztertzeak bere denbora behar du, eta hargatik hamarrera mugatu dugu *link* edo esteka kopurua.

Hona programa bera:

agerpenWeb.pl

```
#!/usr/bin/perl
use warnings;
use strict;
use LWP::Simple;

print("Ze helbidetan begiratu behar dut?\n");
my $html = <STDIN>;
chomp($html);

$html = "http://" . $html;

# Jaso bilaketa-patroiak
my $sarBat = $ARGV[0];
my $sarBi = $ARGV[1];

my $agerpenBat = 0;
my $agerpenBi = 0;
my ($weba, @linkak);

# Bisitautako web orrien helbideak
my @bisitatuak;

# Bisitatu gabeko web orrien helbideak
my @bisiGabeak;

# Esteka kopuru maximoa
my $linkMax = 10;

# Bisitatutako link edo web orri kopurua
my $linkop = 0;

# Gehitu hasierako helbidea bisitatu gabekoen taldeari
push(@bisiGabeak, $html);
```



```
# Bisitatu gabeko helbiderik gertatzen den artean eta
# esteka kopuru maximora iritsi bitartean

while ($#bisiGabeak > -1 and $linkop < $linkMax) {
    $html = shift(@bisiGabeak);

    # array-ko lehen helbidea hartu.
    # bisitatua dagoeneko?
    if (bisitatua($html) == 0) {
        $linkop++;
        # lortu web orriaren edukia
        $weba = get($html);
        # web orriko estekak atera eta gorde
        @linkak = linkak_atera($weba);
        # Gehitu estekak bisitatu gabekoen zerrendara
        push(@bisiGabeak, @linkak);

        $linkop++;
        while ($weba =~ /($sarBat|$sarBi)/gi) {
            # Adierazpen erregularraren aukerak:
            # g globala (agerpen guztiak)
            # i letra larri/xeheak ez bereizi

            # bat egitea $sarrera1 edo $sarrera2-k
            # eragin du?
            if ($1 eq $sarBat) {
                $agerpenBat++;
            }
            if ($1 eq $sarBi) {
                $agerpenBi++;
            }
        }

        # Uneko orria bisitatuaren zerrendara pasa
        push(@bisitatuak, $html);
    }
}

print("$sarBat agerpen kopurua: $agerpenBat\n");
print("$sarBi agerpen kopurua: $agerpenBi");
```

```

# Azpiprograma
# Helbidea dagoeneko bisitatu badugu 1 itzuliko du, 0
# bestela
sub bisitatu {
    my $helb = shift();
    foreach my $i (@bisitatuak) {
        # kontuz, aldagai orokorra erabiltzen ari gara
        if ($i eq $helb) {
            return 1;
        }
    }
    return 0;
}

# Argumentu bezala pasatako web orriaren estekak atera
# eta array batean itzuli
sub linkak_atera {
    my $orri = shift();
    my @lerroak = split(/\n/, $orri);
    my @URL;
    foreach my $lerro (@lerroak) {
        if ($lerro =~ /<a href="(http:[^"]+)"(.*)>(.*)<\a>/) {
            if ($1 !~ /(jpg|gif|png|bmp|tiff|pdf|doc)/) {
                # Ez dadila ez irudia ezta dokumentua izan
                push(@URL, $1);
            }
        }
    }
    return @URL;
}

```



10. Amaiera da hasiera

Hementxe amaitzen da liburua. Ez esan beharreko guztiak esan ditugulako, ezta gutxiagorik ere, bidearen hasiera erakustea besterik ez baitugu egin. Baina hasieratik garbi utzi dugu ez zela gure asmoa Perl lengoaiaren zirrikitu guztiak miatzea, bere mamia ezagutaraztea baizik; eta batez ere testu digitalen tratamendurako nola erabili erakustea, bide batez euskara eta euskal kulturaren edukietara egokituz. Orain, aurrerako bidean lagungarri izan ditzakezun erreferentziak ematea besterik ez zaigu gelditzen.

Perl lengoaia oso erabilia da hizkuntzalaritzan eta testuen prozesamenduan ez bakarrik adierazpen erregularrak idatzi eta erabiltzeko erraztasunagatik, baita dagoeneko programa asko daudelako ere Perl lengoaian idatziak eginda edozeinek eskuratu eta erabiltzeko moduan. Hartu tarte bat eta aztertu CPAN modulu-biltegia, modulu erabilgarri mordoa topatuko duzu-eta bertan.

Irakurri kodea beldurrik gabe. Aztertu ataza bera ebazteko erabilitako estrategiak, baita bakoitza Perl lengoaian nola izan den inplementatua ere.

Dokumentazioa. Perl-ek erabiltzaile-komunitate oso aktiboa dauka, eta nahi beste dokumentazio aurki genezake, bai liburuetan baita Interneten ere. Ge-hiago jakin nahi duenarentzat, lerro hauen azpialdean liburuen eta web helbideen zerrenda idatzi dugu.

Praktikatu ahalik eta gehien. Ez dago hori bezalakorik.

Noraino iritsi nahi duzun, hainbeste lan izango duzu egiteko. Probatu bide guztiak, ondoren erakutsiko dizkizuegunak eta gehiago. Eta ahal bada, gozatu ikasten duzunarekin.

10.1. DOKUMENTAZIOA

Perl-ek dokumentazio iturri oparoa dakar berarekin (ingelesez, hori bai). Windows erabiltzaileek, *ActivePerl* banaketa dutenak, dokumentazioa *html* formatuan atzitu dezaketen arren, ohiko bidea komando-lerroa eta `perldoc` funtzioa erabiltzea da. Informazio erabilgarria eskain diezaguke komando honek. Hona labur-labur nola erabil dezakegun:

```
>perldoc perl
```

Komando-lerroan aurrekoa idatziz gero, Perl-i buruzko informazioa itzuliko digu `perldoc` komandoak: lengoaiari buruzko oinarritzko kontu batzuk, eta dokumentazioaren eduki-taula.

Dena den, `perldoc` komandoaren erabilera ohikoena hauxe da: `perldoc -f fun`, non `fun` Perl komando edo funtzioa den. Adibidez, `chomp` funtzioari buruzko informazioa nahiko bagenu:

```
>perldoc -f chomp
```

Funtzio baten erabilera edo argumentu kopurua ezagutzen ez dugunean, berehala jakin dezakegu `perldoc` erabiliz. Esku-eskura dugun tresna honek informazio zehatza eskainiko digu.

10.2. LIBURUAK

Liburu mordo bat idatzi dira dagoeneko Perl-i buruz. Aipagarrienak iruditzen zaizkigun gutxi batzuk baino ez ditugu hemen aipatuko:

Schwartz, R. L. eta Phoenix, T. (2008): *Learning Perl (5th edition)*, O'Reilly Press. Perl lengoaiaren ikasketan hasiberria denarentzat abiapuntu egokia.

Wall, L.; Christiansen, T. eta Orwant, J. (2000): *Programming Perl (3rd edition)*, O'Reilly and Associates. Primerako erreferentzia-liburua, nahiz eta hasiberrientzako gogor samarra gerta litekeen.

Hammond, M. (2003): *Programming for Linguist*, Blackwell Publishing. Liburu zoragarria, Perl lengoaiaren oinarriak erakusten ditu hizkuntzaren azterketarako aplikazioetan indar eginez.

Christiansen, T. eta Torkington, N. (2003): *The Perl Cookbook*, O'Reilly and Associates. Errezeta-liburua, programazio-ataza andana eta hauen soluzio ezberdinak biltzen dituena. Programatzen esperientzia duenarentzat egokia, adibide praktikoz hornitua.

10.3. SAREA

Informazio andana topatuko dugu Perl-i buruz Interneten. Komunitate aktiboa eta lagunkoia da Perl-ena, eta sarean topa dezakegun dokumentazioaz gain, erabil-tzaile eta programatzaileen posta-zerrendetan beti izango da norbait gure programazio-arazoekin laguntzeko prest.

<http://www.cpan.org>

“Comprehensive Perl Archive Network” Perl-en idatzitako modulu guztiak biltzen dituen gunea. Dokumentazioa eta bestelako informazio erabilgarri ugari ere topa daiteke bertan.

<http://www.perl.org>

Perl-en direktorioa, denetarik aurki daiteke bertan.

<http://www.perl.com>

O'Reilly sarearen Perl gunea, informazio erabilgarri mordoarekin.

<http://learn.perl.org>

Perl ikasteko baliabideei buruzko erreferentziak: liburuak, tutorialak, erabiltzaile-taldeak, etab.

<http://www.perlmonks.org>

Perl erabiltzaileen elkartea.

11. Kasu praktikoen aurkibidea

Moneta-bihurtzailea	46
Alfabetoarekin jolasean	52
Produktu-sailkatzailea	62
Biderketa-aula	66
Hitz-zerrenda ordenatu	76
Ezetz asmatu!	77
Pasahitza berreskuratzeko sistema	80
Hitzen karaktere kopurua	83
Bistaratu lehen N lerroak	101
Bistaratu azken N lerroak	103
Bilaketak testuan	124
Perlsolaritza	127
Gurutzegramak	132
Hitz-egiturak	136
Nola hasi hala amaitu	144
N. bat-egitea aurkitu	150
Esku bakarrarekin idatzitako hitzak	151
Bilatu eta ordeztu	156
Lehen eta azken hitzak trukatu	157
rot13 zifratze-metodoa	160
Testua hitzetan banatu	163
N karaktereko hitzak	164
Palindromoak	167
Urkatuaren jokoak	169
Hitzen agerpen kopurua testuan	178
Sailkatu hitzak luzeraren arabera	195
Zuzentzaile ortografikoa	209
Datuak grafikoki bistaratu	213

Hitz laguntzaileak	216
Karaktere erabiliena	217
Hitza bere testuinguruan	219
Karaktere eta hitz errepikatuak	220
Testua esalditan banatu	222
Ordezkapen anitz	223
Karaktereen maiztasunak testuan	225
Hitzen maiztasunak testuan	228
Bigrama-ereduak	229
Unigrama, bigrama eta trigramak	233
Fitxategien edukiak konparatu	237
Kopia-detektatzailea	238
Hitz propioak eta komunak	242
Indize onomastikoa	244
Definizioa topatu hiztegian	247
Atzizki bidezko bilaketa	249
Atzizkidun definizioak aztertzen	250
Definizio zirkularrak	252
Web orriak atzitu	253
Bilaketak sarean	254

12. Indize alfabetikoa

A		espresio logikoa	54
ActivePerl	25	exists	176
adierazpen erregularrak	121	F	
agindu-blokea	54	fitxategiak	95
aldagai bereziak	113	flag	150
aldagaiak	42	for	69
argumentuak	185	foreach	67
ARGV	92	funtzio-deiak	184
<i>array</i>	49, 72	funtzioaren gorputza	186
atzera begirako erreferentziak	139	G	
azpiprogramak	183	g (globala, agerpen guztiak)	150
B		ge	55
baldintzazko egiturak	54	get	253
begizta habiaratu	81	gt	55
begizta infinitua	65	H	
begiztak	63	hash	173
bektorea	49	html	253
birbideratu	94	I	
C		i (letra larri/xeheak ez bereizi)	123
chomp	49	if	54
close	98	if-else	59
cmp (konparaketa eragilea)	74	if-elsif	60
corpus	233, 259	int	78
CPAN	200, 259	iruzkina	36
D		iterazio-egitura	63
delete	175	J	
deskribatzaile	90	join	162
die	59, 97	K	
E		karaktere-klaseak	128
each	180	keys	177
Emacs	30	komando-lerroa	20, 92
eq	55	komatxoak	45
erreferentziak	189		
eskalarrak	42		

kontrol-egiturak	53	S	
kuantifikatzaile	142	s///	155
L		Sarrera/Irteera	47, 89
le	55	script	25
length	83	seek	134
lerro-jauzia	37	sententzia	35
lt	55	shebang	33, 36
LWP::Simple	253	shell	20
M		shift	72
m//	122	sort	73
metakaraktere	126	splice	75
moduluak	200	split	162
my	43	STDERR	90
N		STDIN	47, 90
ne	55	STDOUT	90
new	205	<i>string</i>	26
Ngramak	201	sub	183
Notepad	30	T	
Notepad++	116	tartea eragilea	51
O		testu-editorea	29
open	94, 104	testu-fitxategi	29
P		Text::Ngrams	201
patroiak	122	tr///	158
Perl interpretatzailea	25	U	
perldoc	205, 259	unshift	73
pop	72	use	205
print	36	use strict	37
printf	215	use warnings	37
programazio-erroreak	107	V	
programazio-estiloa	108	values	178
prozedurak	188	W	
push	72	Web	253
Q		while	63
QWERTY	151	Word	29
R		Writer	29
rand	78	Z	
return	186	zenbakiak	41
reverse	73		

^		\	
^ (bilaketa-mugatzaila)	126	\ (karaktere beraziak ez interpretatu)45, 126	
-		\ (erreferentzia)	191
- (kenketa eragilea)	41	\1	140
- (tarteak karaktere-klaseetan)	129	\2	140
!		\b (hitz-muga)	131
!=	55	\D	130
!~	123	\d (digitua)	130
.		\n	37
.	130	\s	130
.. (tarteak eragilea)	51	\S	130
. = (kateatu eta esleitu)	161	\w	130
. pl (Perl programaren luzapena)	30	\W	130
(		&	
() (biltze metarakterea)	139	&& (eragile logikoa)	57
[		#	
[] (karaktere-klasea)	128	# (iruzkina)	36
[] (array-aren indizea)	49	#! (shebang)	33, 36
{		%	
{ } (agindu blokea zehazteko)	54, 63, 70	% (hash aldagaia)	173
{ } (azpiprogramaren gorputza)	183, 186	% (hondarra eragilea)	42
+{ } (hash-aren elementua atzitu)	174		
{ } (metarakterea)	126, 142	+	
{n, } (kuantifikatzaile)	143	+ (batuketa eragilea)	41
{n,m} (kuantifikatzaile)	143	+ (kuantifikatzaile)	142
{n} (kuantifikatzaile)	143	++	70
@		+=	87
@	19	<	
@_	185	< (txikiago eragilea)	55
*		<= (txikiago edo berdin)	55
* (biderketa eragilea)	41	<=> (konparaketa eragilea)	74
* (kuantifikatzaile)	143	<> (sarrera estandarretik irakurri)	90
** (berreketa eragilea)	41	=	
/		= (esleipen eragilea)	43
/ (zatiketa eragilea)	41	== (berdin eragilea)	55
		=> (gezia)	173
		~= (lotura, adierazpen erregularrak)	122
		>	
		> (handiago eragilea)	55
		> (irteera birbideratu)	94

>= (handiago edo berdin)	55	\$` (parekatze aldagaia)	132
>> (irteera birbideratu)	95	\$_ (aldagai lehenetsia)	114
		\$! (errore mezuen aldagaia)	98
(aukera metakarakterea)	138	\$. (aldagai berezia)	115
(eragile logikoa)	57	\$' (parekatze aldagaia)	132
		\$& (parekatze aldagaia)	132
		\$1 (parekatze aldagaia)	139
\$		\$2 (parekatze aldagaia)	139
\$ (ikurra)	45		
\$ (bilaketa-mugatzaila)	126		

Sailean argitaratu diren beste liburu batzuk

Algoritmika

Rosa Arruabarrena
1997an argitaratua
ISBN: 84-86967-82-1

Ordenadore bidezko irudigintza

Joseba Makazaga, Asier Lasa
1998an argitaratua
ISBN: 84-86967-90-2

Oinarrizko programazioa. Ariketa-bilduma

Arantza Diaz de Illaraza, Kepa Sarasola
1999an argitaratua
ISBN: 84-8438-002-5

Zirkuitu elektriko eta elektronikoen oinarrizko analisia

Olatz Arbelaiz, Txelo Ruiz
2001ean argitaratua
ISBN: 84-8438-018-1

LINUX Sistemaren eta sarearen administrazioa

Iñaki Alegria
2003an argitaratua
ISBN: 84-8438-040-8

Sistema Digitalen Diseinu-hastapenak. Oinarrizko kontzeptuak eta adibideak

Olatz Arbelaiz eta beste
2005ean argitaratua
ISBN: 84-8438-069-6

Softwarearen ingeniariatza [I. atala: Softwarearen garapenaren zenbait arlo]

Jose Ramon Zubizarreta

2006an argitaratua

ISBN: 84-8438-085-8

Softwarearen ingeniariatza [II. ATALA: Garapen monolitikotik hiru mailako arkitekturara bezero/zerbitzariak bisitatuz]

Jose Ramon Zubizarreta

2009an argitaratua

ISBN: 978-84-8438-165-5

LINUX Sistemaren eta sarearen administrazioa. 2. argitaraldia (Debian eta Ubuntu)

Iñaki Alegria eta Roberto Cortiñas

2008an argitaratua

ISBN: 978-84-8438-178-5

